Structuring Formal Requirements Specifications for Reuse and Product Families NAG-1-2242

Deliverable

Title: Final Report

WBS/Task: 2

Date: September 9, 2001

Grant

Number: NAG-1-2242

Project Title: Structuring Formal Requirements Specifications for Reuse and Product Families

Contractor: University of Minnesota

Principal Investigator

Name: Dr. Mats P.E. Heimdahl

Title: Associate Professor

Phone: (612) 625-2068

Fax: (612) 625-0572

Email: heimdahl@cs.umn.edu

Structuring Formal Requirements Specifications for Reuse and Product Families:

Final Report

Mats P.E. Heimdahl

(612)-625-2068
heimdahl@cs.umn.edu

Department of Computer Science and Engineering
University of Minnesota
4-192 EE/SC Building
200 Union Street S.E.
Minneapolis, Minnesota 55455

Abstract

In this project we have investigated how formal specifications should be structured to allow for requirements reuse, product family engineering, and ease of requirements change. The contributions of this work include (1) a requirements specification methodology specifically targeted for critical avionics applications, (2) guidelines for how to structure state-based specifications to facilitate ease of change and reuse, and (3) examples from the avionics domain demonstrating the proposed approach.

Table of Contents

1 Introduction

Incomplete, ambiguous, or rapidly changing requirements are routinely cited as one of the major cost drivers for software development. In addition, in the domain of safety critical systems researchers have found that requirements errors are more likely to impair safety than errors introduced during design or implementation.

Using a formal notation to specify the requirements addresses most of the problems with incompleteness and ambiguity. Languages based on finite state machines such as Statecharts, SCR (Software Cost Reduction), SpecTRM (Specification Tools and Requirements Methodology), and RSML (Requirements State Machine Language), have been successfully used in a number of projects related to NASA's mission. These languages are easy to use, allow automated verification of properties such as completeness and consistency, and support execution and dynamic evaluation.

However, a formal requirements specification does not solve the problems incurred by rapidly changing requirements. The specification must be structured in such a way that it is easy to change and the impact of the changes is limited. Moreover, to reduce the cost of software development the requirements specification should be structured in a way that allows for requirements reuse and the development of product families. Few guidelines have been defined describing how formal requirements specifications should be structured to achieve these objectives. Those that do exist are inadequate and do not sufficiently address the tradeoffs affecting the structure of a state-based specification.

In this project we have investigated how formal specifications should be structured to allow for requirements reuse, product family engineering, and ease of requirements change. The contributions of this work include (1) a requirements specification methodology specifically targeted for critical avionics applications, (2) guidelines for how to structure state-based specifications to facilitate ease of change and reuse, and (3) examples from the avionics domain demonstrating the proposed approach.

1.1 Reading This Report

This report contains a short overview of the project as well as the product resulting from the work—a draft methodology description. Since the work grew beyond the initial scope of the project, we have also included a copy of a dissertation that resulted from this project.

2 Project Overview

2.1 Literature Survey

To ensure that we did not overlook any important trends or duplicated work, we performed a literature study covering the material relevant for this project. In particular, we investigated the current state of the art with respect to requirements modeling and product families. This survey is included as part of the methodology in Appending A as well as in the dissertation included as Appendix B.

2.2 Development of a Methodology

A methodology consists of a set of strategies that make an approach work and the steps that must be followed to apply that approach. None of the methodologies we investigated at the start of this project, including SCR, RSML, SpecTRM, Statecharts, and object-oriented methods, adequately addressed the issue relevant for this investigation, namely structuring for product family engineering. Since no acceptable approach was found during the literature survey, we defined a methodology that addresses the issues by emphasizing the best strategies of the existing approaches. A partial list of these strategies include:

- Making a clear distinction between the environment and the system.
- Stating requirements as constraints on the environment.
- Clearly relating the system and software requirements.
- Presenting requirements in a form that can be read by all stakeholders.
- Using executable requirements models to drive simulations of the user interface.
- Separating the essential requirements of the system from user interface requirements.
- Separating the essential requirements of the system from the hardware interface requirements.
- Anticipating change and organizing the requirements to minimize the effects of change.

These strategies were used to guide the development of the methodology. The methodology defines both the strategies and the set of steps to be followed in developing the requirements model. The final draft of the methodology is included in Appendix A and constitutes the final deliverable in this project.

Appendix A - Draft Methodology

Product Families, Formality, and Reuse: A Guide to the FORM$_{PCS}$ Method

Jeffrey M. Thompson
Mats P.E. Heimdahl
Department of Computer Science and Engineering
University of Minnesota

Draft produced on September 29, 2002

Contents

List of Figures

List of Tables

Acknowledgements

The authors wish to thank Dr. Steven P. Miller from the Advanced Technology Center at Rockwell Collins, Inc, Cedar Rapids, Iowa. Dr. Miller provided the initial outline of a methodology that initiated this work and has been instrumental in the evolution of our thinking. Much of the material in Chapters 1 and 2 was contributed by Dr. Miller in the early stages of this project. His contributions, feedback, and knowledge of the domain are deeply appreciated.

Chapter 1

Introduction

Incomplete, ambiguous, or rapidly changing requirements are known to have a profound impact on the quality and cost of software development. Fred Brooks states the problem succinctly in [5]

> *The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements...No other part of the work so cripples the resulting system if done wrong. No other part is as difficult to rectify later.*

Studies have shown that the majority of software development errors are made during requirements analysis, and that most of these errors are not found until the later phases of a project. Other studies have shown that due to the amount of rework that has to be done, the cost of fixing a requirements error grows dramatically the later it is corrected [4],[7],[29],[24]. In one well-known study conducted at TRW, it was found that it costs ten times as much to correct a requirements error during unit testing than during requirements analysis. Correcting a requirements error after a product had been deployed increased the cost by 100 to 200 times [4]. Moreover, requirements errors are often the most serious errors. Investigators focusing on safety-critical systems have found that requirements errors are most likely to affect the safety of embedded system than errors introduced during design or implementation [16], [19].

The need for better methods and tools for requirements analysis has long been cited as one of industry's primary needs. for example, in 1990, Rockwell Collins Inc. identified improving requirements capture as its highest priority to the Software Productivity Consortium, stating that "requirements are incomplete, misunderstood, poorly defined, and change in ways that are difficult to manage"[8].

Perhaps even more importantly, solving the requirements problem is an essential step in solving many other software development problems. The disjunction between systems and

software engineering, often cited as a major cost in the development of avionics systems, is precisely a problem in requirements allocation and translation. Software verification is widely recognized as one of the largest costs in developing safety-critical systems, but most of the remaining ways of reducing verification costs, such as automating the testing process or automatic generation of test cases, require formal requirements and design models. Out-sourcing software development has been proposed as a way of reducing development costs, but this is impractical until we are able to generate precise, readable, and unambiguous specifications of the requirements—requirements that can then be used as the basis for out-sourcing.

This document describes the FORM$_{PCS}$ requiremetns development method for creating models of subsystem and software requirements that are precise, readable by a wide audience, and robust in the face of change. The methodology integrates the perspectives of system and software engineering, supports the concept of product family engineering, and helps to identify oversights and inconsistencies early in the lifecycle. While strongly influenced by CoRE [8], [9], [10], SCR [14], and RSML [13], [15], the methodology is largely notation independent and is meant to be compatible with a number of commercially available tools and notations, including many of the emerging object-oriented notations.

FORM$_{PCS}$ is a coherent requirements development method that aims to achieve the following:

Separate System Requirements from Software Requirements: Confusion over what constitutes *system requirements* versus *sub-systems* or *software* requirements is a source of problems in requirements modeling. If the initial requirements are modeled at the wrong level of abstraction, changes can be very difficult to accomplish. FORM$_{PCS}$ defines the distinction between these levels of abstraction and provides guidelines on how to refine the *system requirements* to *software requirements*.

Provide Guidance: FORM$_{PCS}$ provides *explicit* steps on how to move from the initial informal requirements, through product family engineering, to a rigorous statement of the required software behavior. The steps are developed to provide guidance at a level suitable for industrial application and this document could serve as a text in a requirements modeling course in the critical systems domain.

Integrate Product Family Engineering and Formal Modeling: Although FORM$_{PCS}$ is intended to be notation and language independent, the primary target notation for the work is some formal modeling language (such as RSML^{-e}, SCR, or VDM-SL). FORM$_{PCS}$ provide guidelines on how the ideas in product family engineering can be integrated into these formal modeling techniques. The aim is to achieve flexible reuse in the *requirements domain* as well as in the code domain.

Avoid Premature Design: FORM$_{PCS}$ allows the analyst to model the desired behavior of the proposed system at increasingly refined levels of abstraction. These models

capture the desired relationship between the various variables in the system. This allows an analyst to specify what the software shall do with little introduction of (software) design details.

Include Continual Requirements Evaluation: To allow an analyst to evaluate the proposed behavior of a new system, FORM$_{PCS}$ advocates modeling in an executable language and continual dynamic evaluation of the model through simulation—a development approach we call *specification based prototyping*. This evaluation will allow early customer involvement, enhance requirements elimination, and provide better risk management as compared to not using executable models.

1.1 Purpose of this Guide Book

this method guide provides a detailed set of guidelines on how to apply the FORM$_{PCS}$ appraoch to requirement modeling. It is intended both as a reference guide for experienced analysts and a self-study guide for the inexperienced analyst. The guide addressed the following:

- An overview of common problems in requirements modeling.
- An overview of the fundamentals of requirements; what is a requirement and how do we separate system requirements from software requirements. We provide an short description of the most influential related work and present the FORM$_{PCS}$ view of this issue.
- A new way of viewing product families, their commonalities, and their variabilities.
- Guidelines and step-by-step instructions on how to scope a system, structure the requirements, and refine the system requirements to software requirements.
- A process for developing requirements for control applications.
- Illustration of the technique through a running example and two completed requirements models.

1.2 Intended Audience

This guidebook is intended for the practicing engineer that is developing requirements for various control oriented applications. Our primary concern is safety critical applications, but the techniques are applicable to all types of control systems.

We assume the reader has experience in the development of control applications, especially the critical issues involving time and interaction with hardware, to full appreciate the material covered in this guide. In addition, we assume familiarity with finite state machines, sets, and Boolean expressions. Any computer science textbook on discrete mathematics will serve as an appropriate reference for the reader unfamiliar with these topics.

The notation used in the running example (RSML^{-e}) will not be fully described in this report—detailed discussions of RSML^{-e} are readily available in, for example, [58, 60, 61] and a formal description of the language is available in [64].

1.3 Scope of the Method and Guidebook

The FORM_{PCS} covers modeling of the required behavior of control systems. It addresses the identification of the system boundary, identification of commonalities and variabilities for product family engineering purposes, and the structuring of the requirements models.

Although the thinking presented in this guide is widely applicable, it is mostly applicable to systems where it is intuitive to think of the software as controlling some physical system, that is, a system where the software is responsible for monitoring changes in the environment (using sensors) and effecting the environment through control commands (using some actuators). Naturally, one can view organizations as physical systems and much of the thinking could be extended to apply in the information systems domain.

Chapter 2

The Problems with Requirements

In developing a methodology for specifying and modeling requirements, there are countless choices that must be made. These choices should be made to solve the most important problems faced by system developers. This section discusses the most important issues in requirements specification and modeling that arise in industrial use. Every methodology choice should be traceable back to one of these issues. To be successful, a methodology must address all of these issues.

2.1 Integrating Systems and Software Engineering

System and software requirements are inextricably intertwined. Unfortunately, systems and software engineers often use different paradigms, different notations, and have different areas of expertise. Since the system engineers usually work more closely with the customer, this mismatch can cause discrepancies to be found late in the program, possibly even after delivery. In many cases, even the format and level of detail of the requirements are not agreed upon. This can lead to an expensive and error prone process as the system requirements are translated or expanded into the form needed by the software engineers. This difference in paradigms can also lead to an incorrect perception of the cost of change. Systems engineers often make what seem to be very simple changes without appreciating the full cost of implementing those changes in software.

2.2 Volatility of Requirements

Few things cause more havoc on a software project than constantly changing requirements, yet the requirements almost always change. Requirements change due to a variety of reasons, including changing customer expectations, not understanding what the customer wanted in the first place, changing system architectures due to a lack of good systems

engineering practices, and changing the hardware interfaces to the software. Requirements may even change due to competition between vendors that impact the schedule or scope of ongoing projects.

2.3 Legacy Systems

Legacy systems are both one of a company's greatest assets and one of their greatest liabilities. During maintenance, requirements changes must be written, implemented, and verified. Unfortunately, legacy systems are usually based on textual requirements and are not structured so that a new method or notation can be easily introduced. Compounding the problem, the cost of modeling requirements may appear so great as to not justify the benefits, particularly if the project manager has not seen requirements models used on other projects. Even new projects are usually patterned after existing systems. As a result, determining how a project can migrate from its current practices to a new method and system architecture can be a major hurdle.

2.4 Planning For Reuse

Closely related to the issue of legacy systems is that of planning for reuse. In today's competitive environment, companies tend to make variations of the same product over and over. A significant cost of each such product is defining its requirements and ensuring that they meet the customers needs. While the requirements of each such product may appear to vary widely, the essential behavior of these systems are largely the same. Unfortunately, most methods and notations for requirements specifications do not provide guidance on how to reuse common portions of the requirements and how to minimize the cost of incorporating changes from one product to the next.

2.5 Satisfying All Stakeholders

Software requirements have to meet the needs of a diverse set of stakeholders, including but not limited to the customer, end users, program management, systems engineers, software engineers, hardware engineers, test engineers, and regulatory agencies. They must be clear enough that end users can understand them, yet complete and precise enough that the software engineers can implement the correct system and develop a comprehensive set of test cases. It is very difficult for one notation to meet all these needs. Often, the solution is to produce a different presentation for each audience. However, this introduces a new problem of maintaining consistency between the various presentations.

2.6 Identifying the Customer's True Needs

Software requirements seldom specify the system actually needed by the customer. They are invariably incomplete, incorrect, and inconsistent. This occurs for a variety of reasons. The customer usually does not have a precise understanding of the system they want. The requirements for the software may not be complete until well into the project, so design may have to proceed with incomplete requirements to meet schedule. The level of detail needed to fully specify the software is seldom appreciated, so requirements are usually incomplete. Requirements are usually specified in English, which is notorious for being ambiguous. It is also very difficult to check an informal English specification for omissions, inconsistencies, and errors. Finally, there are normally so many requirements that they can easily contain inconsistencies that don't surface until design or implementation. Unfortunately, current tools for requirements capture do little to help identify these errors.

2.7 Avoiding Implementation Bias

At the same time, it is also important to avoid over-specification of the requirements, introducing design and implementation issues. Over-specification constrains the designers of the software, preventing them from using their expertise to make choices that would result in a better system. It also makes it difficult to determine what part of the requirements can be changed once the product is fielded in order to meet new demands and constraints. The reason that implementation bias creeps into design is that most methods fail to distinguish clearly between requirements and high level design. As a result, there is a need for good criteria for deciding when the requirements have been adequately specified.

2.8 Support for Automated Tools

Automated tools are not essential to address the issues raised above. In fact, a completely manual method that addresses these issues will produce better results than an automated method that does not. However, automated tools can be of immense value in keeping a requirements model consistent, checking for certain forms of errors, generating test cases, and producing different view of the model. Executable models, when combined with a mock-up of the system and user interfaces, can be invaluable in validating the requirements with the customer. For all of these reasons, it is important that a method be supported by automated tools.

Chapter 3

Limitations with Current Approaches

Surprisingly, methods and tools for requirements modeling and analysis are much less mature than those available for the later stages of coding and design. Requirements have traditionally been stated as English statements, and requirements analysis is often limited to informal inspections, tracing between system, software, and hardware requirements, and tracing between software requirements, design, code, and tests. However, English specifications are infamous for being ambiguous, incomplete, incorrect, and inconsistent. If we are lucky, these problems are found at considerable cost during design, implementation, or testing. In this chapter we provide a short overview of the state of the art in requirements specification and modeling, and point out the limitations.

3.1 Natural Language Requirements

The overwhelmingly most popular language for requirements capture is natural language (for example, English). Natural language has many advantages.

Flexible: A natural language specifications is infinitely flexible and can be used to express the requirements for essentially any system. Natural language is not subject to the arbitrary modeling restrictions more formal notations impose. Natural language can be sued to capture both the intended behavior of a proposed system as well as all non-functional requirements that can often not be captured formally (for example, maintainability, scalability, etc.).

Universally understandable: Natural language is universally understood by the development teams and no extra training is needed for a team to understand and use a natural language requirements document.

Easily accepted: Natural language requirements are the accepted practice in industry

and there is little or no resistance introducing natural language as a requirements language into an organization with a non-existing requirements process.

Naturally, there are many disadvantages with natural language documents—they are notorious for being ambiguous, incomplete, poorly organized, and generally very large and cumbersome to read.

Ambiguous: Natural language is inherently ambiguous and the meaning of a natural language statement is always open to multiple interpretations. This can be somewhat alleviated by using a restricted and well defined subset of the language for requirements definition. This does, however, not solve this problem to a large extent.

Incomplete: We make a distinction between internal and external completeness. Internal completeness means that we have covered all aspects of the cases the requirements address, that is, if we have covered what to do in case condition C holds, we also have to cover what to do if C does not hold. External completeness means we have identified all the relevant customer requirements. Naturally, external completeness is much harder to achieve. In a natural language requirements document, both notions of completeness have to be assured manually through inspections—a very difficult task. Consequently, natural language requirements are typically both internally and externally incomplete.

Large and Un-organized: these properties are not inherent in natural language requirements documents—they just seem to become large and poorly organized over time. Well written natural language requirements do not suffer from these problems. Most natural language requirements documents, however, are not well written—they typically treat the document as a combination of requirements, design, users manual, and tutorial. Consequently, the document grows out of control and becomes basically useless as a requirements document.

3.2 Formal Models

A better approach is to refine the English statement of the requirements into a precise model that can be executed. Requirements models are written in notations specifically developed to make requirements readable and mathematically precise, such as SCR [14], RSML [15], SpecTRM [18], Statecharts [12], and RSML^{-e}. Creating models based on these notations have been shown to find a wealth of errors in textual specifications[13],[21]. Moreover, such models can be connected to a mock-up of the user interface and executed with the customer in the same way as a simulation, or some tools also support "hardware-in-the-loop" simulations, for example, the NIMBUS environment for RSML^{-e}. In the best

approaches, the underlying notation has been carefully designed to support automated analyses. These make possible a variety of consistency and completeness checks that find many errors, as well as the ability to check for properties specific to the application being modeled. Finally, the requirements model itself becomes a detailed statement of the desired behavior. This enhances design and testing, and makes it far more feasible to outsource the software development.

3.2.1 The Early Work

Early work in this area resulted in a collection of notations collectively calls *executable specification languages.* An executable specification language is a formally well defined, very high-level specialized programming language. Most executable specification languages are intended to play multiple roles in the software development process. For instance, languages such as PAISLey [65], ASLAN [4], and REFINE [1] are intended to replace requirements specifications, design specifications, and, in some instances, implementation code. Executable specification languages have achieved some success and have been applied to industrial size projects. Many languages have elaborate tool support and facilitate refinement of a high-level specification into more detailed design descriptions or implementation code.

Nevertheless, current executable specification languages have several drawbacks. Most importantly, the syntax and semantics are close to traditional programming languages. Therefore, they currently do not provide the level of abstraction and readability necessary for a requirements notation [15, 16].

3.2.2 The State-based Notations

Notable exceptions to the first generation of executable specificaiton languages are a collection of state-based notations. Statecharts [20, 21], SCR (Software Cost Reduction) [26, 27], and the RSML [42], are very high-level and provide excellent support for inspections since they are relatively easy to use and understand for all stakeholders in a specification effort. These languages allow automated verification of properties such as completeness and consistency [22, 26], and efforts are underway to model check state-based specifications of large software systems [3, 12].

Software Cost reduction—SCR: SCR is a research prototype created by the Naval Research Laboratory [14]. Based on Parnas's four-variable model [27] (discussed in detail in Chapter XX), SCR specifies a zero-time, black-box model of the system, effectively defining a transform function from the inputs to the outputs. There is no notion of sending or receiving messages or events in SCR. Instead, communication between components is based on change events (a change in value of a component) and simple references to

variables. This discourages the introduction of implementation bias into the requirements (see the discussion of RSML and SpecTRM below). While SCR supports an impressive array of automated analyses, the tool set itself is not ready for industrial use except on small pilots. Also, SCR has limited mechanisms for organizing a model into parts that are likely to change together. This makes it difficult to manage requirements volatility and plan for change and reuse of the model.

Requirements State Machine Language—RMSL: RSML is also a research prototype produced by the Nancy Leveson's group at the University of California at Irvine (later moved to the University of Washington) [15]. RSML was used to specify TCAS-II and this specification was ultimately adopted by the FAA as the official specification for TCAS-II. RSML was heavily influenced by both Statecharts and SCR, and makes heavy use of internal broadcast events for communication between components. In the course of developing the TCAS-II specification, Leveson's group discovered that their most common source of errors was this dependence on internal broadcast [18]. In effect, the specifiers were lured into creating overly complex models that contained implementation bias. To address this concern, they have eliminated the use of internal broadcast events in favor of the change events found in SCR. The new language and tool set is be released shortly under the name SpecTRM by the Safeware Engineering Corporation [18].

Statecharts: One of the most widely known requirements modeling notations is Statecharts. A commercial version of Statecharts is available in the STATEMATE tool [12], but it is based on a functional decomposition paradigm that, besides running counter to the trend towards object orientation among software engineers, also makes it more difficult to extend models, minimize the cost of requirements volatility, and plan for change and reuse. Like RSML, Statecharts makes extensive use of internal broadcast events, rather than change events, for communication between components. The complexity and richness of the notation also makes it difficult to analyze STATEMATE models with automated tools.

RSML Without Events—RSML^{-e}: Initial experiences with RSML (see above) were a success and the language was well-liked by users, engineers, and computer scientists. The explicit event propagation mechanism (shared with Statecharts) , however, was a major source of errors and misconceptions [41]. Therefore, as an evolution, researchers eliminated the events from RSML and ordered the computation based solely on the data dependencies of the specification entities. The resulting language, RSML^{-e}, has a fully formal semantics [64] and interfaces for the specification of inter-component communication [23]. RSML^{-e} is a cousin to SpecTRM-RL (see above) described in [41] in that they share similar semantics but the syntax is substantially different. RSML^{-e} has a fully for-

mal semantics and is supported with the NIMBUS environment. The NIMBUS environment is based on the ideas that (1) the engineers would like to have an executable specification of the system early in the project and (2) as the specification is refined it is desirable to integrate it with more detailed models of the environment.

Other Approaches: In addition to these tools, a number of design modeling tools have appeared on the market, such as ObjecTime, Object Geode, SCADE, and Rhapsody. While many of these tools do a fine job of modeling software designs, there are problems with using them to model requirements. Since they are intended as design tools, they blur the distinction between requirements and design, force implementation bias into the requirements, and obscure the requirements with design artifacts.

Unfortunately, there are no clear front runners in the list described above. Choosing between commercially available tools emphasizes the importance of first identifying the issues to be addressed, then defining a methodology to be followed in developing the requirements. Tools can then be selected or developed based on their ability to support the methodology. Unfortunately, there are also few good methods for requirements modeling. The CoRE method from the Software Productivity Consortium is one of the most complete methods available [8], [9], [10], but SPC is not evolving the method further and provides only limited support for it. SCR is based on the same methodology as CoRE, but a definitive description of the methodology does not exist. While RSML has been very successfully practiced by its originators, they have not published a full description of the methodology they followed in using RSML. A similar criticism can be leveled against SpecTRM and RSML^{-e} also—they have been language and tool oriented, but method support has been lacking.

3.2.3 The Role of Object Orientation:

Recently, object oriented methods have received a great deal of attention. In particular, the Unified Modeling Language (UML) is gaining considerable acceptance in industry. On the surface, these notations appear similar to languages such as SCR and RSML. However, they all currently lack a precise semantics. As a result, users may interpret the same specification in very different ways and tool vendors are free to provide different interpretations with their tools. Also, most of these notations were not developed as requirements modeling languages. As a result, they include constructs that should not be used in modeling requirements, lack some constructs that are needed, and seldom come with a sound methodology for specifying requirements. The attraction of such notations is their widespread appeal to industry and the promise of commercially available tools.

3.3 Prototyping

A common approach to finding requirements errors sooner is to create a rapid prototype of the system before starting the design phase. Besides flushing out many oversights, this provides a simulation that can be explored with the customers to ensure their needs are completely understood. The disadvantage is that simulations can be expensive to create, often model only a portion of the behavior, and are usually discarded once the actual product is developed.

There are two main approaches to prototyping. One approach is to develop a draft implementation to learn more about the requirements, throw the prototype away, and then develop production quality code based on the experiences from the prototyping effort. The other approach is to develop a high quality system from the start and then evolve the prototype over time. Unfortunately, there are problems with both approaches.

The most common problem with throw away prototyping is managerial, many projects start developing a throw away prototype that is later, in a futile attempt to save time, evolved and delivered as a production system. This misuse of a throw-away prototype inevitably leads to unstructured and difficult to maintain systems.

Dedicated prototyping languages have been developed to support evolutionary prototyping [36, 46]. These languages simplify the prototyping effort by supporting execution of partial models and providing default behavior for under-specified parts of the software. Although prototyping languages have achieved some initial success, it is not clear that they provide significant advantages over traditional high-level programming languages. Evolutionary prototyping often lead to unstructured and difficult to maintain systems. Furthermore, incremental changes to the prototype may not be captured in the requirements specification and design documentation which leads to inconsistent documentation and a maintenance nightmare.

Software prototypes have been successfully used for certain classes of systems, for example, human-machine interfaces and information systems. However, their success in process-control systems development has been limited [13]. Clearly, a discussion of every other prototyping technique is beyond the scope of this paper. Nevertheless, most work in prototyping is, in our opinion, too close to design and implementation or is not suitable to the problem domain of safety-critical systems.

Notable examples of work in prototyping include PSDL [36, 45] and Rapide [43, 44]. PSDL is based on reusable libraries of Ada modules which can be used to animate the prototype. Nevertheless, it seems that this approach would preclude execution until a fairly detailed specification was developed. Rapide is a useful prototyping system, but it does not have the capability to integrate as easily with other tools that we desired. In addition, Rapide's scope is too broad for our needs; we wanted a tool-set that was focused on the challenges presented by process-control systems.

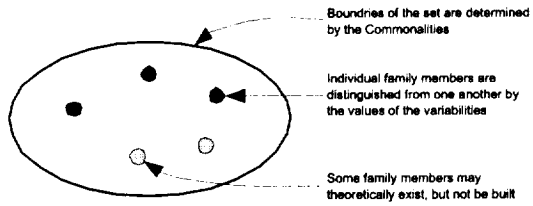

Figure 3.1: A simple product family

3.4 Product Family Engineering

The notion of a product family was introduced by David Parnas in [51]. According to Parnas, it is desirable to study a group of programs as a whole whenever the programs share extensive commonalities. Parnas observed that often programmers would create new programs by modifying existing programs. This process usually involved a reverse step where parts of the working program were discarded. Furthermore, the new program was sometimes crippled by design assumptions made for the original program which did not apply to the new program. Thus, Parnas postulated that it would better to start out by defining what was common about all such programs and successively refining the design until you had working programs as the leaves of a tree structure, with nodes within the tree representing the various design decisions.

Batory and O'Mally [5] discussed how to reuse large portions of a system based on breaking it into components and introduced a simple language for describing the components and their composition. Gomma [18] discusses using domain modeling [56] to create a centralized library of components which are then used by a generation facility to produce the target application.

Weiss and Ardis [63, 2] developed the FAST (Family-oriented Abstraction, Specification and Translation) process that integrates the above with specialized languages [49, 6] for each domain. A similar process is mentioned by Campbell *et al.* in [11, 10] and also by Lam [39, 38]. The differences between these works are primarily in the sort of artifacts produced by domain engineering.

The commonality analysis [62] is a central feature of product-line engineering. This is the document that notes all the commonalities, i.e., features which are present in all systems in the domain, and variabilities, i.e., features which distinguish the different members of the domain. The commonality analysis defines the requirements for the product line.

One way to view a product family is as a set, where the boundaries of the set are determined by the commonalities, and the individual members of the set are distinguished by the values of their variabilities (Figure 3.1). As the figure demonstrates, it is entirely possible that some members of the family may theoretically exist but not yet be built

(shown in gray). Furthermore, the family may be undefined at some points within the boundaries due to, for example, illegal or nonsensical combinations of variability values.

3.5 Summary

To summarize the discussion in this chapter—no current approach satisfactorily addresses the needs for requirements modeling and evaluation of families of safety critical systems. Natural language does not provide the preciseness and analyzability required in this domain. On the other hand, many of the current formal notations are not acceptable by the engineers and software professionals developing these systems. Partial exceptions are notations such as SCR, SpecTRM, RSML, and RSML^{-e} that have has some success in practice. These notations provide support for analysis and execution at an early stage in software development. Therefore, they can serve as prototypes of the proposed system. In particular, RSML^{-e} is supported with the NIMBUS environment that is specifically developed to support specification based prototyping.

Current work in product family engineering has been successful at achieving reuse in limited domains. Many lines of research are helping push the current state-of-the-art including new techniques for implementing product lines and expressing product line architectures. In this report, we will address techniques for recording and reasoning about the structure of the product family requirements; a topic that is inadequately addressed by current work in the field.

Chapter 4

The System Model

In this chapter we present an overview of the model that underlies our requirements modeling method. The model is heavily based on traditional process control thinking—therefore, we first present an overview of process control systems. This overview provided the background to understand why we have chosen the our underlying framework. We then discuss two related models so that we can contrast and compare our approach to related work. Finally, we present the FORM$_{PCS}$ model of systems and discuss how it is used in our development method.

4.1 Process Control Systems

A system is a set of components working together to achieve some common purpose or objective. A process-control system usually involves an environment (i.e., the world), a program (or multiple programs) whose purpose it is to establish or maintain certain conditions in the environment, sensors and actuators that allow the program to get information about the environment and affect the environment, and finally the operator who can usually input various parameters to the running program and receive feedback from the running program. This is summarized in Figure 4.1.

Consider the environment of aircraft moving along in three dimensional space. In this unconstrained environment, airplanes are free to have midair collisions, disrupt take-off and landing of other aircraft, and so forth. Clearly, this is not desirable; therefore, we need a process-control system for air traffic control that will allow us to enforce certain restrictions in the environment, for example, that planes do not run into one another. To do this, we will have to have some sensors, which will give us data about the position of the aircraft in the system, some actuators which will allow us to make course corrections for the aircraft in the system, and possibly have some operator input to guide these choices.

There are a number of difficulties in constructing process-control systems. First, the

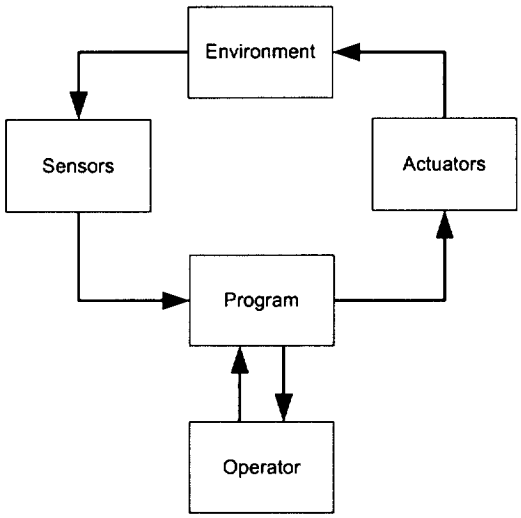

Figure 4.1: A basic process-control model

environment is a key element that is often under specified and/or misunderstood. Misunderstandings about the environment in which the system operates have been the cause of numerous accidents. Second, the sensors and actuators often provide an imperfect, or noisy, view of the real world; sensors can introduce errors, and actuators can fail. Therefore, the program may lose track of the true state of the environment and error conditions in the sensors and actuators can be difficult (or impossible) to detect. Finally, the controller often has only partial control over the process; therefore, state changes can occur in the environment when no actuator commands were given by the program.

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

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

The behavior of the *process* is monitored through *monitored variables* (V_m) and controlled by *controlled variables* (V_c). The process can be described by the process function F_p, a mapping from $V_m \times I_s \times D \times t \rightarrow O_s \times V_c$. Unfortunately, it is usually difficult to

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

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

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

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

This model is an abstraction—responsibility for implementing the control function may actually be distributed among several components including analog devices, digital computers, and humans. Furthermore, the controller most often has only partial control over the process—state changes in the process may occur due to internal conditions in the process or because of external disturbances or the actuators may not perform as expected.

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

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

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

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

Information about the process state has to be inferred from measurements. Theoretically, the function F_c can be defined using only the true values of the controlled variables or component states (e.g., true aircraft positions). However, at any time, the controller has only measured values of the component states (which may be subject to time lags[1] or measurement inaccuracies), and the controller must use these measured values to infer the true conditions in the process and possibly to output corrective actions (O) to maintain F.

This model is an abstraction—responsibility for implementing the control function may actually be distributed among several components including analog devices, digital computers, and humans. The next sections discuss elaborations of this model and what are considered system versus software requirements.

The next sections discuss abstractions of this model and what are considered system versus software requirements.

4.2 The Four-Variable Model and CoRE

The system model that is most closely related to the model used in this paper is the four-variable model developed by Parnas and Madey [54], which in turn evolved out of early efforts to specify the requirements for the A-7 aircraft in SCR [28, 27].

An overview of the four-variable model is shown in Figure 4.2. The variables in this model are continuous functions of time and consist of monitored variables in the environment that the system responds to M, controlled variables in the environment that the system is to control C, input variables through which the software senses the monitored variables I, and output variables through which the software changes the controlled variables O. Note that M and C do not have to be disjoint, some environmental quantities may be both monitored and controlled. For example, monitored values might be the actual altitude of an aircraft and its air-speed, while the corresponding input values would be the ARINC-429 bus words which the software reads to sense these quantities. Examples of controlled variables might be the desired pitch and roll of the aircraft, position of a control surface such as an aileron, or the displayed value of the altitude on the primary flight display.

Four mathematical relations are defined between these variables. The NAT and REQ relations describe how the controlled variables change in response to changes in the monitored variables, i.e., they define the system level view of the specification. The NAT relation defines the constraints imposed by the environment, such as the maximum rate of climb of an aircraft based on its physical characteristics. The REQ relation imposes additional

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

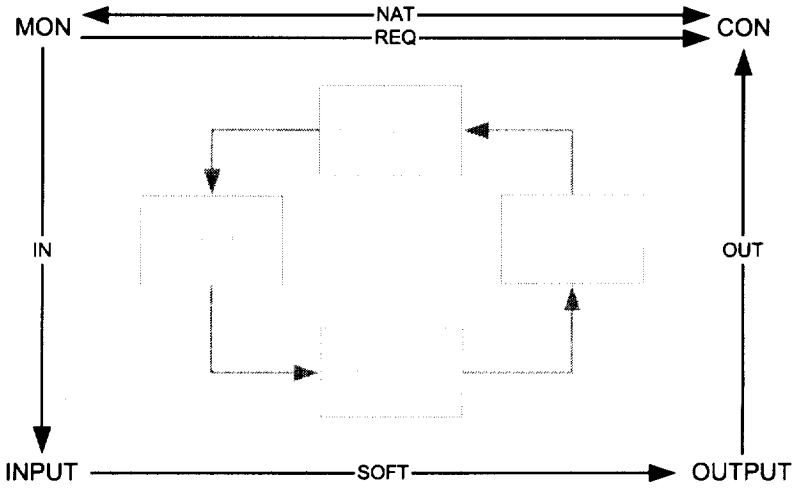

Figure 4.2: The four-variable model.

constraints on how the environmental quantities M and C may change. REQ defines how the controlled variables are to respond to changes in the monitored variables—REQ defines the required control behavior. In other words, NAT describes how the environment behaves in the absence of the controller to be built, while REQ describes how the environment is to be constrained by that controller.

The model is completed by defining the IN relationship relating the monitored variables M to the input variables I and the OUT relation relating the output variables O to the controlled variables C, effectively modeling sensors and actuators surrounding the software. Specification of the NAT, REQ, IN, and OUT relations define the allowable behavior of the control software, shown in Figure 1 as SOFT, without specifying its design—from REQ, NAT, IN, and OUT, the software relation SOFT can be derived. In addition, this separation into relations provides a useful separation of concerns by partitioning the specification of the hardware from the system level specification.

To augment the four variable model and support the SCR language, the CoRE (Consortium Requirements Engineering) [57] methodology was produced by the Software Productivity Consortium (SPC). Many talented people contributed to the development of CoRE and it contains many valuable ideas for the development of process-control systems. In particular, The CoRE guidebook [57] provides technical information on how to document the environmental variables and how they fit into the four-variable model, and they provide some guidance on which environmental quantities are suitable candidates as monitored and controlled variables.

The CoRE process begins with the system requirements and ends with a software

requirements specification. The overall CoRE process is divided up into five main phases:

1. **Identify Environmental Variables:** In this phase, the specifiers identify environmental quantities that the software can monitor and control. Environmental constraints, i.e. constraints which would exist without the presence of the system, are defined; this is called the NAT relation. Finally, the structure of the system is represented as an entity-relationship (ER) diagram.

2. **Preliminary Behavior Specification:** In this phase, a first draft of the high-level behavioral specification, the REQ relation, is developed. The decision is made as to which environmental quantities are monitored, controlled, or both. The domains of the controlled functions are defined and the monitored variables which effect the value of the controlled variable are recorded. Finally, the number and type of mode machines needed is decided.

3. **Class Structuring:** In this phase, the structure of the system is decided. The CoRE methodology attempts to support a pseudo-object oriented structuring technique which includes specialization and generalization. The primary structuring guidance is to choose the objects based on the physical structure of the system and as an extension to the ER diagram developed in the first phase.

4. **Detailed Behavior Specification:** This phase culminates in the completion of the behavioral specification of the classes identified in the previous phase. The controlled variable functions are completely defined and the other classes are refined. Timing constraints, in terms of when each mode machine is recomputed, are also addressed.

5. **Define Hardware Interface:** In this phase, the characteristics of the sensors and actuators are defined by defining the IN and OUT relations.

In practice, the developer must iterate between these phases of the CoRE methodology rather than proceeding through them in a waterfall-like fashion. The CoRE manual addresses this iterative nature in and provides an overview of both the ideal and the interactive (realistic) development process. This enables CoRE to provide both guidelines on *what* should be contained in the specification as well as *how* the specification should be developed. CoRE further addresses the *how* question by providing entry and exit criteria for each of the key steps in the model.

CoRE includes many good ideas and suggestions for developers. The guidelines on identifying the monitored and controlled variables for the system are useful in focusing the construction of the REQ relation. Also valuable is the process of developing a dependency tree for the monitored and controlled variables early in the specification life cycle. This helps to clarify thinking and avoids circular dependencies, which are not permitted in SCR

and not recommended by Parnas [52]. Finally, the overall process is good and provides important guidance to specification developers on how to proceed with the development effort and what information should be included at the various stages. These guidelines provide some help, but in our experience more guidance is needed to correctly make the crucially important selection and classification of the environmental variables.

4.2.1 Discussion

The four variable model has served as the foundation for several research efforts. Most notably, the work at the Naval research Laboratory on the SCR notation [28, 27, 25, 24, 26] and at the University of Minnesota in their work on specification-based prototyping [59].

The main problems encountered when applying the four-variable model are (1) difficulty in identifying appropriate monitored and controlled variables and (2) the difficulty in refining the requirements (REQ, NAT, IN, and OUT) to the SOFT relation. These problems are not fundamental to the model—there are simply no appropriate guidelines available for these two activities.

Identify environmental variables: The guidelines available for this activity are not sufficient for the practitioner. In [54], the originators of the four-variable model state

> *The environmental quantities include: physical properties (such as temperature and pressure), the readings on user visible displays, administrative information (such as the number of people assigned to a given task), and even the wishes of a human user.*

In a footnote, they provide some additional guidance:

> *Frequently, it is not possible to monitor or control exactly the variables of interest to the user. Instead one must monitor or control other variables whose values are related to the variable of real interest. Usually, one obtains the clearest and simplest document by writing them in the terms of the variables of interest to the user in spite of the fact that the system will monitor other variables in order to determine the value of those mentioned in the document.*

The CoRE guidebook [57] provides technical information on how to document the environmental variables and how they fit into the four-variable model, and they provide some guidance on which environmental quantities are suitable candidates as monitored and controlled variables. From the CoRE guidebook [57]:

- Variable properties of physical objects in the problem scope, e.g., positions, velocities, and temperatures.

- Physical quantities, such as dimensions of physical objects.
- Information passed across the interfaces of physical devices, e.g., device status or device commands. Environmental variables typically abstract the interfaces of physical devices. We look at physical devices because they give us insight into which physical quantities the software system can monitor and control.
- Information provided by or supplied to a human user, e.g., user commands or user displays.
- Undesired events, e.g., failures of components of the system or the software system itself, to which the software system is required to respond.

These guidelines provide some help, bet in our experience, more guidance is needed to correctly make the crucially important selection and classification of the environmental variables.

Refinement to SOFT: Assuming we have captured the relations REQ, NAT, IN, and OUT, we need to derive the SOFT relation. There is preciously little guidance in the CoRE guidebook as well as in the original work on how to achieve this task. This is, in our opinion, a serious shortcoming which we attempt to address in the FORM$_{PCS}$ method.

Finally, this basic paradigm of the four-variable model was extended by the Software Productivity Consortium to use object-oriented concepts to make the specification robust in the face of change and to support product families [8], [9], [10]. These extensions were briefly discussed in the previous chapter and will not be covered further in this guide book—we provide an entirely new structuring approach in this guide.

4.3 The WRSPM Model and REVEAL

Michael Jackson and Pamela Zave have presented a reference model for requirements specifications—the world-machine model [30, 32, 33, 66]. The discussion in this section is based on the formalization of this model provided by Gunter, Gunter, Jackson, and Zave [19].

The main idea behind the world-machine model is a separation of concerns between the world (or the environment) and the machine (or, the system to be built). Jackson *et al.* state that the requirements and problems exist in the world, because it is the world that we wish to change via the introduction of the machine. Thus the WRSPM is based on five artifacts grouped roughly into two categories—the ones relating mostly to the environment (or world) and those that pertain mostly to the computer and software (or the machine). These artifacts are denoted by W, R, S, P, and M as illustrated in Figure 4.3. The artifacts are:

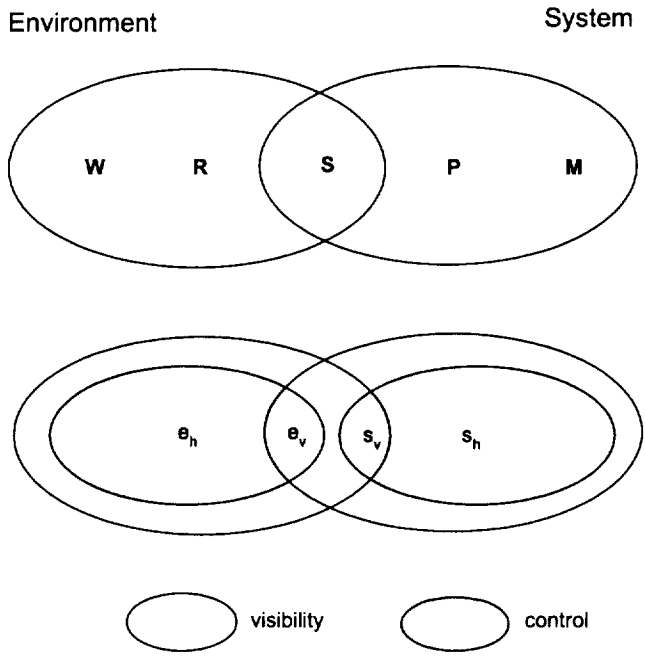

Figure 4.3: The world, requirements, specification, program, and machine (WRSPM) model.

The World (W): This is domain knowledge that captures knowledge of environmental facts.

The Requirements (R): Describes what the customer need from the system expressed in terms of its effect on the environment.

The Specification (S): A less abstract description of the desired behavior that provides enough information for a software developer to design and implement a system that satisfies the requirements.

The Program (P): The program (implemented in some programming language) that implements the specification and runs on some machine.

The Machine (M): The system (computer, associated hardware, operating system, etc.) that executes the program.

Variables that belong in the world are collectively called e—the ones belonging in the machine are called s. The variables in the world e are split into two mutually exclusive sets e_h and e_v—the variables in e_h are hidden from the system and are considered to be exclusively in the domain of the environment. The variables in e_v are visible to both the environment and the system. The variables in s are decomposed in a similar way into s_v and s_h where all variables in s_h are hidden from the environment.

With this decomposition of the variables, e_h, e_v, and s_v are visible to the environment and used in W and R. Variables in e_v, s_v, and s_h are visible to the system and used in P and M. The only variables shared between the environment and the system are in e_v and s_v—therefore, the specification S is restricted to use only variables in e_v and s_v and they form the interface between the environment and the system. Figure 4.3 (from [19]) illustrates the relationship between the variables and the various artifacts.

The WRSPM is related to the four-variable model discussed in the previous section. W corresponds to NAT in the four-variable model. R corresponds to REQ. In the four variable model, REQ and NAT are somewhat more restrictive than W and R in that it can seemingly only make assertions about the variables that are shared between the environment and the system. W and R allow us to make statements about variables that are hidden from the system (e_h). SOFT corresponds to P, and IN and OUT together correspond to M.

The real difference between these two models is in the consistency and sufficiency constraints imposed on these various relations. We will not consider these technical details further in this guide—the interested reader is referred to [19] for a detailed discussion.

The WRSPM model is intended as a reference model only and does not discuss how the various variables in e and s are selected. Nor does the method discuss how the various artifacts are derived or structured—this is a pure reference that simply discusses the required relationship between these different artifacts.

The REVEAL methodology [55] was developed by Praxis Critical Systems, Limited as a method based on the world-machine model. REVEAL consists of six stages:

1. **Defining the Problem Context:** In this stage the goal is to develop an understanding of the problem (i.e., what it is about the world that you wish the system to help to achieve) and explore the boundaries of the problem.

2. **Identifying Stake holders and Eliciting Requirements:** This second stage is associated with identifying stake holders to the project and eliciting requirements and domain knowledge.

3. **Analyzing and Writing:** In the third stage, the requirements and domain knowledge are written down and analyzed using the completeness criteria of the WRSPM model.

4. **Verification and Validation:** The fourth stage involves checking the work that was done in the first three stages to ensure its accuracy.

5. **Use:** After the fourth stage, the requirements will be used throughout the rest of the development life cycle.

6. **Maintenance:** Should any changes to the requirements be discovered, then we must perform maintenance on the description. This is discussed in the final stage of REVEAL.

The REVEAL methodology is based on two key processes: (1) conflict management, and (2) managing requirements. The work in REVEAL on managing requirements is the most relevant to this work.

REVEAL implements a unique notion of traceability of the requirements based on the WRSPM model. In the WRSPM model the requirements are satisfied when the World (W), and the Specification (S) imply the requirements. That is,

$$W, S \vdash R$$

This concept is referred to as the *Adequacy Check* by the REVEAL method. REVEAL uses the adequacy check as a basis for the entire requirements process.

Suppose, for example, that we start out writing down the general requirements for a system that we are building. We would record these requirements, R_{gen}, along with a description of the World, W, and specification, S. Then, we demonstrate that $W, S \vdash R_{gen}$ and life is good.

Now we want to introduce more detail to R_{gen} and produce a set of requirements at a lower level of abstraction. Of course, the detailed requirements, R_{det}, are certainly

related to R_{gen} and certainly that relationship should be preserved in the requirements documentation. Thus in REVEAL, we would prove the property:

$$W, R_{det} \vdash R_{gen}$$

then, by transitivity, we can reuse the original adequacy check on the general requirements so show that the requirements are still satisfied. Doing this provides traceability to the high-level requirements from the detailed requirements and ensures that if the high-level requirements change, the proofs for the detailed requirements will no longer work (as you would expect). This notion of traceability is similar to that proposed by Leveson [17, 40] for Intent Specifications except that in REVEAL the traceability is organized around a more formal framework.

The traceability information recorded by the REVEAL methodology combined with its use of the WRSPM system model make REVEAL a good complement to the CoRE methodology that we discussed earlier. However, a combination of REVEAL and CoRE would still not serve the needs of practitioners because neither methodology adequately addresses the issues associated with recording the requirements for product families. Furthermore, REVEAL does not address specifically the issues associated with state-based specification of process-control systems in a formal language (as CoRE does).

4.4 The FORM$_{PCS}$ More Variable Model

There are variations of the four-variable model that are useful on occasion. For example, it may be helpful to layer the IN and OUT relations into levels much like the ISO Reference Model for communication protocols. Another variation is to "glue" the controlled variables of one or more models to the monitored variables of another model to create a larger system specification or to split a large model up into several smaller models (although care must be taken not to fall into the trap of introducing implementation bias). However, Figure 4.4 depicts the basic paradigm and is adequate for the current discussion.

One problem (or advantage) with the traditional four variable model shown in Figure 4.2 is that it leaves the software developer with the question of how to structure an implementation of SOFT, i.e., how to write the software. One appealing approach is to "stretch" SOFT into the relations IN', REQ', and OUT' as shown in Figure 4.4. In this figure, IN' and OUT' are nothing more than a collection of hardware interface routines designed to isolate the software from changes in the hardware. This conceptual view creates a virtual image of the MON and the CON variables in software, an approach often advocated in object-oriented design methods.

Decomposing the software in this way has several benefits. First, if MON and CON are chosen correctly, the portion of the software specified by IN' will change only as the input hardware changes. Likewise, the portion of the software specified by OUT' will change

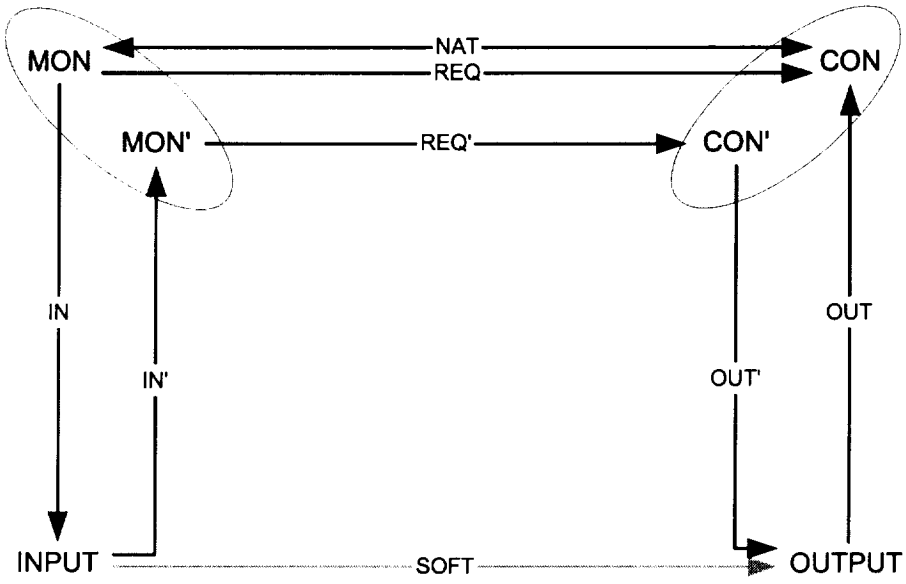

Figure 4.4: The FORM$_{PCS}$ system model adapted from [48, 59]

only as the output hardware changes. In a similar fashion, the portion of the software specified by REQ' will be isolated from hardware changes and will change only in response to changes in REQ, the system requirements. Since customer driven changes and hardware driven changes arise for different reasons, this helps to make the software more robust in the face of change, partially addressing the issues of requirements volatility identified earlier. It also greatly simplifies tracing the system requirements to the software requirements.

Of course, it is important to note that MON' and CON' are not the same as the system level variables represented by MON and CON. Small differences in value are introduced both by the hardware and the software. Differences in timing are introduced when sensing and setting the input and output variables. For example, the value of an aircraft's altitude created in software is always going to lag behind and differ somewhat from the aircraft's true altitude. In safety-critical applications, the existence of these differences must be taken into account. However, if they are well within the tolerances of the system, the paradigm of Figure 4.4 provides a natural conceptual model relating the system and the software requirements. This directly addresses the issue of integrating systems and software engineering identified earlier.

> *Frequently, it is not possible to monitor or control exactly the variables of interest to the user. Instead one must monitor or control other variables whose values are related to the variable of real interest. Usually, one obtains the clearest and simplest document by writing them in the terms of the variables of interest to the user in spite of the fact that the system will monitor other variables in order to determine the value of those mentioned in the document.*

The specification starts as a high-level model of the *system requirements* (i.e., the REQ relation). This model is is then iteratively refined, adding more detail as the system becomes better understood. During each iteration, if a formal, executable specification language is used, the specification is executable and can therefore be used as the prototype of the proposed system. Eventually, the system requirements will be well-defined and the system engineer must allocate requirements to particular hardware and software components within the system. At that point, the *system requirements* can be refined to the *software requirements* by adding descriptions pertaining to the actual hardware with which the software must interact.

From the start of the modeling effort, we know that we will not be able to directly access the monitored and controlled variables—we must use sensors and actuators. At this early stage, we may not know exactly what hardware will be used for sensors and actuators; but, we do know that we must use something and we may as well prepare for it. By simply encapsulating the monitored and controlled variables we can get a model that is essentially isomorphic to the requirements model; the only difference is that this model is more suited for the refinement steps that will follow as the surrounding system is completed.

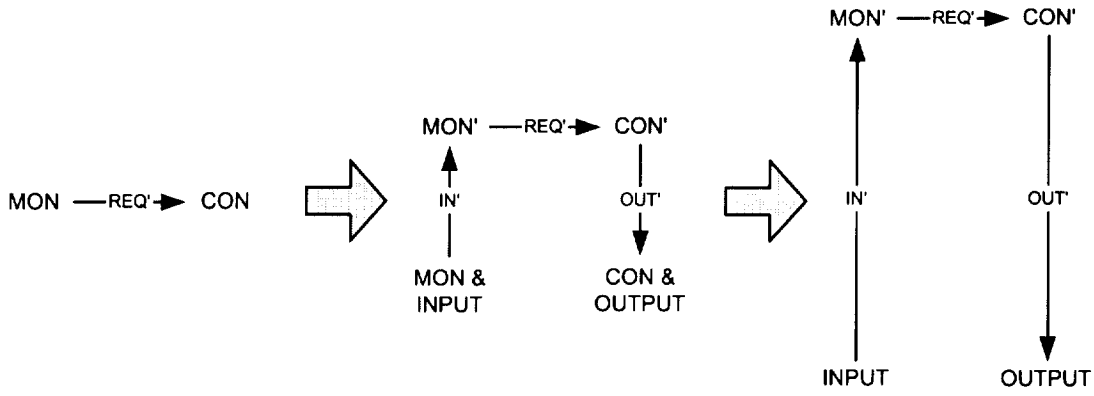

Figure 4.5: Refining REQ to SOFT

The method of this encapsulation differs depending on the language used. If the language does not have a modularity construct, then extra variables or functions can be introduced in the specification to isolate the REQ' behavior from the hardware specification. If the language does have a modularity construct, the specifier may choose to define a module that computes the REQ' relation and then the module's interface naturally provides the encapsulation.

As the hardware components of the system are defined (either developed in house or procured), the IN and OUT relations can be rigorously specified. Figure 4.5 shows a high-level view of the refinement process. At the far left of the figure, we start the process with just a notion of the REQ' relation and evaluate REQ' with the monitored and controlled variables (we basically assume that the sensors and actuators are perfect–there are no delays or noise). Next, we move into an intermediate stage as we add more and more detail to the IN' and OUT' relations. During this stage, the specifications for some sensors and actuators might be completely finished while the specifications of others are under development; this is the reason that both MON and INPUT are noted as the sources for the IN' relation (and similarly for the OUT' relation). Finally, we will arrive at a complete specification of both the IN' and OUT' relations, shown at the far right of the figure.

We have shown in the abstract how the SOFT relation should be structured and our conception of the process that should be used to refine the REQ relation to the SOFT relation. In the next sections, we illustrate this approach by applying it to the ASW and the Mobile Robotics examples.

Chapter 5

Product-Line Engineering Concepts

Today's consumers want devices and systems custom fitted to their needs. Software product-line engineering has the potential to deliver great cost savings and productivity gains to organizations that provide families of products, as well as give those organizations a competitive edge in the market-place. For safety-critical systems, software-product line engineering has the potential to produce systems that are more safe than their serially produced counterparts while being cheaper and faster overall to build.

Even if the goal of the development effort is to the production of a product family, techniques oriented towards product families may still be used to provide for increased flexibility in the long term. One can view the iterative version of a program which are produced through maintenance as a product family.

The commonality analysis [62] is the document which describes the product family. Any product family may be described by listing its *commonalities*, i.e., those features which are shared by all members of the families, and its *variabilities*, i.e., those features which are allowed to vary accross members of the family.

One way to view a product family is as a set, where the boundaries of the set are determined by the commonalities, and the individual members of the set are distinguished by the values of their variabilities (Figure 5.1). As the figure demonstrates, it is entirely possible that some members of the family may theoretically exist but not yet be built (shown in gray). Furthermore, the family may be undefined at some points within the boundaries due to, for example, illegal or nonsensical combinations of variability values.

A family where all the variabilities have a sensical value for all family members, a single product family may work for the entire domain. These sorts of families are most commonly used as examples of product family engineering. However, for many families some variabilities do not have a sensible value for all family members; instead, whether or not a particular variability has a sensical value may depend on the values choosen for *other* variabilities. In this way, the choice of the second variability is effected by the choice of the first; this can be viewed as a hierarchical relationship between these variabilities.

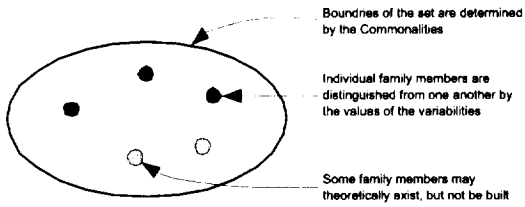

Figure 5.1: A simple product family

Furthermore, many product families contain multiple dimensions; that is, many families have groups of related

5.1 n-Dimensional and Hierarchical Product Lines

Current techniques for product-line engineering work well if the following conditions are met:

- The systems in the family share significant commonalities, and
- The variabilities which define each family member have a straightforward decision model, i.e., it does not require many complicated rules to describe how the variability values are assigned to produce each family member.

The first point describes the essential feature of product families that Parnas noticed in his work. However, the second point originates in the practical experience of many researchers who have labored to construct software product-lines. Robyn Lutz observed that the primary limitations of the product family approach stem from difficulties in handling "*near-commonalities* and relationships among the variabilities"[emphasis added] [47]. Thus, the more simple the relationships among the variabilities, the easier it is to construct the product family.

5.1.1 n-Dimensional product families

Attempts have been made to organize the product family requirements in a hierarchical fashion [47, 51, 37, 38]. Lutz noted in her attempt to organize the variabilities into a tree that "there were several possible trees, with often no compelling reason to select one possible tree over another" [47].

Brownsword and Clements present a shipboard command and control systems family which contained 3000-5000 parameters of variation for each ship [9]. They state that "the multitude of configuration parameters raises an issue which may well warrant serious

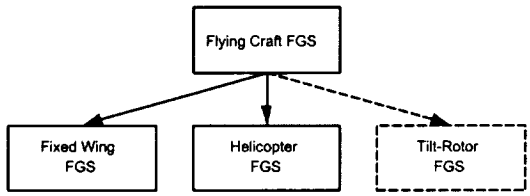

Figure 5.2: FGS product family covering flying craft

attention." In addition, they present three different views of the architectural layering of the base system that "do not conflict with each other; rather they provide complementary explanations of the same ideas."

Both these examples, as well as our own experience in the domain of mobile robotics, illustrate the fact that often a product family is multi-dimensional; therefore, a hierarchical decomposition is not sufficient to capture the structure of the domain. We call such domains *n-dimensional product families.*

5.1.2 Hierarchical product families

Suppose that a company wished to construct a flight guidance system (FGS) for both fixed-wing aircraft and helicopters[1]. The FGS is responsible for issuing commands that keep the aircraft level, cause it to climb or descend, and so forth. Furthermore, the FGS must interact with other airborne systems. Many of the tasks that the system has to perform might be common across these two radically different aircraft: interaction with other systems, deciding to level off at a particular altitude, mode transition logic related to when it is legal to switch between the various operating modes. Therefore, many requirements between these two systems will be the same, or very similar. Nevertheless, the actual control of the aircraft is very different. Therefore, developing a single set of commonalities and variabilities which span this entire domain is difficult.

Some would argue that this difficulty stems from the fact that the family is simply too diverse to be considered a product line. However, it is clear that these systems share much in common, which was the original, and in our view the most important, criterion for being a family. Thus, we propose the concept of a *hierarchical product family.*

Most previous attempts at product family structuring have focused on hierarchically grouping the *variabilities* while the *commonalities* remain the same for all family members [47, 38]. Notable exceptions are Parnas [51] and Brownsword and Clements who noted in their case study at CelciusTech [9] that sometimes product-lines exist within the main product line. However, Parnas' work is based heavily on design and coding choices;

[1]We would like to thank Steven P. Miller of Rockwell-Collins Inc. for this example

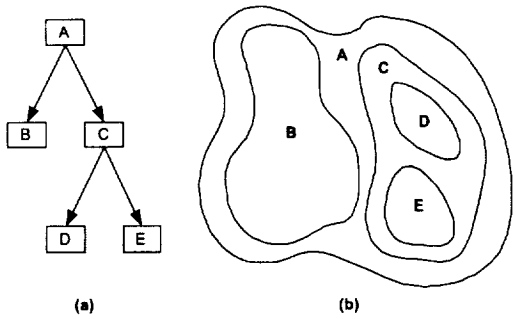

Figure 5.3: Hierarchical decomposition and subset structure

therefore, it is difficult to apply to requirements. Brownsword and Clements mention this phenomenon in passing and apply it in a more limited way than what we advocate.

In our approach, *additional commonalities* which are *unrelated* to the parent product family can be added in the sub-families. The hierarchical decomposition of the FGS family is shown in Figure 5.2. Thus, the helicopter sub-family can have significantly different requirements than for fixed-wing aircraft, yet share many things in common as well.

This will eventually effect the architecture and structure of the systems. For example, the product of the domain engineering for the parent family, Flying Craft FGS, might be a set of reusable components, whereas the product of domain engineering for the children might be a reference architecture or generation facility. The architectures for the fixed-wing aircraft and the helicopters could differ significantly and use the components from the parent family in different ways.

By structuring the requirements in this way, we have avoided imposing restrictive design constraints on the family members and instead focus on the structure of the domain itself. Furthermore, should the company wish to start building FGS systems for an entirely new set of aircraft, for example, tilt-rotor aircraft, this could be done while reusing many aspects of the FGS systems already implemented. This is also shown in Figure 5.2.

5.2 Structuring Families

This section describes how set theory can be used to think about structuring product families. The most basic structure that can be represented with the set theoretic approach is the subset. Figure 5.3 shows a product family, **A**, which has been divided into two subsets, **B** and **C**. Furthermore, **C** has been further divided into subsets **D** and **E**. This corresponds to a hierarchical decomposition of the family.

Consider a member of family **E**, $\mathbf{e}_1$. The member $\mathbf{e}_1$ must have all the commonalities

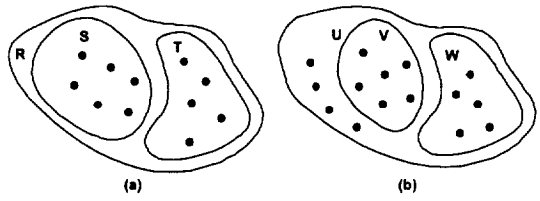

Figure 5.4: Abstract verses non-abstract families

defined for **E** as well as have some values for all the variabilities in **E**. Furthermore, because **E** is a subset of **C** and **A**, e_1 is also a member of families **C** and **A**. The general definition for any family **E** which is a subset of another family **C** is as follows:

- **E** must include all of the commonalities in **C**.
- **E** must include all of the variabilities in **C**; however, **E** may restrict the range or options available in the variabilities.
- **E** can add additional commonalities which are not present in **C** as long as the additional commonalities do not conflict with the commonalities or variabilities in **C**. These new commonalities might come from a refinement of variabilities in **C** or might be completely unrelated.
- **E** can define additional variabilities which are not present in **C** as long as those variabilities do not conflict with the above.

The first criterion is straightforward and necessary for the subset **E** to be completely contained within **C**. The second criterion defines the fact that **E** may wish to refine or restrict the values of the variabilities of **C**. For example, in the mobile robotics domain, a variability across the entire domain might be that the maximum speed of the mobile robot can vary from one to five miles per hour. However, subsets might define a lesser maximum speed depending on the hardware involved. It is possible for this refinement to result in an additional commonality, for example, a subset that instantiates a boolean choice variability to a particular value. Additional commonalities can also be added which are unrelated to the parent family. For example, it is likely that the family of helicopters will need different commonalities than the family of fixed-wing aircraft. Finally, it is possible to add additional variabilities.

The two cases of hierarchical decomposition are shown in Figure 5.4. Part (a) of the figure demonstrates that the family **R** need not have any members that only exist in **R**. In a sense, **R** is an *abstract family*, because any member of **R** must be either a member of **S** or a member of **T**. This is similar to our FGS example from earlier, where all family members are either helicopters or fixed-wing aircraft and it does not make sense to talk

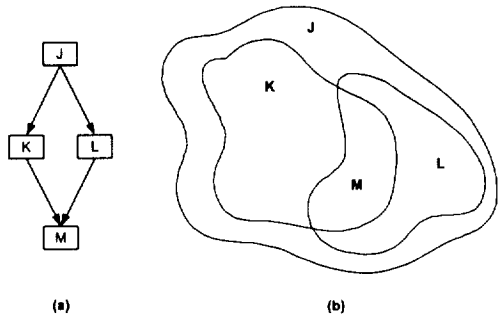

Figure 5.5: Set intersection and non-hierarchical structure

about member which are only of the parent family. However, this need not be the case, as Figure 5.4(b) demonstrates.

Another structure that can be represented using a set-theoretic approach is that of set intersection. The ability to represent a set intersection distinguishes this approach from the purely hierarchical structures which have been applied by others. This is shown in Figure 5.5.

Consider a member, m_1 of **M**. By definition, m_1 is also a member of families **K**, **L**, and **J**. Thus, m_1 must have all the commonalities of both **K** and **L**. In addition, **M** is a subfamily of both families **K** and **L** (this is shown in the figure). The constraints on any family **M** which is a subset of families **K** and **L** are as follows:

- **M** must include all the commonalities of both **K** and **L**.
- **M** must include all the variabilities of both **K** and **L**; however, it may restrict those variabilities as above for subsets.
- **M** may introduce additional commonalities which are not present in either **K** or **L**.
- **M** may introduce additional variabilities which are not present in either **K** or **L**.

These structures can be used to document and reason about the two problems explored in the previous section; they can be used to describe product families which are both *n-dimensional* and *hierarchical*.

5.3 Addressing existing issues

This section describes how our approach can assist with well-known documented issues in product-line engineering. We describe how our structuring method can deal with near-commonalities as well as variability dependencies.

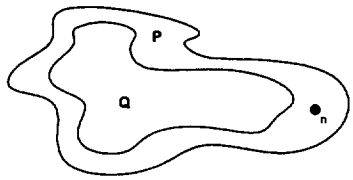

Figure 5.6: Set representation of a near-commonality

Near-commonalities: A near-commonality (NC) is a commonality which is true for almost all (e.g., all except one) member of the product family. Lutz states that in her experience near commonalities "frequently had to be modeled" [47]. One solution for near commonalities is to model them as variabilities; however, this is, in some sense, a misrepresentation of their basic properties. The solution that Lutz advises is to model it as a constrained commonality of the form "If not member n then NC_1." However, a complex domain might contain numerous constrained commonalities with conditions significantly more complex than the example just mentioned.

Figure 5.6 shows how a near-commonality is represented in our approach. The near commonality, NC_1, would simply be a property of family **Q** (and not of **P**). Thus, the commonality naturally does not apply to n, a member of only **P** but does apply to any member of **Q**. This has several advantages. First, NC_1 is now a pure commonality of **Q**. Second, if another member of the family is introduced with reduced functionality [47] it need only be added as a member of **P** and **Q** may remain untouched. Finally, the subset structure can act as a guide in determining that certain components in the eventual application engineering environment will not be needed for n.

Dependencies among options: In [47], Lutz cites modeling dependencies among options as one issues that must be addressed in product family engineering effort. A dependency is typically a constraint among the variabilities, for example, if variability V_1 has value B then variability V_2 must have option C. Ardis recommends treating this constraint as a commonality. However, in our experience, without some additional structuring, the domain could become littered with such commonalities; in addition, it may not be clear given a set of constraints whether or not a particular variability is viable.

In our approach, we can also represent constraints like these as commonalities. However, we isolate them into logical groups by forming different subfamilies so that their numbers do not become overwhelming. In the abstract example given above, a set would be defined where "V_1 has option B" and "V_2 has option C" are both commonalities.

In this section, we have discussed how our approach can help deal with existing issues which have been raised in the literature regarding product families. In the next section,

we go on to describe how this structuring mechanism can help deal with more difficult product families illustrated with an example in the mobile robotics domain.

5.4 Benefits

The structuring technique presented results in the creation of more families within the domain than with a traditional approach. However, these sub-families are more cohesive and simpler than would be the case if we created just one top level-family. We believe that this provides several benefits. First, the top-level family can now be much broader than was previously possible. Second, the overall family can be expanded and contracted by adding and subtracting sub-families. Finally, these techniques will allow a family to be more easily refactored as the definition of the family evolves over time.

The ability to draw a larger product family was an essential requirement for the structuring technique. This grows out of our own experiences with mobile robotics [14], where we had difficulty in applying the product family approach. This difficulty stems from the fact that the mobile robotics domain is both *n-dimensional* and *hierarchical*.

The mobile robotics domain breaks down along two clear dimensions: the hardware platform and the desired behavior. Each hardware platform conforms to a basic specification: it can move forward and backward, turn left and right, sense whether or not an object is in front of it. The hardware platform may also be equipped with a variety of sensors and actuators that give it additional capabilities; and, the various sensors differ greatly in the speed and accuracy with which they provide information. Thus, on the hardware side, there are many different configurations that must be modeled.

On the behavior side, we can imagine that a basic behavior might be a random exploration where the primary goal of the robot is collision avoidance and recovery. More complex behaviors can be added, for example, wall following, going through doors, and finding particular objects. Furthermore, those behaviors may be composed and combined to form a composite behavior. We might envision a behavior which includes the door navigation, a wall following behavior, and a high-level planner. The high-level planning behavior needs to communicate with the random exploration, door navigation, and wall following to direct the robot towards high-level goals. However, if the robot collides with an obstacle, then the lower level behavior will take over and recover from the collision. Thus structure of the behavioral dimension is much different from the hardware dimension and resembles Brooks' subsumptive architecture [8].

Certainly, a domain such as mobile robotics which absolutely requires n-dimensional and hierarchical product families will necessarily be more complex than a domain that does not require these techniques. Nevertheless, any domain can benefit from reuse of the artifacts at the top of the family hierarchy and a more traditional cost-benefit will exist towards the leaves of the family (along each particular dimension).

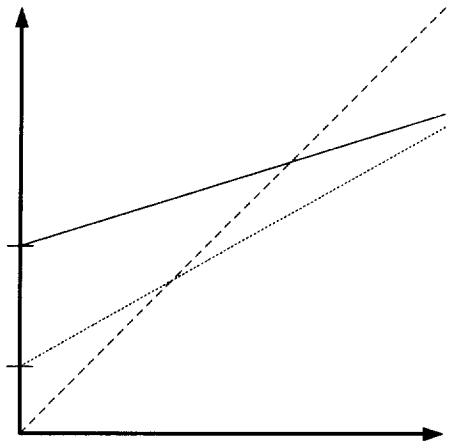

Figure 5.7: Cost-benefit of the FGS Family

Another benefit of the technique is the ability to expand and contract the family as necessary. This ability is essential because it allows a more incremental development of product-lines than is facilitated by current approaches. Furthermore, it facilitates *family refactoring*; that is, the family can be redefined more easily as the product line evolves over time. Thus, this structuring technique has much potential to increase the usefulness of the product family approach.

One of the barriers to traditional product family approaches is that the whole organization must change to accommodate product-line oriented development. Many resources are required to develop the domain engineering support for the entire product line while at the same time continuing to produce products for existing customers. Our approach allows an organization to start out with a high-level product family and reuse just a few key pieces between the major product areas. As the payoff from this reuse makes more organizations resources available, the organization can then afford to make the family more rich (by refactoring and/or adding sub-families) and thus achieving more payoff from the effort.

Of course, these benefits do not come for free. The broader and more flexible view of product families allowed by our techniques will result in families which are more complex than traditional families. In addition, because of this broader view, it may be more difficult to determine what constitutes a viable family under our approach. Almost anything is related in some fashion or other and it may be difficult for organizations to decide when to define an encompassing family for a particular group of subfamilies. Nevertheless, we feel that these techniques hold promise and may serve to advance the frontiers of product-line engineering.

The cost-benefit analysis of our product-line engineering approach is more difficult because one must not only consider the cost of developing domain engineering support of the particular sub-family in which the member resides, but also all sub-families above that one in the product family hierarchy. For example, the cost-benefits for the FGS family is shown in Figure 5.7. The payoff for the fixed-wing FGS is shown by the thick black line and the payoff for the helicopter FGS family is shown by the thick gray line. As the figure demonstrates, the payoff point for the two sub-families is different, because the cost of implementing each over and above the functionality provided by the flying craft FGS family is different. As the structure of the family becomes more complex, for example, through the creation of a deeper hierarchies and/or the use of multiple dimensions with constraints between them, this relationship will become more complex.

Chapter 6

The Methodology at a Glance

Now that we have setup the background of what problems there are in requirements engineering and presented several different models that help analysts think about the requirements for families of embedded systems, this chapter presents and overview of the FORM$_{PCS}$ methodology, which is the subject of the remainder of this report.

We begin by discussing FORM$_{PCS}$ in an idealized setting where the specifier always has all the correct information necessary to make correct decisions at each stage of the process. Often, however, this is not the case [53]. Thus, the idealized process is not necessarily a realistic one. Neverthess, the requirements document, to be useful, should be organized according to the idealized process.

The second half of this chapter deals with the desparity between the realistic process and the idealized process by noting the common says in which one typically iterates between the activities of FORM$_{PCS}$. This organization is similar to that of the CoRE methodology [57].

6.1 Idealized FORM$_{PCS}$ Process

This section describes the idealized FORM$_{PCS}$ process activities. Each subsection below describes a phase of the methodology, begining with the commonality analysis and ending with the specification related to the sensors and actuators in the final, physical system. Along the way, we will define environmental quantities and operator set points, develop an overall structure for the requirements and then develop a draft specification, finally, we will refine that specification (adding, for example, error handling and fault recovery behaviors).

6.1.1 Commonality Analysis

The commonality analysis is the first phase of the methodology. The commonality analysis begins with a short (i.e., one to 5 paragraphs) high-level description of the intended family. This high-level paragraph is then refined until the analyst can begin to identify the *commonalities*, i.e., those features which are present in all family members, and the *variabilities*, i.e, those features which vary accross members of the family. This initial set of commonalities and variabilities forms the basis for the rest of the process.

In the FORM$_{PCS}$ approach, we allow a family to be broken down along different dimensions, for example, a hardware dimension and a behavioral diminesion. In addition, we allow a family to be broken into several sub-families, for example, a general family of flying craft might be broken down into fixed-wing aircraft and helicopters. This family-level structuring occurs as a result of discovering additional commonalities and variabilities during the commonality analysis. Finally, we will examine the commonalities and variabilities in terms of whether they apply to the REQ relation or whether they apply to the IN' or OUT' relations.

At the end of the commonality analysis, you will have a description of the family including all the sub-families and dimensions involved; and, you will have a subset of the commonalities and variabilities that you will use to specify the REQ relation in the next stages.

6.1.2 Environmental Variables

In the environmental variables phase, the goal is to identify quantities in the environment that are important to the specification. Earlier, we discussed several models of viewing the system's interaction with the environment, including our own *more variabile* model. This phase of the methodology provides concrete guidance on how to choose monitored and controled quantities. In addition, this section will demonstrate the characteristics of the various types of environmental variables.

At the end of this phase, you will have a list of all the monitored and controlled variables used in the system cataloged according to thier type. This will form the boundaries of the REQ relation. Furthermore, you will have a statement of the NAT relation, i.e., a statement of the constraints that are imposed upon the environmental variables in the absence of the proposed system(s).

6.1.3 Initial Structure

In the initial structure phase, you will use the environmental variable descriptions developed in the previous phase along with the product family structure identified in the first phase to develop an initial structure of the REQ relation. In languages which support

a module construct, specification entities may be grouped together into pieces that can be reused accross the product family. In languages that do not support a module construct, specification pieces can be formed by textual delimitation and physical grouping. Component reuse can be accomplished by cut-and-paste.

The outcome of the this phase phase will be that the REQ relation is divided into a series of manageable pieces each of which will be specified in detail in the next phase of the methodology.

6.1.4 Draft Specification

In this phase, a preliminary behavioral specification of the system requirements is developed. This first version of the specification will deal primarily with the intended, normal case behavior. While, failure modes and fault tolerance must be kept in mind, these characteristics will be added to the specification in later stages. This phase concentrates on refining the module definitions developed in the previous stage into working pieces of the specification. When all the modules have been defined, then the specification is complete.

The outcome of the draft specification is a document which can be reviewed so that all interested parties can agree on the *essential* behavior of the REQ relation without getting bogged down in details about particular sensors and actuators, or about complex failure modes and error handling. Using RSML^{-e} with the NIMBUS environment, it is possible to simulate the high-level behavior at this point; therefore, everyone involved on the specification effort can get a very good idea of the behavior that was specified.

6.1.5 Detailed Requirements

When producing the Detailed Requirements, the analyst will begin to add to the REQ relation all things that were initially left out of the preliminary behavioral specification. In this phase, we will consider the fault tolerance of the specification, error conditions which may arrise due to the fact that we are using sensors and actutors, and so forth. Also, hear is where we need to consider in more detail the startup and shutdown behavior of the system.

As these new behaviors are added, we may find it necessary to revisit decisions which were made about the preliminary specification as well as about the requirements structure. Thus, it is natural to iterate between these phases.

At this point, the analysts should start to analyze and/or test the completeness and consistency of the REQ specification. Therefore, if analysis tools are available, the REQ specification should be run through these tools and any errors which are found should be corrected.

The outcome of this phase is a completed specification of REQ.

6.1.6 Sensors and Actuators

Phases two through five have illustrate how to move from the commonality analysis in phase one to a completed REQ specifiation. In this final phase, the process will be repeated for the IN' and OUT' phases. In discussing this phase, we point out which parts of the process are generalizable and what information needs to be considered specificially for the hardware.

The outcome of this phase is the completed behavioral specification of the SOFT relation.

6.2 Normal Iteration Among the Phases

In an ideal world, the specifier would proceed through the phases one after the other and never have to go back and modify the products of an earlier phase once that phase was complete. However, researchers know from thier experience with the Waterfall model of software engineering that this is not the case. This section discusses the various common ways that iteration occurs between the various phases.

6.2.1 Constructing Partial Specifications

It is common in process to have one portion of the specification more refined than another portion. This is sometimes a concious choice – focusing on some aspects of the system while ignoring others – but it can also be that certain details were overlooked by accident when the specification was first constructed.

One case where this can happen is after during the Sensors and Actuators phase. Here, some of the specification will still be at the detailed requirements phase while you are refining the specification of a particular sensor or group of sensors and actuators.

Another case is when you abstract away certain portions of the computation. For example, in avionics systems sometimes there are complex conditions that must be satisfied for certain mode transitions. These often depend on control laws, continuous functions, etc. that might not be convient to represent. Furthermore, the way in which these conditions are met is often well understood. Thus, you might wish to put off defining exactly how these conditions are satisfied until later in the specification effort. When this information is added, the new parts of the specification will need to go through the phases of just like the other parts of the specification.

6.2.2 Monitored and Controlled quantities

Sometimes, new monitored and controlled quantities will emerge as you are constructing the preliminary behavior specification. There are several reasons why this might be the

case.

First, you may discover that you need additional information from the environment to be able to compute the values of the controlled quantities. These may be, for example, operator inputs that you did not anticipate.

Second, as you study the system in more detail, it may become clear that there are more controlled variables. For example, the system needs to accomplish 'x' but to do that, we need to introduce a controlled variable which makes that possible.

Finally, sometimes to make it easier to say certain things about the domain it is easier to adjust the particular choices of monitored and controlled variables rather than express a very complex relationship between the ones that you have. For example, in a system that monitores the fuel level in a tank, there are many different monitored quantities that you could choose, a boolean indicating whether or not the liquid is at a certain level, a numeric measurement of the liquid level in the tank, etc. Each of these choices are valid, but have different implications when you construct the preliminary behavior specification and as you refine the specification.

In general, you should think carefully about the choice of monitored and controlled variables but realize that you may have to revist those choices later in the specification effort due to unforseen difficulties.

6.2.3 Draft Requirements and Requirements Structure

It is natural to switch back and forth between structuring and the requirements and developing a specification of the behavior. As you specify the behavior in more detail, you may discover modules or pieces of the computation that may be reused accross different sections. In addition, you may want to reorganize the computation, or refine the interfaces of the modules. Similarly, as you develop the module structure, you may change your ideas about how to specify the behavior, and in what order various computations need to be performed.

The iteration between these activities is similar to the iteration that you would normally see in an object oriented development between the creation of the class diagrams and the creation of sequence diagrams.

6.2.4 Detailed Requirements and Prior Phases

When adding the information in the detailed requirements phase, sometimes you may discover that the structures that you have choosen for the requirements are not conducive to adding fault tolerance, etc. Thus, you may have to restructure or add structure to the requirements to support these additional behaviors.

In general, it is necessary to keep these things in mind from the begining of the specification effort, but beneficial to not get bogged down in the details when first understanding

the system. This is a delcate balance which becomes easier with experience in specification.

Chapter 7

Phase 1: Commonality Analysis

Today's consumers want devices and systems custom fitted to their needs and software product-line engineering has the potential to deliver great cost savings and productivity gains to organizations that provide families of products, as well as give those organizations a competitive edge in the market-place. For safety-critical systems, software-product line engineering has the potential to produce systems that are more safe than their serially produced counterparts while being cheaper and faster overall to build.

Even if the goal of the development effort is the creation of a product family, techniques oriented towards product families may still be used to provide for increased flexibility in the long term. One can view the iterative version of a program that are produced through maintenance as a product family.

Although it is certainly not a definitative reference on illiciting commonalities and variabilities, this chapter describes how to construct a specification of the product family within the FORM_{PCS} method.

7.1 Goals

The commonality analysis [62] is the document that describes the product family. Any product family may be described by listing its *commonalities*, i.e., those features that are shared by all members of the families, and its *variabilities*, i.e., those features that are allowed to vary accross members of the family. For example, in the altitude switch family, all family members will posses some method of assessing the current altitude of the aircraft (using various combinations and types of altimeters). However, altimeters may vary in terms of the quality or charaterists of the altitude that they provide (more on that in the next sections).

The goals of this stage, then, are to do the following:

- Define the top-level family commonalities and variabilities for the system.

- Define sub-families encompassing different areas or dimensions of flexibility within the overall family.
- For each sub-family, the commonalities and variabilities should be cohesive.
- Construct a decision model for the family.

7.2 Entrance Criteria

This is the first phase of the methodology. Before starting, you should make sure that that you have access to domain experts who know about the systems that are present in the domain.

7.3 Activities

This section describes the activities that take place in the Commonality Analysis phase of the methodology. The first objective the phase is to create a high-level description of the family; this is described in the first section. The next two activities involve creating commonalities and variabilities and identifying dimensions and structure in the product line. You may find that you iterate between these three activities has your understanding of the family progresses. Next, the family structure is refined and the decision model for the family is specified. For more information about how to describe a product family, see [63, 62].

7.3.1 Define the Top-Level Family

This section describes the first activity of the FORM$_{PCS}$ method: Defining the top-level family that will form the basis for the specification(s) developed in the later phases. Thus, this is an important activity. We will begin discussing this activity by describing our running example: the altitude switch. Next, we will describe how this family is scoped for the purposes of this methodology. Finally, this activity ends when a one to three paragraph description of the family has been generated.

In avionics, the altitude of the aircraft is an essential environmental quantity. Many devices on board the plane react to changes in the altitude, for example, the autopilot must know the plane's current altitude in order to know whether to climb or descend. In addition, there are many other devices on board the plane that rely on altitude. However, these different devices vary greatly in the types of actions that the perform in response to the altitude data. In addition, the types of altitude data differ significantly from system to system and from aircraft to aircraft. We might make an intial attempt at a family description such as the following:

The ASW family consists of systems on board the aircraft that utilize the values from the various altimeters on board to make a choice among various options for actions (one of which being to do nothing) and perform the choosen action.

Our family could be viewed as a sub-family of a larger family that would include all aspects of avionics systems. This description does describe all the systems on board the plane that use the altitude and is therefore a good starting point for describing our family. However, notice that the particular actions that the system performs can be largely separated from tasks relating to measuring the altitude and fusing the results from various different types of altimeters. We will refine this further in the next section where we talk about the high-level commonalities and variabilities for the ASW family.

In summary, to develop the high-level definition of the product family you do the following:

- Brainstorm about the family: What systems are potentially members of this family? What sorts of functionality do those systems have? What are the common threads that tie these sytems together?
- Next, develop a first pass description of the family. Have other members of the product team and the domain experts review this description.
- Finally, develop a specific, one paragraph description of the product family that clearly conveys the basic ideas behind the systems under consideration. Be specific, but avoid introducing too much detail.

In the next section, we will start to elaborate on our understanding of the ASW product family by developing an initial list of commonalities and variabilities for the ASW family.

7.3.2 Initial Commonalities and Variabilities

We can begin to develop a list of initial commonalities by examining the system description. Furthermore, it is unrealistic to try and list either all the commonalities or all the variabilities at one time. Sometimes, it is easier for domain experts to identify the variabilities; however, it may be difficult for them to the precise way in that the variability values must be assigned to produce a viable family member. We denote commonalities by a 'C' and then a number so that they can be referenced elsewhere in the document (variabilities are noted in simliar fashion).

Generally, commonalities and variabilites can be highly related and should be grouped together if they related to the same parts of the system description. These grouping provide the basis for discovering the dimensions and sub-families that define the structure of the domain.

In is often useful to start out with high-level commonalities and variabilities and work towards a more refined description of the family. The highest level commonalities define the boundaries of the broadest possible product family. As more commonalities are added, the definition of the family becomes more refined. It is useful to preserve those commonalities that define the outermost scope of the family – these are the least likey to change in the future and, thus, should depend on physical principles of the essential purpose of the system.

Along the same lines, commonalities and variabilities can also be separated by whether or not they apply generally to the REQ relation or to the IN' and OUT' relations. This is a useful separation because in the next phases of the methodology, we will be concentrating on the REQ relation and will get to the IN' and OUT' relations in the last phase.

Obviously, there is much information on illiciting commonalities and variabilities, managing the process and meetings, etc. that we have not included here. That is a topic that is covered in many other books and references, including [63, 7, 35] among others.

All of the commonalities and variabilties for the ASW system are listed in Appendix A. As an example, consider the highest level ASW commonalities and variabilities for the ASW.

C1 All ASW systems will have a way to measure the altitude of the aircraft

- C1.1 The ASW system will use the information about the aircraft's altitude to make a decision as to what action the ASW should perform

V1 The actions that the ASW takes in response to the altitude and the criteria to perform those actions varies from aircraft to aircraft

This is really just an alternative way to specify our high-level family description. Next, we add a few more details on how the ASW system gets its information.

V2 The number and type of Altimeters, devices that measure altitude, on board each aircraft may vary.

- V2.1 Some altimeters provide a numeric measure of the altitude (digital altimeters) where as some altimeters simply indicate whether or not the altitude is above or below a constant threshold that is determined when the altimeter is manufactured (analog altimeters).

Now, we know that there are a number of different sensors on the aircraft that can measure altitude (i.e., the altimeters) and we know that there are two types of fundementally different altimeters: analog and digital. We can add some information on how the ASW handles fusing the data frome these various sensors into one estimate of the altitude:

V3 In family members where there is more than one altimeter, a variety of smoothing and/or thresholding algorithms may be used to determine the estimated value for the true altitude or estimated value of whether or not the aircraft is truely above or below a certain threshold.

V3.1 Methods for choosing numeric altitude from several numeric sources will be mean, median, smallest, largest

V3.2 Methods for choosing whether or not the aircraft is above or below a certain threshold from a variety of altimeters that are either thresholded or numeric are any one above/below, all above/below, and majority above/below.

And, we can add information about how all altimeters on the plane are supposed to function.

C2 All Altimeters will provide an indication of whether or not the supplied altitude is valid or not

C2.1 An altitude that is denoted to be *invalid* shall not be used in a compuation to determine the action to be performed by the ASW

C2.2 If no altitude can be determined (i.e., all altimeters report invalide altitudes) for a specified period of time, then the ASW will declare that the system has failed. This period of time shall be constant for each family member (i.e., determined at specification time).

V4 The period of time that the altitude must be invalid before the ASW will declare a failure may vary from family member to family member.

Finally, there are a few more properties of the ASW family that we need to express in relation to the various indications that the ASW should produce and user controls on the ASW.

C3 All ASW systems will provide a failure indication to the environment.

C3.1 The indication that the ASW has failed will be the fact that the ASW has not strobed a watchdog timer within a specified amount of time. This period of time shall be a constant for each family member (i.e., known at specification time).

C4 The ASW shall except an inhibit signal. While inhibited, the ASW shall not attempt to perform any action other than declaring a failure.

C5 The ASW shall except a reset signal. When the reset signal is recieved, the ASW shall return to its initial state.

For the purposes of the running example for this methodology, we will define a sub-family of the ASW pertaining only the ASW family members that turn on or off and particular Device of Interest on board the aircraft. By making it a sub-family, we keep open the option of reusing all the work that we have done in defining commonalities and variabilties in altitude processing that we have specified thus far. Hopefully, with the family structured in this way the implemention of these features will also be reusable.

We denote the commonalities and variabilities for the DOI subfamily as C_{DOI} and V_{DOI} respectively.

$C_{DOI}1$ The ASW shall change the status (turn on or off) a Device of Interest (DOI) when it crosses a certain threshold

$V_{DOI}1$ The threshold for the ASW varies from 0 to 8024 feet from aircraft to aircraft

$V_{DOI}2$ Whether the ASW turns on/off the DOI when passing above/below the threshold is a variability with nine possible choices:

- do nothing going above or below;
- turn on going below, do nothing going above;
- turn off going below, do nothing going above;
- do nothing going below, turn on going above;
- turn on going below, turn on going above;
- turn off going below, turn on going above;
- do nothing going below, turn off going above;
- turn on going below, turn off going above; or,
- turn off going below, turn off going above;

To deal with noisy data, or the aircraft flying near to the threshold altitude, the DOI controlling ASW needs to have a certain hysteresis factor that is used to deterimine how much the altitude of the plane must change in order to have the DOI powered on or off again. The commonalities and variabilities that govern the hysteresis function of the ASW are given below.

$C_{DOI}2$ The ASW shall employ a hystersis factor to ensure that when the aircraft is flying at approximately the threshold altitude noisy data from the altimeters or slight variations in altitude do not cause the ASW to turn on/off the DOI in rapid succession

$V_{DOI}3$ The hysteresis factor may vary from aircraft to aircraft

$V_{DOI}4$ The hysteresis factor may vary depending whether or not the aircraft is going above or below the threshold.

$C_{DOI}3$ Both the hysteresis factor for going above and the hysteresis factor for going below shall be a constant for each particular aircraft (i.e., known at specification time).

Finally, the ASW will received updates from the DOI whenever that status of the DOI changes. This is important to confirm whether or not the DOI is responding to the commands issued by the ASW as well as fofill the requirement denoted by the final commonality.

$C_{DOI}4$ The DOI shall give the ASW an indication of its status (on or off) whenever that status changes

$C_{DOI}5$ Whenever the ASW submits a command to the DOI, it shall wait for a specified period of time for the status of the DOI to change to reflect the command. If the status does not change within the specified period of time, then the ASW shall declare a failure. The period of time will be a constant for each aircraft

$V_{DOI}5$ The period of time that the ASW will wait after issuing a command to the DOI before indicating a failure if the DOI does not change status shall vary from aircraft to aircraft.

$C_{DOI}6$ The ASW shall not attempt to power on the DOI if the DOI is already on or attempt to power off the DOI if the DOI is already off.

In the next section, we discuss how to view the structure of the ASW family that we have started to define.

7.3.3 Identify Family Structure

Even for a family as small and simple as the ASW, we can identify elements of structure in the family. This identification is useful because it helps us to understand the family and it is invaluable if, in the future, we would like to refactor the family or incorporate the family as a part of a larger family. For example, we might like to have one family that encompasses all the avionics devices built (not just the ASW).

When you are writting a lot of commonalities that start with the word "if" you may consider making a sub-family. For example, we could have written all the DOI commonalities as "If the action to be formed is turning on or off a DOI, then ..." However, this is wordy; and, when the family structure becomes more complex it is very difficult to understand the commonalities. It is often better to define a subfamily when there are conditions on the commonalites.

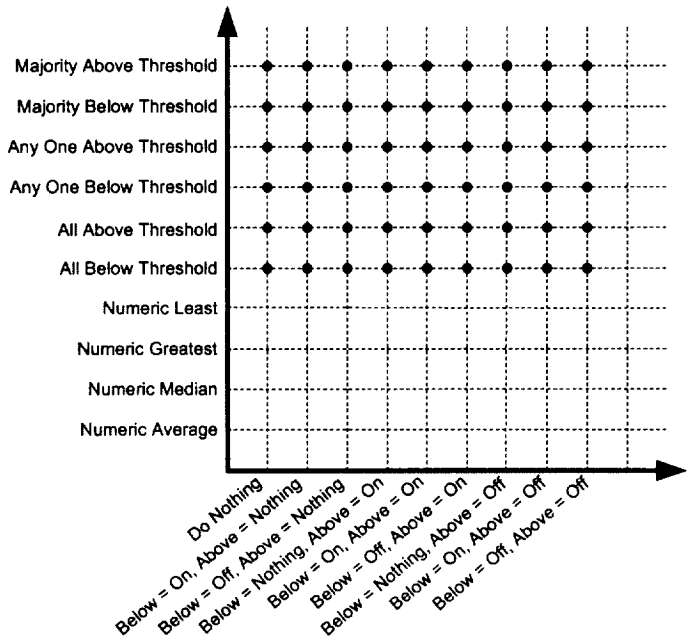

Figure 7.1: The ASW family structure visualized in 2 dimensions

Diminensions of the family, are used as a visualization technique to separate out the major choices of the family. Dividing a family into dimensions does not necessarily mean partitioning *all* the commonalities and variabilities of the family. For the ASW, we identified two possible dimensions: 1) the choice of the altitude smoothing and/or thresholding algorithm and 2) the major choice of functionality for the DOI. This decomposition is show in Figure 7.1.

Figure 7.1 depicts the various possible members of the ASW family, as we have currently defined it in the phase 1 appendix. An interesting property of the figure is that there are no family members currently that use the numeric altitude methods that we dicussed in the commonalities and variabilities. This is because we have only looked at a small sub-family of the possible behaviors of the ASW family. In the future, we can envision adding all sorts of behaviors some of which might use the numeric methods.

The reader might note that the dimension of the family that shows the choice of smoothing or thresholding algorithms has some structure. That is, either the algorithm will have a numeric result and be a smoothing algorithm or it will have a boolean result and be a thresholding algorithm. This structure is visualized in Figure 7.2.

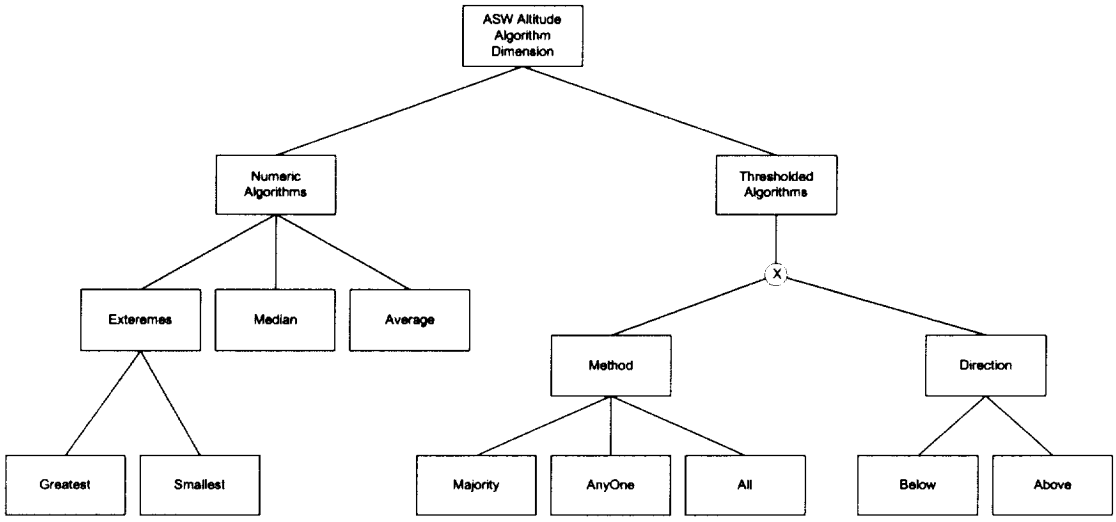

Figure 7.2: The structure of the Altitude Dimension for the ASW

Visualizing the structure of the family in this way can be useful in developing a better understanding of the system. You may find that some commonalities should be made into more general statements and moved to the top-level family. Alternatively, you may discover that certain commonalities and variabilites may be very tied to the current way of doing things and likely to change. These, you may wish to isolate by placing them in a subfamily. In the next section, we dicuss how to refine the family specifications.

7.3.4 Elaborate Variabilities and Commonalities

In the next phase of developing the family description the commonalities and (especially) the variabilities should be refined so that they contain actual quantities (or choices) for the variations. This was done for one of the DOI variabilities above, but values should be filled in for other variabilities as well.

For example, we might like to refine REF VAR 4 so that we specify the tolerances on the failure indication time for the family.

V4 The period of time that the altitude must be invalid before the ASW will declare a failure may vary between 2 seconds and 10 seconds from family member to family member.

7.3.5 Define the Decision Model

The decision model represents a recording of which choices for all the possible variabilities result in current family members. Obviously, the more complex the structure of the family, the more complex the decision model will be.

Building the decision model can often help to identify commonalities or variabilities that may have been forgotten in the initiation draft of the family requirements. This is because engineers, familar with the products, may recall items that must be specified about a particular family members that they did not recall when attempting to generalize to all family members. For example, in our first draft of the ASW commonalities and variabilities, we had forgotten to add a variability to the DOI subfamily for the threshold.

One way that the decision model can be written down is by simply noting which choices are made for each family member. For the ASW family, we have done that below for several ASW family members.

- **CS-123**: This aircraft as one analog and one digital altimeter, turns on the DOI when at least one altimeter is below 2000 feet, will not turn the DOI back on until going 200 ft above the threshold, has a timeout of 4 seconds for altitude staleness and 2 seconds for the DOI.

- **CS-134**: This aircraft as one analog and two digital altimeter, turns on the DOI when at least one altimeter is below 2000 feet, will not turn the DOI back on until going 200 ft above the threshold, has a timeout of 4 seconds for altitude staleness and 2 seconds for the DOI.

- **DD-123**: This aircraft as one analog and one digital altimeter, turns on the DOI when at least one altimeter is below 2000 feet, will not turn the DOI back on until going 250 ft above the threshold, has a timeout of 2 seconds for altitude staleness and 2 seconds for the DOI.

Even so, there are a number of disadvantages to listing the family member configurations in this way. First, it is difficult to tell whether all required variabilities have been given values. Second, it is difficult to easily see family members that have the same choices for the variability values. A tabular format is often used to represent the decision model. A tabular decision model for the ASW family members that we will consider in this methodology is presented in Figure 7.3.

In a family with a more complex structure, a hierarchical series of tables might be used with one table for each sub-family, for example.

Variability	CS-123	CS-134	DD-123	DD-134	EF-155
# of Analog Alt.	1	1	1	1	2
# of Digital Alt.	1	2	1	2	3
Threshold Algo.	Any	Any	Any	Majority	Majority
Invalid Alt. Failure	4 s	2 s	2 s	2 s	2 s
Threshold	2000 ft	2000 ft	2000 ft	2000 ft	1500 ft
Go Above Action	None	None	None	None	Turn Off
Go Below Action	Turn On	Turn On	Turn On	Turn On	Turn On
Go Above Hyst.	200 ft	200 ft	250 ft	200 ft	200 ft
Go Below Hyst.	NA	NA	NA	NA	200 ft
DOI timeout	2 s	2 s	2 s	2 s	2 s

Figure 7.3: A tabular representation of the ASW family decision model

7.4 Evaluation Criteria

It is difficult to tell whether or not a list of commonalities and variabilities is "good" or "bad." Nevertheless, many of the same criteria that apply to requirements can be applied to commonalities and variabilities.

For each commonality, a review should be conducted to determine the following:

- Is the commonality truely common for the subfamily underwhich it is defined? If not, then the family should be refactored.
- Can the commonality be moved to any other, larger subfamily?

Similarly, for each variability a review should be conducted

- Do parameters of variation need to be specified for this variability?
- If parameters of variation are necessary, are all known variation values included?

In addition to evaluation of the commonalities and variabilities, the analyst also needs to evaluate the structure of the family that has been created at this point. The analyst should look at each sub-family and dimension that has been expressed in the commonality analysis and determine the following:

- What does this sub-family or dimension *mean* in the context of my system?
- Does this sub-family or dimension contribute to understanding of the system?

Some sub-families or dimensions that are possible, may not be meaningful and therefore, can be discarded.

Finally, you should have a completely specified decision model for the family.

7.5 Exit Criteria

- Each commonality and variability passes the evaluation criteria.
- The family structure (sub-families and dimensions) meets the criteria outlined above.
- The decision model has been created.

Chapter 8

Phase 2: Identify Environmental Variables

This chapter describes how to take the description of the family developed in the last chapter and create a list of the environmental quantities which are, in some sense, important to the proposed chapter.

In this chapter, we will explore the types of environmental quantities (introduced Chapter 4) used in the system. Fundementally, these are quantities which are either used by the system, i.e., *monitored* quantities, or are affected by the system, i.e., *controlled* quantites. We will also explore the ways in which the environmental quantities can vary, the absense of the proposed system (i.e., without any control).

8.1 Goals

The focus of this chapter is first to identify the monitored and controled quantities and then to describe them in such a way that the resultant list will form an interface between the proposed system and its environment. Furthermore, we desire to develop an understanding of how the physical environment behaves. Therefore, the goals of this phase are

- To develop a complete list of all the environmental quantities needed by the system so that it can know what the required behavior should be (i.e., the monitored quantities),
- To develop a complete list of all the environmental quantities needed by the system so that it can accomplish the control necessary to achieve the requirements (i.e., the controlled quantities),
- To identify the existing relationships between the monitoried and controlled quantities.

8.2 Entrance Criteria

Before starting this phase, you should have

- The completed product family specification developed in the first phase.
- An idea of the devices which will be used in the system (if possible or available).
- Access to domain experts.

8.3 Activities

The activities for this phase of the methodology follow in a straightforward fashion from the goals outlined above: identifying monitored and controlled variables and then defining them and their inter-relationships. Although we have separated out these tasks below, in practice you will most likely iterate between the activities as your understanding of the system developes.

8.3.1 Identifying Controlled Variables

The focus of this activity is to identify the environmental quantities that are under the system's control. Many environmental quantities will be mentioned in the commonalities and variabilities were created in the previous phase. Now, the key is to recognize those quantities and start to write them down.

In general, the question to ask when identifing controlled variables is: what do I want the system to be able to do? A good potential source of information about controlled variables might by an existing system specification. However, care must be taken that *environmental* quantities are captured, rather than values which might be tied to particular actuators in the system.

Controlled quantities can be broken down into several different types. This will help us in identifying them. The types of controlled variables are:

- **Environmental Quantities:** These are values in the environment that you wish to change as a result of the some action of the system. These should not be tied to any particular actuators, but should represent actions that that system is capable of performing.
- **User Displays:** These are values that need to be displayed to the user. These sort of controlled variables often represent indicator lights, gauges, etc. that are present in the physical system. Their purpose is to help the user develop a mental model about the state of the system being controlled; thus, indicates of the state of the controler are also often included.

- **Values for Another Subsystem:** These are values that go to another subsystem. You will see these sorts of controlled variables when you are specifying one piece out of a system or subsystem and there are certain details that must be abstracted away.

In the ASW, the first controlled variable that comes to mind is the state of the DOI state, which can be set by the ASW. As it happens, the DOI is an interesting case, because the state of the DOI is both controlled *and* monitored by the ASW. This is due to the fact that other systems can control DOI. In terms of our catagories of controlled variables, the DOI fits best as an environmental variable. The DOI is something which will exist on the aircraft presumably whether or not the ASW is on board.

Another controlled variable is the failure indication of the ASW. The ASW is required to supply an indication of whether or not it is operating correctly. Therefor, a controlled variable is required to support this indication. In terms of the catagories, the failure indication fits best as a user display, but could also be viewed as a subsystem interface because it may be used by either component on board the aircraft.

In summary, the controlled variables represent those pieces of the interface between the environment on the controller that can be manipulated by the controller to affect the environment. Controlled variables fall into several catagories: environemental quantities, user displays, and values for other subsystems. For the ASW, we discovered two controlled variables: the DOI status which was an environmental quantity and the the failure indication which was more of a user display.

8.3.2 Identifying Monitored Variables

In addition to controlled variables, we must also identify the quantities which the system must monitor. In general, the best approach is to look at the controlled quantities and ask the question: what information do I need to determine what the value this controlled variable? This should lead you to the monitored variables. Another approach, if this is an existing system, is to examine the sensors that are used and ask: what sort of information about the environment is given to me by these sensors?

Monitored quantities, similar to controlled quantities, can be broken down into several different types. This will help us in identifying them. The types of Monitored variables are:

- **Environmental Quantities:** Values or conditions that exist in the environment and are observable that you can use to compute the values of controlled variables.
- **User set-points:** Values that are specified by the user of the system. These values change the way in which you compute the controlled quantities.
- **Abstracted quantities:** values that you expect to receive from another subsystem that you introduce because you want to concentrate on the current subsystem.

- **Quality Indications:** These are variables which indicate the quality or obserability of other monitored variables. These variables are often Boolean, for example, indicating that you can or cannot know the altitude.

Certainly, the most obvious monitored quanitiy in the ASW is that of the Altitude. This is clearly an environmental quantity, because the plan will have some altitude whether or not the ASW is present. In addition, we know that eventually we will have some kind of sensors in the system that actually measure the altitude. Thus, it is possible that there will times when the alitude will not be measurable, for example, if the sensors are failed. Therefor, another monitored variable is the Altitude_Quality variable.

8.3.3 Define the Variables

The monitored and controlled variables represent the interface of the system requirements, the REQ relation, to the environment. It is important to capture the essential information about each variable. Many important quantities to capture about input and output values were noted in [57] and also in [34]. In the FORM$_{PCS}$ we have provided a template for the user to specify the following information:

- Name and purpose. The purpose should include a statement about the physical meaning of the variable as well as the rational for why this is a monitored or controlled quantity.
- The type: boolean, floating point, integer, or enumerated
- The expected minium (if numeric)
- The expected maximum (if numeric)
- The units
- A description of the meaning of each enumeration (if enumerated), or a description of the precision or other physical characteristics required if numeric with physical units.

In addition to the above information, you may also want to note the precision that the requirements are required to maintain about the variables.

If some of this information is not available yet (e.g., the expected minimum and maximum) do not worry. Specify it as UNDEFINED for now and leave that choice for later in the process. Do not make up non-sensical values for these values: it is far better to have UNDEFINED listed than to have a non-sensical value propgated through the requirements process (and even, potentially, into design and implementation).

For controlled variables, you will also want to specify a short description of the conditions under which the variable can take on its various values. For examle, if a particular variable can be either "on" or "off", what conditions cause it to take on these values. This activity will help you in later states as you refine this informal description of each controlled variable into a formal description of the REQ relation (and later the SOFT relation).

As you begin to define the conditions under which each variable takes on its various values, you may find that even with these informal descriptions you can find previously overlooked errors in your conceptions of the requirements. If these sorts of issues arrise, you should list them at the end of the variable definition, until they can be resolved.

Figure 8.1 shows an example of a controlled variable specification from this phase of the ASW development. It shows the definition of the DOI variable as desribed above. Figure 8.2 shows the MON_Altitude variable from this phase.

This activity centers around starting to fill in the details about the monitored and controlled variables which were identified. It is normal for some information to be unknown at this point; however, resist the temptation to simply make up a value in these cases. Instead, try to define unknown quantities as UNDEFINED and list this lack of knowledge as an issue for that particular variable. This ensures that these issues will be resolved at some later point in the effort, and not just be forgotten.

8.3.4 Define Relationships Among Variables

In this activity, you will denote the relationships between the monitored and controlled variables that exist as part of the environment (and in the absense of the proposed system). Thus, by this activity we are encoding the NAT relation.

A system context diagram is helpful in starting to think about the environment of the system. This is a diagram which shows each input and output to the system. The key in capturing the NAT relation is to begin to think about how the rectangular boxes (i.e, the monitored and controlled variable sources) interact with one another in the environment. Figure 8.3 shows the system context diagram for the ASW.

Michael Jackson provides some good guidance on how to identify the actors in the environment that the system interacts with in his books [29, 31]. Jackson builds on the system context diagram to include techniques for identifying and describing the interaction of the system with its environment. That work is complementary to this methodology, and will not be reproduced here.

In addition, some points to consider when specifying the NAT relation are the following:

- Identify how quickly a monitored or controlled variable may change. For example, altitude cannot change from 0 to 10,000 ft in one second.

- Identify relationships between variabls. For example, if variable X has value y then

STATE VARIABLE

CON_DOI_P2

Parent: NONE

Possible Values: On, Off, Uncommanded

Initial Value: UNDEFINED

Classified as: Controlled

Purpose: This variable represents the ASW's commanded status of the Device of Interest (DOI).

Interpretation:

On: Indicates that the DOI is commanded to be On. The DOI is commanded to be on when the aircraft enters the target region for turning the DOI on, the DOI is not already on, and the ASW is not inhibitied.

Off: Indicates that the DOI is commanded to be Off. The DOI is commanded to be off when the aircraft leaves the target region and after a certain period of time has passed. If this time is UNDEFINED, then the ASW will never turn the DOI Off.

Uncommanded: Indicates that the DOI is not commanded by the ASW. This CON_DOI variable will be equal to Uncommanded in any step were the ASW does not issue a command to the device of interest.

Issues:

- If the aircraft leaves the target area and the DOI is on, but was *not* commanded to be on by the ASW, should the ASW turn it off?

Figure 8.1: The CON_DOI variable in Phase 2 of the methodology

INPUT VARIABLE

MON_Altitude_P2

Type: INTEGER

Initial Value: UNDEFINED

Units: ft

Expected Minimum Value: 0

Expected Maximum Value: 50000

Classified as: Monitored

Purpose: This variable represents the ASW's idea of what the altitude of the aircraft is. It is related to the Altitude_Quality variable.

Interpretation:

Precision: We will know the altitude to within ± 10 ft.

Figure 8.2: The MON_Altitute variable in Phase 2 of the methodology

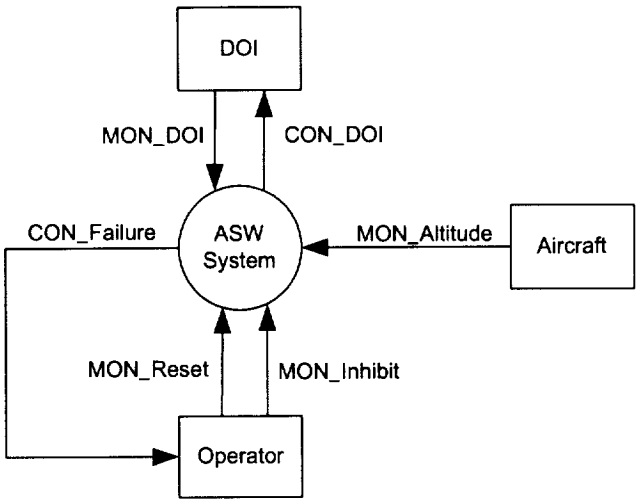

Figure 8.3: The System Context Diagram for the ASW in this Phase

variable Z cannot have value q. These conditions, which should hold over the whole system, should be noted as assertation.

These constraints between the variables will be usefull in the development of the REQ relation in the following phases and also later to provide as inputs to static analysis. The NAT relation might be represented as simply a series of conditions or equations that are always true about the environment; or, that type of specification may be combined with a language similar to the language used to specify the REQ relation. This choice is highly dependent on what level of formality and/or detail is desired in the NAT relation as well as who suitable to the languages involved are to representing NAT. Therefore, it is highly dependent on the particular system involved.

In any event, because the NAT and REQ relations are intimately related, it will probably be necessary to revisit the NAT relation upon further investigation of the REQ relation.

8.4 Evaluation Criteria

For each monitored and controlled variable, the following questions should be answered:

- Does the definition of each monitored and controlled quantity contain a description of its units, rational for including it in the list of environmental quantities, ways in which the environment of the system constrains its values.

- For controlled variables, does there exist monitored quantities which would allow the computation of the correct value of the controlled variable.
- Does the NAT relation given adaquately specify the contraints on the monitored and controlled variables?

8.5 Exit Criteria

At the end of this phase, you have essentially defined the interface for a module that will encapsulate the REQ relation. The only element that we have not talked about explictly is the points at which each family member differs. This phase is complete when

- A list of environmental quantities has been developed
- A specification of the how the environment or the values of other variables accompanies each variable in the list.

Chapter 9

Phase 3: Initial Requirements Structure

In this chapter the commonalities and variabilities developed in the first phase will be combined with the monitored and controlled variabled discovered in the previous phase to form the basis to reason about what the high-level structure of the REQ relation should be.

This high-level structure will represent the module structure in a language that supports module encapsulation. In a language that does not support encapsulation, the specifier will have to manually provide for points in the specification so that pieces can be easily separated.

9.1 Goals

The goal of this chapter is to make establish a high-level structure for the requirements.

- Establish a structure for the requirements, grouping together those entities in the draft specification which "belong" together in some sense.
- Avoid the introduction of design details while structuring the requirements.
- Support the reuse of pieces of the draft specification and, also, support the families and sub-families developed in phase 1, the Commonality Analysis.

9.2 Entrance Criteria

Before entering the structuring phase, you will need the following:

- Family and sub-family relationships from phase 1
- Environmental variables from phase 2

9.3 Activities

The activities in this phase involve getting a broad overview of the computeration that will be preformed as well as defining modules to perform the computation. This phase is tightly coupled with the draft requirements phase of the next chapter and you will most likely find that you shall iterate between these phases.

9.3.1 Define Dependancy Relationships

In this activity, you identify which monitored variables and modes are necessary for the computation of each controlled variable. The goal of this activity is not to produce a detailed graph; although if one does not have a tool-supported language that work may have to be done by hand. Rather, the goal is to formulate a solid idea of the *order* in which entities in the system must be computed so that there are no circular dependencies between the various variables.

The first step is to make a sort list in each controlled variable definition of which other controlled variables, monitored variables, and mode machines it depends upon. Then you can go through and see any controlled variables depend on each other, or if any mode machines depend circularly on controlled variables.

Circular dependencies must be resolved in some way. One way to resolve them is to use the PRE value for one of the variables. That is, instead of using the value of the variable which will be computed during this step, you use the value that the variable had at the start of the computation (which, obviously, can be known without any computation and therefore does not introduce a circular execution dependency). Care must be taken to ensure that one can, in fact, use the PRE values – note that they will always be "one step behind" in some sense. It is *not* desirable to have a specification which takes more than one step to settle into a valid value; each step of the specification must result in a valid and meaningful set of controlled variable values.

Often, it is helpful to view a large specification as a series of functional blocks. The different blocks can then be drilled down into in a functional-decomposition type style. In addition, these functional blocks may be candidates to be made into modules.

9.3.2 Define Modules and Interfaces

In the preceeding section, we identified the rough dependency relationships for the REQ specification. In this activity, we will use the dependency relationships to start to group

pieces of the computation together to form modules.

Parnas [50] defined a critera to be used in decomposing a system into modules called information hiding. Using this philosophy, every module in the system should be choosen so as to encapsulate a decision or several decisions about the system. The interface of such a module exposes only the essential information that the rest of the specification requires. It has been suggested in CoRE [57] that a method for determining which decisions should be grouped together should be whether they are expected to change together.

Another way to view a module is as an addition to the vocabulary that you use to express the requirements. This is the reasoning that lies behind the standard modules used in functional declaration style in the RSML^{-e} language. A module may allow the specifier map to a construct in the physical domain to a single construct in the specification.

One building block that we might like for the ASW is a module that exports the thresholded altitude taking into consideration the hysteresis factor that is required. We will then be able to use such a module to make decisions about whether to turn the ASW on or off. The definition of the interface to this module is shown in Figure 9.1.

After starting work on the draft specification, we realized for the ASW that in order to properly specficy the reset behavior of the system in RSML^{-e}, we had to provide for a top-level ASW mode and an operating module. This is an example of how iteration occurs between these two phases.

9.4 Evaluation Criteria

- Each module should have a purpose. The start of a module block should include a paragraph describing what the purpose of the module is and why it imports and exports certain values.
- Each import and export should have a purpose within the module.

9.5 Exit Criteria

You are done with the Requirements Structure phase when you all the modules in the system pass the above evaluation criteria. The products of the Requirements Structure phase are the following:

- A series of module definitions (see above)
- A diagram of the structure of the specification.
- A specification of how each import of every module is provided by the enclosing scope (i.e., the specific module interconnections).

```
MODULE ThresholdedAltitude_P3 :

  INTERFACE :

    IMPORT Altitude_P3 : Integer
      UNITS : ft
      EXPECTED_MIN : 0
      EXPECTED_MAX : 50000
    END IMPORT

    IMPORT CONSTANT Threshold_P3 : Integer
      UNITS : ft
      EXPECTED_MIN : 0
      EXPECTED_MAX : 8024
    END IMPORT

    IMPORT CONSTANT Hysteresis_P3 : Integer
      UNITS : ft
      EXPECTED_MIN : 50
      EXPECTED_MAX : 500
    END IMPORT

    IMPORT CONSTANT Direction_P3 : UpDownType

      Purpose : &*L This parameter tells the thresholding algorithm
      which direction we are interested in applying the hysteresis
      to.  If the direction is specified as Down, then we will have to
      go above threshold altitude by the hysteresis amount before we
      can declare that we are above (and, thus, be allowed to declare
      below again). L*&

    END IMPORT

    EXPORT AboveOrBelow : AboveBelowType

      Purpose : &*L this export reports whether or not the altitude is
      above or below the threshold given the hysteresis factor L*&

    END EXPORT

  END INTERFACE

  DEFINITION :
  END DEFINTION

END MODULE
```

Figure 9.1: Module Defined to threshold altitude

Chapter 10

Phase 4: Draft Requirements Specification

In this chapter, a prelimary draft of the requirements specification will be constructed. While the examples in this and later chapters are given in RSML^{-e}, the guidance on which aspects of the system to concentrate on and how to evaluate the work should be applicable to a wide variety of languages.

10.1 Goals

The goal of this phase is to capture the *essential* behavior of the system: what was the system meant to do. For the time being, we will put on the back burner questions of fault tolerance, error conditions, etc. These will be delt with in more detail after we have developed a good normal-case understanding of the system.

In the previous chapter, we blocked out the computation of the REQ relatoin. The specific goals of this section, then, are the following:

- Define how each module will be computed from inputs to outputs
- To define how each controlled variable is computed under normal operating conditions.
- To define when each controlled variable must be computed (i.e., on demand or periodic).
- To define how and when each monitored variable will be recorded from the environment.
- To remove unnecessary monitored and controlled quantities.

10.2 Entrance Criteria

Before staring to work on the draft behavior specification, you should have completed identification of the monitored and controlled quantities from the previous section. Thus, you should have the following:

- The initiation specifiation structure from the previous phase.
- A list of the monitored and controlled quantities in the system.
- A specification of the NAT relation, i.e., a specification of the existing relationships between the monitored and controlled quantities.

10.3 Activities

The activities in this section focus on refining the informal specification of how each controlled variable takes on each of its values to a formal specification of this information. The goal is to provide a detailed and formal specification such that given a set of values for the monitored variables, a set of values for the controlled variables can be known.

In practice, you may iterate between this phase and the previous phase as your understanding of the system increases. In addition, you may wish to define modules to use within the computation of other modules. In which case, you would use the module creation guidelines in the previous phase to help decide which modules you needed.

10.3.1 Specify Each Controlled Variable

In this activity, you will specify formally how each controlled variable assumes its various values. This activity is a natural extension of the previous one, because it involves not only thinking about what values are necessary to compute the controlled variables, but exactly how those variables contribute to the controlled values. Furthermore, you may also iterate between this activity and identifying potential modes because you may discover the need to keep some system state or system mode information as you try to determine what values the controlled variables should be.

There are two main styles for defining a state variable in a system: the transitional style and the equivalence style. These two styles are explained in the following paragraphs, with examples.

Equivalence-Style Specifications: Equivalence-style specification of a state variable is, perhaps, the most straightforward. In this style, the specifier states explicitly in a series of cases what value the state variable assumes. The value of the variable is, thus, always

defined unless explicitly noted otherwise by the specifier or unless it is a child underneath another state variable.

For any computation of the specification, it is expected that one and only one case of the variable will be true; the state variable then assumes the value specified by the one unique case. If the state variable does not have a case which evalutes to true in some step, then we say that the variable definition is *incomplete* because for the particular sequence of inputs events leading up to this step the variable does not have a defined value. If the state variable has more than one case which is true then we say that the specification is *inconsistent*; how can we know which case is the one that was intented by the specifier?

Variables specified in this way are similar to condition tables in SCR.

An example from the ASW of a equivalence-style specification is the CON_Failure variable (below, from phase 4). In this case, we want to declare a failure of the ASW as soon as one of our designated failure conditions exists and stays in the failure mode until a reset occurs.

```
EXPORT CON_Failure_P4 :
  PARENT : NONE
  DEFAULT_VALUE : False

  EQUALS TRUE IF
    TABLE
      DURATION(AttemptingOn(), 0 S, Clock) > DOI_Timeout_P4     : T * * * ;
      DURATION(AttemptingOff(), 0 S, Clock) > DOI_Timeout_P4    : * T * * ;
      DURATION(MON_Altitude_Quality_P4 = Invalid, 0 S, Clock)   : * * T * ;
      PRE(CON_Failure_P4) = False                               : * * * T ;
    END TABLE

  EQUALS FALSE IF
    TABLE
      DURATION(AttemptingOn(), 0 S, Clock) > DOI_Timeout_P4     : F ;
      DURATION(AttemptingOff(), 0 S, Clock) > DOI_Timeout_P4    : F ;
      DURATION(MON_Altitude_Quality_P4 = Invalid, 0 S, Clock)   : F ;
      PRE(CON_Failure_P4) = False                               : F ;
    END TABLE

END EXPORT
```

Transitional-Style Specifications: Sometimes, we are not so interested in what values a variable should have in each step but, rather, it is desireable to specify when the variable should change values. A transitional-style specification consists of a series of transitions, each with a source state, a destination state, and a condition. When the condition is true and the variable has the value specified by the source state, then the variable will become the value specified by the destination state.

Some languages include the notion of a triggering event for transitions (RSML^{-e} does not). In these languages, a transition is taken with the triggering event occurs (and possibly when the gaurding condition is true in addition to the trigger happening). However, much semantic information can be embedded in such events and we find that it is preferrable to state explicitly the conditions under which an event occurs. Therefore, in RSML^{-e} (and similar languages) an event is simply that a set of conditions are true in this step which were not true in the previous step (or vice-versa).

Transitional-style specifications can share the same notion of consistency as equivalence-type specifications. Nevertheless, for a transitional-style specification, it is usually expected that the variable will retain its current value in the absence of any need to change. Therefore, transitional-style specifications cannot make use of the notion of completeness because it is expected that there will be some steps (probably many steps) in which the none of the transitions evaluate to true.

Although the topic can be debated, the notion of completeness can be extended to transitional-style specifications if we require the specifier to include transitions in the specification from each state back to itself which will be taken in steps were we do not wish the variable to change value.

An example from the ASW of a transition style specification is the CON_DOI variable (below, from phase 4).

```
EXPORT CON_DOI_P4 :
  PARENT : NONE
  DEFAULT_VALUE : Uncommanded

  TRANSITION Uncommanded TO On IF
    TABLE
       DOI_Action_Ok(On)                                  : T T ;
       WHEN(ThresholdedAlt_P4.Result_P4 = Below, False)   : T * ;
       GoBelowAction = TurnOn                             : T * ;
       WHEN(ThresholdedAlt_P4.Result_P4 = Above, False)   : * T ;
       GoAboveAction = TurnOn                             : * T ;
    END TABLE

  TRANSITION Uncommanded TO Off IF
    TABLE
       DOI_Action_Ok(Off)                                 : T T ;
       WHEN(ThresholdedAlt_P4.Result_P4 = Below, False)   : T * ;
       GoBelowAction = TurnOff                            : T * ;
       WHEN(ThresholdedAlt_P4.Result_P4 = Above, False)   : * T ;
       GoAboveAction = TurnOff                            : * T ;
    END TABLE

  TRANSITION On TO Uncommanded IF WHEN(MON_DOI_P4 = On, False)
```

```
    TRANSITION Off TO Uncommanded IF WHEN(MON_DOI_P4 = Off, False)

END EXPORT
```

10.3.2 Identify Potential Modes

In this activity, you will examine the informal descriptions of how each controlled variable takes on its values and begin to identify potential modes of the system. The first step in this activity is to make a list of what information is necessary to compute each controlled variable. Some of this information will be monitored variables, or previous values of monitored variables. Other information that may be needed might include the results of previous computations on the monitored variables.

In general, modes of the system are points at which changes of the values of the monitored variables causes changes of the values of controlled variables. For example, a controlled variable might depend on a specific series of user inputs are events before it can take on a particular value; thus, we will require a mode machine of some kind which will record for us *where* in the sequence of actions we are and what input we expect to occur next.

A concrete example is that of a weapons firing interlock. It is usually true a number of conditions must become true before pressing the 'fire' button will cause the weapon to fire, for example, perhaps that airplane must be traveling at a particular speed, or at least a certain altitude. Furthermore, it is usually *not* desirable to have the press of the firing button precede these events: what if the firing button is stuck down and we cross a threshold altitude which makes the preconditions true? We probably do not want to fire in that case. To model this type of behavior, we must store internal state information so that we can track where in the squence we are.

Modes partition of the functionality of the system. When a mode variable has one value, the system behaves in one way and when the mode has a different value, the system behaves in another way. The above example of a squence of values is not the only time when this occurs. For example, in the ASW, a mode of the system is whether or not the ASW is inhibited. If the ASW is inhibitied, then it will not turn on the DOI regardless of crossing a threshold altitude; if it is not inhibited, then it will attempted to turn on the DOI when passing below the threshold.

Modes may represent some alternate or reduced functionalty operation of the system. For example, many systems have a startup or shutdown mode in addition to the normal operation mode. Another example is when a system has some reduced functionality modes; for example, when the values of some environmental quantities are not available, the system may only be allowed to perform a subset of the available actions.

Finally, modes may be introduced to represent to the enivironment or controller what the system is doing. For example, in an aircraft, the various systems can be on autopilot,

or in landing or take-off modes. If the system being built is responsible for implementing one or more of these modes, then it will be useful to represent them explicitly in the requirements because they are the language in which the customer will be most able to communicate. In addition, it will be a common desire to state properties about these modes, for example, " the system will not lower the landing gear while in take-off mode."

There are a number of examples of variables which might be considered modes in the ASW specification. The first that comes to mind is the ASW_System_Mode variable (below, from phase 4). This variable controlls the overall functioning of the ASW. Although, in this case, there is only the reset mode (and that mode has no functionality), this same structure could be used to represent a startup and shutdown mode, or it could be used to represent different modes of reduced functionality simply by added values to the ASW_System_Mode variable and then defining appropriate behavior for those modes. Using the module construct (or cut and paste) it is possible to allow modes to share functionality while still differing significantly in some areas.

```
STATE_VARIABLE ASW_System_Mode_P4 :
  VALUES : {Reset, Operating}
  PARENT : NONE

  Purpose : &*L This is the top-level mode of the ASW.  If the ASW
  were to have a startup mode, etc., we could put those modes as
  children of this controlling mode.  Currently, we have only two
  states, the reset mode which is used for when the reset signal
  is received and the operating mode that handles the main
  behavior. L*&

  DEFAULT_VALUE : Operating

  TRANSITION Operating TO Reset IF
    WHEN(MON_Reset_P4, False)

  TRANSITION Reset TO Operating IF
    DURATION(PRE(ASW_System_Mode_P4), 0 s, Clock) >= 0 S

END STATE_VARIABLE
```

Another example of variable that functions as a mode is the ApplyHisteresis variable that is defined inside of the ThresholdedAltitude_P4 module. This mode determines whether or not the system should apply the Histeresis factor when determining whether the aircraft is above or below the threshold.

```
STATE_VARIABLE ApplyHisteresis_P4 :
  VALUES : {NoHyst, Above, Below}
  PARENT : NONE
```

```
  DEFAULT_VALUE : NoHyst

  TRANSITION NoHyst TO Above IF
    TABLE
      DEFINED(Altitude_P4)                    : T ;
      WHEN(Altitude_P4 < Threshold_P4, False) : T ;
    END TABLE

  TRANSITION NoHyst TO Below IF
    TABLE
      DEFINED(Altitude_P4)                    : T ;
      WHEN(Altitude_P4 > Threshold_P4, False) : T ;
    END TABLE

  TRANSITION Above TO NoHyst IF
    TABLE
      DEFINED(Altitude_P4)                                          : T T ;
      WHEN(Altitude_P4 < Threshold_P4 + AboveHysteresis_P4, False) : T * ;
      WHEN(Altitude_P4 > Threshold_P4 - BelowHysteresis_P4, False) : * T ;
    END TABLE

  TRANSITION Below TO NoHyst IF
    TABLE
      DEFINED(Altitude_P4)                                          : T T ;
      WHEN(Altitude_P4 > Threshold_P4 + AboveHysteresis_P4, False) : T * ;
      WHEN(Altitude_P4 < Threshold_P4 - BelowHysteresis_P4, False) : * T ;
    END TABLE

END STATE_VARIABLE
```

10.3.3 Using Tools to Visualize the Preliminary Behavoral Specification

This section describes how using a formal language that is supported by tools can help in visualizing the requirements at this stage. Many forma languages are supported by tools, including RSML^{-e}, which is supported by the NIMBUS tools. The examples given in this section are from the NIMBUS toolset; however, almost any reasonable good formal language tools will provide this information.

Viewing the System Dependency Graph One of the artifacts of this stage in the processs is the system dependency graph. For a simple specification, it may be easy enough to generate this by hand. For a more complex specification doing it by hand will be much more complex.

In practice, it is easier to think about stages or blocks of the computation for a complex specification. Nevertheless, a detailed visualization of the actual dependency group that is generated by tools can be very useful.

Simulating the Draft Specification Simulating the draft specification allows the analyst, customers, and others involved (managers, regulatory agencies, researchers, etc.) to see what the specified behavior of the system is and make corrections early in the process. Thus, simulation is an invaluable tool for validating the specification and is especially useful if it can be done early and continuously throughout the effort.

Many languages support tools which allow the user to input data into the input variables and see what values the outputs take on. More advanced toolsets, like NIMBUS allow the user to connect the draft requirements to more advanced models and simulations of the environment. For example, in a avionics context, the draft specification could be connected to a cockpit simulator and tested with actual pilots. In a medical devices context, the draft requirements might be connected to an accurrate simulation of the body or be run through actual patient data. By accurately simulating the environment and input sequences to the draft specification, you can achieve a much higher quality product than just making up the inputs yourself (from your own mental model of the environment).

10.4 Evaluation Criteria

To evaluate the entities produced in this phase, you should ask the following questions:

- Does each variable used in the system have a complete definition?
- Are there any monitored variables which are not used to compute the value of any controlled quantity? If so, can they be eliminated?
- Are there any controlled quantities which are never produced as a result a of a change in the monitored quantities? If so, can they be eliminated?
- Are there any cycles in the system dependency graph? If so, they should be eliminated.
- There should be no imports which are not used in the computation of the exports. In addition, there should be no exports which are never generated.
- All imports for all modules have been given a value in the enclosing scope.

10.5 Exit Criteria

You have completed a good draft specification when you have the following:

- Complete definitions of all monitored, controlled, and state variables
- View of the variable dependency graph

Chapter 11

Phase 5: Detailed Requirements Specification

In this chapter, the draft requirements specification is updated to include information about fault tolerance and error conditions which was intentially left out of earlier drafts. By the end of this phase, a complete specification of the REQ relation, the system requirements, will be produced.

11.1 Goals

The goal of this chapter is to complete the specification of REQ so that it includes all necessary information. In addition, we will prepare the REQ relation for the realities of sensor failures, etc. by adding in failure modes.

- Specify startup and shutdown behavor
- Specify the tolerances of each controlled and monitored variable
- Identify possible error conditions and specify the error handling behavior
- Specify degraded modes of functionality in response to tolerance violations or error conditions

11.2 Entrance Criteria

Before starting this phase, you should have the following:

- The structured draft of REQ from the previous phases
- A rough idea of what types of sensors and actuators might be used in the system.

11.3 Activities

The activities of this section help you to focus on the special cases of the specification. These are areas of the systems operation which are do not represent the normal operating modes of the system, but rather the boundary cases and error conditions that the system must handle.

Given that the activities in this section are very important, the reader may question why it is that we are just getting around to them now. We do them at this stage in the methodology because it is easier to deal with these cases after one developes a thourough understanding of the system. We do not focus on them at the begining because it is easy to become bogged down in special cases before developing an understanding about the essence of what it is the system is supposed to do.

11.3.1 Specify Initialization and Shutdown Activities

Most controllers have (or should have!) a different operational profile immediately after they are turned on and just before they are about to turn off. The reason for this is that the environment in which the controller operates is a system of its own right (and is described by the NAT relation); it exists with or without the presence or operation of the controller. Certainly, there are two different systems: one with the controller turned on and one with the controller turned off. And, these systems behave differently from one another.

The startup and shutdown modes of a system are designed to handle the fact that is necessary to transition from the system were the controller is off to the system where it is on and vice-versa. In particular, for the startup mode, it is necessary to ensure that the model of the environment within the controller matches the real environment and for the shutdown mode it is necessary to ensure that once the controller goes offline the system will be in a safe state (and will remain in a safe state in the absense of the controller's actions).

Consider the accident in which a chemical plan explosion was caused by a system which was designed to use a metering pump to put a certain amount of catalyst into a reaction. The control system had begun this operation and then was taken offline. While the system was offline, the pump continued to run. However, when the control system was turned on again, it started counting from where it had left off, not taking into consideration the amount which had been pumped while it was offline. This is an example of improper startup and shutdown behavior for the system.

The ASW's startup mode is very simple: it just has to receive five seconds of valid altitude in order to transfer to normal operation. Thus, it can be represented with only a single transition and does not need other behavior. In other systems, the controller may need to wait until it develops a certain confidence in the estimates of the monitored quantities before it issues any commands to the environment.

11.3.2 Specify Error Handling

The first thing to do in specifying the error handling behavior of the specification is to create a list of potential error conditions in the specification. Note that all of the possible error conditions may not be known at this time; some error conditions may only come to light when we add information about the sensors and actuators. Nevertheless, we will know about many possible error conditions during our development of the REQ relation and we should attempt to handle those error conditions in the best way possible.

It is often useful to have a global failure mode that encapsulates the failure behavior for the system. The ASW's failure mode is given in the example below.

```
EXPORT CON_Failure_P5 :
  PARENT :  NONE
  DEFAULT_VALUE : False

  TRANSITION False TO True IF
    TABLE
      ASW_System_Mode_P5 = NormalOperating   : T * ;
      ASW_Operating_Mode_P5.CON_Failure_P5   : T * ;
      ASW_System_Mode_P5 = Degraded          : * T ;
      ASW_Operating_Mode_P5.CON_Failure_P5   : * T ;
    END TABLE

  TRANSITION True TO False IF ASW_System_Mode_P5 = Reset

END EXPORT
```

The ASW is a faily simple example. In more complex systems, it is useful to have each module below the main module also export a failure indication that covers failures local to that module. Then, the global failure mode checks each of these local failure indications and, if they are true, may decide declare a failure or to enter some reduced functionality mode as is discussed in the next section.

11.3.3 Degraded Modes of Functionality

Often, we wish to have a system which has some behavior under ideal conditions, i.e., good knowledge about the environment, but which will continue to function in a safe manner event if conditions are not ideal (for example, with broken sensors or actuators). If we know that the desired controller has these properties, then we can plan ahead and establish several different modes of functionality ranging from fully operational where all information is known to an acceptable confidence to a shutting down mode where the system will turn itself off and leave the process in a safe state.

This sort of system is difficult to construct because, in a sense, many different systems are being specified - one for each degrated functionality mode. However, it may be that

the system behavior is more or less the same in these various modes. In that case, the modes may be able to be treated as a family.

The various modes of the ASW and how the ASW switches between them are shown below. We have simply added additional states to the undeveloped ASW_System_Mode from the previous phase. We have added an overall failure mode to deal with system failures and also a value for the started and degraded functionality modes.

```
STATE_VARIABLE ASW_System_Mode_P5 :
  VALUES : {Startup, NormalOperating, Degraded, Failed, Reset}
  PARENT : NONE

  Purpose : &*L This is the top-level mode of the ASW.  If the ASW
  were to have a startup mode, etc., we could put those modes as
  children of this controlling mode.  Currently, we have only two
  states, the reset mode which is used for when the reset signal
  is received and the operating mode that handles the main
  behavior. L*&

  DEFAULT_VALUE : Startup

  TRANSITION NormalOperating TO Reset IF MON_Reset_P5

  TRANSITION Degraded TO Reset IF MON_Reset_P5

  TRANSITION NormalOperating TO Degraded IF
    EpisodeMonitor_P5 = QualifyingEpisode

  TRANSITION Degraded TO NormalOperating IF
    DURATION (MON_Altitude_Quality_P5 = Valid, 0 S, Clock) > 1 M

  TRANSITION Reset TO NormalOperating IF
    DURATION(PRE(ASW_System_Mode_P5), 0 s, Clock) >= 0 S

END STATE_VARIABLE
```

In order to enter the degraded functionality mode, we must know whether two episodes of invalid altitude lasting at least one second have occured within one minute of each other. This requires state information, so we have introduced the EpisodeMonitor_P5 variable to track the occurence of episodes and inform the ASW_System_Mode variable when a qualifying episode as occured and it is necessary to enter degraded functionality mode.

```
STATE_VARIABLE EpisodeMonitor_P5 :
  VALUES : {NoEpisode, FirstEpisode, QualifyingEpisode}
  PARENT : NONE
```

```
Purpose : &*L This simple state variable tracks whether or not
we have met the conditions for being in degraded functionality
mode.  Namely, whether or not we have seen two periods of
invalid altitude lasting 1 second or more within 1 minute. L*&

 DEFAULT_VALUE : NoEpisode

 TRANSITION NoEpisode TO FirstEpisode IF
   DURATION(MON_Altitude_Quality_P5 = Invalid, 0 S, Clock) > 1 S

 TRANSITION FirstEpisode TO QualifyingEpisode IF
   TABLE
     DURATION(MON_Altitude_Quality_P5 = Invalid, 0 S, Clock) > 1 S : T ;
     DURATION(PRE(EpisodeMonitor_P5) = FirstEpisode) > 1 S         : T ;
   END TRANSITION

 TRANSITION FirstEpisode TO NoEpisode IF
   DURATION(PRE(EpisodeMonitor_P5) = FirstEpisode) >= 1 M

 TRANSITION QualifyingEpisode TO NoEpisode IF
   DURATION(MON_Altitude_Quality_P5 = Valid, 0 S, Clock) >= 2 M

END STATE_VARIABLE
```

11.3.4 Specify Tolerances and Handle Violations

In the ideal world of the REQ specification, we know the value of each controlled variable with exact precision. Nevertheless, we know that eventually we will build a physical implementation of the system and that in that implementation we cannot know the values for certain or to an infinite accuracy.

In many cases, the tolerance of a controlled variable is constant throughout the entire specification. In that case, the tolerance may be specified in much the same way as the precision was specified for monitored variables.

In other cases, the tolerance of a controlled variable may be a function of one or more modes of the system. For example, some cases when tolerance may be a function include

- When particular variables in the specification increase in value, for example, the altitude of an aircraft may be required to be controlled to a much greater tolerance when the aircraft is near to the ground than when the aircraft is at a high altitude;
- When the system has several degrated modes of functionality, the controlled variables may be specified to a wider tolerance in a mode of decreased functionality; and,
- When the system has a high load the controlled variables may have a wider tolerance, for example, in the case of a tracking system if the system is tracking 30 aircraft it

may track each one to a certain tolerance; however, if the system had only 5 aircraft to track it may be able to track each one to a greater level of accuracy.

11.4 Evaluation Criteria

The specification of REQ is complete when the following are true:

- All errors for the system should have a specified behavior or an explaination of why the system does not need to handle that error differently from the normal case.
- All variables specified in the system should be complete and consistent

11.5 Exit Criteria

You are done when the specification of REQ meets the criteria expressed above.

Chapter 12

Phase 6: Including Sensors and Actuators

This chapter describes how to take a specification of the REQ relation and refine that specification into a specification of the SOFT relation.

12.1 Goals

The goal of chapter is to produce the finished specification of the SOFT relation. In a sense, the activities done in this phase are a microcosm of the activities done in earlier phases as you first describe the IN and OUT relations at a high level and then begin to specify the IN^{-1} and OUT^{-1} relations.

12.2 Entrance Criteria

- The complete specification of REQ
- A list of all the sensors and actuators which will be or could be used in the system.
- A description of the properties of each sensor and actuator.

12.3 Activities

The activities of this phase are essentially just a replication of the previous phases, except for the IN^{-1} and OUT^{-1} relations as opposed to the REQ relation. Thus, we will begin by identifying the sensors and actuators in the system from the commonalities and variabilities in phase one. Then we will identify the input and output variables as we did in phase two.

We will then move on to the overall structure of the IN^{-1} and OUT^{-1} relations just as in phase three, and construct a draft of the relations as in phase four. Finally we will add all the necessary error handling and polishing as in phase five. At that point we will have completely specified the SOFT relation.

12.3.1 Identify and Describe the Sensors and Actuators

The first step in adding the IN^{-1} and OUT^{-1} relations is to identify and describe the sensors and actuators involved in the system. After that, you identify the input and output variables for the software. This activity is analogous to phase two for the REQ relation.

For the ASW, each aircraft as a number of altimeters that measure the altitude, a status indication from the DOI, a reset signal, and an inhibit signal. All inputs except for the altimeters pretty much map directly to the existing monitored variables. Therefore, on the input side we will concentrate in refining the IN^{-1} relation for the Altitude monitored quantity.

The commonality anaylsis from phase one tells us that we will have a varying number and type of altimeters for each aircraft that we wish to build. Furthermore, we know that the different types of altimeters yield different information: analog altimeters give only above or below whereas digital altimeters yield a numeric altitude.

On the output side, we have the DOI command indication and the failure output. Only the failure output needs significant changes to specify the output relation.

For the failure indication, the ASW must produce a pulse on a watchdog timer at least every 200 MS or else the other devices on board the aircraft will believe that the ASW has failed. This is the opposite from the way that the REQ relation works, where we only produce an indication if there *was* a failure. Thus, we need a small state machine that will produce a pulse if there is not a failure.

12.3.2 Outline the IN^{-1} and OUT^{-1} Relations

The first step in specify inteh IN^{-1} and OUT^{-1} relations is to outline the computation, just like we did for REQ in phase. For the ASW, the IN^{-1} relation is the most interesting, so we will focus on that one.

Each aircraft differs in the number and type of altimeters and in the algorithm used to determine whether the aircraft is above or below the threshold from the various altimeters. The first thing to notice is that the specification of REQ from phase five expects a numeric altitude input. For compatability, we will change the input to REQ to be a thresholded value and move the thresholding of the digital altimeters into the IN^{-1} relation.

Thus, the overall structure of the IN^{-1} relatin for Altitude is given by the following module definition:

```
MODULE Altimeters_IN_P6 :

  INTERFACE :

    IMPORT CONSTANT NumDigitalAlt_P6 : INTEGER
      UNITS : NA
      EXPECTED_MIN : 0
      EXPECTED_MAX : 10
    END IMPORT

    IMPORT CONSTANT NumAnalogAlt_P6 : INTEGER
      UNITS : NA
      EXPECTED_MIN : 0
      EXPECTED_MAX : 10
    END IMPORT

    IMPORT DigialAlt_P6 : [1 TO NumDigitalAlt] OF INTEGER
      UNITS : ft
      EXPECTED_MIN : 0
      EXPECTED_MAX : 50000
    END IMPORT

    IMPORT CONSTANT Threshold_P6 : INTEGER
    END IMPORT

    IMPORT CONSTANT GoAboveHyst_P6 : INTEGER
      UNITS : ft
      EXPECTED_MIN : 50
      EXPECTED_MAX : 500

      Purpose : &*L This defines the hysteresis factor for going above
      the threshold altitude. L*&

    END IMPORT

    IMPORT CONSTANT GoBelowHyst_P6 : INTEGER
      UNITS : ft
      EXPECTED_MIN : 50
      EXPECTED_MAX : 500

      Purpose : &*L This defines the hysteresis factor for going above
      the threshold altitude. L*&

    END IMPORT

    IMPORT AnalogAlt_P6 : [1 TO NumAnalogAlt] OF AboveBelowType
    END IMPORT
```

```
    IMPORT DigitalQuality_P6 : [1 TO NumDigitalAlt] OF AltitudeQualityType
    END IMPORT

    IMPORT AnalogQuality_P6 : [1 TO NumAnalogAlt] OF AltitudeQualityType
    END IMPORT

    IMPORT INTERFACE AltitudeVoter_P6 :
    END IMPORT

    EXPORT Altitude_P6 : AboveBelowType
    END EXPORT

    EXPORT AltitudeQuality_P6 : AltitudeQualityType
    END EXPORT

  END INTERFACE

  DEFINITION :
  END DEFINITION

END MODULE
```

The interface AltitudeVoter will be used by all the various implementations of the altitude voting algorithm. The specification for each aircraft will decide how many altimeters and which algorithm to use.

12.3.3 Specify the Normal-Case

For the next activity, we need to fill in the actual behavior of the IN^{-1} and OUT^{-1} modules that we have declared. The Definition part of the Altitude module is shown below (for brevity, we will not duplicate the interface specificatoin).

```
  DEFINITION :

    MODULE_INSTANCE ThresholdedDigital_P6 : [1 TO NumDigitalAlt] OF ThresholdedAltitude_P6
      PARENT : NONE
      ASSIGNMENT
        Altitude_P6        := DigitalAlt_P6,
        Threshold_P6       := EXTEND Threshold_P6 TO [1 TO NumDigitalAlt] OF INTEGER,
        AboveHysteresis_P6 := EXTEND GoAboveHyst_P6 TO [1 TO NumDigitalAlt] OF INTEGER,
        BelowHysteresis_P6 := EXTEND GoBelowHyst_P6 TO [1 TO NumDigitalAlt] OF INTEGER
      END ASSIGNMENT
    END MODULE_INSTANCE

    SLOT_INSTANCE AltitudeVoter_P6 :
      ASSIGNMENT
```

```
      Num_of_Alt    := NumDigitalAlt_P6 + NumAnalogAlt_P6,
      Altitudes     := ThresholdedDigital_P6.Result_P6 | AnalogAlt_P6,
      Qualities     := DigitialQuality_P6 | AnalogQuality_P6
    END ASSIGNMENT
  END SLOT_INSTANCE

  EXPORT Altitude_P6 :
    PARENT : NONE
    DEFAULT_VALUE : AltitudeVoter_P6.Altitude_P6
    EQUALS AltitudeVoter_P6.Altitude_P6
  END EXPORT

  EXPORT AltitudeQuality_P6 :
    PARENT : NONE
    DEFAULT_VALUE : AltitudeVoter_P6.AltitudeQuality_P6
    EQUALS AltitudeVoter_P6.AltitudeQuality_P6
  END EXPORT

END DEFINITION
```

We also need to specify the various altitude voting algorithms. These can be found in Appendix F. We will not duplicate them here.

At this point, it is possible to simulate the entire SOFT relation by wiring the IN^{-1} OUT^{-1} and REQ relations together.

12.3.4 Specify Detailed SOFT Relation

With the preliminary version of the IN^{-1} and OUT^{-1} relations completed, it is possible to move on and consider the startup, shutdown, and degraded functionality modes of the IN^{-1} and OUT^{-1} relations. For the ASW, there is not much here. But, you would construct the detalied version of these relations in the same way as for phase five of the REQ relation.

All the analyses that were done on the REQ relation are also applicable to the IN^{-1} and OUT^{-1} relations. They should be consistent and (ideally) complete just as the REQ relation was refined to be. In addition, analysis to deterine the timing properties of the SOFT relation, and the deviation of the output under noisy data should be performed.

At the end of this activity, you should have a complete specification of the SOFT relation.

12.4 Evaluation Criteria

- Are assumptions (tolerance, frequency, etc.) placed on the input variables compatible with the assumptions formed for the monitored variables in the previous phase?

- Has each additional error condition that was recognized been supplied with a specified behavior?
- Have all known error conditions of the sensors and actuators been accounted for?

12.5 Exit Criteria

The specification is complete when you can answer "yes" to all of the above questions.

Appendix A

The Altitude Switch in RSML^{-e}-Phase 1

This chapter describes the running example for the FORM$_{PCS}$ methodology, the Altitude Switch (ASW) family. One chapter in the appendix is devoted to the complete ASW family specification as it exists at the end of each phase in the idealized FORM$_{PCS}$ process. Of course, in constructing these idealized versions, we may have iterated between several phases. It would be confusing to attempt to provide all of that information here. Nevertheless, where this iteration took place and was significant in nature, we have attempted to explain it in these examples.

A.1 Commonalities and Variabilities for the ASW

The ASW family consists of systems on board the aircraft that utilize the values from the various altimeters on board to make a choice among various options for actions (one of which being to do nothing) and perform the choosen action. Therefore, some high-level commonalities and variabilities are the following:

C1 All ASW systems will have a way to measure the altitude of the aircraft

- C1.1 The ASW system will use the information about the aircraft's altitude to make a decision as to what action the ASW should perform

V1 The actions that the ASW takes in response to the altitude and the criteria to perform those actions varies from aircraft to aircraft

At this point, the ASW is essentially a family of systems that process the altitude and then can perform some action based on the altitude that is measured. Of course, the ASW

exists on board and aircraft of some kind and that aircraft will have a specified number and type of altimeters. This is noted in the following two variabilities.

V2 The number and type of Altimeters, devices that measure altitude, on board each aircraft may vary.

V2.1 Some altimeters provide a numeric measure of the altitude (digital altimeters) where as some altimeters simply indicate whether or not the altitude is above or below a constant threshold which is determined when the altimeter is manufactured (analog altimeters).

Different manufacters and/or different situations may dictate using different algorithms to process and threshold the altitude. This is noted in the following variabilities.

V3 In family members where there is more than one altimeter, a variety of smoothing and/or thresholding algorithms may be used to determine the estimated value for the true altitude or estimated value of whether or not the aircraft is truely above or below a certain threshold.

V3.1 Methods for choosing numeric altitude from several numeric sources will be mean, median, smallest, largest

V3.2 Methods for choosing whether or not the aircraft is above or below a certain threshold from a variety of altimeters which are either thresholded or numeric are any one above/below, all above/below, and majority above/below.

All the altimeters that are used on board the aircraft are required to provide a measure of the validity of the measure. Futhermore, if the ASW cannot get a valid (or high enough precision) estimation of the altitude, it should declare that the system has failed. Therefore, we would like to record that fact as a commonality for the ASW family.

C2 All Altimeters will provide an indication of whether or not the supplied altitude is valid or not

C2.1 An altitude which is denoted to be *invalid* shall not be used in a compuation to determine the action to be performed by the ASW

C2.2 If no altitude can be determined (i.e., all altimeters report invalide altitudes) for a specified period of time, then the ASW will declare that the system has failed. This period of time shall be constant for each family member (i.e., determined at specification time).

V4 The period of time that the altitude must be invalid before the ASW will declare a failure may vary between 2 seconds and 10 seconds from family member to family member.

In order for other devices on board the aircraft to know that the ASW has failed, the ASW must provide some kind of failure indication. Usually, this is done by having the system in question cease to strobe a watchdog output. If the watchdog is not present, then other devices on board the aircraft know that that piece of the system is no long functioning for some reason.

C3 All ASW systems will provide a failure indication to the environment.

C3.1 The indication that the ASW has failed will be the fact that the ASW has not strobed a watchdog timer within a specified amount of time. This period of time shall be a constant for each family member (i.e., known at specification time).

V5 The time interval with which the ASW must strobe the watchdog timer varies from aircraft to aircraft.

The ASW also accepts an inhibit and a reset signal. The inhibit signal should prevent the ASW from performing any action other than declaring a failure. The reset signal should return the ASW to its initial state.

C4 The ASW shall except an inhibit signal. While inhibited, the ASW shall not attempt to perform any action other than declaring a failure.

C5 The ASW shall except a reset signal. When the reset signal is recieved, the ASW shall return to its initial state.

Finally, the ASW has several operating modes in addition to the normal one described above. The ASW should wait until receiving at least 5 seconds of valid altitude before performing any action.

C6 The ASW shall receive at least 5 seconds of valid altitude upon startup before entering normal operation.

In addition, the ASW has a reduced functionality mode that is activated when two episodes of invalid altitude lasting at least one second occur within a minute of each other. In the reduced functionality mode, if the ASW detects that an action should be performed, it shall wait for a minumum of 2 seconds before checking the conditions for action again. If, after that minumum delay, the conditions for action are still statisfied, then it will perform the action. However, if after the six seconds the conditions are not satisfied then the ASW will disgard that action and go back to waiting for the aircraft to cross the threshold.

C7 The ASW shall enter reduced functionality mode when two episodes of invalid altitude lasting at least one second occur within one minute of each other

C7.1 While in reduced functionality mode, the ASW will delay performing any action by a minumum delay period (2 seconds) at which time if the conditions for action are still satisfied the ASW will perform the action

C7.2 While in reduced functionality mode, the ASW will not wait to peform an action longer than the maximum delay time (6 seconds).

C7.3 The ASW shall exit the reduce functionality mode upon receipt of one minute of valid altitude data

As defined, the ASW system currently allows for almost any action to be performed as a result of the estimated altitude. A subfamily of the broad ASW famly would be the class of ASW devices responsible for turning on or off a particular Device of Interest (DOI) on board the aircraft.

$C_{DOI}1$ The ASW shall change the status (turn on or off) a Device of Interest (DOI) when it crosses a certain threshold

$V_{DOI}1$ The threshold for the ASW varies from 0 to 8024 feet from aircraft to aircraft

$V_{DOI}2$ Whether the ASW turns on/off the DOI when passing above/below the threshold is a variability with nine possible choices:

- do nothing going above or below;
- turn on going below, do nothing going above;
- turn off going below, do nothing going above;
- do nothing going below, turn on going above;
- turn on going below, turn on going above;
- turn off going below, turn on going above;
- do nothing going below, turn off going above;
- turn on going below, turn off going above; or,
- turn off going below, turn off going above;

To deal with noisy data, or the aircraft flying near to the threshold altitude, the DOI controlling ASW needs to have a certain hysteresis factor that is used to deterimine how much the altitude of the plane must change in order to have the DOI powered on or off again. The commonalities and variabilities that govern the hysteresis function of the ASW are given below.

$C_{DOI}2$ The ASW shall employ a hystersis factor to ensure that when the aircraft is flying at approximately the threshold altitude noisy data from the altimeters or slight variations in altitude do not cause the ASW to turn on/off the DOI in rapid succession

$V_{DOI}3$ The hysteresis factor may vary from aircraft to aircraft between 50 ft and 500 ft.

$V_{DOI}4$ The hysteresis factor may vary depending whether or not the aircraft is going above or below the threshold.

$C_{DOI}3$ Both the hysteresis factor for going above and the hysteresis factor for going below shall be a constant for each particular aircraft (i.e., known at specification time).

Finally, the ASW will received updates from the DOI whenever that status of the DOI changes. This is important to confirm whether or not the DOI is responding to the commands issued by the ASW as well as fofill the requirement denoted by the final commonality.

$C_{DOI}4$ The DOI shall give the ASW an indication of its status (on or off) whenever that status changes

$C_{DOI}5$ Whenever the ASW submits a command to the DOI, it shall wait for a specified period of time for the status of the DOI to change to reflect the command. If the status does not change within the specified period of time, then the ASW shall declare a failure. The period of time will be a constant for each aircraft

$V_{DOI}5$ The period of time that the ASW will wait after issuing a command to the DOI before indicating a failure if the DOI does not change status shall vary between 1 second and 5 seconds from aircraft to aircraft.

$C_{DOI}6$ The ASW shall not attempt to power on the DOI if the DOI is already on or attempt to power off the DOI if the DOI is already off.

In this section, we have discussed the commonalities and variabilities for the ASW family. In the next section, we will examine the structure of the ASW family and we will present the decision model for the ASW family.

A.2 Structure and Members of the ASW Family

As discussed in Chapter 7, the high-level structure of the ASW can be visualized as in Figures A.1 and A.2.

The next section presents the decision model for the ASW.

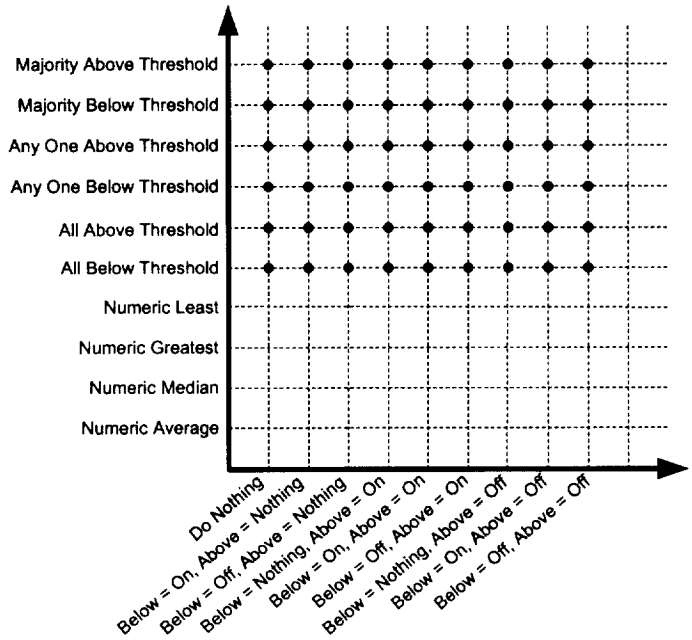

Figure A.1: The ASW family structure visualized in 2 dimensions

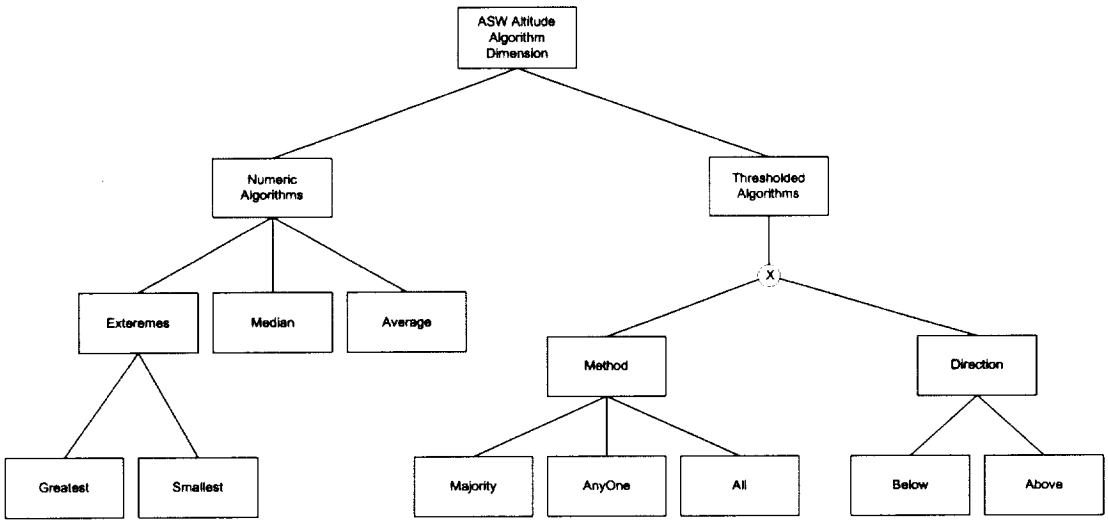

Figure A.2: The structure of the Altitude Dimension for the ASW

Variability	CS-123	CS-134	DD-123	DD-134	EF-155
# of Analog Alt.	1	1	1	1	2
# of Digital Alt.	1	2	1	2	3
Threshold Algo.	Any	Any	Any	Majority	Majority
Invalid Alt. Failure	4 s	2 s	2 s	2 s	2 s
Threshold	2000 ft	2000 ft	2000 ft	2000 ft	1500 ft
Go Above Action	None	None	None	None	Turn Off
Go Below Action	Turn On	Turn On	Turn On	Turn On	Turn On
Go Above Hyst.	200 ft	200 ft	250 ft	200 ft	200 ft
Go Below Hyst.	NA	NA	NA	NA	200 ft
DOI timeout	2 s	2 s	2 s	2 s	2 s

Figure A.3: A tabular representation of the ASW family decision model

A.3 Decision Model for the ASW

This section presents the decision model for the ASW family used in the methodology document. For the purposes of the methodology, we will consider an ASW family with five members. Figure A.3 shows the tabular decision model for the family.

Appendix B

The Altitude Switch in RSML^{-e}-Phase 2

```
TYPE_DEF OnOffType_P2 { On, Off }
TYPE_DEF ActionType {NoAction, TurnOn, TurnOff}

MODULE ASW_REQ :

  INTERFACE :

    EXPORT CON_DOI_P2 : {On, Off, Uncommanded}
    Purpose : &*L  This variable represents the ASW's
    commanded status of the Device of Interest (DOI). L*&

    Interpretation : &*L
      \begin{quote}
      \begin{mydescription}
      \item[On:] Indicates that the DOI is commanded to be On.  The DOI
      is commanded to be on when the aircraft enters the target region
      for turning the DOI on, the DOI is not already on,
      and the ASW is not inhibitied.
      \item[Off:] Indicates that the DOI is commanded to be Off.  The
      DOI is commanded to be off when the aircraft leaves the target
      region and after a certain period of time has passed.  If this
      time is \RUndefined, then the ASW will never turn the DOI Off.
      \item[Uncommanded:] Indicates that the DOI is not commanded by the
      ASW.  This CON\_DOI variable will be equal to Uncommanded in any
      step were the ASW does not issue a command to the device of interest.
      \end{mydescription}
```

```
  \end{quote}
  L*&

Issues : &*L
  \begin{myitemize}
  \item If the aircraft leaves the target area and the DOI is on,
  but was {\em not} commanded to be on by the ASW, should the ASW
  turn it off?
  \end{myitemize}
  L*&

END EXPORT

EXPORT CON_Failure_P2 : Boolean
 Purpose : &*L This variable represents the ASW's indication of
 whether or not it has failed to the external world.  It is
 potentially displayed to the pilot and/or used by other subsystems
 on board the aircraft. L*&

 Interpretation : &*L
  \begin{quote}
  \begin{mydescription}
  \item[True:] Indicates that the ASW has failed.  The ASW is
  considered to be failed if it attempts to turn on the DOI, but the
  DOI does not turn on after a certain timeout period.
  \item[False:] Indicates that the ASW has not failed.  The ASW is
  considered to be operating normally if none of the failure
  conditions are true.
  \end{mydescription}
  \end{quote}
  L*&
END EXPORT

IMPORT MON_Altitude_P2 : INTEGER
  UNITS : ft
  EXPECTED_MIN : 0
  EXPECTED_MAX : 50000
  CLASSIFICATION : Monitored

  Purpose : &*L This variable represents the ASW's idea of what the
  altitude of the aircraft is.  It is related to the Altitude\_Quality
  variable. L*&
```

```
  Interpretation : &*L
    \begin{quote}
    \begin{mydescription}
    \item[Precision:] We will know the altitude to within $\pm 10$ ft.
    \end{mydescription}
    \end{quote}
    L*&

END IMPORT

IMPORT MON_DOI_P2 : OnOffType_P2
  Purpose : &*L This variable indicates the monitored status of the
  DOI.  The DOI can be turned on or off by other devices/systems on
  board the aircraft, so the ASW needs an accurate accounting of the
  status of the DOI L*&

  Interpretation : &*L
    \begin{quote}
    \begin{mydescription}
    \item[On:] Indicates that the DOI is currently on.
    \item[Off:] Indicates that the DOI is currently off.
    \end{mydescription}
    \end{quote}
    L*&

 END IMPORT

IMPORT MON_Reset_P2 : Boolean

  Purpose : &*L This variable indicates the whether the ASW should be
  reset or not.  In a step where the ASW is reset, this variable will
  have the value true.  In all others, this variable will have the
  value false. L*&

  Interpretation : &*L
    \begin{quote}
    \begin{mydescription}
    \item[True:] Indicates that the ASW as been reset.
    \item[False:] Indicates that the ASW has not been reset.
    \end{mydescription}
    \end{quote}
```

```
    L*&

END IMPORT

IMPORT MON_Inhibit_P2 : Boolean

   Purpose : &*L This variable is true when the ASW is inhibited and
   false otherwise.  The value is determined by the user and/or other
   systems on board the aircraft. L*&

  Interpretation : &*L
    \begin{quote}
    \begin{mydescription}
    \item[True:] Indicates that the operation of the ASW has been
    inhibited; the ASW shall not attempt to change the status of the
    DOI.
    \item[False:] Indicates that the ASW has not been inhibited; the
    ASW will behave as specified by other requirements.
    \end{mydescription}
    \end{quote}
    L*&

END IMPORT

IMPORT CONSTANT Threshold : INTEGER
  UNITS : ft
  EXPECTED_MIN : 0
  EXPECTED_MAX : 8024

  Purpose : &*L This constant will be defined by each family
  member when the REQ module is instantiated.  It is the altitude
  at which the ASW is required to turn on or off the ASW. L*&

END IMPORT

IMPORT CONSTANT Invalid_Alt_Failure : Time
  UNITS : NA
  EXPECTED_MIN : 2 s
  EXPECTED_MAX : 10 s

  Purpose : &*L This constant will be defined by each family
  member.  It is the length of time after which the ASW will
```

```
  declare a failure if there is not valid altitude. L*&

END IMPORT

IMPORT CONSTANT DOI_Timeout : Time
  UNITS : NA
  EXPECTED_MIN : 1 s
  EXPECTED_MAX : 5 s

  Purpose : &*L This constant will be defined by each member of
  the ASW family to represent the amount of time before the ASW
  declares a failure if the DOI does not respond to a command. L*&

END IMPORT

IMPORT CONSTANT GoAboveAction : ActionType

  Purpose : &*L This constant specifies the action that the ASW
  will perform when it crosses the Threshold going up. It is
  specified by the decision model for each family member. L*&

END IMPORT

IMPORT CONSTANT GoBelowAction : ActionType

  Purpose : &*L This constant specifies the action that the ASW
  will perform when it crosses the Threshold going down. It is
  specified by the decision model for each family member. L*&

END IMPORT

IMPORT CONSTANT GoAboveHyst : INTEGER
  UNITS : ft
  EXPECTED_MIN : 50
  EXPECTED_MAX : 500

  Purpose : &*L This defines the hysteresis factor for going above
  the threshold altitude. L*&

END IMPORT

IMPORT CONSTANT GoBelowHyst : INTEGER
```

```
      UNITS : ft
      EXPECTED_MIN : 50
      EXPECTED_MAX : 500

      Purpose : &*L This defines the hysteresis factor for going above
      the threshold altitude. L*&

    END IMPORT

  END INTERFACE

  DEFINITION :
  END DEFINITION

END MODULE
```

Appendix C

The Altitude Switch in RSML^{-e}- Phase 3

```
INCLUDE "asw-alltypes.nimbus"

MODULE ASW_REQ_P3 :

  INTERFACE :

    EXPORT CON_DOI_P3 : {On, Off, Uncommanded}
    Purpose : &*L  This variable represents the ASW's
    commanded status of the Device of Interest (DOI). L*&

    Interpretation : &*L
      \begin{quote}
      \begin{mydescription}
      \item[On:] Indicates that the DOI is commanded to be On.  The DOI
      is commanded to be on when the aircraft enters the target region
      for turning the DOI on, the DOI is not already on,
      and the ASW is not inhibitied.
      \item[Off:] Indicates that the DOI is commanded to be Off.  The
      DOI is commanded to be off when the aircraft leaves the target
      region and after a certain period of time has passed.  If this
      time is \RUndefined, then the ASW will never turn the DOI Off.
      \item[Uncommanded:] Indicates that the DOI is not commanded by the
      ASW.  This CON\_DOI variable will be equal to Uncommanded in any
      step were the ASW does not issue a command to the device of interest.
      \end{mydescription}
      \end{quote}
      L*&

    Issues : &*L
      \begin{myitemize}
```

```
  \item If the aircraft leaves the target area and the DOI is on,
  but was {\em not} commanded to be on by the ASW, should the ASW
  turn it off?
  \end{myitemize}
  L*&

END EXPORT

EXPORT CON_Failure_P3 : Boolean
 Purpose : &*L This variable represents the ASW's indication of
 whether or not it has failed to the external world.  It is
 potentially displayed to the pilot and/or used by other subsystems
 on board the aircraft. L*&

 Interpretation : &*L
  \begin{quote}
  \begin{mydescription}
  \item[True:] Indicates that the ASW has failed.  The ASW is
  considered to be failed if it attempts to turn on the DOI, but the
  DOI does not turn on after a certain timeout period.
  \item[False:] Indicates that the ASW has not failed.  The ASW is
  considered to be operating normally if none of the failure
  conditions are true.
  \end{mydescription}
  \end{quote}
  L*&
END EXPORT

IMPORT MON_Altitude_P3 : INTEGER
  UNITS : ft
  EXPECTED_MIN : 0
  EXPECTED_MAX : 50000
  CLASSIFICATION : Monitored

  Purpose : &*L This variable represents the ASW's idea of what the
  altitude of the aircraft is.  It is related to the Altitude\_Quality
  variable. L*&

  Interpretation : &*L
    \begin{quote}
    \begin{mydescription}
    \item[Precision:] We will know the altitude to within $\pm 10$ ft.
    \end{mydescription}
    \end{quote}
    L*&

END IMPORT
```

```
IMPORT MON_Altitude_Quality_P3 : AltitudeQualityType
  CLASSIFICATION : Monitored

  Purpose : &*L This variable represents the quality of the
  Altitude of the aircraft is. L*&
END IMPORT

IMPORT MON_DOI_P3 : OnOffType_P3
  Purpose : &*L This variable indicates the monitored status of the
  DOI.  The DOI can be turned on or off by other devices/systems on
  board the aircraft, so the ASW needs an accurate accounting of the
  status of the DOI L*&

  Interpretation : &*L
    \begin{quote}
    \begin{mydescription}
    \item[On:] Indicates that the DOI is currently on.
    \item[Off:] Indicates that the DOI is currently off.
    \end{mydescription}
    \end{quote}
    L*&

 END IMPORT

IMPORT MON_Reset_P3 : Boolean

  Purpose : &*L This variable indicates the whether the ASW should be
  reset or not.  In a step where the ASW is reset, this variable will
  have the value true.  In all others, this variable will have the
  value false. L*&

  Interpretation : &*L
    \begin{quote}
    \begin{mydescription}
    \item[True:] Indicates that the ASW as been reset.
    \item[False:] Indicates that the ASW has not been reset.
    \end{mydescription}
    \end{quote}
    L*&

END IMPORT

IMPORT MON_Inhibit_P3 : Boolean

   Purpose : &*L This variable is true when the ASW is inhibited and
   false otherwise.  The value is determined by the user and/or other
   systems on board the aircraft. L*&
```

```
  Interpretation : &*L
    \begin{quote}
    \begin{mydescription}
    \item[True:] Indicates that the operation of the ASW has been
    inhibited; the ASW shall not attempt to change the status of the
    DOI.
    \item[False:] Indicates that the ASW has not been inhibited; the
    ASW will behave as specified by other requirements.
    \end{mydescription}
    \end{quote}
    L*&

END IMPORT

IMPORT CONSTANT Threshold_P3 : INTEGER
  UNITS : ft
  EXPECTED_MIN : 0
  EXPECTED_MAX : 8024

  Purpose : &*L This constant will be defined by each family
  member when the REQ module is instantiated.  It is the altitude
  at which the ASW is required to turn on or off the ASW. L*&

END IMPORT

IMPORT CONSTANT Invalid_Alt_Failure_P3 : Time
  UNITS : NA
  EXPECTED_MIN : 2 s
  EXPECTED_MAX : 10 s

  Purpose : &*L This constant will be defined by each family
  member.  It is the length of time after which the ASW will
  declare a failure if there is not valid altitude. L*&

END IMPORT

IMPORT CONSTANT DOI_Timeout_P3 : Time
  UNITS : NA
  EXPECTED_MIN : 1 s
  EXPECTED_MAX : 5 s

  Purpose : &*L This constant will be defined by each member of
  the ASW family to represent the amount of time before the ASW
  declares a failure if the DOI does not respond to a command. L*&

END IMPORT

IMPORT CONSTANT GoAboveAction_P3 : ActionType
```

```
      Purpose : &*L This constant specifies the action that the ASW
      will perform when it crosses the Threshold going up. It is
      specified by the decision model for each family member. L*&

    END IMPORT

    IMPORT CONSTANT GoBelowAction_P3 : ActionType

      Purpose : &*L This constant specifies the action that the ASW
      will perform when it crosses the Threshold going down. It is
      specified by the decision model for each family member. L*&

    END IMPORT

    IMPORT CONSTANT GoAboveHyst_P3 : INTEGER
      UNITS : ft
      EXPECTED_MIN : 50
      EXPECTED_MAX : 500

      Purpose : &*L This defines the hysteresis factor for going above
      the threshold altitude. L*&

    END IMPORT

    IMPORT CONSTANT GoBelowHyst_P3 : INTEGER
      UNITS : ft
      EXPECTED_MIN : 50
      EXPECTED_MAX : 500

      Purpose : &*L This defines the hysteresis factor for going above
      the threshold altitude. L*&

    END IMPORT

  END INTERFACE

  DEFINITION :
  END DEFINITION

END MODULE


MODULE ThresholdedAltitude_P3 :

  INTERFACE :

    IMPORT Altitude_P3 : Integer
```

```
      UNITS : ft
      EXPECTED_MIN : 0
      EXPECTED_MAX : 50000
    END IMPORT

    IMPORT CONSTANT Threshold_P3 : Integer
      UNITS : ft
      EXPECTED_MIN : 0
      EXPECTED_MAX : 8024
    END IMPORT

    IMPORT CONSTANT AboveHysteresis_P3 : Integer
      UNITS : ft
      EXPECTED_MIN : 50
      EXPECTED_MAX : 500
    END IMPORT

    IMPORT CONSTANT BelowHysteresis_P3 : Integer
      UNITS : ft
      EXPECTED_MIN : 50
      EXPECTED_MAX : 500
    END IMPORT

    EXPORT AboveOrBelow : AboveBelowType

      Purpose : &*L this export reports whether or not the altitude is
      above or below the threshold given the hysteresis factor L*&

    END EXPORT

  END INTERFACE

  DEFINITION :
  END DEFINTION

END MODULE


INCLUDE "standard-modules.nimbus"
```

Appendix D

The Altitude Switch in RSML^{-e}-Phase 4

```
INCLUDE "asw-alltypes.nimbus"

MODULE ASW_REQ_P4 :

  INTERFACE :

    EXPORT CON_DOI_P4 : DOIControlledType
    Purpose : &*L  This variable represents the ASW's
    commanded status of the Device of Interest (DOI). L*&

    Interpretation : &*L
      \begin{quote}
      \begin{mydescription}
      \item[On:] Indicates that the DOI is commanded to be On.  The DOI
      is commanded to be on when the aircraft enters the target region
      for turning the DOI on, the DOI is not already on,
      and the ASW is not inhibitied.
      \item[Off:] Indicates that the DOI is commanded to be Off.  The
      DOI is commanded to be off when the aircraft leaves the target
      region and after a certain period of time has passed.  If this
      time is \RUndefined, then the ASW will never turn the DOI Off.
      \item[Uncommanded:] Indicates that the DOI is not commanded by the
      ASW.  This CON\_DOI variable will be equal to Uncommanded in any
      step were the ASW does not issue a command to the device of interest.
      \end{mydescription}
      \end{quote}
      L*&

    Issues : &*L
      \begin{myitemize}
```

```
  \item If the aircraft leaves the target area and the DOI is on,
  but was {\em not} commanded to be on by the ASW, should the ASW
  turn it off?
  \end{myitemize}
  L*&

END EXPORT

EXPORT CON_Failure_P4 : Boolean
 Purpose : &*L This variable represents the ASW's indication of
 whether or not it has failed to the external world.  It is
 potentially displayed to the pilot and/or used by other subsystems
 on board the aircraft. L*&

 Interpretation : &*L
  \begin{quote}
  \begin{mydescription}
  \item[True:] Indicates that the ASW has failed.  The ASW is
  considered to be failed if it attempts to turn on the DOI, but the
  DOI does not turn on after a certain timeout period.
  \item[False:] Indicates that the ASW has not failed.  The ASW is
  considered to be operating normally if none of the failure
  conditions are true.
  \end{mydescription}
  \end{quote}
  L*&
END EXPORT

IMPORT MON_Altitude_P4 : INTEGER
  UNITS : ft
  EXPECTED_MIN : 0
  EXPECTED_MAX : 50000
  CLASSIFICATION : Monitored

  Purpose : &*L This variable represents the ASW's idea of what the
  altitude of the aircraft is.  It is related to the Altitude\_Quality
  variable. L*&

  Interpretation : &*L
    \begin{quote}
    \begin{mydescription}
    \item[Precision:] We will know the altitude to within $\pm 10$ ft.
    \end{mydescription}
    \end{quote}
    L*&

END IMPORT
```

```
IMPORT MON_Altitude_Quality_P4 : AltitudeQualityType
  CLASSIFICATION : Monitored

  Purpose : &*L This variable represents the quality of the
  Altitude of the aircraft is. L*&
END IMPORT

IMPORT MON_DOI_P4 : OnOffType_P4
  Purpose : &*L This variable indicates the monitored status of the
  DOI.  The DOI can be turned on or off by other devices/systems on
  board the aircraft, so the ASW needs an accurate accounting of the
  status of the DOI L*&

  Interpretation : &*L
    \begin{quote}
    \begin{mydescription}
    \item[On:] Indicates that the DOI is currently on.
    \item[Off:] Indicates that the DOI is currently off.
    \end{mydescription}
    \end{quote}
    L*&

 END IMPORT

IMPORT MON_Reset_P4 : Boolean

  Purpose : &*L This variable indicates the whether the ASW should be
  reset or not.  In a step where the ASW is reset, this variable will
  have the value true.  In all others, this variable will have the
  value false. L*&

  Interpretation : &*L
    \begin{quote}
    \begin{mydescription}
    \item[True:] Indicates that the ASW as been reset.
    \item[False:] Indicates that the ASW has not been reset.
    \end{mydescription}
    \end{quote}
    L*&

END IMPORT

IMPORT MON_Inhibit_P4 : Boolean

   Purpose : &*L This variable is true when the ASW is inhibited and
   false otherwise.  The value is determined by the user and/or other
   systems on board the aircraft. L*&
```

```
  Interpretation : &*L
    \begin{quote}
    \begin{mydescription}
    \item[True:] Indicates that the operation of the ASW has been
    inhibited; the ASW shall not attempt to change the status of the
    DOI.
    \item[False:] Indicates that the ASW has not been inhibited; the
    ASW will behave as specified by other requirements.
    \end{mydescription}
    \end{quote}
    L*&

END IMPORT

IMPORT CONSTANT Threshold_P4 : INTEGER
  UNITS : ft
  EXPECTED_MIN : 0
  EXPECTED_MAX : 8024

  Purpose : &*L This constant will be defined by each family
  member when the REQ module is instantiated.  It is the altitude
  at which the ASW is required to turn on or off the ASW. L*&

END IMPORT

IMPORT CONSTANT Invalid_Alt_Failure_P4 : Time
  UNITS : NA
  EXPECTED_MIN : 2 s
  EXPECTED_MAX : 10 s

  Purpose : &*L This constant will be defined by each family
  member.  It is the length of time after which the ASW will
  declare a failure if there is not valid altitude. L*&

END IMPORT

IMPORT CONSTANT DOI_Timeout_P4 : Time
  UNITS : NA
  EXPECTED_MIN : 1 s
  EXPECTED_MAX : 5 s

  Purpose : &*L This constant will be defined by each member of
  the ASW family to represent the amount of time before the ASW
  declares a failure if the DOI does not respond to a command. L*&

END IMPORT

IMPORT CONSTANT GoAboveAction_P4 : ActionType
```

```
    Purpose : &*L This constant specifies the action that the ASW
    will perform when it crosses the Threshold going up. It is
    specified by the decision model for each family member. L*&

  END IMPORT

  IMPORT CONSTANT GoBelowAction_P4 : ActionType

    Purpose : &*L This constant specifies the action that the ASW
    will perform when it crosses the Threshold going down. It is
    specified by the decision model for each family member. L*&

  END IMPORT

  IMPORT CONSTANT GoAboveHyst_P4 : INTEGER
    UNITS : ft
    EXPECTED_MIN : 50
    EXPECTED_MAX : 500

    Purpose : &*L This defines the hysteresis factor for going above
    the threshold altitude. L*&

  END IMPORT

  IMPORT CONSTANT GoBelowHyst_P4 : INTEGER
    UNITS : ft
    EXPECTED_MIN : 50
    EXPECTED_MAX : 500

    Purpose : &*L This defines the hysteresis factor for going above
    the threshold altitude. L*&

  END IMPORT

END INTERFACE

DEFINITION :

  STATE_VARIABLE ASW_System_Mode_P4 :
    VALUES : {Reset, Operating}
    PARENT : NONE

    Purpose : &*L This is the top-level mode of the ASW.  If the ASW
    were to have a startup mode, etc., we could put those modes as
    children of this controlling mode.  Currently, we have only two
    states, the reset mode which is used for when the reset signal
    is received and the operating mode that handles the main
```

```
      behavior. L*&

      DEFAULT_VALUE : Operating

      TRANSITION Operating TO Reset IF
        WHEN(MON_Reset_P4, False)

      TRANSITION Reset TO Operating IF
        DURATION(PRE(ASW_System_Mode_P4), 0 s, Clock) >= 0 S

    END STATE_VARIABLE

    MODULE_INSTANCE ASW_Operating_Mode_P4 : ASW_Operating_Mode_Def_P4
      PARENT :  ASW_System_Mode_P4.Operating
      ASSIGNMENT
         MON_Altitude_P4         := MON_Altitude_P4,
         MON_Altitude_Quality_P4 := MON_Altidue_Quality_P4,
         MON_DOI_P4              := MON_DOI_P4,
         MON_Inhibit_P4          := MON_Inhibit_P4,
         Threshold_P4            := Threshold_P4,
         Invalid_Alt_Failure_P4  := Invalid_Alt_Failure_P4,
         DOI_Timeout_P4          := DOI_Timeout_P4,
         GoAboveAction_P4        := GoAboveAction_P4,
         GoBelowAction_P4        := GoBelowAction_P4,
         GoAboveHyst_P4          := GoAboveHyst_P4,
         GoBelowHyst_P4          := GoBelowHyst_P4
      END ASSIGNMENT
    END MODULE_INSTANCE

    EXPORT CON_DOI_P4 :
      PARENT :  ASW_System_Mode_P4.Operating
      DEFAULT_VALUE : ASW_Operating_Mode_P4.CON_DOI_P4
      EQUALS          ASW_Operating_Mode_P4.CON_DOI_P4
    END EXPORT

    EXPORT CON_Failure_P4 :
      PARENT :  ASW_System_Mode_P4.Operating
      DEFAULT_VALUE : ASW_Operating_Mode_P4.CON_Failure_P4
      EQUALS          ASW_Operating_Mode_P4.CON_Failure_P4
    END EXPORT

  END DEFINITION

END MODULE


MODULE ASW_OperatingMode_Def_P4 :
```

```
INTERFACE :

  EXPORT CON_DOI_P4 : DOIControlledType
  END EXPORT

  EXPORT CON_Failure_P4 : Boolean
  END EXPORT

  IMPORT MON_Altitude_P4 : INTEGER
  END IMPORT

  IMPORT MON_Altitude_Quality_P4 : AltitudeQualityType
  END IMPORT

  IMPORT MON_DOI_P4 : OnOffType_P4
  END IMPORT

  IMPORT MON_Inhibit_P4 : Boolean
  END IMPORT

  IMPORT CONSTANT Threshold_P4 : INTEGER
    UNITS : ft
    EXPECTED_MIN : 0
    EXPECTED_MAX : 8024
  END IMPORT

  IMPORT CONSTANT Invalid_Alt_Failure_P4 : Time
    UNITS : NA
    EXPECTED_MIN : 2 s
    EXPECTED_MAX : 10 s
  END IMPORT

  IMPORT CONSTANT DOI_Timeout_P4 : Time
    UNITS : NA
    EXPECTED_MIN : 1 s
    EXPECTED_MAX : 5 s
  END IMPORT

  IMPORT CONSTANT GoAboveAction_P4 : ActionType
  END IMPORT

  IMPORT CONSTANT GoBelowAction_P4 : ActionType
  END IMPORT

  IMPORT CONSTANT GoAboveHyst_P4 : INTEGER
    UNITS : ft
    EXPECTED_MIN : 50
    EXPECTED_MAX : 500
```

```
  END IMPORT

  IMPORT CONSTANT GoBelowHyst_P4 : INTEGER
    UNITS : ft
    EXPECTED_MIN : 50
    EXPECTED_MAX : 500
  END IMPORT

END INTERFACE

DEFINITION :

  EXPORT CON_DOI_P4 :
    PARENT : NONE
    DEFAULT_VALUE : Uncommanded

    TRANSITION Uncommanded TO On IF
      TABLE
         DOI_Action_Ok(On)                                    : T T ;
         WHEN(ThresholdedAlt_P4.Result_P4 = Below, False) : T * ;
         GoBelowAction = TurnOn                               : T * ;
         WHEN(ThresholdedAlt_P4.Result_P4 = Above, False) : * T ;
         GoAboveAction = TurnOn                               : * T ;
      END TABLE

    TRANSITION Uncommanded TO Off IF
      TABLE
         DOI_Action_Ok(Off)                                   : T T ;
         WHEN(ThresholdedAlt_P4.Result_P4 = Below, False) : T * ;
         GoBelowAction = TurnOff                              : T * ;
         WHEN(ThresholdedAlt_P4.Result_P4 = Above, False) : * T ;
         GoAboveAction = TurnOff                              : * T ;
      END TABLE

    TRANSITION On TO Uncommanded IF WHEN(MON_DOI_P4 = On, False)

    TRANSITION Off TO Uncommanded IF WHEN(MON_DOI_P4 = Off, False)

  END EXPORT

  MACRO DOI_Action_Ok(act IS ActionType) :
    TABLE
      MON_Inhibit_P4     : F F ;
      CON_Failure_P4     : F F ;
      MON_DOI_P4 = On    : T * ;
      act = On           : F * ;
      MON_DOI_P4 = Off   : * T ;
      act = Off          : * F ;
```

```
  END TABLE
END MACRO

EXPORT CON_Failure_P4 :
  PARENT : NONE
  DEFAULT_VALUE : False

  EQUALS TRUE IF
    TABLE
      DURATION(AttemptingOn_P4(), 0 S, Clock) > DOI_Timeout_P4     : T * * * ;
      DURATION(AttemptingOff_P4(), 0 S, Clock) > DOI_Timeout_P4    : * T * * ;
      DURATION(MON_Altitude_Quality_P4 = Invalid, 0 S, Clock)   : * * T * ;
      PRE(CON_Failure_P4) = False                               : * * * T ;
    END TABLE

  EQUALS FALSE IF
    TABLE
      DURATION(AttemptingOn_P4(), 0 S, Clock) > DOI_Timeout_P4     : F ;
      DURATION(AttemptingOff_P4(), 0 S, Clock) > DOI_Timeout_P4    : F ;
      DURATION(MON_Altitude_Quality_P4 = Invalid, 0 S, Clock)   : F ;
      PRE(CON_Failure_P4) = False                               : F ;
    END TABLE

END EXPORT

MACRO AttemptingOn_P4() :
  TABLE
    MON_DOI_P4 = Off    : T ;
    CON_DOI_P4 = On     : T ;
  END TABLE
END MACRO

MACRO AttemptingOff_P4() :
  TABLE
    MON_DOI_P4 = On     : T ;
    CON_DOI_P4 = Off    : T ;
  END TABLE
END MACRO

MODULE_INSTANCE ThresholdedAlt_P4 : ThresholdedAltitude_P4
  PARENT : NONE
  ASSIGNMENT
    Altitude_P4 := MON_Altitude_P4,
    Threshold_P4 := Threshold_P4,
    BelowHysteresis_P4 := GoBelowHyst_P4,
    AboveHysteresis_P4 := GoAboveHyst_P4
  END ASSIGNMENT
END MODULE_INSTANCE
```

```
  END DEFINITION

END MODULE


MODULE ThresholdedAltitude_P4 :

  INTERFACE :

    IMPORT Altitude_P4 : Integer
      UNITS : ft
      EXPECTED_MIN : 0
      EXPECTED_MAX : 50000
    END IMPORT

    IMPORT CONSTANT Threshold_P4 : Integer
      UNITS : ft
      EXPECTED_MIN : 0
      EXPECTED_MAX : 8024
    END IMPORT

    IMPORT CONSTANT AboveHysteresis_P4 : Integer
      UNITS : ft
      EXPECTED_MIN : 50
      EXPECTED_MAX : 500
    END IMPORT

    IMPORT CONSTANT BelowHysteresis_P4 : Integer
      UNITS : ft
      EXPECTED_MIN : 50
      EXPECTED_MAX : 500
    END IMPORT

    EXPORT Result_P4 : AboveBelowType

      Purpose : &*L this export reports whether or not the altitude is
      above or below the threshold given the hysteresis factor L*&

    END EXPORT

  END INTERFACE

  DEFINITION :

    EXPORT Result_P4 :
      PARENT : NONE
```

```
  DEFAULT_VALUE : Above IF
    TABLE
      DEFINED(Altitude_P4)        : T ;
      Altitude_P4 > Threshold_P4  : T ;
    END TABLE

  DEFAULT_VALUE : Below IF
    TABLE
      DEFINED(Altitude_P4)        : T ;
      Altitude_P4 <= Threshold_P4 : T ;
    END TABLE

  DEFAULT_VALUE : UNDEFINED IF NOT (DEFINED(Altitude_P4))

  EQUALS Above IF
    TABLE
      DEFINED(Altitude_P4)                : T ;
      Altitude_P4 > EffectiveThreshold_P4 : T ;
    END TABLE

  EQUALS Below IF
    TABLE
      DEFINED(Altitude_P4)                 : T ;
      Altitude_P4 <= EffectiveThreshold_P4 : T ;
    END TABLE

  EQUALS UNDEFINED IF NOT (DEFINED(Altitude_P4))

END EXPORT

STATE_VARIABLE ApplyHisteresis_P4 :
  VALUES : {NoHyst, Above, Below}
  PARENT : NONE

  DEFAULT_VALUE : NoHyst

  TRANSITION NoHyst TO Above IF
    TABLE
      DEFINED(Altitude_P4)                    : T ;
      WHEN(Altitude_P4 < Threshold_P4, False) : T ;
    END TABLE

  TRANSITION NoHyst TO Below IF
    TABLE
      DEFINED(Altitude_P4)                    : T ;
      WHEN(Altitude_P4 > Threshold_P4, False) : T ;
    END TABLE
```

```
      TRANSITION Above TO NoHyst IF
        TABLE
          DEFINED(Altitude_P4)                                          : T T ;
          WHEN(Altitude_P4 < Threshold_P4 + AboveHysteresis_P4, False) : T * ;
          WHEN(Altitude_P4 > Threshold_P4 - BelowHysteresis_P4, False) : * T ;

        END TABLE

      TRANSITION Below TO NoHyst IF
        TABLE
          DEFINED(Altitude_P4)                                          : T T ;
          WHEN(Altitude_P4 > Threshold_P4 + AboveHysteresis_P4, False) : T * ;
          WHEN(Altitude_P4 < Threshold_P4 - BelowHysteresis_P4, False) : * T ;

        END TABLE

    END STATE_VARIABLE

    STATE_VARIABLE EffectiveThreshold_P4 : INTEGER
      PARENT : NONE
      UNITS : ft
      EXPECTED_MIN : Threshold_P4 - BelowHysteresis_P4
      EXPECTED_MAX : Threshold_P4 + AboveHysteresis_P4

      DEFAULT_VALUE : Threshold_P4

      EQUALS Threshold_P4 + AboveHysteresis_P4
        IF ApplyHysteresis_P4 = Above

      EQUALS Threshold_P4 - BelowHysteresis_P4
        IF ApplyHysteresis_P4 = Below

      EQUALS Threshold_P4
        IF ApplyHysteresis_P4 = NoHyst

    END STATE_VARIABLE

  END DEFINITION

END MODULE


INCLUDE "standard-modules.nimbus"
```

Appendix E

The Altitude Switch in RSML^{-e}-Phase 5

```
INCLUDE "asw-alltypes.nimbus"

MODULE ASW_REQ_P5 :

  INTERFACE :

    EXPORT CON_DOI_P5 : DOIControlledType
    Purpose : &*L  This variable represents the ASW's
    commanded status of the Device of Interest (DOI). L*&

    Interpretation : &*L
      \begin{quote}
      \begin{mydescription}
      \item[On:] Indicates that the DOI is commanded to be On.  The DOI
      is commanded to be on when the aircraft enters the target region
      for turning the DOI on, the DOI is not already on,
      and the ASW is not inhibitied.
      \item[Off:] Indicates that the DOI is commanded to be Off.  The
      DOI is commanded to be off when the aircraft leaves the target
      region and after a certain period of time has passed.  If this
      time is \RUndefined, then the ASW will never turn the DOI Off.
      \item[Uncommanded:] Indicates that the DOI is not commanded by the
      ASW.  This CON\_DOI variable will be equal to Uncommanded in any
      step were the ASW does not issue a command to the device of interest.
      \end{mydescription}
      \end{quote}
      L*&

    Issues : &*L
      \begin{myitemize}
```

```
  \item If the aircraft leaves the target area and the DOI is on,
  but was {\em not} commanded to be on by the ASW, should the ASW
  turn it off?
  \end{myitemize}
  L*&

END EXPORT

EXPORT CON_Failure_P5 : Boolean
 Purpose : &*L This variable represents the ASW's indication of
 whether or not it has failed to the external world.  It is
 potentially displayed to the pilot and/or used by other subsystems
 on board the aircraft. L*&

 Interpretation : &*L
  \begin{quote}
  \begin{mydescription}
  \item[True:] Indicates that the ASW has failed.  The ASW is
  considered to be failed if it attempts to turn on the DOI, but the
  DOI does not turn on after a certain timeout period.
  \item[False:] Indicates that the ASW has not failed.  The ASW is
  considered to be operating normally if none of the failure
  conditions are true.
  \end{mydescription}
  \end{quote}
  L*&
END EXPORT

IMPORT MON_Altitude_P5 : INTEGER
  UNITS : ft
  EXPECTED_MIN : 0
  EXPECTED_MAX : 50000
  CLASSIFICATION : Monitored

  Purpose : &*L This variable represents the ASW's idea of what the
  altitude of the aircraft is.  It is related to the Altitude\_Quality
  variable. L*&

  Interpretation : &*L
    \begin{quote}
    \begin{mydescription}
    \item[Precision:] We will know the altitude to within $\pm 10$ ft.
    \end{mydescription}
    \end{quote}
    L*&

END IMPORT
```

```
IMPORT MON_Altitude_Quality_P5 : AltitudeQualityType
  CLASSIFICATION : Monitored

  Purpose : &*L This variable represents the quality of the
  Altitude of the aircraft is. L*&
END IMPORT

IMPORT MON_DOI_P5 : OnOffType_P5
  Purpose : &*L This variable indicates the monitored status of the
  DOI.  The DOI can be turned on or off by other devices/systems on
  board the aircraft, so the ASW needs an accurate accounting of the
  status of the DOI L*&

  Interpretation : &*L
    \begin{quote}
    \begin{mydescription}
    \item[On:] Indicates that the DOI is currently on.
    \item[Off:] Indicates that the DOI is currently off.
    \end{mydescription}
    \end{quote}
    L*&

 END IMPORT

IMPORT MON_Reset_P5 : Boolean

  Purpose : &*L This variable indicates the whether the ASW should be
  reset or not.  In a step where the ASW is reset, this variable will
  have the value true.  In all others, this variable will have the
  value false. L*&

  Interpretation : &*L
    \begin{quote}
    \begin{mydescription}
    \item[True:] Indicates that the ASW as been reset.
    \item[False:] Indicates that the ASW has not been reset.
    \end{mydescription}
    \end{quote}
    L*&

END IMPORT

IMPORT MON_Inhibit_P5 : Boolean

   Purpose : &*L This variable is true when the ASW is inhibited and
   false otherwise.  The value is determined by the user and/or other
   systems on board the aircraft. L*&
```

```
  Interpretation : &*L
    \begin{quote}
    \begin{mydescription}
    \item[True:] Indicates that the operation of the ASW has been
    inhibited; the ASW shall not attempt to change the status of the
    DOI.
    \item[False:] Indicates that the ASW has not been inhibited; the
    ASW will behave as specified by other requirements.
    \end{mydescription}
    \end{quote}
    L*&

END IMPORT

IMPORT CONSTANT Threshold_P5 : INTEGER
  UNITS : ft
  EXPECTED_MIN : 0
  EXPECTED_MAX : 8024

  Purpose : &*L This constant will be defined by each family
  member when the REQ module is instantiated.  It is the altitude
  at which the ASW is required to turn on or off the ASW. L*&

END IMPORT

IMPORT CONSTANT Invalid_Alt_Failure_P5 : Time
  UNITS : NA
  EXPECTED_MIN : 2 s
  EXPECTED_MAX : 10 s

  Purpose : &*L This constant will be defined by each family
  member.  It is the length of time after which the ASW will
  declare a failure if there is not valid altitude. L*&

END IMPORT

IMPORT CONSTANT DOI_Timeout_P5 : Time
  UNITS : NA
  EXPECTED_MIN : 1 s
  EXPECTED_MAX : 5 s

  Purpose : &*L This constant will be defined by each member of
  the ASW family to represent the amount of time before the ASW
  declares a failure if the DOI does not respond to a command. L*&

END IMPORT

IMPORT CONSTANT GoAboveAction_P5 : ActionType
```

```
    Purpose : &*L This constant specifies the action that the ASW
    will perform when it crosses the Threshold going up. It is
    specified by the decision model for each family member. L*&

  END IMPORT

  IMPORT CONSTANT GoBelowAction_P5 : ActionType

    Purpose : &*L This constant specifies the action that the ASW
    will perform when it crosses the Threshold going down. It is
    specified by the decision model for each family member. L*&

  END IMPORT

  IMPORT CONSTANT GoAboveHyst_P5 : INTEGER
    UNITS : ft
    EXPECTED_MIN : 50
    EXPECTED_MAX : 500

    Purpose : &*L This defines the hysteresis factor for going above
    the threshold altitude. L*&

  END IMPORT

  IMPORT CONSTANT GoBelowHyst_P5 : INTEGER
    UNITS : ft
    EXPECTED_MIN : 50
    EXPECTED_MAX : 500

    Purpose : &*L This defines the hysteresis factor for going above
    the threshold altitude. L*&

  END IMPORT

END INTERFACE

DEFINITION :

  STATE_VARIABLE ASW_System_Mode_P5 :
    VALUES : {Startup, NormalOperating, Degraded, Failed, Reset}
    PARENT : NONE

    Purpose : &*L This is the top-level mode of the ASW.  If the ASW
    were to have a startup mode, etc., we could put those modes as
    children of this controlling mode.  Currently, we have only two
    states, the reset mode which is used for when the reset signal
    is received and the operating mode that handles the main
```

```
    behavior. L*&

    DEFAULT_VALUE : Startup

    TRANSITION NormalOperating TO Reset IF MON_Reset_P5

    TRANSITION Degraded TO Reset IF MON_Reset_P5

    TRANSITION NormalOperating TO Degraded IF
      EpisodeMonitor_P5 = QualifyingEpisode

    TRANSITION Degraded TO NormalOperating IF
      DURATION (MON_Altitude_Quality_P5 = Valid, 0 S, Clock) > 1 MIN

    TRANSITION Reset TO NormalOperating IF
      DURATION(PRE(ASW_System_Mode_P5), 0 s, Clock) >= 0 S

  END STATE_VARIABLE

  STATE_VARIABLE EpisodeMonitor_P5 :
    VALUES : {NoEpisode, FirstEpisode, QualifyingEpisode}
    PARENT : NONE

    Purpose : &*L This simple state variable tracks whether or not
    we have met the conditions for being in degraded functionality
    mode.  Namely, whether or not we have seen two periods of
    invalid altitude lasting 1 second or more within 1 minute. L*&

     DEFAULT_VALUE : NoEpisode

     TRANSITION NoEpisode TO FirstEpisode IF
       DURATION(MON_Altitude_Quality_P5 = Invalid, 0 S, Clock) > 1 S

     TRANSITION FirstEpisode TO QualifyingEpisode IF
       TABLE
         DURATION(MON_Altitude_Quality_P5 = Invalid, 0 S, Clock) > 1 S : T ;
         DURATION(PRE(EpisodeMonitor_P5) = FirstEpisode) > 1 S         : T ;
       END TABLE

     TRANSITION FirstEpisode TO NoEpisode IF
       DURATION(PRE(EpisodeMonitor_P5) = FirstEpisode) >= 1 MIN

     TRANSITION QualifyingEpisode TO NoEpisode IF
       DURATION(MON_Altitude_Quality_P5 = Valid, 0 S, Clock) >= 2 MIN

  END STATE_VARIABLE

  MODULE_INSTANCE ASW_Operating_Mode_P5 : ASW_Operating_Mode_Def_P5
```

```
  PARENT :  ASW_System_Mode_P5.NormalOperating
  ASSIGNMENT
     MON_Altitude_P5          := MON_Altitude_P5,
     MON_Altitude_Quality_P5  := MON_Altidue_Quality_P5,
     MON_DOI_P5               := MON_DOI_P5,
     MON_Inhibit_P5           := MON_Inhibit_P5,
     Threshold_P5             := Threshold_P5,
     Invalid_Alt_Failure_P5   := Invalid_Alt_Failure_P5,
     DOI_Timeout_P5           := DOI_Timeout_P5,
     GoAboveAction_P5         := GoAboveAction_P5,
     GoBelowAction_P5         := GoBelowAction_P5,
     GoAboveHyst_P5           := GoAboveHyst_P5,
     GoBelowHyst_P5           := GoBelowHyst_P5,
     DOI_Delay_P5             := 0 S
  END ASSIGNMENT
END MODULE_INSTANCE

MODULE_INSTANCE ASW_Degraded_Mode_P5 : ASW_Operating_Mode_Def_P5
  PARENT :  ASW_System_Mode_P5.Degraded
  ASSIGNMENT
     MON_Altitude_P5          := MON_Altitude_P5,
     MON_Altitude_Quality_P5  := MON_Altidue_Quality_P5,
     MON_DOI_P5               := MON_DOI_P5,
     MON_Inhibit_P5           := MON_Inhibit_P5,
     Threshold_P5             := Threshold_P5,
     Invalid_Alt_Failure_P5   := Invalid_Alt_Failure_P5,
     DOI_Timeout_P5           := DOI_Timeout_P5,
     GoAboveAction_P5         := GoAboveAction_P5,
     GoBelowAction_P5         := GoBelowAction_P5,
     GoAboveHyst_P5           := GoAboveHyst_P5,
     GoBelowHyst_P5           := GoBelowHyst_P5,
     DOI_MinDelay_P5          := 2 S,
     DOI_MaxDelay_P5          := 6 S
  END ASSIGNMENT
END MODULE_INSTANCE

EXPORT CON_DOI_P5 :
  PARENT :  NONE
  DEFAULT_VALUE : Uncontrolled

  EQUALS ASW_Operating_Mode_P5.CON_DOI_P5
    IF ASW_System_Mode_P5 = NormalOperating

  EQUALS ASW_Degraded_Mode_P5.CON_DOI_P5
    IF ASW_System_Mode_P5 = Degraded

  EQUALS Uncontrolled IF
    TABLE
```

```
        ASW_System_Mode_P5 = Failed : T * ;
        ASW_System_Mode_P5 = Reset  : * T ;
      END TABLE

  END EXPORT

  EXPORT CON_Failure_P5 :
    PARENT :  NONE
    DEFAULT_VALUE : False

    TRANSITION False TO True IF
      TABLE
        ASW_System_Mode_P5 = NormalOperating   : T * ;
        ASW_Operating_Mode_P5.CON_Failure_P5   : T * ;
        ASW_System_Mode_P5 = Degraded          : * T ;
        ASW_Operating_Mode_P5.CON_Failure_P5   : * T ;
      END TABLE

    TRANSITION True TO False IF ASW_System_Mode_P5 = Reset

  END EXPORT

END DEFINITION

END MODULE


MODULE ASW_OperatingMode_Def_P5 :

  INTERFACE :

    EXPORT CON_DOI_P5 : DOIControlledType
    END EXPORT

    EXPORT CON_Failure_P5 : Boolean
    END EXPORT

    IMPORT MON_Altitude_P5 : INTEGER
    END IMPORT

    IMPORT MON_Altitude_Quality_P5 : AltitudeQualityType
    END IMPORT

    IMPORT MON_DOI_P5 : OnOffType_P5
    END IMPORT

    IMPORT MON_Inhibit_P5 : Boolean
    END IMPORT
```

```
IMPORT CONSTANT Threshold_P5 : INTEGER
  UNITS : ft
  EXPECTED_MIN : 0
  EXPECTED_MAX : 8024
END IMPORT

IMPORT CONSTANT Invalid_Alt_Failure_P5 : Time
  UNITS : NA
  EXPECTED_MIN : 2 s
  EXPECTED_MAX : 10 s
END IMPORT

IMPORT CONSTANT DOI_Timeout_P5 : Time
  UNITS : NA
  EXPECTED_MIN : 1 s
  EXPECTED_MAX : 5 s
END IMPORT

IMPORT CONSTANT GoAboveAction_P5 : ActionType
END IMPORT

IMPORT CONSTANT GoBelowAction_P5 : ActionType
END IMPORT

IMPORT CONSTANT GoAboveHyst_P5 : INTEGER
  UNITS : ft
  EXPECTED_MIN : 50
  EXPECTED_MAX : 500
END IMPORT

IMPORT CONSTANT GoBelowHyst_P5 : INTEGER
  UNITS : ft
  EXPECTED_MIN : 50
  EXPECTED_MAX : 500
END IMPORT

IMPORT DOI_MinDelay_P5 : TIME

  Purpose : &*L This parameter to the ASW operating module
  determines whether or not we will wait to turn the DOI on. If it
  is greater than zero, then we will wait.  It represents the
  minium waiting time L*&

END IMPORT

IMPORT DOI_MaxDelay_P5 : TIME
```

```
    Purpose : &*L This parameter to the ASW operating module
    determines the maximum waiting time that we will stay in a
    Delayed action state before giving up and returning to NoAction
    L*&

  END IMPORT

END INTERFACE

DEFINITION :

  EXPORT CON_DOI_P5 :
    PARENT : NONE
    DEFAULT_VALUE : Uncommanded

    TRANSITION Uncommanded TO On IF
      TABLE
         GoBelowAction = TurnOn             : T * ;
         ActionBelow_P5.PerformAction_P5    : T * ;
         GoAboveAction = TurnOn             : * T ;
         ActionAbove_P5.PerformAction_P5    : * T ;
      END TABLE

    TRANSITION Uncommanded TO Off IF
      TABLE
         GoBelowAction = TurnOff            : T * ;
         ActionBelow_P5.PerformAction_P5    : T * ;
         GoAboveAction = TurnOff            : * T ;
         ActionAbove_P5.PerformAction_P5    : * T ;
      END TABLE

    TRANSITION On TO Uncommanded IF WHEN(MON_DOI_P5 = On, False)

    TRANSITION Off TO Uncommanded IF WHEN(MON_DOI_P5 = Off, False)

  END EXPORT

  MODULE_INSTANCE ActionBelow_P5 : DOI_Action_P5
    PARENT : NONE
    ASSIGNMENT
      Direction_P5           := Down,
      ThresholdedAltitude_P5 := ThresholdedAlt_P5.Result_P5,
      MinDelay_P5            := DOI_MinDelay_P5,
      MaxDelay_P5            := DOI_MaxDelay_P5,
      AltitudeQuality_P5     := MON_AlitudeQuality_P5,
      ActionOK_P5            := DOI_Action_Ok_P5(),
      Clock                  := Clock
    END ASSIGNMENT
```

```
END MODULE_INSTANCE

MODULE_INSTANCE ActionAbove_P5 : DOI_Action_P5
  PARENT : NONE
  ASSIGNMENT
    Direction_P5           := Up,
    ThresholdedAltitude_P5 := ThresholdedAlt_P5.Result_P5,
    MinDelay_P5            := DOI_MinDelay_P5,
    MaxDelay_P5            := DOI_MaxDelay_P5,
    AltitudeQuality_P5     := MON_AlitudeQuality_P5,
    ActionOK_P5            := DOI_Action_Ok_P5(),
    Clock                  := Clock
  END ASSIGNMENT
END MODULE_INSTANCE

MACRO DOI_Action_Ok_P5(act IS ActionType) :
  TABLE
    MON_Inhibit_P5      : F F ;
    CON_Failure_P5      : F F ;
    MON_DOI_P5 = On     : T * ;
    act = On            : F * ;
    MON_DOI_P5 = Off    : * T ;
    act = Off           : * F ;
  END TABLE
END MACRO

EXPORT CON_Failure_P5 :
  PARENT : NONE
  DEFAULT_VALUE : False

  EQUALS TRUE IF
    TABLE
      DURATION(AttemptingOn(), 0 S, Clock) > DOI_Timeout_P5     : T * * * ;
      DURATION(AttemptingOff(), 0 S, Clock) > DOI_Timeout_P5    : * T * * ;
      DURATION(MON_Altitude_Quality_P5 = Invalid, 0 S, Clock)   : * * T * ;
      PRE(CON_Failure_P5) = False                               : * * * T ;
    END TABLE

  EQUALS FALSE IF
    TABLE
      DURATION(AttemptingOn(), 0 S, Clock) > DOI_Timeout_P5     : F ;
      DURATION(AttemptingOff(), 0 S, Clock) > DOI_Timeout_P5    : F ;
      DURATION(MON_Altitude_Quality_P5 = Invalid, 0 S, Clock)   : F ;
      PRE(CON_Failure_P5) = False                               : F ;
    END TABLE

END EXPORT
```

```
    MACRO AttemptingOn() :
      TABLE
        MON_DOI_P5 = Off    : T ;
        CON_DOI_P5 = On     : T ;
      END TABLE
    END MACRO

    MACRO AttemptingOff() :
      TABLE
        MON_DOI_P5 = On     : T ;
        CON_DOI_P5 = Off    : T ;
      END TABLE
    END MACRO

    MODULE_INSTANCE ThresholdedAlt_P5 : ThresholdedAltitude_P5
      PARENT : NONE
      ASSIGNMENT
        Altitude_P5 := MON_Altitude_P5,
        Threshold_P5 := Threshold_P5,
        BelowHysteresis_P5 := GoBelowHyst_P5,
        AboveHysteresis_P5 := GoBelowHyst_P5
      END ASSIGNMENT
    END MODULE_INSTANCE

  END DEFINITION

END MODULE


MODULE ThresholdedAltitude_P5 :

  INTERFACE :

    IMPORT Altitude_P5 : Integer
      UNITS : ft
      EXPECTED_MIN : 0
      EXPECTED_MAX : 50000
    END IMPORT

    IMPORT CONSTANT Threshold_P5 : Integer
      UNITS : ft
      EXPECTED_MIN : 0
      EXPECTED_MAX : 8024
    END IMPORT

    IMPORT CONSTANT AboveHysteresis_P5 : Integer
      UNITS : ft
      EXPECTED_MIN : 50
```

```
    EXPECTED_MAX : 500
  END IMPORT

  IMPORT CONSTANT BelowHysteresis_P5 : Integer
    UNITS : ft
    EXPECTED_MIN : 50
    EXPECTED_MAX : 500
  END IMPORT

  EXPORT Result_P5 : AboveBelowType

    Purpose : &*L this export reports whether or not the altitude is
    above or below the threshold given the hysteresis factor L*&

  END EXPORT

END INTERFACE

DEFINITION :

  EXPORT Result_P5 :
    PARENT : NONE

    DEFAULT_VALUE : Above IF
      TABLE
        DEFINED(Altitude_P5)        : T ;
        Altitude_P5 > Threshold_P5  : T ;
      END TABLE

    DEFAULT_VALUE : Below IF
      TABLE
        DEFINED(Altitude_P5)        : T ;
        Altitude_P5 <= Threshold_P5 : T ;
      END TABLE

    DEFAULT_VALUE : UNDEFINED IF NOT (DEFINED(Altitude_P5))

    EQUALS Above IF
      TABLE
        DEFINED(Altitude_P5)                : T ;
        Altitude_P5 > EffectiveThreshold_P5 : T ;
      END TABLE

    EQUALS Below IF
      TABLE
        DEFINED(Altitude_P5)                 : T ;
        Altitude_P5 <= EffectiveThreshold_P5 : T ;
      END TABLE
```

```
    EQUALS UNDEFINED IF NOT (DEFINED(Altitude_P5))

  END EXPORT

  STATE_VARIABLE ApplyHisteresis_P5 :
    VALUES : {NoHyst, Above, Below}
    PARENT : NONE

    DEFAULT_VALUE : NoHyst

    TRANSITION NoHyst TO Above IF
      TABLE
        DEFINED(Altitude_P5)                    : T ;
        WHEN(Altitude_P5 < Threshold_P5, False) : T ;
      END TABLE

    TRANSITION NoHyst TO Below IF
      TABLE
        DEFINED(Altitude_P5)                    : T ;
        WHEN(Altitude_P5 > Threshold_P5, False) : T ;
      END TABLE

    TRANSITION Above TO NoHyst IF
      TABLE
        DEFINED(Altitude_P5)                                          : T T ;
        WHEN(Altitude_P5 < Threshold_P5 + AboveHysteresis_P5, False) : T * ;
        WHEN(Altitude_P5 > Threshold_P5 - BelowHysteresis_P5, False) : * T ;

      END TABLE

    TRANSITION Below TO NoHyst IF
      TABLE
        DEFINED(Altitude_P5)                                          : T T ;
        WHEN(Altitude_P5 > Threshold_P5 + AboveHysteresis_P5, False) : T * ;
        WHEN(Altitude_P5 < Threshold_P5 - BelowHysteresis_P5, False) : * T ;

      END TABLE
  END STATE_VARIABLE

  STATE_VARIABLE EffectiveThreshold_P5 : INTEGER
    PARENT : NONE
    UNITS : ft
    EXPECTED_MIN : Threshold_P5 - BelowHysteresis_P5
    EXPECTED_MAX : Threshold_P5 + AboveHysteresis_P5

    DEFAULT_VALUE : Threshold_P5
```

```
      EQUALS Threshold_P5 + AboveHysteresis_P5
        IF ApplyHysteresis_P5 = Above

      EQUALS Threshold_P5 - BelowHysteresis_P5
        IF ApplyHysteresis_P5 = Below

      EQUALS Threshold_P5
        IF ApplyHysteresis_P5 = NoHyst

    END STATE_VARIABLE

  END DEFINITION

END MODULE

MODULE DOI_Action_P5 :

  INTERFACE :

    IMPORT MinDelay_P5 : TIME
    END IMPORT

    IMPORT MaxDelay_P5 : TIME
    END IMPORT

    IMPORT CONSTANT Direction_P5 : UpDownType
    END IMPORT

    IMPORT ThresholdedAltitude_P5 : AboveBelowType
    END IMPORT

    IMPORT AltitudeQuality_P5 : AltitudeQualityType
    END IMPORT

    IMPORT ActionOK_P5 : Boolean
    END IMPORT

    IMPORT Clock : TIME
    END IMPORT

    EXPORT PerformAction_P5 : Boolean
    END EXPORT

  END INTERFACE

  DEFINITION :

    EXPORT PerformAction_P5 :
```

```
  PARENT : NONE
  DEFAULT_VALUE : False
  EQUALS WHEN(_internal = Perform)
END EXPORT

STATE_VARIABLE internal_P5 :
  VALUES : {NoAction, Delayed, Perform}
  PARENT : NONE

  DEFAULT_VALUE : NoAction

  TRANSITION NoAction TO Delayed IF
    TABLE
      MinDelay_P5 > 0 S                    : T T ;
      ActionOK_P5                          : T T ;
      WHEN(ThresholdedAltitude_P5 = Below) : T * ;
      Direction_P5 = Below                 : T * ;
      WHEN(ThresholdedAltitude_P5 = Above) : * T ;
      Direction_P5 = Above                 : * T ;
    END TABLE

  TRANSITION NoAction TO Peform IF
    TABLE
      MinDelay_P5 > 0 S                    : F F ;
      ActionOK_P5                          : T T ;
      WHEN(ThresholdedAltitude_P5 = Below) : T * ;
      Direction_P5 = Down                  : T * ;
      WHEN(ThresholdedAltitude_P5 = Above) : * T ;
      Direction_P5 = Up                    : * T ;
    END TABLE

  TRANSITION Delayed TO Perform IF
    TABLE
      DURATION(PRE(internal_P5) IN_STATE Delayed, 0 S, Clock) >= MinDelay_P5 : T T ;
      ActionOK_P5                                                           : T T ;
      AltitudeQuality_P5 = Valid                                            : T T ;
      Direction_P5 = Down                                                   : T * ;
      ThresholdedAltitude_P5 = Below                                        : T * ;
      Direction_P5 = Up                                                     : * T ;
      ThresholdedAltitude_P5 = Above                                        : * T ;
    END TABLE

  TRANSITION Delayed TO NoAction IF
    DURATION(PRE(internal_P5) IN_STATE Delayed, 0 S, Clock) >= MaxDelay_P5

  TRANSITION Perform TO NoAction IF
    DURATION(PRE(internal_P5) IN_STATE Perform, 0 S, Clock) >= 0 S
```

```
    END STATE_VARIABLE

  END DEFINITION

END MODULE

INCLUDE "standard-modules.nimbus"
```

Appendix F

The Altitude Switch in RSML^{-e}-Phase 6

```
/*L

In this chapter, we add to the REQ specification for the ASW a
specification of the ASW's IN' and OUT' relations.  These relations
are developed in a similar way to the REQ relation, but starting out
at a high level and then refining the structure and computation,
finally taking into consideration completeness and error handling
constraints.

For this Phase, we will be defining a number of new modules.  The
Altimeters\_IN\_P6 module will transform the inputs from the digital altimeters

L*/

INCLUDE "asw-alltypes.nimbus"

MODULE Altimeters_IN_P6 :

  INTERFACE :

    IMPORT CONSTANT NumDigitalAlt_P6 : INTEGER
      UNITS : NA
      EXPECTED_MIN : 0
      EXPECTED_MAX : 10
    END IMPORT

    IMPORT CONSTANT NumAnalogAlt_P6 : INTEGER
      UNITS : NA
      EXPECTED_MIN : 0
      EXPECTED_MAX : 10
```

```
END IMPORT

IMPORT DigialAlt_P6 : [1 TO NumDigitalAlt] OF INTEGER
  UNITS : ft
  EXPECTED_MIN : 0
  EXPECTED_MAX : 50000
END IMPORT

IMPORT CONSTANT Threshold_P6 : INTEGER
END IMPORT

IMPORT CONSTANT GoAboveHyst_P6 : INTEGER
  UNITS : ft
  EXPECTED_MIN : 50
  EXPECTED_MAX : 500

  Purpose : &*L This defines the hysteresis factor for going above
  the threshold altitude. L*&

END IMPORT

IMPORT CONSTANT GoBelowHyst_P6 : INTEGER
  UNITS : ft
  EXPECTED_MIN : 50
  EXPECTED_MAX : 500

  Purpose : &*L This defines the hysteresis factor for going above
  the threshold altitude. L*&

END IMPORT

IMPORT AnalogAlt_P6 : [1 TO NumAnalogAlt] OF AboveBelowType
END IMPORT

IMPORT DigitalQuality_P6 : [1 TO NumDigitalAlt] OF AltitudeQualityType
END IMPORT

IMPORT AnalogQuality_P6 : [1 TO NumAnalogAlt] OF AltitudeQualityType
END IMPORT

IMPORT INTERFACE AltitudeVoter_P6 :
END IMPORT

EXPORT Altitude_P6 : AboveBelowType
END EXPORT

EXPORT AltitudeQuality_P6 : AltitudeQualityType
END EXPORT
```

```
  END INTERFACE

  DEFINITION :

    MODULE_INSTANCE ThresholdedDigital_P6 : [1 TO NumDigitalAlt] OF ThresholdedAltitude_P6
      PARENT : NONE
      ASSIGNMENT
        Altitude_P6        := DigitalAlt_P6,
        Threshold_P6       := EXTEND Threshold_P6 TO [ 1 TO NumDigitalAlt] OF INTEGER,
        AboveHysteresis_P6 := EXTEND GoAboveHyst_P6 TO [ 1 TO NumDigitalAlt] OF INTEGER,
        BelowHysteresis_P6 := EXTEND GoBelowHyst_P6 TO [1 TO NumDigitalAlt] OF INTEGER
      END ASSIGNMENT
    END MODULE_INSTANCE

    SLOT_INSTANCE AltitudeVoter_P6 :
      ASSIGNMENT
        Num_of_Alt   := NumDigitalAlt_P6 + NumAnalogAlt_P6,
        Altitudes    := ThresholdedDigital_P6.Result_P6 | AnalogAlt_P6,
        Qualities    := DigitialQuality_P6 | AnalogQuality_P6
      END ASSIGNMENT
    END SLOT_INSTANCE

    EXPORT Altitude_P6 :
      PARENT : NONE
      DEFAULT_VALUE : AltitudeVoter_P6.Altitude_P6
      EQUALS AltitudeVoter_P6.Altitude_P6
    END EXPORT

    EXPORT AltitudeQuality_P6 :
      PARENT : NONE
      DEFAULT_VALUE : AltitudeVoter_P6.AltitudeQuality_P6
      EQUALS AltitudeVoter_P6.AltitudeQuality_P6
    END EXPORT

  END DEFINITION

END MODULE


INTERFACE AltitudeVoter_P6 :

  IMPORT CONSTANT Num_of_Alt_P6 : INTEGER
    UNITS : NA
    EXPECTED_MIN : 0
    EXPECTED_MAX : 50
  END IMPORT
```

```
  IMPORT Altitudes_P6 : [1 TO Num_of_Alt_P6] OF AboveBelowType
  END IMPORT

  IMPORT Qualities_P6 : [1 TO Num_of_Alt_P6] OF AltitudeQualityType
  END IMPORT

  EXPORT Altitude_P6 : AboveBelowType
  END EXPORT

  EXPORT Quality_P6 : AltitudeQualityType
  END EXPORT

END INTERFACE

MODULE Alt_and_Quality_P6 :

  INTERFACE :

    IMPORT Altitude_P6 : AboveBelowType
    END IMPORT

    IMPORT Quality_P6 : AltitudeQualityType
    END IMPORT

    EXPORT Result : Alt_and_QualityType
    END EXPORT

  END INTERFACE

  DEFINITION :

    EXPORT Alt_and_QualityType :
      PARENT : NONE

      EQUALS Above IF
        TABLE
          Altitude_P6 = Above : T ;
          Quality_P6 = Valid  : T ;
        END TABLE

      EQUALS Below IF
        TABLE
          Altitude_P6 = Below : T ;
          Quality_P6 = Valid  : T ;
        END TABLE

      EQUALS Invaid IF Quality_P6 = Invalid
```

```
    END EXPORT

  END DEFINITION

END MODULE

MODULE Most_P6 : AltitudeVoter_P6

  DEFINITION :

    EXPORT Altitude_P6 :
      PARENT : NONE

      DEFAULT_VALUE : Below

      EQUALS Below IF
        COUNT(EXTEND Alt_and_Quality_P6(Altitudes_P6, Qualities_P6) TO [1 TO Num_of_Alt_P6] OF Altit
              EXTEND Below TO [1 TO Num_of_Alt_P6] OF AltitudeQualityType, Num_of_Alt_P6) >
        COUNT(EXTEND Alt_and_Quality_P6(Altitudes_P6, Qualities_P6) TO [1 TO Num_of_Alt_P6] OF Altit
              EXTEND Above TO [1 TO Num_of_Alt_P6] OF AltitudeQualityType, Num_of_Alt_P6)

      EQUALS Above IF
        COUNT(EXTEND Alt_and_Quality_P6(Altitudes_P6, Qualities_P6) TO [1 TO Num_of_Alt_P6] OF Altit
              EXTEND Below TO [1 TO Num_of_Alt_P6] OF AltitudeQualityType, Num_of_Alt_P6) <=
        COUNT(EXTEND Alt_and_Quality_P6(Altitudes_P6, Qualities_P6) TO [1 TO Num_of_Alt_P6] OF Altit
              EXTEND Above TO [1 TO Num_of_Alt_P6] OF AltitudeQualityType, Num_of_Alt_P6)

    END EXPORT

    EXPORT Quality_P6 :
      PARENT : NONE
      DEFAULT_VALUE : Valid

      EQUALS Valid IF
       EXISTS(Qualities_P6 = EXTEND Valid TO [1 TO Num_of_Alt_P6] OF AltitudeQualityType, Num_of_Alt

      EQUALS Invalid IF
       FORALL(Qualities_P6 = EXTEND Invalid TO [1 TO Num_of_Alt_P6] OF AltitudeQualityType, Num_of_A


    END EXPORT

  END DEFINITION

END MODULE

MODULE AnyCrossed_P6 : AltitudeVoter_P6
```

```
  DEFINITION :

    EXPORT Altitude_P6 :
      PARENT : NONE

      DEFAULT_VALUE : Below

      TRANSITION Below TO Above IF
          EXISTS(EXTEND Alt_and_Quality_P6(Altitudes_P6, Qualities_P6) TO [1 TO Num_of_Alt_P6] OF Al
              EXTEND Above TO [1 TO Num_of_Alt_P6] OF AltitudeQualityType, Num_of_Alt_P6)

      TRANSITION Above TO Below IF
          EXISTS(EXTEND Alt_and_Quality_P6(Altitudes_P6, Qualities_P6) TO [1 TO Num_of_Alt_P6] OF Al
              EXTEND Below TO [1 TO Num_of_Alt_P6] OF AltitudeQualityType, Num_of_Alt_P6)

    END EXPORT

    EXPORT Quality_P6 :
      PARENT : NONE

      EQUALS Valid IF
       EXISTS(Qualities_P6 = EXTEND Valid TO [1 TO Num_of_Alt] OF AltitudeQualityType, Num_of_Alt)

      EQUALS Invalid IF
       FORALL(Qualities_P6 = EXTEND Invalid TO [1 TO Num_of_Alt] OF AltitudeQualityType, Num_of_Alt)

    END EXPORT

  END DEFINITION

END MODULE


MODULE AllCrossed_P6 : AltitideVoter_P6

  DEFINITION :

    EXPORT Altitude_P6 :
      PARENT : NONE

      DEFAULT_VALUE : Below

      TRANSITION Below TO Above IF
          FORALL(EXTEND Alt_and_Quality_P6(Altitudes_P6, Qualities_P6) TO [1 TO Num_of_Alt_P6] OF Al
              EXTEND Above TO [1 TO Num_of_Alt_P6] OF AltitudeQualityType, Num_of_Alt_P6)

      TRANSITION Above TO Below IF
          FORALL(EXTEND Alt_and_Quality_P6(Altitudes_P6, Qualities_P6) TO [1 TO Num_of_Alt_P6] OF Al
```

```
            EXTEND Below TO [1 TO Num_of_Alt_P6] OF AltitudeQualityType, Num_of_Alt_P6)

    END EXPORT

    EXPORT Quality_P6 :
      PARENT : NONE

      EQUALS Valid IF
       EXISTS(Qualities_P6 = EXTEND Valid TO [1 TO Num_of_Alt] OF AltitudeQualityType, Num_of_Alt)

      EQUALS Invalid IF
       FORALL(Qualities_P6 = EXTEND Invalid TO [1 TO Num_of_Alt] OF AltitudeQualityType, Num_of_Alt)

    END EXPORT

  END DEFINITION

END MODULE


MODULE Failure_OUT_P6 :

  INTERFACE :

     IMPORT Failure_P6 : Boolean
     END IMPORT

     IMPORT PulseInterval_P6 : TIME
     END IMPORT

     IMPORT Clock : TIME
     END IMPORT

     EXPORT Watchdog_Pulse_P6 : Boolean
     END EXPORT

  END INTERFACE

  DEFINITION :

    EXPORT Watchdog_Pulse_P6 :
      PARENT : NONE

      DEFAULT_VALUE : false

      TRANSITION False TO True IF
        TABLE
          DURATION(PRE(Watchdog_Pulse_P6) IN_STATE False, 0 S, Clock) >= PulseInterval_P6 : T ;
```

```
        Failure_P6                                                    : F ;
      END TABLE

    TRANSITION True TO False IF
      DURATION(PRE(Watchdog_Pulse_P6) IN_STATE True, 0 S, Clock) >= 0 S

  END EXPORT

END DEFINITION

END MODULE



MODULE ASW_REQ_P6 :

  INTERFACE :

    EXPORT CON_DOI_P6 : DOIControlledType
    Purpose : &*L  This variable represents the ASW's
    commanded status of the Device of Interest (DOI). L*&

    Interpretation : &*L
      \begin{quote}
      \begin{mydescription}
      \item[On:] Indicates that the DOI is commanded to be On.  The DOI
      is commanded to be on when the aircraft enters the target region
      for turning the DOI on, the DOI is not already on,
      and the ASW is not inhibitied.
      \item[Off:] Indicates that the DOI is commanded to be Off.  The
      DOI is commanded to be off when the aircraft leaves the target
      region and after a certain period of time has passed.  If this
      time is \RUndefined, then the ASW will never turn the DOI Off.
      \item[Uncommanded:] Indicates that the DOI is not commanded by the
      ASW.  This CON\_DOI variable will be equal to Uncommanded in any
      step were the ASW does not issue a command to the device of interest.
      \end{mydescription}
      \end{quote}
      L*&

    Issues : &*L
      \begin{myitemize}
      \item If the aircraft leaves the target area and the DOI is on,
      but was {\em not} commanded to be on by the ASW, should the ASW
      turn it off?
      \end{myitemize}
```

```
  L*&

END EXPORT

EXPORT CON_Failure_P6 : Boolean
 Purpose : &*L This variable represents the ASW's indication of
 whether or not it has failed to the external world.  It is
 potentially displayed to the pilot and/or used by other subsystems
 on board the aircraft. L*&

 Interpretation : &*L
  \begin{quote}
  \begin{mydescription}
  \item[True:] Indicates that the ASW has failed.  The ASW is
  considered to be failed if it attempts to turn on the DOI, but the
  DOI does not turn on after a certain timeout period.
  \item[False:] Indicates that the ASW has not failed.  The ASW is
  considered to be operating normally if none of the failure
  conditions are true.
  \end{mydescription}
  \end{quote}
  L*&
END EXPORT

IMPORT MON_Altitude_P6 : AboveBelowType
  CLASSIFICATION : Monitored

  Purpose : &*L This variable represents the ASW's idea of what the
  altitude of the aircraft is.  It is related to the Altitude\_Quality
  variable. L*&
END IMPORT

IMPORT MON_Altitude_Quality_P6 : AltitudeQualityType
  CLASSIFICATION : Monitored

  Purpose : &*L This variable represents the quality of the
  Altitude of the aircraft is. L*&
END IMPORT

IMPORT MON_DOI_P6 : OnOffType_P6
  Purpose : &*L This variable indicates the monitored status of the
  DOI.  The DOI can be turned on or off by other devices/systems on
  board the aircraft, so the ASW needs an accurate accounting of the
  status of the DOI L*&

  Interpretation : &*L
    \begin{quote}
    \begin{mydescription}
```

```
    \item[On:] Indicates that the DOI is currently on.
    \item[Off:] Indicates that the DOI is currently off.
    \end{mydescription}
    \end{quote}
    L*&

 END IMPORT

IMPORT MON_Reset_P6 : Boolean

  Purpose : &*L This variable indicates the whether the ASW should be
  reset or not.  In a step where the ASW is reset, this variable will
  have the value true.  In all others, this variable will have the
  value false. L*&

  Interpretation : &*L
    \begin{quote}
    \begin{mydescription}
    \item[True:] Indicates that the ASW as been reset.
    \item[False:] Indicates that the ASW has not been reset.
    \end{mydescription}
    \end{quote}
    L*&

END IMPORT

IMPORT MON_Inhibit_P6 : Boolean

   Purpose : &*L This variable is true when the ASW is inhibited and
   false otherwise.  The value is determined by the user and/or other
   systems on board the aircraft. L*&

  Interpretation : &*L
    \begin{quote}
    \begin{mydescription}
    \item[True:] Indicates that the operation of the ASW has been
    inhibited; the ASW shall not attempt to change the status of the
    DOI.
    \item[False:] Indicates that the ASW has not been inhibited; the
    ASW will behave as specified by other requirements.
    \end{mydescription}
    \end{quote}
    L*&

END IMPORT

IMPORT CONSTANT Threshold_P6 : INTEGER
  UNITS : ft
```

```
  EXPECTED_MIN : 0
  EXPECTED_MAX : 8024

  Purpose : &*L This constant will be defined by each family
  member when the REQ module is instantiated.  It is the altitude
  at which the ASW is required to turn on or off the ASW. L*&

END IMPORT

IMPORT CONSTANT Invalid_Alt_Failure_P6 : Time
  UNITS : NA
  EXPECTED_MIN : 2 s
  EXPECTED_MAX : 10 s

  Purpose : &*L This constant will be defined by each family
  member.  It is the length of time after which the ASW will
  declare a failure if there is not valid altitude. L*&

END IMPORT

IMPORT CONSTANT DOI_Timeout_P6 : Time
  UNITS : NA
  EXPECTED_MIN : 1 s
  EXPECTED_MAX : 5 s

  Purpose : &*L This constant will be defined by each member of
  the ASW family to represent the amount of time before the ASW
  declares a failure if the DOI does not respond to a command. L*&

END IMPORT

IMPORT CONSTANT GoAboveAction_P6 : ActionType

  Purpose : &*L This constant specifies the action that the ASW
  will perform when it crosses the Threshold going up. It is
  specified by the decision model for each family member. L*&

END IMPORT

IMPORT CONSTANT GoBelowAction_P6 : ActionType

  Purpose : &*L This constant specifies the action that the ASW
  will perform when it crosses the Threshold going down. It is
  specified by the decision model for each family member. L*&

END IMPORT

IMPORT CONSTANT GoAboveHyst_P6 : INTEGER
```

```
    UNITS : ft
    EXPECTED_MIN : 50
    EXPECTED_MAX : 500

    Purpose : &*L This defines the hysteresis factor for going above
    the threshold altitude. L*&

  END IMPORT

  IMPORT CONSTANT GoBelowHyst_P6 : INTEGER
    UNITS : ft
    EXPECTED_MIN : 50
    EXPECTED_MAX : 500

    Purpose : &*L This defines the hysteresis factor for going above
    the threshold altitude. L*&

  END IMPORT

END INTERFACE

DEFINITION :

  STATE_VARIABLE ASW_System_Mode_P6 :
    VALUES : {Startup, NormalOperating, Degraded, Failed, Reset}
    PARENT : NONE

    Purpose : &*L This is the top-level mode of the ASW.  If the ASW
    were to have a startup mode, etc., we could put those modes as
    children of this controlling mode.  Currently, we have only two
    states, the reset mode which is used for when the reset signal
    is received and the operating mode that handles the main
    behavior. L*&

    DEFAULT_VALUE : Startup

    TRANSITION NormalOperating TO Reset IF MON_Reset_P6

    TRANSITION Degraded TO Reset IF MON_Reset_P6

    TRANSITION NormalOperating TO Degraded IF
      EpisodeMonitor_P6 = QualifyingEpisode

    TRANSITION Degraded TO NormalOperating IF
      DURATION (MON_Altitude_Quality_P6 = Valid, 0 S, Clock) > 1 MIN

    TRANSITION Reset TO NormalOperating IF
      DURATION(PRE(ASW_System_Mode_P6), 0 s, Clock) >= 0 S
```

```
END STATE_VARIABLE

STATE_VARIABLE EpisodeMonitor_P6 :
  VALUES : {NoEpisode, FirstEpisode, QualifyingEpisode}
  PARENT : NONE

  Purpose : &*L This simple state variable tracks whether or not
  we have met the conditions for being in degraded functionality
  mode.  Namely, whether or not we have seen two periods of
  invalid altitude lasting 1 second or more within 1 minute. L*&

   DEFAULT_VALUE : NoEpisode

   TRANSITION NoEpisode TO FirstEpisode IF
     DURATION(MON_Altitude_Quality_P6 = Invalid, 0 S, Clock) > 1 S

   TRANSITION FirstEpisode TO QualifyingEpisode IF
     TABLE
       DURATION(MON_Altitude_Quality_P6 = Invalid, 0 S, Clock) > 1 S : T ;
       DURATION(PRE(EpisodeMonitor_P6) = FirstEpisode) > 1 S         : T ;
     END TABLE

   TRANSITION FirstEpisode TO NoEpisode IF
     DURATION(PRE(EpisodeMonitor_P6) = FirstEpisode) >= 1 MIN

   TRANSITION QualifyingEpisode TO NoEpisode IF
     DURATION(MON_Altitude_Quality_P6 = Valid, 0 S, Clock) >= 2 MIN

END STATE_VARIABLE

MODULE_INSTANCE ASW_Operating_Mode_P6 : ASW_Operating_Mode_Def_P6
  PARENT :  ASW_System_Mode_P6.NormalOperating
  ASSIGNMENT
     MON_Altitude_P6          := MON_Altitude_P6,
     MON_Altitude_Quality_P6  := MON_Altidue_Quality_P6,
     MON_DOI_P6               := MON_DOI_P6,
     MON_Inhibit_P6           := MON_Inhibit_P6,
     Threshold_P6             := Threshold_P6,
     Invalid_Alt_Failure_P6   := Invalid_Alt_Failure_P6,
     DOI_Timeout_P6           := DOI_Timeout_P6,
     GoAboveAction_P6         := GoAboveAction_P6,
     GoBelowAction_P6         := GoBelowAction_P6,
     GoAboveHyst_P6           := GoAboveHyst_P6,
     GoBelowHyst_P6           := GoBelowHyst_P6,
     DOI_Delay_P6             := 0 S
  END ASSIGNMENT
END MODULE_INSTANCE
```

```
MODULE_INSTANCE ASW_Degraded_Mode_P6 : ASW_Operating_Mode_Def_P6
  PARENT :  ASW_System_Mode_P6.Degraded
  ASSIGNMENT
     MON_Altitude_P6          := MON_Altitude_P6,
     MON_Altitude_Quality_P6  := MON_Altidue_Quality_P6,
     MON_DOI_P6               := MON_DOI_P6,
     MON_Inhibit_P6           := MON_Inhibit_P6,
     Threshold_P6             := Threshold_P6,
     Invalid_Alt_Failure_P6   := Invalid_Alt_Failure_P6,
     DOI_Timeout_P6           := DOI_Timeout_P6,
     GoAboveAction_P6         := GoAboveAction_P6,
     GoBelowAction_P6         := GoBelowAction_P6,
     GoAboveHyst_P6           := GoAboveHyst_P6,
     GoBelowHyst_P6           := GoBelowHyst_P6,
     DOI_MinDelay_P6          := 2 S,
     DOI_MaxDelay_P6          := 6 S
  END ASSIGNMENT
END MODULE_INSTANCE

EXPORT CON_DOI_P6 :
  PARENT :  NONE
  DEFAULT_VALUE : Uncontrolled

  EQUALS ASW_Operating_Mode_P6.CON_DOI_P6
    IF ASW_System_Mode_P6 = NormalOperating

  EQUALS ASW_Degraded_Mode_P6.CON_DOI_P6
    IF ASW_System_Mode_P6 = Degraded

  EQUALS Uncontrolled IF
    TABLE
      ASW_System_Mode_P6 = Failed : T * ;
      ASW_System_Mode_P6 = Reset  : * T ;
    END TABLE

END EXPORT

EXPORT CON_Failure_P6 :
  PARENT :  NONE
  DEFAULT_VALUE : False

  TRANSITION False TO True IF
    TABLE
      ASW_System_Mode_P6 = NormalOperating   : T * ;
      ASW_Operating_Mode_P6.CON_Failure_P6   : T * ;
      ASW_System_Mode_P6 = Degraded          : * T ;
      ASW_Operating_Mode_P6.CON_Failure_P6   : * T ;
```

```
        END TABLE

      TRANSITION True TO False IF ASW_System_Mode_P6 = Reset

    END EXPORT

  END DEFINITION

END MODULE


MODULE ASW_OperatingMode_Def_P6 :

  INTERFACE :

    EXPORT CON_DOI_P6 : DOIControlledType
    END EXPORT

    EXPORT CON_Failure_P6 : Boolean
    END EXPORT

    IMPORT MON_Altitude_P6 : AboveBelowType
    END IMPORT

    IMPORT MON_Altitude_Quality_P6 : AltitudeQualityType
    END IMPORT

    IMPORT MON_DOI_P6 : OnOffType_P6
    END IMPORT

    IMPORT MON_Inhibit_P6 : Boolean
    END IMPORT

    IMPORT CONSTANT Threshold_P6 : INTEGER
      UNITS : ft
      EXPECTED_MIN : 0
      EXPECTED_MAX : 8024
    END IMPORT

    IMPORT CONSTANT Invalid_Alt_Failure_P6 : Time
      UNITS : NA
      EXPECTED_MIN : 2 s
      EXPECTED_MAX : 10 s
    END IMPORT

    IMPORT CONSTANT DOI_Timeout_P6 : Time
      UNITS : NA
      EXPECTED_MIN : 1 s
```

```
    EXPECTED_MAX : 5 s
  END IMPORT

  IMPORT CONSTANT GoAboveAction_P6 : ActionType
  END IMPORT

  IMPORT CONSTANT GoBelowAction_P6 : ActionType
  END IMPORT

  IMPORT CONSTANT GoAboveHyst_P6 : INTEGER
    UNITS : ft
    EXPECTED_MIN : 50
    EXPECTED_MAX : 500
  END IMPORT

  IMPORT CONSTANT GoBelowHyst_P6 : INTEGER
    UNITS : ft
    EXPECTED_MIN : 50
    EXPECTED_MAX : 500
  END IMPORT

  IMPORT DOI_MinDelay_P6 : TIME

    Purpose : &*L This parameter to the ASW operating module
    determines whether or not we will wait to turn the DOI on. If it
    is greater than zero, then we will wait.  It represents the
    minium waiting time L*&

  END IMPORT

  IMPORT DOI_MaxDelay_P6 : TIME

    Purpose : &*L This parameter to the ASW operating module
    determines the maximum waiting time that we will stay in a
    Delayed action state before giving up and returning to NoAction
    L*&

  END IMPORT

END INTERFACE

DEFINITION :

  EXPORT CON_DOI_P6 :
    PARENT : NONE
    DEFAULT_VALUE : Uncommanded

    TRANSITION Uncommanded TO On IF
```

```
    TABLE
       GoBelowAction = TurnOn              : T * ;
       ActionBelow_P6.PerformAction_P6     : T * ;
       GoAboveAction = TurnOn              : * T ;
       ActionAbove_P6.PerformAction_P6     : * T ;
    END TABLE

  TRANSITION Uncommanded TO Off IF
    TABLE
       GoBelowAction = TurnOff             : T * ;
       ActionBelow_P6.PerformAction_P6     : T * ;
       GoAboveAction = TurnOff             : * T ;
       ActionAbove_P6.PerformAction_P6     : * T ;
    END TABLE

  TRANSITION On TO Uncommanded IF WHEN(MON_DOI_P6 = On, False)

  TRANSITION Off TO Uncommanded IF WHEN(MON_DOI_P6 = Off, False)

END EXPORT

MODULE_INSTANCE ActionBelow_P6 : DOI_Action_P6
  PARENT : NONE
  ASSIGNMENT
    Direction_P6           := Down,
    ThresholdedAltitude_P6 := MON_Altitude_P6,
    MinDelay_P6            := DOI_MinDelay_P6,
    MaxDelay_P6            := DOI_MaxDelay_P6,
    AltitudeQuality_P6     := MON_AlitudeQuality_P6,
    ActionOK_P6            := DOI_Action_Ok_P6(),
    Clock                  := Clock
  END ASSIGNMENT
END MODULE_INSTANCE

MODULE_INSTANCE ActionAbove_P6 : DOI_Action_P6
  PARENT : NONE
  ASSIGNMENT
    Direction_P6           := Up,
    ThresholdedAltitude_P6 := MON_Altitude_P6,
    MinDelay_P6            := DOI_MinDelay_P6,
    MaxDelay_P6            := DOI_MaxDelay_P6,
    AltitudeQuality_P6     := MON_AlitudeQuality_P6,
    ActionOK_P6            := DOI_Action_Ok_P6(),
    Clock                  := Clock
  END ASSIGNMENT
END MODULE_INSTANCE

MACRO DOI_Action_Ok_P6(act IS ActionType) :
```

```
    TABLE
      MON_Inhibit_P6       : F F ;
      CON_Failure_P6       : F F ;
      MON_DOI_P6 = On      : T * ;
      act = On             : F * ;
      MON_DOI_P6 = Off     : * T ;
      act = Off            : * F ;
    END TABLE
  END MACRO

  EXPORT CON_Failure_P6 :
    PARENT : NONE
    DEFAULT_VALUE : False

    EQUALS TRUE IF
      TABLE
        DURATION(AttemptingOn(), 0 S, Clock) > DOI_Timeout_P6     : T * * * ;
        DURATION(AttemptingOff(), 0 S, Clock) > DOI_Timeout_P6    : * T * * ;
        DURATION(MON_Altitude_Quality_P6 = Invalid, 0 S, Clock)   : * * T * ;
        PRE(CON_Failure_P6) = False                               : * * * T ;
      END TABLE

    EQUALS FALSE IF
      TABLE
        DURATION(AttemptingOn(), 0 S, Clock) > DOI_Timeout_P6     : F ;
        DURATION(AttemptingOff(), 0 S, Clock) > DOI_Timeout_P6    : F ;
        DURATION(MON_Altitude_Quality_P6 = Invalid, 0 S, Clock)   : F ;
        PRE(CON_Failure_P6) = False                               : F ;
      END TABLE

  END EXPORT

  MACRO AttemptingOn() :
    TABLE
      MON_DOI_P6 = Off    : T ;
      CON_DOI_P6 = On     : T ;
    END TABLE
  END MACRO

  MACRO AttemptingOff() :
    TABLE
      MON_DOI_P6 = On     : T ;
      CON_DOI_P6 = Off    : T ;
    END TABLE
  END MACRO

END DEFINITION
```

```
END MODULE


MODULE ThresholdedAltitude_P6 :

  INTERFACE :

    IMPORT Altitude_P6 : Integer
      UNITS : ft
      EXPECTED_MIN : 0
      EXPECTED_MAX : 50000
    END IMPORT

    IMPORT CONSTANT Threshold_P6 : Integer
      UNITS : ft
      EXPECTED_MIN : 0
      EXPECTED_MAX : 8024
    END IMPORT

    IMPORT CONSTANT AboveHysteresis_P6 : Integer
      UNITS : ft
      EXPECTED_MIN : 50
      EXPECTED_MAX : 500
    END IMPORT

    IMPORT CONSTANT BelowHysteresis_P6 : Integer
      UNITS : ft
      EXPECTED_MIN : 50
      EXPECTED_MAX : 500
    END IMPORT

    EXPORT Result_P6 : AboveBelowType

      Purpose : &*L this export reports whether or not the altitude is
      above or below the threshold given the hysteresis factor L*&

    END EXPORT

  END INTERFACE

  DEFINITION :

    EXPORT Result_P6 :
      PARENT : NONE

      DEFAULT_VALUE : Above IF
        TABLE
          DEFINED(Altitude_P6)        : T ;
```

```
      Altitude_P6 > Threshold_P6  : T ;
    END TABLE

  DEFAULT_VALUE : Below IF
    TABLE
      DEFINED(Altitude_P6)        : T ;
      Altitude_P6 <= Threshold_P6 : T ;
    END TABLE

  DEFAULT_VALUE : UNDEFINED IF NOT (DEFINED(Altitude_P6))

  EQUALS Above IF
    TABLE
      DEFINED(Altitude_P6)                : T ;
      Altitude_P6 > EffectiveThreshold_P6 : T ;
    END TABLE

  EQUALS Below IF
    TABLE
      DEFINED(Altitude_P6)                 : T ;
      Altitude_P6 <= EffectiveThreshold_P6 : T ;
    END TABLE

  EQUALS UNDEFINED IF NOT (DEFINED(Altitude_P6))

END EXPORT

STATE_VARIABLE ApplyHisteresis_P6 :
  VALUES : {NoHyst, Above, Below}
  PARENT : NONE

  DEFAULT_VALUE : NoHyst

  TRANSITION NoHyst TO Above IF
    TABLE
      DEFINED(Altitude_P6)                    : T ;
      WHEN(Altitude_P6 < Threshold_P6, False) : T ;
    END TABLE

  TRANSITION NoHyst TO Below IF
    TABLE
      DEFINED(Altitude_P6)                    : T ;
      WHEN(Altitude_P6 > Threshold_P6, False) : T ;
    END TABLE

  TRANSITION Above TO NoHyst IF
    TABLE
      DEFINED(Altitude_P6)                                    : T T ;
```

```
          WHEN(Altitude_P6 < Threshold_P6 + AboveHysteresis_P6, False) : T * ;
          WHEN(Altitude_P6 > Threshold_P6 - BelowHysteresis_P6, False) : * T ;

        END TABLE

      TRANSITION Below TO NoHyst IF
        TABLE
          DEFINED(Altitude_P6)                                          : T T ;
          WHEN(Altitude_P6 > Threshold_P6 + AboveHysteresis_P6, False) : T * ;
          WHEN(Altitude_P6 < Threshold_P6 - BelowHysteresis_P6, False) : * T ;

        END TABLE
    END STATE_VARIABLE

    STATE_VARIABLE EffectiveThreshold_P6 : INTEGER
      PARENT : NONE
      UNITS : ft
      EXPECTED_MIN : Threshold_P6 - BelowHysteresis_P6
      EXPECTED_MAX : Threshold_P6 + AboveHysteresis_P6

      DEFAULT_VALUE : Threshold_P6

      EQUALS Threshold_P6 + AboveHysteresis_P6
        IF ApplyHysteresis_P6 = Above

      EQUALS Threshold_P6 - BelowHysteresis_P6
        IF ApplyHysteresis_P6 = Below

      EQUALS Threshold_P6
        IF ApplyHysteresis_P6 = NoHyst

    END STATE_VARIABLE

  END DEFINITION

END MODULE

MODULE DOI_Action_P6 :

  INTERFACE :

    IMPORT MinDelay_P6 : TIME
    END IMPORT

    IMPORT MaxDelay_P6 : TIME
    END IMPORT

    IMPORT CONSTANT Direction_P6 : UpDownType
```

```
  END IMPORT

  IMPORT ThresholdedAltitude_P6 : AboveBelowType
  END IMPORT

  IMPORT AltitudeQuality_P6 : AltitudeQualityType
  END IMPORT

  IMPORT ActionOK_P6 : Boolean
  END IMPORT

  IMPORT Clock : TIME
  END IMPORT

  EXPORT PerformAction_P6 : Boolean
  END EXPORT

END INTERFACE

DEFINITION :

  EXPORT PerformAction_P6 :
    PARENT : NONE
    DEFAULT_VALUE : False
    EQUALS WHEN(_internal = Perform)
  END EXPORT

  STATE_VARIABLE internal_P6 :
    VALUES : {NoAction, Delayed, Perform}
    PARENT : NONE

    DEFAULT_VALUE : NoAction

    TRANSITION NoAction TO Delayed IF
      TABLE
        MinDelay_P6 > 0 S                      : T T ;
        ActionOK_P6                            : T T ;
        WHEN(ThresholdedAltitude_P6 = Below)   : T * ;
        Direction_P6 = Below                   : T * ;
        WHEN(ThresholdedAltitude_P6 = Above)   : * T ;
        Direction_P6 = Above                   : * T ;
      END TABLE

    TRANSITION NoAction TO Peform IF
      TABLE
        MinDelay_P6 > 0 S                      : F F ;
        ActionOK_P6                            : T T ;
        WHEN(ThresholdedAltitude_P6 = Below)   : T * ;
```

```
        Direction_P6 = Down                    : T * ;
        WHEN(ThresholdedAltitude_P6 = Above)   : * T ;
        Direction_P6 = Up                      : * T ;
      END TABLE

    TRANSITION Delayed TO Perform IF
      TABLE
        DURATION(PRE(internal_P6) IN_STATE Delayed, 0 S, Clock) >= MinDelay_P6 : T T ;
        ActionOK_P6                                                            : T T ;
        AltitudeQuality_P6 = Valid                                             : T T ;
        Direction_P6 = Down                                                    : T * ;
        ThresholdedAltitude_P6 = Below                                         : T * ;
        Direction_P6 = Up                                                      : * T ;
        ThresholdedAltitude_P6 = Above                                         : * T ;
      END TABLE

    TRANSITION Delayed TO NoAction IF
      DURATION(PRE(internal_P6) IN_STATE Delayed, 0 S, Clock) >= MaxDelay_P6

    TRANSITION Perform TO NoAction IF
      DURATION(PRE(internal_P6) IN_STATE Perform, 0 S, Clock) >= 0 S

  END STATE_VARIABLE

END DEFINITION

END MODULE

INCLUDE "standard-modules.nimbus"
```

Bibliography

[1] L. Abraido-Fandino. An overview of REFINE 2.0. In *Proceedings of the second symposium on knowledge engineering, Madrid, Spain*, 1987.

[2] Mark A. Ardis and David M. Weiss. Defining families: The commonality analysis. In *Nineteenth International Conference on Software Engineering (ICSE'97)*, pages 649–650, 1997.

[3] J.M. Atlee and M.A. Buckley. A logic-model semantics for SCR software requirements. In S.J. Zeil, editor, *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'96)*, pages 280–292, January 1996.

[4] B. Auernheimer and R. A. Kemmerer. RT-ASLAN: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 12(9), September 1986.

[5] D. Batory and S. O'Mally. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[6] J.L. Bentley. Programming pearls: Little languages. *Communications of the ACM*, 29(8):711–721, August 1986.

[7] Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

[8] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, March 1986.

[9] Lisa Brownsword and Paul Clements. A case study in successful product line development. Technical Report CMU/SEI-96-TR-016, Software Engineering Institute, Carnegie-Mellon University, October 1996.

[10] G. Campbell, J O'Connor, C. Mansour, and J. Turner-Harris. Reuse in command and control systems. *IEEE Software*, 11(5):70–79, September 1994.

[11] Grady H. Jr. Campbell, Stuart R. Faulk, and David M. Weiss. Introduction to synthesis. Technical Report INTRO-SYNTHESIS-PROCESS-90019-N, Software Productivity Consortium, Herdon, VA, 1990.

[12] W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J.D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.

[13] A. M. Davis. Operational prototyping: A new development approach. *IEEE Software*, 6(5), September 1992.

[14] Debra M. Erickson. Structuring formal requirements specifications for reuse: A mobile robotics case study. Masters Project, University of Minnesota, April 2000.

[15] S. Gerhart, D. Craigen, and T. Ralston. Experience with formal methods in critical systems. *IEEE Software*, vol-11(1):21–39, January 1994.

[16] S. Gerhart, D. Craigen, and T. Ralston. Formal methods reality check: Industrial usage. *IEEE Transactions on Software Engineering*, 21(2):90–98, February 1995.

[17] Nancy G.Leveson. Intent specifications:an approach to building human-centered specifications.

[18] H. Gomaa. Reusable software requirements and architectures for families of systems. *Journal of Systems and Software*, 25(3):189–202, August 1995.

[19] Carl A. Gunter, Elsa L. Gunter, Michael Jackson, and Pamela Zave. A reference model for requirements and specifications. *IEEE Software*, 17(3):37–43, May/June 2000.

[20] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[21] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[22] Mats P. E. Heimdahl and Nancy G. Leveson. Completeness and consistency in hierarchical state-base requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.

[23] Mats P.E. Heimdahl, Jeffrey M. Thompson, and Barbara J. Czerny. Specification and analysis of intercomponent communication. *IEEE Computer*, pages 47–54, April 1998.

[24] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance, COMPASS 95*, 1995.

[25] C. L. Heitmeyer, B. L. Labaw, and D. Kiskis. Consistency checking of SCR-style requirements specifications. In *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, March 1995.

[26] C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

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

[28] K.L. Heninger, J.W. Kallander, J.E. Shore, and D.L. Parnas. Software Requirements for the A-7e Aircraft. Technical Report 3876, Naval Research Laboratory, Washington, D.C., November 1978.

[29] Michael Jackson. *Software Requirements and Specifications.* ACM Press and Addison-Wesley, 1995.

[30] Michael Jackson. The world and the machine. In *Proceedings of the 1995 Internation Conference on Software Engineering*, pages 283–292, 1995.

[31] Michael Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems.* ACM Press and Addison-Wesley, 2001.

[32] Michael Jackson and Pamela Zave. Domain descriptions. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 56–64, 1992.

[33] Michael Jackson and Pamela Zave. Deriving specifications from requirements: An example. In *Proceedings of the Seventeenth International Conference on Software Engineering (ICSE'95)*, pages 15–24, May 1995.

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

[35] Mehdi Jazayeri, Alexander Ran, and Frank van der Linden. *Software Architecture for Product Families: Principles and Practice*. Addison-Wesley, 2000.

[36] B. Kramer, Luqi, and V. Berzins. Compositional semantics of a real-time prototyping language. *IEEE Transactions on Software Engineering*, 19(5):453–477, May 1993.

[37] Juha Kuusela and Juha Savolainen. Requirements engineering for product families. In *Proceedings of the Twenty-Second International Conference on Software Engineering (ICSE'00)*, pages 60–68, June 2000.

[38] W. Lam. Creating reusable architectures: Initial experience report. *ACM SIGSOFT Software Engineering Notes*, 22(4):39–43, 1997.

[39] W. Lam, J.A. McDermid, and A.J. Vickers. Ten steps towards systematics requiremens reuse. *Requirements Engineering*, 2(2):120–113, 1997.

[40] Nancy G. Leveson. Sample tcas intent specification.

[41] Nancy G. Leveson, Mats P.E. Heimdahl, and Jon Damon Reese. Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, volume 1687 of *LNCS*, pages 127–145, September 1999.

[42] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering*, 20(9):684–706, September 1994.

[43] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–354, April 1995.

[44] David C. Luckham, James Vera, Doug Bryan, Larry Augustin, and Frank Belz. Partial orderings of event sets and their application to prototyping concurrent timed systems. *Journal of Systems Software*, 21(3):253–265, June 1993.

[45] Luqi. Real-time constraints in a rapid prototyping language. *Computer Languages*, 18(2):77–103, 1993.

[46] Luqi and V. Berzins. Execution of a high level real-time language. In *Proceedings of the Real-Time Systems Symposium*, 1988.

[47] Robyn R. Lutz. Extenting the product family approach to support safe reuse. *Journal of Systems and Software*, 53:207–217, 2000.

[48] Steven P. Miller and Alan C. Tribble. Extending the four-variable model to bridge the system-software gap. In *Proceedings of the Twentith IEEE/AIAA Digital Avionics Systems Conference (DASC'01)*, October 2001.

[49] J. Neighbors. The draco approach to constructing sofware from reusable components. *IEEE Transactions on Software Engineering*, 10(5):564–574, 1984.

[50] D.L. Parnas. On the criteria to be used in decomposing a system into modules. *Communications of the ACM*, 15:1053–1058, December 1972.

[51] D.L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(1):1–9, March 1976.

[52] D.L. Parnas. Designing software for ease of extension and contraction. In *Third International Conference on Software Engineering*, 1978.

[53] D.L. Parnas and P.C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, 1986.

[54] D.L. Parnas and J. Madey. Functional documentation for computer systems engineering. *Science of Computer Programming*, 25(1):41–61, 1991.

[55] Praxis Critical Systems Limited. *REVEAL: A Keystone of Modern Systems Engineering*, issue 1.1 edition, July 2000.

[56] R. Prieto-Diaz. Domain analysis: An introduction. *ACM SIGSOFT Software Engineering Notes*, 15(2):47–54, 1990.

[57] Software Productivity Consortium. *Consortium Requirements Engineering Handbook*, 1993. SPC-92060-CMC.

[58] Jeffrey M. Thompson. NIMBUS: A framework for static analysis and simulation of system-level inter-component communication. Master's thesis, University of Minnesota, December 1999.

[59] Jeffrey M. Thompson, Mats P.E. Heimdahl, and Steven P. Miller. Specification based prototyping for embedded systems. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, number 1687 in LNCS, pages 163–179, September 1999.

[60] Jeffrey M. Thompson, Michael W. Whalen, and Mats P.E. Heimdahl. Requirements capture and evaluation in NIMBUS: The light-control case study. *Journal of Universal Computer Science*, 6(7):731–757, July 2000.

[61] Jeffrey Michael Thompson. *Structuring Formal State-Based Specifications for Reuse and the Development of Product Families*. PhD thesis, University of Minnesota, 2002.

[62] David M. Weiss. Defining families: The commonality analysis. Technical report, Lucent Technologies Bell Laboratories, 1000 E. Warrenville Rd, Naperville, IL 60566, 1997.

[63] David M. Weiss and Chi Tau Robert Lai. *Software Product Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.

[64] Michael W. Whalen. A formal semantics for RSML^{-e}. Master's thesis, University of Minnesota, May 2000.

[65] P. Zave. An insider's evaluation of PAISLey. *IEEE Transactions on Software Engineering*, 17(3), March 1991.

[66] Pamela Zave. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–29, January 1997.

Department of Computer Science and Engineering
4-192 EE/CS Building
200 Union Street SE
Minneapolis, Minnesota

Appendix B - Jeffrey M. Thompson's Dissertation

UNIVERSITY OF MINNESOTA

This is to certify that I have examined this copy of a doctoral thesis by

Jeffrey Michael Thompson

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Dr. Mats P.E. Heimdahl and Dr. Maria Gini
Name of Faculty Adviser(s)

Signature of Faculty Adviser(s)

Date

GRADUATE SCHOOL

Structuring Formal State-Based Specifications for Reuse and the Development of Product Families

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Jeffrey Michael Thompson

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Dr. Mats P.E. Heimdahl and Dr. Maria Gini, Advisor
June, 2002

Dedication

To my dear friends, Mike, Tim, Andy, and Jeffrey who provided support and encouragement.

To my family, who provided my foundation.

Acknowledgments

I would like to again thank all my family and friends for their support and encouragement as I have finished up my doctoral work.

I am indebted to my committee who have provided guidance and support. Nikolaos Papanikolopoulos who has written recommendation letters for me resulting in my successful application for the Doctoral Dissertation Fellowship. Maria Gini who taught me about mobile robotics, was my co-advisor, and agreed to serve both on my Masters degree and Doctoral degree committees and Rajesh Rajamani who has also served on both my Masters and Doctoral degree committees.

Steve Miller from the Rockwell-Collins Advanced Technology center has been an invaluable resource for the work that is presented in this dissertation. Steve's years of experience in safety-critical systems have been essential to the evaluation of the work presented herein.

I would especially like to thank Mike Whalen for his insight, knowledge, and support. Mike and I have went through the entire graduate school process together and his ideas, and insights are woven into the work presented here. I cannot imagine what my graduate school experience, or life, would have been like without Mike.

Finally, I would like to thank my primary co-advisor, Mats Heimdahl. First, for encouraging me to get a doctorate in the first place. Second, for his unwavering support, patience, and dedication as I undertook the work. I will always be happy to have attended graduate school with Mats as my advisor.

Abstract

The software in a safety critical system has the potential to cause loss of life, loss of property/money, or environmental disaster. Researchers have found that most safety-critical errors are introduced in the requirements, rather than the design and implementation stages of development. These errors are conceptual in nature and reflect misunderstandings about the intended operation of the system or the system's environment. Furthermore, requirements for safety critical systems can be difficult to express: the software must interact with a variety of analog and digital components and be able to detect and recover from error conditions in the environment. To compound the problems, a requirements specification goes through many changes before it is completed–these changing requirements are a major cost driver in industrial projects.

A mathematically precise, or formal, specification of the requirements provides an unambiguous representation; therefore, use of a formal specification language to model the requirements promises to improve the quality of (and thus, assurance in) the requirements. Nevertheless, formal specifications are costly to develop and little research has been conducted on structuring formal requirements specifications. In most cases, there is a lack of a clear methodology for specification development. Ideally, such specifications would be easy to maintain and reuse, particularly in light of the fact the many companies build families of related systems. Unfortunately, this is beyond the current state-of-the-art and is a critical barrier to industrial acceptance of these techniques.

To address these concerns, this dissertation makes three key contributions. First,

we have extended the state-of-the-art in expressing the structure of product families. Second, we have defined a methodology for creating formal specifications of safety-critical process-control systems that includes the overall process for creating the specifications as well as techniques directed specifically at reuse. Finally, a module construct designed to support the methodology and product family structuring has been added to the formal specification language RSML^{-e}.

Contents

List of Figures

Chapter 1

Introduction

Software plays an increasingly important role in safety-critical systems. An error in such systems has the potential to cause loss of life, environmental disaster, or loss of property/money. Examples include medical devices, avionics systems, anti-lock brakes, and control of nuclear power plants. Unfortunately, the state-of-the-art in software development for critical systems does not provide industry with the theory, tools, and techniques to produce the high-quality software needed at a reasonable cost; the software is often poorly engineered, has hidden errors, and is very expensive to develop. Therefore, techniques to help increase the quality of software while at the same time reduce its cost are of utmost importance.

Software development typically begins with a high-level concept for a new system. Next, a document describing the intended software behavior, i.e., the *requirements* of the software, is written (called the requirements specification). Exactly how the software accomplishes the requirements is determined in the software design and implementation phase, where the actual working program is constructed. After the implementation phase, the working program is tested to detect and eliminate errors. Finally, the software is taken into operation.

Researchers have found that most errors leading to an accident are introduced in the requirements stage, rather than the design and implementation stages. Traditionally, requirements specifications have been written in a natural language, such as English. Unfortunately, these natural language specifications are inherently ambigu-

ous and unclear. Therefore, during the design and implementation, misunderstandings of the requirements often lead to design and implementation mistakes that may cause accidents. To combat this issue, a *formal specification language* can be used to describe the requirements.

A formal specification language has a mathematically well defined semantics; thus, a requirements specification expressed in such a language has a precise and unambiguous meaning. Another advantage of formal specifications is that they can be supported by tools that allow for visualization, animation, and mathematical analysis of the requirements. One such formal specification language is RSML^{-e} (Requirements State Machine Language without Events) [39].

Despite these advantages, formal specification languages have *not* achieved widespread use in industry. One major obstacle is lack of guidance on the process of writing a formal requirements specification and descriptions of the most effective techniques for capturing the requirements in the desired language. Although some work has been done on such guidelines, it is fragmented and incomplete. Another barrier is cost: although the use of formal requirements specifications holds the potential to reduce overall development costs, formal requirements specifications are typically much more costly to develop than their natural language counterparts.

In today's marketplace, many companies that build critical systems often create lines of similar products, or *product families.* Product family members (i.e., the individual products) share many common features (called *commonalities*) but vary in certain well-defined ways (called *variabilities*). The concept of a product family is well-understood in most industries. For example, in the auto industry, many cars from the same maker might share parts; some cars are even built on the same chassis with different sheet metal and trim, e.g., the Chrysler Concorde and Dodge Intrepid, or the Ford Taurus and Mercury Sable. Unfortunately, in the software industry it is

common to see different (but related) products being developed by different project teams with little coordination and essentially no reuse.

This duplication of effort is costly; and, considering the cost of creating a formal specification, creating a completely original formal specification for each family member is out of the question. Therefore, reuse of the formal requirements specifications in the context of product families is absolutely essential for successful adoption by industry of these techniques. Fortunately, a formal foundation for the requirements should make reuse easier than with traditional, informal techniques.

1.1 Contributions

The basic problem addressed by this dissertation is the underutilization of formal requirements specification languages in industry. This work reduces a major barrier to industrial acceptance of formal specification techniques through the development of a set of guidelines, i.e., a methodology, for the creation of formal requirements specifications for safety-critical systems.

The methodology must address how to organize, or *structure*, the requirements. This research has produced structuring techniques which support the development of software families (for example, a line of pacemakers for different heart conditions), as well as more ad-hoc techniques suitable for special situations. The techniques are illustrated in formal specification language RSML^{-e}.

Structuring techniques for requirements are best supported by special requirements specification language features that allow related pieces of the formal specification to be grouped together. However, researchers have focused on the development of tools to assist in the creation of formal requirements, analysis of the requirements, and execution of the formal specifications, but not on techniques describing *how* to create the specifications. Thus, many formal specification languages lack such an

organizational construct. A part of the research was to design an organizational construct for RSML^{-e}. This construct facilitates the techniques presented in the methodology and allows RSML^{-e} to be used in more extensive, industrial-sized case examples.

Finally, the methodology integrates many existing techniques and previous work. This makes the methodology comprehensive: practitioners who wish to apply these techniques to their own systems will not have to read volumes of research journals. Rather, the essentials of the current state-of-the-art, including my own work described above, will be presented in one location.

Thus, the contributions of the dissertation are in three main areas:

1. the development of techniques to structure formal requirements specifications,
2. the addition of a module construct to RSML^{-e}, and
3. integration with existing work to form a methodology, called the Family Oriented Requirements Method for Process Control Systems (FORM$_{PCS}$).

Within these general areas, there are a number different dimensions of contributions as well as contributions at different depths. This is summarized in Figure 1.1. The figure contains two dimensions: (1) the dimension from the most general (i.e., applicable to all software systems) to the most specific (i.e., applicable only to our specific specification language and techniques) and (2) from the most fundamental to the most detailed. Along these dimensions, we have three facets of contribution: (1) product family structuring, (2) methodology work, and (3) RSML^{-e} additions. Bubbles in the figure that have a grey background indicate an area where work has been done. Bubbles with a bold (thicker) border indicate areas in which this dissertation has made significant contributions. Bubbles with a white background and non-bold border indicate areas that are future work.

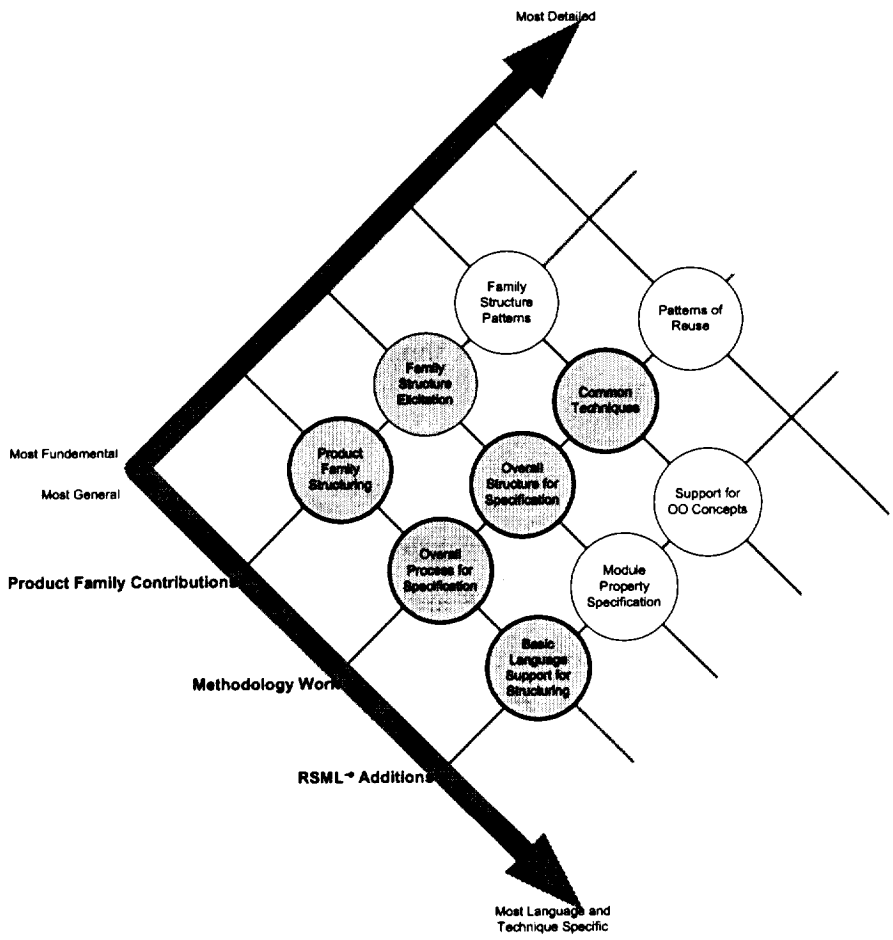

Figure 1.1: Framework of Contributions. Bubbles with a bold outline indicate areas of contribution by this dissertation; bubbles with a grey background indicate areas where significant research results have been achieved.

The first line of work is the product family facet. As shown in the figure, this facet is the most general. Techniques that were developed for product families are extensible to *all software systems* including the safety-critical systems of the most interest to this work. Furthermore, the product family structuring techniques are also based on the fundamental principles of product families and represent a basic contribution to the field of product-line development. The next facet of contributions has been the most explored by this work, and that is not surprisingly the methodology dimension. The original contributions of the methodology are highlighted and the methodology is available as a separate document [41]. The final facet of contribution for this research is the addition of the module construct to RSML^{-e}.

1.2 Organization

The rest of the dissertation is organized more or less around the contribution framework described above with some introductory material where necessary. First, we will present some background and related work that forms the foundation for much of the dissertation work in Chapter 2. Next, Chapter 3 presents the three primary examples for the dissertation work: the Altitude Switch (ASW), Flight Guidance System (FGS), and Mobile Robotics (MR).

The product family work is the first that will be presented as it is the most fundamental and allows us to give a very thorough overview of the high-level requirements for the case studies (Chapter 4). In this chapter, we concentrate on the structures that are present in the product family domain and do not delve deeply into the process which is used in elicitation of the family requirements (much has already been written on elicitation and management in the literature, for example [117, 116, 5, 16, 18, 20, 53, 12, 21, 28, 96, 95, 112, 100, 50]).

Moving on, Chapter 5 then explains the fundamentals about the types of systems

of interest in this report– safety-critical process-control systems– and illustrates various models for describing the entities involved. We relate how process control systems can be thought of in terms of the product family structuring discussed in the previous chapter. And, we also discuss the low-level micro-process, specification-based prototyping [109], on which the methodology is based.

The activities and processes of the methodology are presented in Chapter 6. Instead of reproducing the entire methodology, only an overview is given here (however, the entire methodology is available as a technical report [41]). We do illustrate the overall process and point out where original contributions were made in the development of the methodology.

Chapter 6 also presents an overview of a number of formal and semi-formal languages to which the methodology is applicable. Highlighted in this chapter are the reuse and modularization capabilities of the language as well as how suitable each language is to working at the *requirements* level as opposed to design or implementation. This chapter also contains an introduction to RSML^{-e}, the formal specification language developed at the University of Minnesota that we will use in our examples.

The next chapter (Chapter 7) illustrates how a modularity construct can be added to the RSML^{-e} llanguage to improve its ability to be used in conjunction with the methodology. These additions, make RSML^{-e} one of the most suitable languages in which to develop formal state-based requirements. The additions are primarily centered around the addition of a reusable module-type construct to the language.

Finally, Chapter 8 presents our conclusions and future work that may be completed. The dissertation also includes a number of appendices. Appendix A includes the definition of all the standard modules that are meant to be included with every specification in the new version of RSML^{-e}. Appendix B includes the completed Altitude Switch (ASW) requirements specification, and Appendix C includes the

completed ASW software specification.

Chapter 2

Related Work

Much of the thinking behind the work that has been done in structuring programs and software designs is also applicable to software requirements. Indeed, often a structure is imposed at the requirements level in large software projects to make the requirements easier to comprehend. Nevertheless, structuring at the requirements phase is fraught with its own, unique problems, e.g., the desire to avoid implementation bias through the structure of the requirements. This chapter attempts to overview the most relevant work related to requirements structuring. In addition, later chapters add more related work that is specific to the topics at hand.

We will begin by examining early work in software system structuring that provides the basis for much of the object oriented work and component software work that has followed. Next, we will examine product families (sets of related software artifacts) and the work that has been done there to facilitate reuse. Related to product families is work that has been done on macro-structuring of software systems in the software architecture community. Finally, we will wrap up the chapter by discussion what previous attempts have been made at methodologies for formal requirements and process-control systems.

2.1 Early Work

Much of the early work on structuring and design of software systems was written by David Parnas *et al.* This seminal research lays the groundwork for many later developments including object oriented analysis and design. Thus, it is good place to begin to examine research about the structuring and modularization of process control systems.

In [90] Parnas describes common manifestations of software which is not easily extensible or contractable. First, he suggests that defining the subsets of the program functionality belongs in the requirements phase. In particular, he describes searching for the minimal subset and then building incrementally on this initial functionality. Indeed, this approach is at the forefront of modern software design and implementation. Second, Parnas describes information hiding and module definition. Third, Parnas introduces the concept of thinking about software modules as "software machine extensions that will be useful in writing many ... programs." This idea is fundamental to object oriented design and analysis was well as the even more modern component-based technologies. Finally, Parnas describes avoiding loops in the uses graph of the software modules. Avoiding uses loops produces a software system which is much less interdependent, and therefore less complex. These ideas are illustrated in the paper through an address processing system example.

Parnas *et al.* also addresses abstract interfaces in several other papers. In [13,92] they describe the creation of abstract interfaces for the A7E aircraft specification [46]. This work discusses the importance of defining an abstract "virtual device," which could be a combination of software and hardware. This enables the developers to isolate changes in the hardware from the rest of the system. The work also describes a number of problems with using the methods, for example, virtual devices that are likely to change or that do not correspond to hardware devices.

The A7E project resulted in the creation of the SCR (Software Cost Reduction) [45] language for expressing the required behavior of process-control systems (discussed in more detail in the next section). During this time, the four-variable model for process-control systems was developed and later publicized in [93] (more on the four variable model and other reference models for process-control systems will be presented later in this chapter).

Nevertheless, the work discussed above does not include sufficient guidance for practitioners to be able to develop and structure formal models of process-control systems. The work does not address reuse specifically and also does not address issues specific to requirements, for example, how to avoid biasing or limiting the eventual implementation of the software because of the structure chosen for the requirements. More work is needed to further refine the basic ideas present in this foundational work so that it is (1) complete and (2) accessible to practitioners.

2.2 Product Family Engineering

Reuse of in the software domain has been the most successful when reusing software artifacts across members of a series of related products, i.e., a *product family.* In this section, we give an overview of the most relevant work that has been done in the area of product family engineering, starting out with background of the field, and moving on to specific research results and results in the related field of software architecture.

2.2.1 Background

A software product-line is a family of related software products designed to take advantage of their similarities and predicted variabilities. Domain engineering is the process of studying families of similar software artifacts so as to make it easier to build the individual members of the family. The concept of a program family was

originally developed by Parnas in [89] and later expanded in [90]. For the purposes of this discussion, we will take the terms domain, product family, program family and product line to be equivalent. Parnas gives a pragmatic definition of program families:

> *We consider a set of programs to constitute a family whenever it is worthwhile to study the programs from the set by first studying the common properties of the individual members.* [89]

Parnas observed that often programmers would create new programs by modifying existing programs. This process usually involved a reverse step where parts of the working program were discarded. The new program was sometimes crippled by design assumptions made for the original program that did not apply to the new program. Thus, Parnas postulated that it would better to start out by defining what was common about all such programs and successively refining the design until you had working programs as the leaves of a tree structure, with nodes within the tree representing the various design decisions.

The basic idea behind software product line engineering is shown in Figure 2.1. The development process begins by studying the domain – the family of software programs that is desired by the customer. The result of this analysis is some application engineering support. Researchers differ in what exactly should be provided to support the application engineering process, for example, in FAST (Family-oriented Abstraction, Specification, and Translation) [117] the application support is typically a domain specific language (DSL) and associated application generation facilities. In other approaches, the product of domain engineering might be a reference architecture that can be used to build each family member. After the domain engineering artifacts have been created, they are then used in the application engineering process to produce the individual family members.

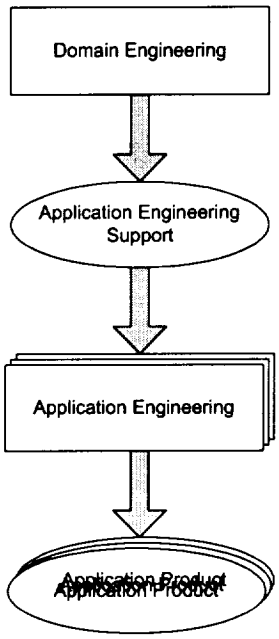

Figure 2.1: An overview of a domain engineering process

The cost benefit ratio of product-line engineering is clear and is illustrated in Figure 2.2. Domain engineering has the effect of making the slope of the cost line less because (presumably) the application engineering support (the product of domain engineering) makes it less costly to build each family member. Thus, given the cost savings to build each family member due to the application engineering support, the team must build enough family members to make up the cost of the initial domain engineering.

Of course, if product family engineering was as easy as it sounds from this simple introduction it would not be the active area of research it is today. There are many problems with establishing the requirements for and creating support for software product lines that are currently being addressed by researchers. In the next section, we will give an overview of various lines of foundational research in the product family area.

2.2.2 Product Family Research Concentrations

In recent months, this area of research has gown dramatically as many that were formally primarily concerned with software architecture have become more concerned with product families. Therefore, a complete view of all research that is happening in the field can not be presented here. Instead, we will focus on the foundational work as well as work that deals specifically with the *requirements* for a product line, which is the work most relevant to this dissertation.

FAST by David Weiss *et al.*: The work on FAST (Family-oriented Abstraction, Specification, and Translation) [117, 4] produced the first significant results in the field of product-line engineering. Based directly on Parnas' work, FAST focuses on using domain engineering to develop a domain specific language (DSL) and application

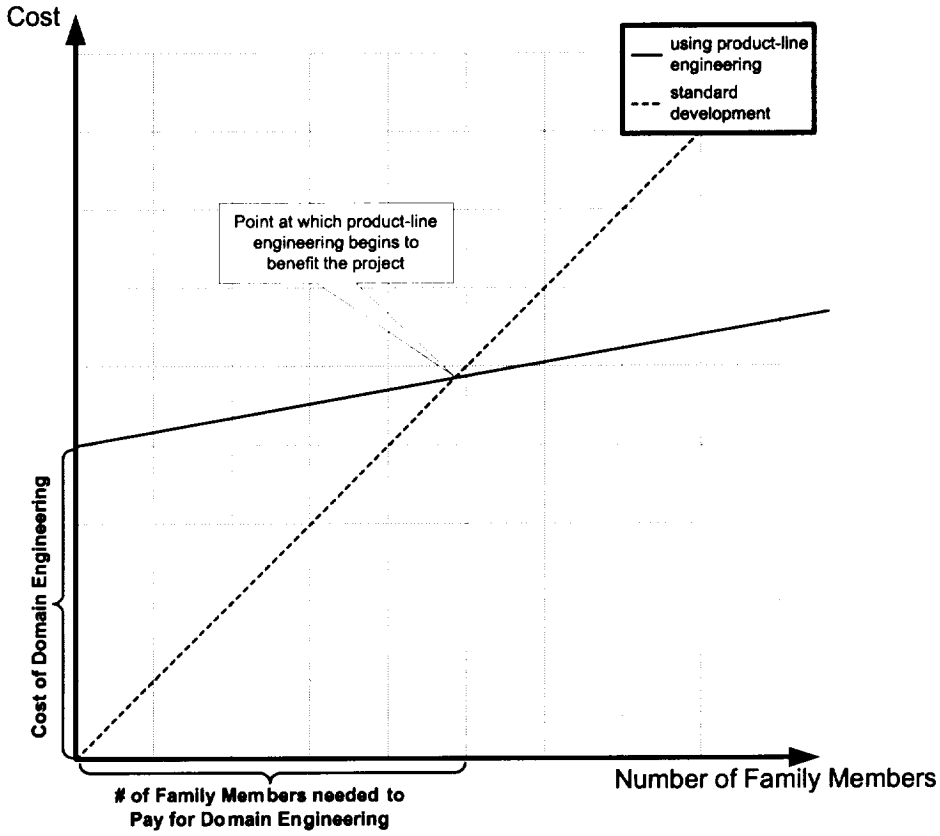

Figure 2.2: Cost-benefit analysis of software product-line engineering

generation facilities as the application engineering support (see Figure 2.1). The FAST approach has been applied to over 25 domains including a floating weather station [117], a commands and reports for the AT&T 5ESS telephone switch [19], and auditing software for the 5ESS [33]. The FAST process is one of the most developed of the product-line engineering approaches and is formally documented in [117].

The aspect of FAST most useful to the work presented in this dissertation is commonality and variability analysis [117, 116, 18], which focuses on identifying the aspects of the product family which are common across all members versus those which vary from member to member. The FAST researchers identified what information should be specified about commonalities and variabilities as well as the process that should be used to discover them.

Commonalities and variabilities can be used as a basis for thinking about the structure of the system, which is why they are of interest. Nevertheless, FAST (or any other product-line engineering approach) does not say how to use the commonalities and variabilities to help to structure the system.

Lam's Work: Unlike the FAST work described above, the goal of Lam's approach to product-line engineering was not the explicit development of a product line but rather a way to facilitate requirements reuse. In [59] Lam describes the RACE (Reusable Architecture Creation and Employment) process for product-line engineering. In RACE, the result of domain engineering is a reusable architecture that is adapted to work with each member of the product family. This work is based closely on previous work that was done by Tracz on domain specific software architectures (DSSA) [112, 113].

Lam's work is based on experiences in specifying the requirements for aero-engine control systems at Rolls-Smiths Engine Controls [58, 60, 61, 62]. The work is useful because Lam approaches the problem from the standpoint of requirements reuse

rather than concentrating entirely on just the idea of the product-line. This makes Lam's work especially interesting for this dissertation, since one of the goals is to reuse formal requirements for process-control systems. Nevertheless, Lam work is made up of a collection of approaches, some overlapping, that worked in specific situations related to the Rolls-Smiths Engines case-study. Lam does not generalize the work, and therefore the simpler and more elegant formulation provided by Weiss *et al.* and the FAST approach is a more solid foundation on which to build new techniques.

Lutz' work: In the past several years, Robyn Lutz has started to do some work on the safety analysis of product families [67, 68]. The basis of this work is Lutz' extensive experience with projects at the Jet Propulsion Laboratory (JPL). She has discovered a number of issues with current product-line engineering approaches that are of immediate concern when looking at adapting the product-line approach to work with the methodology that we are proposing. These issues are:

- **Near commonalities.** These are properties that are true for almost all the systems in the domain. Lutz presents two approaches to the problem: (1) model them as variabilities, or (2) model them as constrained commonalities. Nevertheless, Lutz states that she expects that the question of how to deal with near commonalities will be a recurring issue in product-line engineering.

- **Dependencies among options.** In this case, the problem is that among the variabilities there are constrains as to which options can occur together.

- **Hierarchy of variabilities.** Lutz discusses organizing the variabilities hierarchically such that all the family members at a certain node share the same value for a set of parameters of variability. Ultimately, there are many different trees that could be constructed in this fashion. It is an open issue whether or

not a hierarchy of variabilities would be beneficial and how to determine the structuring of the hierarchy.

Lutz' work is useful because it exposes issues encountered when applying the product-line engineering approach to real systems. The solution to the three questions above, for example, would contain essential information about the structuring of the commonalities and variabilities and hence the structuring of the specification. Major steps towards such a solution is exactly what is proposed in this dissertation.

Recent Developments: Software product-line engineering has the potential to deliver great cost savings and productivity gains to organizations that provide families of products, as well as give those organizations a competitive edge in the marketplace. For safety-critical systems, software-product line engineering has the potential to produce systems that are more safe than their serially produced counterparts while being cheaper and faster overall to build.

Although one of the main barriers to the use of product family techniques is one of process and organizational acceptance, technical issues have not been completely solved for product-line engineering. The techniques available work best for cohesive product families, where the variabilities do not have complex interdependencies. When this is not the case, it can be difficult to apply the product family approach even though there might be significant commonalities between the members of the family.

Current techniques for product-line engineering work well if the following conditions are met:

- The systems in the family share significant commonalities, and
- The variabilities which define each family member have a straightforward decision model, i.e., it does not require many complicated rules to describe how the

variability values are assigned to produce each family member.

The first point describes the essential feature of product families that Parnas noticed in his work. However, the second point originates in the practical experience of many researchers who have labored to construct software product-lines. Recall that Robyn Lutz observed that the primary limitations of the product family approach stem from difficulties in handling "*near-commonalities* and relationships among the variabilities"[emphasis added] [69]. Thus, the simplier the relationships among the variabilities, the easier it is to construct the product family.

Recording the structure of the product family at the *requirements* level before an architecture has been constructed may provide advantages in making an architecture that is more flexible in the face of changes to the domain. Therefore, we feel that it is important to develop a structuring technique for product families that is designed to be used at the requirements level. This would allow the potential to develop analysis techniques for the product family requirements and could provide insight into the high-level structure of the architecture. This high-level structure can guide the later creation of the architecture and is therefore complementary to current work in the field.

Nevertheless, most of the current approaches to product family engineering focus on developing the assets (i.e., reference architectures or generation facilities) using the commonalities and variabilities as a requirements specification for the product family. The issue of how to structure the architecture to overcome difficulties in the family itself (such as near-commonalities) is often intermingled with solutions to general architecture problems. In order to adequately address this large area of work, we have have devoted the next section to that topic.

2.2.3 Software Architecture and Software Structuring

Software architecture research focuses on leveraging patterns of software *design* and *programming*. Why then, discuss software architecture in the context of the *requirements* for product families? The reason is that much of the recent work that has been done in product families has been done with only a cursory look at the requirements problems for product families and with requirements issues intermixed with software architecture, design and implementation details. Therefore, it is somewhat challenging to get a complete picture of the work on product-line requirements without taking at least a small look at research in software architectures.

Much work in the software architecture community has focused on developing formal languages suitable for describing the software architecture. These architecture description languages (ADLs) include MetaH [115, 9, 114], Unicon [98], Darwin [72, 70, 71], Wright [2, 3, 86], Aesop [26, 25], Weaves [29, 30], C2 [76, 77], SADL [85, 84], ArTek [103], and Lilleanna [111] (among others).

Furthermore, as the research in software architectures have progressed, there have been several efforts to provide a survey of and classification of ADLs including Clements [17] and Medvidovic [79, 78]. These surveys have helped the research community distinguish ADLs from requirements languages and programming languages as well as provide researchers and practitioners with a good idea of what constitutes an architecture description language. An ADL is generally expected to include a method of modeling both components and connectors between components as well as providing some kind of type checking and configuration language (to support expressing variabilities of product family members).

Researchers are in agreement that an ADL is not a requirements language; yet, most work on the structuring of product families focuses on the structuring of the architecture and views the commonalities and variabilities as a flat structure when,

in fact, this is not the case. To compound this issue, much of the recent work in product families has focused on building a complete solution for a particular family. This has the positive effect that the work is applicable to real families of systems, but it has the negative effect that it is often difficult to separate concerns with the product family requirements from architectural or implementation concerns. This is especially important for our work, where we wish to discuss a formal expression of the requirements to do not wish to proceed directly to an object-oriented design and implementation of the family.

Therefore, in the work presented in this dissertation (Chapter 4) we make a clear distinction between the structure of the product family *requirements* versus the structure of the product family *architecture*.

2.2.4 Product Family Summary

Current work in product family engineering has been successful at achieving reuse in limited domains. Many lines of research are helping push the current state-of-the-art including new techniques for implementing product lines and expressing product line architectures. In this dissertation, we will address techniques for recording and reasoning about the structure of the product family requirements; a topic that is inadequately addressed by current work in the field.

In addition, we will strive to make a distinction between the product family requirements and the remainder of the product family development effort. This is necessary so that we can integrate formal specifications of the family requirements into the development effort In the next section, we discuss various system existing models system for process-control systems and several methodologies that have been proposed for specification construction.

2.3 Methodological Background

Some of the key related work for the methodology is the spiral model of software development proposed by Boehm [10, 11]. This model advocates managing risk in a software project by building and testing the project in smaller, more manageable phases. This is contrasted with the waterfall model [97], that advocates the unrealistic process of gathering and certifying all the requirements in the project up front and then proceeding on through the design and implementation phases.

As Parnas and Clements have noted [91], it is impossible that the complete requirements can be established in the beginning of a project because often the customer does not know or cannot clearly articulate what it is they want, some details become known only when the implementation or design progresses, and people will naturally make human errors. Therefore, Parnas and Clements focus on "faking" the rational design process; in other words, ordering the documentation for a system such that it appears to have been constructed by an idealized process so that it is accessible to the persons who need to review it, even though such a process was not followed. Furthermore, Parnas and Clements go on to state what they perceive as the rational design process that should be emulated by such documentation.

One of the main reason requirements change is because the customer often does not understand what it is they need or want until they can see a working version of the system. This is especially true of, for example, user interface systems. It is also true for safety-critical process-control systems because it is often difficult to visualize what the system will do in certain situations from a long list of natural language requirements. Thus, a prototyping approach where an executable model of the software is available early in the development life cycle is key to successful development.

We wish to develop requirements for process-control systems. Therefore, we

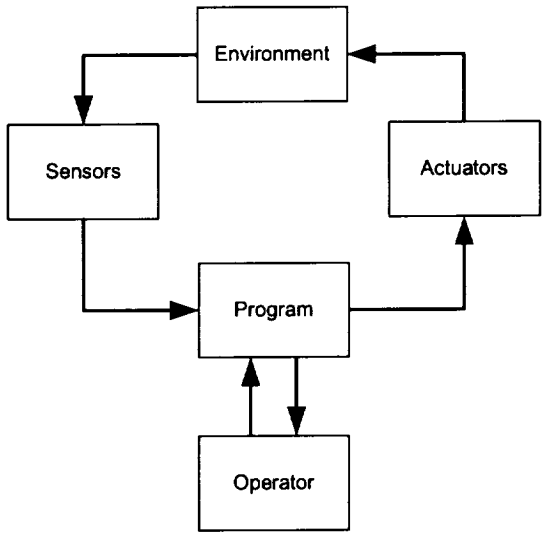

Figure 2.3: A basic process-control model

should first discuss the theory and general background of the process-control field before moving on to several different system models for thinking about process-control.

2.3.1 Introduction to Process-Control Systems

A system is a set of components working together to achieve some common purpose or objective. A process-control system usually involves an environment (i.e., the world), a program (or multiple programs) whose purpose it is to establish or maintain certain conditions in the environment, sensors and actuators that allow the program to get information about the environment and affect the environment, and finally the operator who can usually input various parameters to the running program and receive feedback from the running program. This is summarized in Figure 2.3.

Consider the environment of aircraft moving along in three dimensional space. In this unconstrained environment, airplanes are free to have midair collisions, disrupt take-off and landing of other aircraft, and so forth. Clearly, this is not desirable;

therefore, we need a process-control system for air traffic control that will allow us to enforce certain restrictions in the environment, for example, that planes do not run into one another. To do this, we will have to have some sensors, which will give us data about the position of the aircraft in the system, some actuators which will allow us to make course corrections for the aircraft in the system, and possibly have some operator input to guide these choices.

There are a number of difficulties in constructing process-control systems. First, the environment is a key element that is often under specified and/or misunderstood. Misunderstandings about the environment in which the system operates have been the cause of numerous accidents. Second, the sensors and actuators often provide an imperfect, or noisy, view of the real world; sensors can introduce errors, and actuators can fail. Therefore, the program may lose track of the true state of the environment and error conditions in the sensors and actuators can be difficult (or impossible) to detect. Finally, the controller often has only partial control over the process; therefore, state changes can occur in the environment when no actuator commands were given by the program.

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

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

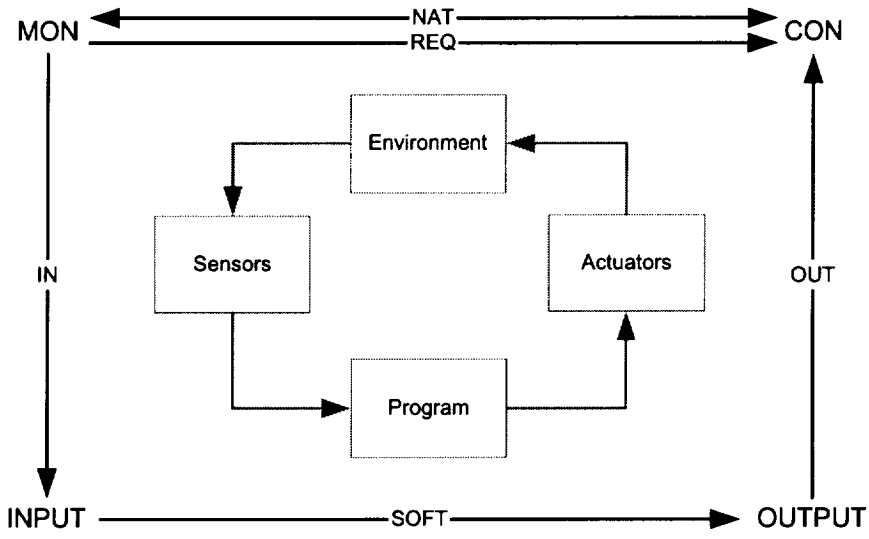

Figure 2.4: The four-variable model.

overall objective as well as potentially part of the constraints.

This model is an abstraction—responsibility for implementing the control function may actually be distributed among several components including analog devices, digital computers, and humans. The next sections discuss elaborations of this model and what are considered system versus software requirements.

2.3.2 The Four Variable Model and CoRE

The four variable model was published by Parnas and Madey [93] and is the closest to the traditional process-control model. The four variable model was developed from early efforts to specify the requirements for the A-7 aircraft [46, 45] in a language called Software Cost Reduction (SCR) [43, 42, 44], which was also developed on the project.

An overview of the four-variable model is shown in Figure 2.4. For reference, the

process-control model has been reproduced in grey inside of the four variable model. The four variable model consists of (not surprisingly) four sets of variables (MON, CON, INPUT, and OUTPUT) and five relations between those variables (REQ, NAT, IN, OUT, and SOFT).

All of the variables in the model are continuous functions of time. The MON, or monitored, variables are those quantities in the *environment* of the system that we can observe. An example of a monitored quantity in the hypothetical air traffic control system mentioned previously might be the altitude of an aircraft. The CON, or controlled, variables are those quantities in the environment that we can affect. Thus, altitude is also a controlled quantity in our example.

The relationships between the MON and CON quantities are essential to understanding the behavior of the system. The NAT relation expresses the environment of the process-control system. Thus, the NAT relation expresses how changing the controlled quantities affects the monitored quantities (CON to MON) as well as constraints that exist on any required behavior and behaviors that already exist in the environment (MON to CON). The REQ relation represents the functions and operations that we desire to introduce into the environment, and is therefore a relation from MON to CON.

Monitored and controlled quantities do not correspond to inputs from sensors or outputs to actuators, they are an idealized representation where we always know their values to infinite accuracy. Because of problems introduced by noisy and inaccurate sensors as well as inaccurate or unreliable actuators, the relationship of the monitored and controlled quantities to the software inputs and outputs is often non-trivial. The four variable model represents the software inputs and outputs as INPUT and OUTPUT respectively and the transformation of monitored quantities to input quantities as the IN relation (similarly for the OUT relation). Given REQ, IN, and OUT a spec-

ification of SOFT, the software requirements, is theoretically achievable. However, the four variable model leaves open how this specification should be constructed, and how it should be structured.

The four variable model has served as the foundation for several research efforts. Most notably, the work at the Naval Research Laboratory (NRL) on the SCR notation and our work on specification-based prototyping [109] (which is discussed further in Chapter 5).

To augment the four variable model and support the SCR language, the CoRE (Consortium Requirements Engineering) [99] methodology was produced by the Software Productivity Consortium (SPC). Many talented people contributed to the development of CoRE and it contains many valuable ideas for the development of process-control systems. In particular, The CoRE guidebook [99] provides technical information on how to document the environmental variables and how they fit into the four-variable model, and they provide some guidance on which environmental quantities are suitable candidates as monitored and controlled variables.

The CoRE process begins with the system requirements and ends with a software requirements specification. The overall CoRE process is divided up into five main phases:

1. **Identify Environmental Variables:** In this phase, the specifiers identify environmental quantities that the software can monitor and control. Environmental constraints, i.e. constraints which would exist without the presence of the system, are defined; this is called the NAT relation. Finally, the structure of the system is represented as an entity-relationship (ER) diagram.

2. **Preliminary Behavior Specification:** In this phase, a first draft of the high-level behavioral specification, the REQ relation, is developed. The decision is made as to which environmental quantities are monitored, controlled, or both.

The domains of the controlled functions are defined and the monitored variables which effect the value of the controlled variable are recorded. Finally, the number and type of mode machines needed is decided.

3. **Class Structuring:** In this phase, the structure of the system is decided. The CoRE methodology attempts to support a pseudo-object oriented structuring technique which includes specialization and generalization. The primary structuring guidance is to choose the objects based on the physical structure of the system and as an extension to the ER diagram developed in the first phase.

4. **Detailed Behavior Specification:** This phase culminates in the completion of the behavioral specification of the classes identified in the previous phase. The controlled variable functions are completely defined and the other classes are refined. Timing constraints, in terms of when each mode machine is recomputed, are also addressed.

5. **Define Hardware Interface:** In this phase, the characteristics of the sensors and actuators are defined by defining the IN and OUT relations.

In practice, the developer must iterate between these phases of the CoRE methodology rather than proceeding through them in a waterfall-like fashion. The CoRE manual addresses this iterative nature in and provides an overview of both the ideal and the interactive (realistic) development process. This enables CoRE to provide both guidelines on *what* should be contained in the specification as well as *how* the specification should be developed. CoRE further addresses the *how* question by providing entry and exit criteria for each of the key steps in the model.

CoRE includes many good ideas and suggestions for developers. The guidelines on identifying the monitored and controlled variables for the system are useful in focusing the construction of the REQ relation. Also valuable is the process of developing a

dependency tree for the monitored and controlled variables early in the specification life cycle. This helps to clarify thinking and avoids circular dependencies, which are not permitted in SCR and not recommended by Parnas [90]. Finally, the overall process is good and provides important guidance to specification developers on how to proceed with the development effort and what information should be included at the various stages. These guidelines provide some help, but in our experience more guidance is needed to correctly make the crucially important selection and classification of the environmental variables.

Nevertheless, the CoRE methodology falls short in a number of areas. First, assuming we have captured the relations REQ, NAT, IN, and OUT, we need to derive the SOFT relation. There is little guidance in the CoRE guidebook as well as in the original four variable model work on how to achieve this task. Second, the only structuring guidelines are based on the physical structure of the system. Such a structure will not work in general and does not facilitate reuse of operational modes. Since multiple structuring techniques are not presented, there are also no tradeoffs between them. Third, the latter phases of the methodology are unclear and, in some cases, self-contradictory. In particular, the "Define Hardware Interface" step is a mere twelve pages long and includes nothing on the structuring or refinement of the IN and OUT relations. Furthermore, their seems to be some confusion about the difference between monitored/controlled variables and input/output variables throughout CoRE. Finally, CoRE does not specifically address reuse (other than saying the reuse is possible using the class structuring). It does not include information on how to plan for reuse or structure for reuse.

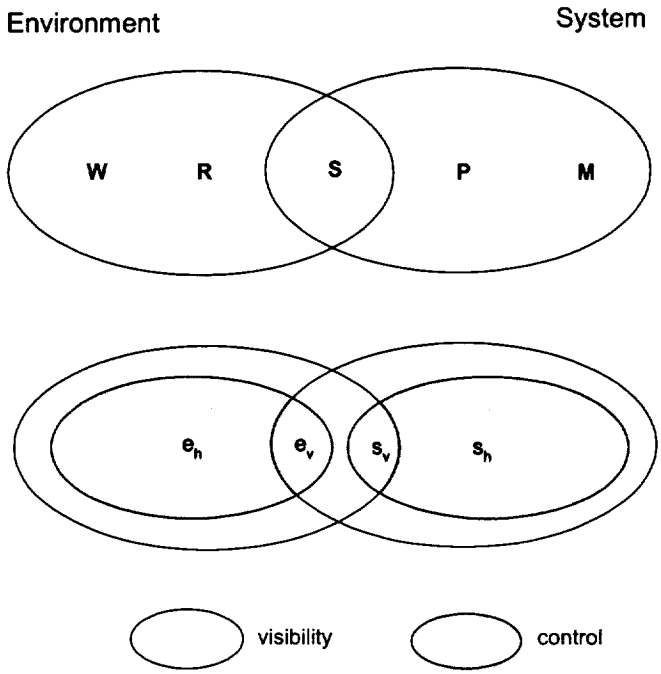

Figure 2.5: The world, requirements, specification, program, and machine (WRSPM) model [32].

2.3.3 The WRSPM Model and REVEAL

Michael Jackson and Pamela Zave have presented a reference model for requirements specifications—the world-machine model [48, 50, 51, 120]. The discussion in this section is based on the formalization of this model provided by Gunter *et al.* [32].

The main idea behind the world-machine model is a separation of concerns between the world (or the environment) and the machine (or, the system to be built). Jackson *et al.* state that the requirements and problems exist in the world, because it is the world that we wish to change via the introduction of the machine. Thus the WRSPM is based on five artifacts grouped roughly into two categories—the ones relating mostly to the environment (or world) and those that pertain mostly to the

computer and software (or the machine). These artifacts are denoted by W, R, S, P, and M as illustrated in Figure 2.5. The artifacts are:

The World (W): This is domain knowledge that captures knowledge of environmental facts.

The Requirements (R): Describes what the customer need from the system expressed in terms of its effect on the environment.

The Specification (S): A less abstract description of the desired behavior that provides enough information for a software developer to design and implement a system that satisfies the requirements.

The Program (P): The program (implemented in some programming language) that implements the specification and runs on some machine.

The Machine (M): The system (computer, associated hardware, operating system, etc.) that executes the program.

Variables that belong in the world are collectively called e—the ones belonging in the machine are called s. The variables in the world e are split into two mutually exclusive sets e_h and e_v—the variables in e_h are hidden from the system and are considered to be exclusively in the domain of the environment. The variables in e_v are visible to both the environment and the system. The variables in s are decomposed in a similar way into s_v and s_h where all variables in s_h are hidden from the environment.

With this decomposition of the variables, e_h, e_v, and $s_v$are visible to the environment and used in W and R. Variables in e_v, s_v, and s_h are visible to the system and used in P and M. The only variables shared between the environment and the system are in e_v and s_v—therefore, the specification S is restricted to use only variables in

$e_v$and s_v and they form the interface between the environment and the system. Figure 2.5 (from [32]) illustrates the relationship between the variables and the various artifacts.

The WRSPM is related to the four-variable model discussed in the previous section. W corresponds to NAT in the four-variable model. R corresponds to REQ. In the four variable model, REQ and NAT are somewhat more restrictive than W and R in that it can seemingly only make assertions about the variables that are shared between the environment and the system. W and R allow us to make statements about variables that are hidden from the system (e_h). SOFT corresponds to P, and IN and OUT together correspond to M.

The real difference between these two models is in the consistency and sufficiency constraints imposed on these various relations. We will not consider these technical details further in this guide—the interested reader is referred to [32] for a detailed discussion.

The WRSPM model is intended as a reference model only and does not discuss how the various variables in e and s are selected. Nor does the method discuss how the various artifacts are derived or structured—this is a pure reference that simply discusses the required relationship between these different artifacts.

The REVEAL methodology [94] was developed by Praxis Critical Systems, Limited as a method based on the world-machine model. REVEAL consists of six stages:

1. **Defining the Problem Context:** In this stage the goal is to develop an understanding of the problem (i.e., what it is about the world that you wish the system to help to achieve) and explore the boundaries of the problem.

2. **Identifying Stake holders and Eliciting Requirements:** This second stage is associated with identifying stake holders to the project and eliciting requirements and domain knowledge.

3. **Analyzing and Writing:** In the third stage, the requirements and domain knowledge are written down and analyzed using the completeness criteria of the WRSPM model.

4. **Verification and Validation:** The fourth stage involves checking the work that was done in the first three stages to ensure its accuracy.

5. **Use:** After the fourth stage, the requirements will be used throughout the rest of the development life cycle.

6. **Maintenance:** Should any changes to the requirements be discovered, then we must perform maintenance on the description. This is discussed in the final stage of REVEAL.

The REVEAL methodology is based on two key processes: (1) conflict management, and (2) managing requirements. The work in REVEAL on managing requirements is the most relevant to this work.

REVEAL implements a unique notion of traceability of the requirements based on the WRSPM model. In the WRSPM model the requirements are satisfied when the World (W), and the Specification (S) imply the requirements. That is,

$$W, S \vdash R$$

This concept is referred to as the *Adequacy Check* by the REVEAL method. REVEAL uses the adequacy check as a basis for the entire requirements process.

Suppose, for example, that we start out writing down the general requirements for a system that we are building. We would record these requirements, R_{gen}, along with a description of the World, W, and specification, S. Then, we demonstrate that $W, S \vdash R_{gen}$ and life is good.

Now we want to introduce more detail to R_{gen} and produce a set of requirements at a lower level of abstraction. Of course, the detailed requirements, R_{det}, are certainly related to R_{gen} and certainly that relationship should be preserved in the requirements documentation. Thus in REVEAL, we would prove the property:

$$W, R_{det} \vdash R_{gen}$$

then, by transitivity, we can reuse the original adequacy check on the general requirements so show that the requirements are still satisfied. Doing this provides traceability to the high-level requirements from the detailed requirements and ensures that if the high-level requirements change, the proofs for the detailed requirements will no longer work (as you would expect). This notion of traceability is similar to that proposed by Leveson [27, 64] for Intent Specifications except that in REVEAL the traceability is organized around a more formal framework.

The traceability information recorded by the REVEAL methodology combined with its use of the WRSPM system model make REVEAL a good complement to the CoRE methodology that we discussed earlier. However, a combination of REVEAL and CoRE would still not serve the needs of practitioners because neither methodology adequately addresses the issues associated with recording the requirements for product families. Furthermore, REVEAL does not address specifically the issues associated with state-based specification of process-control systems in a formal language (as CoRE does).

2.4 Summary

To summarize, currently, there is no focused, up-to-date methodology for developing formal specifications for families of process-control systems. Many useful specification languages exist, and there has been much convergence recently on the types of

languages and language features which are needed to express these types of systems. Nevertheless, the most recent and complete methodologies available are the CoRE methodology written a number of years ago for SCR-style specifications and the REVEAL methodology based on the WRSPM model. The CoRE methodology must be updated to reflect what has been learned about specification construction since its development (some of which is included in REVEAL) and the REVEAL methodology does not contain guidance on problems specific to constructing a formal specification of the requirements.

Both CoRE and REVEAL lack guidance on how to structure and express the requirements for a whole family of process-control systems and how to achieve reuse of the specifications. We reviewed product-line engineering, an approach to software reuse that leverages the similarities between members of a product family to achieve cost savings and reuse. Unfortunately, much work in product family engineering is oriented towards the design and implementation of the family rather than at recording and structuring the product families requirements, which are largely composed of the commonalities and variabilities of the product family. Nevertheless, issues with product-line engineering that Lutz has raised are key to discovering how to structure the commonalities and variabilities and thus the requirements. Structuring specifications, particularly with the goal of reusing operational modes or creating formal requirements for a product family, is also currently not adequately addressed in the literature.

Chapter 3

Case Studies

This chapter introduces the three primary running examples of the dissertation: the Altitude Switch (ASW), the Flight Guidance System (FGS), and Mobile Robots (MR). All of these examples represent families of process-control systems. The techniques in the dissertation were also applied to a family of implantible cardio-defibrillators at Medtronic during the summer of 2001; however, because this information is Medtronic proprietary, details about the family and the specifications of it cannot be included in this dissertation. We also provide an introduction to RSML^{-e}.

3.1 Altitude Switch (ASW)

The first example that we will consider is the Altitude Switch (ASW), which is derived from the ASW example proposed by Steve Miller from the Rockwell-Collins Advanced Technology Center [82]. While our example is based on an original system, it is hypothetical in that we have introduced certain features for demonstration purposes and we may not represent the full spectrum of possibilities in this one example.

In avionics, the altitude of the aircraft is an essential environmental quantity. Many devices on board the plane react to changes in the altitude, for example, the autopilot must know the plane's current altitude in order to know whether to climb or descend. In addition, there are many other devices on board the plane which rely on altitude. However, these different devices vary greatly in the types of actions that

the perform in response to the altitude data. In addition, the types of altitude data differ significantly from system to system and from aircraft to aircraft. We will use the ASW is the primary running example for the dissertation.

We might make an initial attempt at a family description such as the following:

> *The ASW family consists of systems on board the aircraft that utilize the values from the various altimeters on board to make a choice among various options for actions (one of which being to do nothing) and perform the chosen action.*

The ASW family could be viewed as a sub-family of a larger family which would include all aspects of avionics systems. This description does describe all the systems on board the plane which use the altitude and is therefore a good starting point for describing our family. However, notice that the particular actions that the system performs can be largely separated from tasks relating to measuring the altitude and fusing the results from various different types of altimeters. We will refine this further in Chapter 4 where we talk about the high-level commonalities and variabilities for the ASW family.

Clearly, the ASW system must have a method of measuring the altitude. In avionics, there are various types of altimeters that can be used to measure altitude, for example, barometric altimeters (that measure altitude by measuring the air pressure), radio altimeters (that use radio signals to measure altitude), and GPS altitude (that uses the global positioning system (GPS) satellites to measure altitude). As the aircraft moves from an area of high air pressure to an area of low air pressure the barometric altitude will change even if the aircraft remains at the same absolute distance from sea level. Therefore, the values given by these different altimeter types are not necessarily comparable to one another directly.

In addition, in the avionics domain there are two types of altimeters: analog and digital. The type of analog altimeters are designed to be used in thresholding applications. Rather than provide a numeric altitude, these analog altimeters are hard-wired at the factory to report whether the aircraft is above or below a certain threshold. This is done primarily for cost concerns. Digital altimeters, on the other hand, do provide a numeric altitude. All altimeters that we will consider provide an indication of the quality of the altitude measured (i.e., whether the measured altitude is good or bad). The number and type of altimeters on each plane is specific to each family member.

The particular action(s) that the ASW can take as a result of crossing a threshold vary across family members; however, in many ways the action to be performed is orthogonal with decision to perform the action. For the purposes of this dissertation, we will primary concentrate on a sub-family of the ASW family where and ASW turns on or off a particular Device of Interest (DOI). We will however, explore the various methods that the ASW might use to make the decision to perform the action.

Because the ASW is the primary running example for the dissertation, it is used in almost every chapter. Chapter 4 gives a more complete overview of the ASW family by providing the initial commonalities and variabilities, explaining and exploring the family structure, and giving the decision model for the ASW family. A high-level overview of the ASW specification effort is provided in Chapter 5 followed by a more detailed look at some parts of the specification in Chapter 6. The ways that modules facilitate reuse is examined in Chapter 7.

3.2 Mobile Robotics (MR)

The mobile robotics domain is the second case study for the dissertation. Although it will not be discussed under every topic, as is the ASW, we will use it to illustrate the

product family structuring techniques in Chapter 4 as well as some of the high-level process issues in Chapter 5.

The domain of mobile robotics that we will consider encompasses small robots ranging in size from approximately 6 inches long to up to about 2 feet long. The robots have a limited speed, and can operate either autonomously (via a radio modem or radio Ethernet) or via a tether cable going to a personal computer. The robotics platforms come from various vendors and have a wide variety of sensors and actuators available. Also, the robots can support many different behaviors – scouting an area, constructing a map, working collectively, and so forth.

We model the mobile robotics domain as a product family taking into consideration both the different hardware platforms that could be supported and the many different behaviors we wished to specify. This proved to be difficult using conventional product-family techniques because the mobile robot family is both *n-dimensional* and *hierarchical* [22] (n-dimensional and hierarchical product families are discussed in Chapter 4).

The mobile robotics domain breaks down along two clear dimensions: the hardware platform and the desired behavior. Each hardware platform conforms to a basic specification: it can move forward and backward, turn left and right, sense whether or not an object is in front of it, and so forth. In addition, the hardware platform may or may not be equipped with some sort of vision system or infra-red camera; the various sensors used to monitor the environment differ greatly in the speed and accuracy with which they provide information. On the behavior side, we can imagine that a basic behavior might be a random exploration of the robot's environment where the primary goal of the robot is collision avoidance and recovery. Furthermore, more complex behaviors can be added, for example, wall following, going through doors, and finding particular objects. Therefore, along both the hardware and behavior

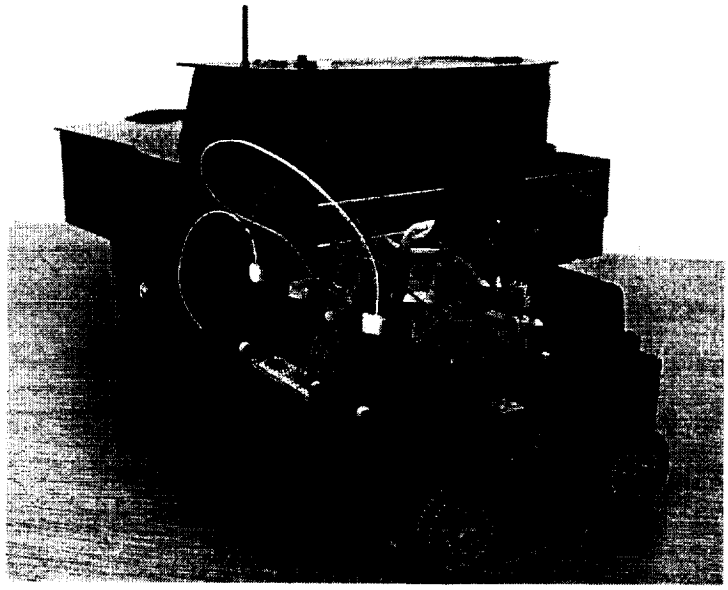

Figure 3.1: Pictures of the Mobile Robots (Photo by Timothy F. Yoon)

dimensions, the mobile robot family can be viewed *hierarchically.*

A family member in the mobile robotics domain will be defined by a pairing of the desired behavior with the robotic platform. Of course, there are constraints between the two dimensions and not all behaviors can run on all hardware platforms. For example, a behavior that requires the robot to find all red objects in a room will not work unless the robot has a sensor capable of distinguishing red objects from non-red objects. These complications of the mobile robotics domain will be discussed in Chapter 4.

As a specific example for this dissertation, we will consider a mobile robotics domain consisting of three classes each of robots and behaviors. Figure 3.1 shows a photograph of two of the mobile robots used as an example. Will will consider the following robotics platforms:

- A custom robot made out of Lego pieces with two infra-red sensors in the front

for obstacle detection, a front bumper, and tank-tread locomotion. We will call this one LegoBot.

- A Pioneer robot made by ActivMedia [1] which has an array of sonar sensors, a gripper, collision detection via motor stalled, and wheels for locomotion. We will call this one Pioneer.
- A Pioneer (see above) with a color vision system. We will call this one Pioneer w/ Vision.
- A small "pickle" robot that can roll around, and jump over small obstacles, and that is equipped with a camera. We will call this one Pickle.

The Pioneer Platform [1], is built and sold by ActivMedia, Inc (background of Figure 3.1). The Pioneer includes an array of sonar sensors in the front and sides that allow it to detect obstacles. To detect collisions, the Pioneer monitors its wheels and signals a collision when the wheels stall. The Pioneer is supported by a comprehensive software library (called Saphira) that manages the communication with the robot over radio modem.

The lego-bot is a smaller platform built from Lego building blocks and small motors and sensors. The lego-bot uses a tank-like track locomotion system and has infrared sensors for range detection. The lego-bot is controlled via a tether to the robot from the personal computer. This tether is connected to a data-acquisition card and the software specification for the lego-bot behavior must directly manage the low-level voltages and signal necessary to control the robot; there is very little support for the actuators and sensors.

Although this is a *significant* simplification of the actual domain, it will be sufficient to illustrate the various concepts in the dissertation. Clearly, the real mobile robotics domain is significantly more complex than space allows us to present in

this dissertation. For example, we have not discussed whether or not the robot can move objects in its environment (with a gripper, for example). Nevertheless, we can illustrate some interesting properties of the domain even with this limited example.

3.3 Flight Guidance System (FGS)

The Flight Guidance System (FGS) example is loosely based on some of the FGS systems built by Rockwell-Collins, but does not represent actual products of the company. This FGS example is essentially the same one that has been introduced in numerous other publications, including [81, 23].

The purpose of the FGS is to compare the measured state of the aircraft to the desired state of the aircraft and then generate pitch and roll commands that attempt to maintain the measured state as close as possible to the desired state. The FGS can be partitioned into two pieces: (1) the continuous control laws that govern the performance of the various control surfaces of the aircraft, and (2) the complicated mode logic that defines how the FGS switches between these control laws.

To perform its job, the FGS must communicate with other systems on board the aircraft as well as other components of the Flight Control System (FCS), of which it is a part. The level zero context diagram for the FGS is shown in Figure 3.2. The FGS accepts input from the Primary Flight Display (PFD), Flight Control Panel (FCP), Flight Management System (FMS), Autopilot, Attitude Heading Reference System (AHRS), and NAV radio. These systems provide the FGS with the information it needs to compute which mode is active. The FGS in turn provides output to the PFD, FMS, FCP, and Autopilot.

The aircraft operators (i.e., the crew) primarily interact with the FGS through the use of the Flight Control Panel. This panel includes a number of buttons for manually selecting and deselecting modes of the FGS; the buttons have lights by

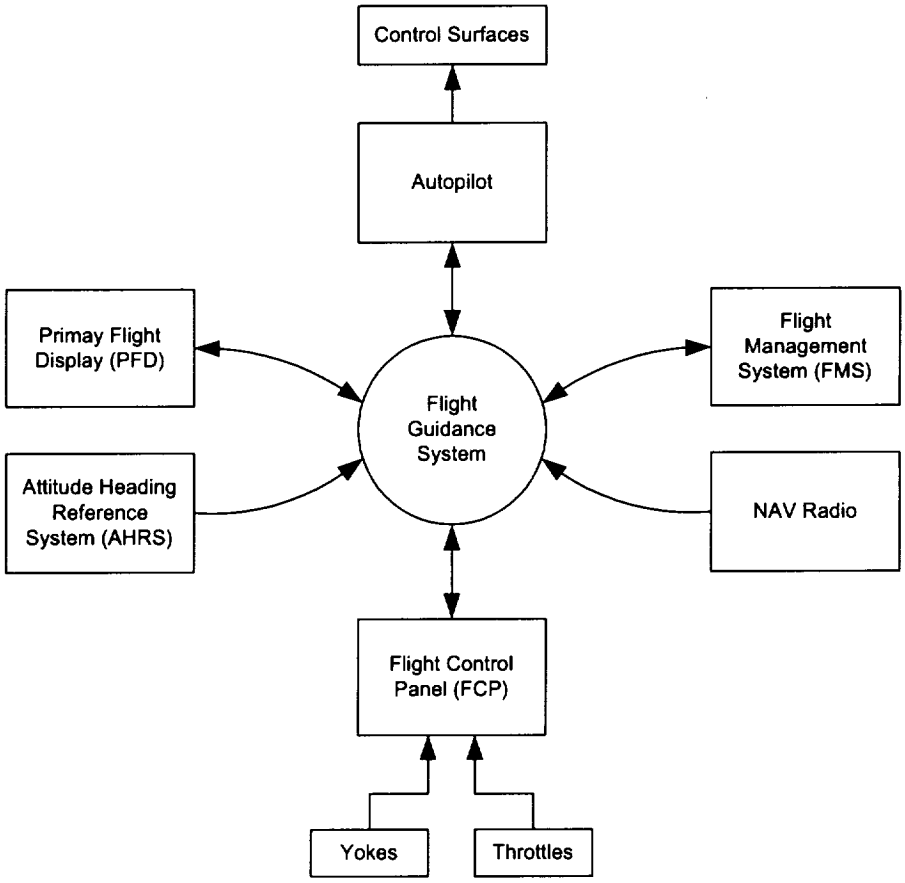

Figure 3.2: The FGS Level 0 context diagram

them that are lit with the mode is active. In addition, the FCP has several knobs with the pilot can use to adjust settings that have wider range of values, such as the selected altitude. The modes of the FGS are also annunciated on the primary flight display using short text strings.

To make matters more complicated, there are actually two FGS systems on the aircraft. This redundancy is present for safety reasons so that if one FGS should fail, the other FGS can be used to fly the aircraft. Nevertheless it should be clear to even the casual observer that only *one* FGS should be allowed to control the aircraft most if not all of the time; otherwise, the two systems would potentially fight over control. Therefore, the FGS systems need to be synchronized a good deal of the time. In addition to two FGS systems, there are also two FMS, AHRS, Air Data, PDF, and NAV radio systems. Usually, the two sets of systems are referred to using left and right (e.g., FGS-left, FGS-right).

The control of the aircraft is divided into the lateral and vertical components. Thus, at all times the FGS is required to select one and only one lateral mode to control the horizontal axis of the aircraft and, similarly, one and only one vertical mode to control the vertical axis of flight. One mode in each axis is designated at the *default* mode, meaning that if no other modes on that axis are active then it will become active.

3.4 Introduction to RSML^{-e}

The majority of the work for the dissertation is independent of the particular choice of formal modeling languages. Nevertheless, to present the examples for the dissertation we have chosen to use the formal modeling language RSML^{-e} (Requirements State Machine Language, without Events). In this section, we will provide an overview of RSML^{-e} for those readers who would like a complete understanding of RSML^{-e} prior

to seeing any examples. In practice, most readers could probably skip this section until Chapter 7 when a detailed proposal for changes to RSML^{-e} is presented.

The first version of RSML^{-e}, RSML, was developed as a requirements specification language for process-control systems and is based on David Harel's Statecharts [36]. One of the main design goals of RSML was readability and understandability by non-computer professionals such as end-users, engineers in the application domain, managers, and representatives from regulatory agencies [66].

Initial projects with RSML were a success and the language was well-liked by users, engineers, and computer scientists. The explicit event propagation mechanism (as mentioned above), however, was a major source of errors and misconceptions [65]. Therefore, the events were eliminated from RSML The resulting language, RSML^{-e}, has a fully formal semantics [118] and interfaces for the specification of inter-component communication [40]. RSML^{-e} is a cousin to SpecTRM-RL described in [65] in that they share the formal semantics but the syntax is substantially different.

RSML^{-e} is state-based specification language. An RSML^{-e} specification consists of a collection of *input variables*, *state variables*, *input interfaces*, *output interfaces*, *functions*, *macros*, and *constants*, which will be discussed below.

In RSML^{-e}, the state of the model is the set of assignment histories of all *variables* and *interfaces*. The state information is used to compute the values of a set of *state variables*, similar to mode classes in SCR [43]. These state variables can be organized in parallel or hierarchically to describe the current state of the system. Parallel state variables are used to represent the inherently parallel or concurrent concepts in the system being modeled. Hierarchical relationships allow *child* state variables to present an elaboration of a particular *parent* state value. Hierarchical state variables allow a specification designer to work at multiple levels of abstraction, and make models simpler to understand.

Assignment relations in RSML^{-e} determine the value of state variables. As discussed in Chapter 6, these relations can be organized as *transitions* or *condition tables.* Condition tables describe under what condition a state variables *assumes* each of its possible values. Transitions describe the condition under which a state variable is to *change* value. A transition consists of a source value, a destination value, and a guarding condition. A transition is taken (causing a state variable to change value) when (1) the state variable value is equal to the source value, and (2) the guarding condition evaluates to true. The two relation types are logically equivalent; mechanized procedures exist to ensure that both functions are complete and consistent [39].

The state variable definition and assignment relation for a state variable in the mobile robotics specification are given in Figure 3.3. The assignment relation is given as a series of transitions from one value of the state variable to another. For example, the first transition is from Startup to Cruise_Forward. On each transition, a condition defines when that transition is taken. These conditions are simply predicate logic statements over the various states and variables in the specification. The conditions are expressed in disjunctive normal form using a notation called AND/OR tables [66]. The far-left column of the AND/OR table lists the logical phrases. Each of the other columns is a conjunction of those phrases and contains the logical values of the expressions. If one of the columns is true, then the table evaluates to true. A column evaluates to true if all of its elements match the truth values of the associated columns. An asterisk denotes "don't care." Although none of the tables given in Figure 3.3 have multiple columns, there will be examples of multiple column tables later in the dissertation.

Input variables in the specification allow the analyst to record the the values reported by the environment or various external sensors. They are assigned based on the messages received by input interfaces (discussed briefly below).

State Variable

Normal

Location: Reactive_Control

Transition: Startup→Cruise_Forward

Condition:

TIME > 2 s	T
..Failure IN_STATE Ok	T

Transition: Cruise_Forward→Collision_Recover

Condition:

CollisionDetectedMacro() = TRUE	T
..Failure IN_STATE Ok	T

Transition: Cruise_Forward →Avoid_Obstacle

Condition:

ObstacleDetectedMacro() = TRUE	T
CollisionDetectedMacro() = FALSE	T
..Failure IN_STATE Ok	T

Transition: Collision_Recover→Cruise_Forward

Condition:

Prev_Step(..Robot_Recover_Action IN_STATE Done)	T
..Failure IN_STATE Ok	T

Transition: Avoid_Obstacle →Cruise_Forward

Condition:

Prev_Step(..Robot_Avoid_Action IN_STATE Done)	T
..Failure IN_STATE Ok	T

Transition: Avoid_Obstacle →Collision_Recover

Condition:

CollisionDetectedMacro() = TRUE	T
..Failure IN_STATE Ok	T

Figure 3.3: The definition of the *Normal* state variable

Interfaces encapsulate the boundaries between the RSML^{-e} model and the external world. There should be a clear distinction between the inputs to a component, the outputs from a component, and the internal state of the component. Every data item entering and leaving a component is defined by the input and output variables (state variables designated as outputs). The state machine can use both input and output variables when defining the transitions between the states in the state machine. However, the input variables represent direct input to the component and can only be set when receiving the information from the environment. The output variables are presented to the environment through output interfaces.

The state variables are placed into a partial order based on data dependencies and the hierarchical structure of the state machine. State variables are data-dependent on any other specification entities that contained in the predicates in their condition tables. A variable is also data dependent on its parent variable (if it has one). The value of the state variable can be computed after the items on which it is data-dependent have been computed. A single computation of all the variables in the specification is referred to as a *step.*

RSML^{-e}supports numerous expressions that allow for the specifier to express the conditions (that define the data dependencies). Of course, RSML^{-e} allows for standard arithmetic and relational expressions, as indicated in Figure 3.4. Also in the figure, RSML^{-e} allows the specifier to reference the expected minimum, etc. that was given in the specification.

RSML^{-e} supports a number of different expressions on the variables of the specification as well. Input variables in RSML^{-e} are assigned only by the interfaces; therefore, there may be steps in which the input variable is not assigned. State variables, however, are assigned a value in every step. Figure 3.5 gives the expressions that are currently available for state variables and input variables. As the figure

Expression	**Meaning**
$x \odot y$ where $\odot$ is one of $+$, $-$, $*$, $\div$, $<$, $>$, $\leq$, $\geq$, $\neq$	Standard mathematical and relational expressions.
NOT x	Standard logical NOT
$-x$	Unary Minus
x EQ_ONE_OF $\{y_1, y_2, \ldots, y_n\}$	True if x is equal to any one of y_1, y_2, thru y_n expressions.
x::EXPECTED_MIN	Returns the expected minimum of variable x.
x::EXPECTED_MAX	Returns the expected maximum of variable x.
i::MIN_SEP	Returns the expected minimum separation for interface i.
i::MAX_SEP	Returns the expected maximum separation for interface i.

Figure 3.4: A summary of the standard mathematical and relational expressions supported in RSML^{-e}

shows, currently RSML^{-e} includes facilities for getting the previous assignment, previous value, and value in the previous step for input variables and state variables. These expressions are powerful, but in our work we have found them more limited than what we would like. These expressions are one of the languages features that will be changed in the new version of RSML^{-e} proposed in this chapter.

RSML^{-e} currently supports a limited form of arrays and the expressions in Figure 3.6 give the syntax for these expressions. The reader can see that currently it is possible to express the concepts of "for all," "exists," and others in the language. Again, this facility, while powerful, does not allow us to add features without extending RSML^{-e}. This is a topic which is addressed in the next several sections of this chapter.

Finally, to further increase the readability of the specification, RSML^{-e} contains many other syntactic conventions. For example, RSML^{-e} allows expressions used in the predicates to be defined as functions, and familiar and frequently used conditions to be defined as macros. *Functions* in RSML^{-e} are mathematical functions that are used to abstract complex calculations. A *macro* is simply a named AND/OR table that is used for frequently repeated conditions and is defined in a separate section of the document.

3.5 Summary

This short chapter has presented an introduction to the case studies and notation that will be used as the primary running examples for the rest of the dissertation. In the following chapters, we will introduce more details about these examples as they are used to illustrate the concepts in the dissertation.

The ASW will serve as the main example, being used in nearly every chapter. The Mobile Robotics example will be used primarily to illustrate the product family

Expression	**Meaning**
ASSIGNED(x)	True if variable x was assigned in this step. Valid for input variables.
CHANGED(x)	True if variable x has changed value in this step.
WHEN(x)	True if variable x has become TRUE in this step.
PREV_STEP(x)	Returns the value that variable x had in the previous step.
PREV_VALUE(x, [y])	Returns the value that variable x had before it took on its current value. y allows for any number of values in the past. Primarily used with State variables
PREV_ASSIGN(x, [y])	Returns the value the variable x had before it was assigned. Primarily used for Input variables; for state variables, $(PREV\_STEP)(x)$ = PREV_ASSIGN(x).
TIME_CHANGED(x, [y])	Returns the time that variable x changed value. Primarily for State variables.
TIME_ASSIGNED(x, [y])	Returns the time that variable x was assigned. Primarily for Input variables.

Figure 3.5: A summary of the previous value expressions supported in RSML^{-e}

Expression	**Meaning**
EXISTS(i, x, c)	True if there exists an i such that c is true for the variable x.
FORALL(i, x, c)	True if for all i c is true for variable x.
COUNT(i, x, c)	Equals the number of conditions c that were true for variable x.
FIRST_INDEX(i, x, c)	Equals the value of the first index i for which c is true for variable x. Similarly, there is a LAST_INDEX.

Figure 3.6: The array expressions currently supported in RSML^{-e}

concepts presented in the next chapter. Finally, the FGS is given as an example of a large, industrial-sized example on which these techniques were applied and validated.

Chapter 4

Product Family Structuring

This chapter discusses the foundational work the has been done in product family structuring in the dissertation. This work was originally published in [106] and a more expanded version is currently under review for a special edition of the Requirements Engineering Journal [107].

Recall that a product family is a group of related programs that share so many common features that it is useful to study the group of programs as a whole before studying each individual program. Current approaches were discussed in Chapter 2. In this chapter, we propose a structuring technique for product families that views the families themselves in a multi-dimensional and hierarchical fashion. This helps us to deal with existing problems, for example, *near commonalities*, and also, helps to extend the approach to domains that, traditionally, would be difficult for product-line engineering.

This chapter is organized as follows. First, we will present some background both from other researcher's case examples and our own that indicate that product families should be thought of in an n-dimensional and hierarchical way as well as what we mean by those terms. Next, we present the structuring technique that allows the product family to be organized n-dimensionally and hierarchically. Then, we devote three sections to explaining how this technique is illustrated on each of the case studies in the dissertation. Finally, we present a brief evaluation of the technique.

4.1 Extending Product Families

We have mentioned in Chapter 2 that current approaches to product family engineering work well when the family contains a cohesive set of commonalities and simple relationships among the variabilities. We hypothesize that it *should* be possible to model and reason able product families with much more complex variability relationships than is currently possible today. We believe that to make this possible, we must first better understand the structures that are present in product families. This section gives the background for the structures that we have observed (and have been observed by others), namely *n-dimensional* and *hierarchical* product families.

4.1.1 n-Dimensional Product Families

A three dimensional object has many different projections into two dimensional space. Furthermore, it can be difficult (or impossible) to determine the shape of the three dimensional object from the projections into two dimensional space unless the projections are carefully chosen. When dealing in higher-dimensional space, the problem is similar. We use the term *n-dimensional* to refer to the fact that most product families have many different possible organizations. This is because each organization is really a projection of the multi-dimensional family into a lesser dimensional space. This section provides justification for the fact that families are n-dimensional objects.

Attempts have been made to organize the product family requirements in a hierarchical fashion [69, 89, 57, 59]. Lutz noted in her attempt to organize the variabilities into a tree that "there were several possible trees, with often no compelling reason to select one possible tree over another" [69].

Brownsword and Clements present a shipboard command and control systems family which contained 3000-5000 parameters of variation for each ship [15]. They state that "the multitude of configuration parameters raises an issue which may well

warrant serious attention." In addition, they present three different views of the architectural layering of the base system that "do not conflict with each other; rather they provide complementary explanations of the same ideas."

Both these examples, as well as our own experience, illustrate the fact that often a product family is multi-dimensional; therefore, a hierarchical decomposition is not sufficient to capture the structure of the domain. As an example, mobile robots form families along the dimensions of hardware platform (common basic features, but different modes of locomotion, different environmental sensors, different manipulators, etc.) and behaviors (common basic behaviors, but they may also require wall following, obstacle avoidance, mapping, etc.). We call families that decompose naturally along such dimensions *n-dimensional product families.*

n-Dimensionality is common in software systems. Thus, software design and implementations have already attempted solutions to the problems associated with having an n-dimensional space. In software implementation, researchers have proposed aspect-oriented programming [54, 55, 80] a technique that allows the programmer to separate out one dimension of the program from another and then use an automated tool to "weave" the dimensions back together again. The approach was first tried successfully on separating out the locking and synchronization code from the rest of of the code base using a tool called AspectJ [56].

Our approach is similar in structure to the notation of design spaces [63] and extended design spaces [7, 6]. Lane states that design spaces were created to allow a system designer to describe and classify the various architectural alternatives for a software system [63]. A dimension in a design space represents a single variation in a system characteristic or a single design choice; thus, a design space dimension is related to the product family concept of a variability.

Both aspect-oriented programming and design spaces are promising techniques

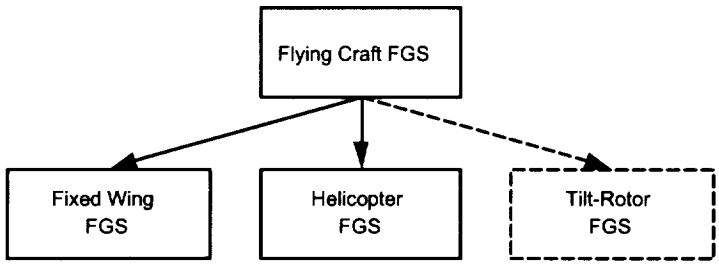

Figure 4.1: FGS product family covering flying craft

aimed at structuring the design and implementation of product lines. In addition, there is much other work in structuring that has been done in the object oriented community, and software architecture community that may be applicable to the requirement phase. We clearly do not have the space to overview all of that work here. Nevertheless, we want a technique that will work on the product family requirements. The key, in our view, is to define a simple structuring mechanism which does not introduce unnecessary design or implementation detail but which is still able to capture the essence of the problem at hand.

4.1.2 Hierarchical Product Families

Suppose that a company wished to construct a flight guidance system (FGS) for both fixed-wing aircraft and helicopters. Many of the tasks that the system has to perform might be common across these two radically different aircraft: interaction with other systems, deciding to level off at a particular altitude, mode transition logic related to when it is legal to switch between the various operating modes. Therefore, many requirements between these two systems will be the same, or very similar. Nevertheless, the actual control of the aircraft is very different. Therefore, developing a single set of commonalities and variabilities that span this entire domain is difficult.

Some would argue that this difficulty stems from the fact that the family is simply

too diverse to be considered a product line. However, it is clear that these systems share much in common, which was the original, and in our view the most important, criterion for being a family. Thus, we propose the concept of a *hierarchical product family.*

Most previous attempts at product family structuring have focused on hierarchically grouping the *variabilities* while the *commonalities* remain the same for all family members [69, 59]. Notable exceptions are Parnas [89] and Brownsword and Clements who noted in their case study at CelciusTech [15] that sometimes product-lines exist within the main product line. However, Parnas' work is rooted in design and coding choices. Brownsword and Clements mention this phenomenon in passing and apply it in a more limited way than what we advocate.

In our approach, *additional commonalities* which are *unrelated* to the commonalities of the parent product family can be added in the sub-families. Of course, these additional commonalities cannot conflict with those in the parent family. The hierarchical decomposition of the FGS family is shown in Figure 4.1. Thus, the helicopter sub-family can have significantly different requirements than for fixed-wing aircraft, yet share many things in common as well.

This will eventually effect the architecture and structure of the systems. For example, the product of the domain engineering for the parent family, Flying Craft FGS, might be a set of reusable components, whereas the product of domain engineering for the children might be a reference architecture or generation facility. The architectures for the fixed-wing aircraft and the helicopters could differ significantly and use the components from the parent family in different ways.

By structuring the requirements in this way, we have avoided imposing restrictive design constraints on the family members and instead focus on the structure of the domain itself. Furthermore, should the company wish to start building FGS systems

for an entirely new set of aircraft, for example, tilt-rotor aircraft, this could be done while reusing many aspects of the FGS systems already implemented. This is also shown in Figure 4.1.

4.1.3 Constraints on the Solution

When starting to develop a structuring technique for product family requirements that would be able to deal with n-dimensional and hierarchical product families, we determined that any such structuring technique must:

- Be based on structures that are present in the *domain* itself, not on implementation or design concerns,
- Be simple, allowing the analysts to capture the structure of the domain without introducing complex notations or concepts,
- Be amenable to the types of structures observed in product family analysis, and
- Produce a readable and usable artifact that facilitates reasoning about the structure of the domain.

We chose to explore a structuring technique based on a set-theoretic view of product families. The notion of sets proves surprisingly useful for thinking about the structure of a software product line, yet is simple and based upon well understood principles.

4.2 Structuring Technique

One way to view a product family is as a set, where the boundaries of the set are determined by the commonalities, and the individual members of the set are distinguished by the values of their variabilities (Figure 4.2). As the figure demonstrates,

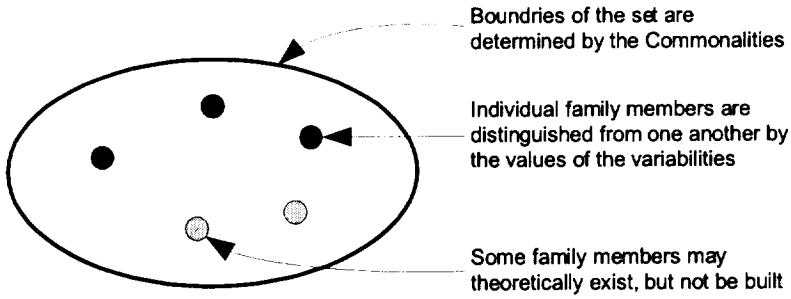

Figure 4.2: A simple product family

it is entirely possible that some members of the family may theoretically exist but not yet be built (shown in gray). Furthermore, the family may be undefined at some points within the boundaries due to, for example, illegal or nonsensical combinations of variability values. We will use this view of a product family throughout this section to demonstrate how current approaches to product-line engineering might be expanded to a greater class of systems.

4.2.1 Representing Hierarchical Product Families

The most basic structure that can be represented with the set theoretic approach is the subset. Figure 4.3 shows a product family, A, which has been divided into two subsets, B and C. Furthermore, C has been further divided into subsets D and E. This corresponds to a hierarchical decomposition of the family.

Consider a member of family E, e_1. The member e_1 must have all the commonalities defined for E as well as have some value for all the variabilities in E. Furthermore, because E is a subset of C and A, e_1 is also a member of families C and A. The general definition for any family E which is a subset of another family C is as follows:

- E must include all of the commonalities in C.

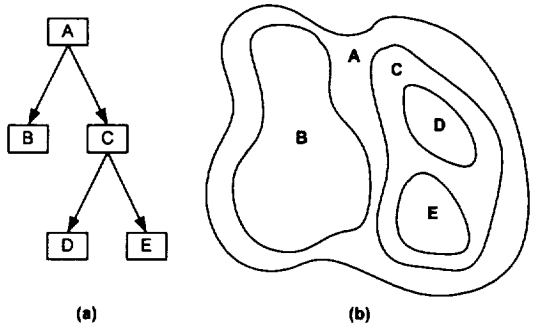

Figure 4.3: Hierarchical decomposition and subset structure

- E must include all of the variabilities in C; however, E may restrict the range or options available in the variabilities.
- E can add additional commonalities which are not present in C as long as the additional commonalities do not conflict with the commonalities or variabilities in C. These new commonalities might come from a refinement of variabilities in C or might be completely unrelated.
- E can define additional variabilities which are not present in C as long as those variabilities do not conflict with the above.

The first criterion is straightforward and necessary for the subset E to be completely contained within C. The second criterion defines the fact that E may wish to refine or restrict the values of the variabilities of C. For example, in the mobile robotics domain, a variability across the entire domain might be that the maximum speed of the mobile robot can vary from one to five miles per hour. However, subsets might define a lesser maximum speed depending on the hardware involved. It is possible for this refinement to result in an additional commonality, for example, suppose that we have aircraft that can use either radio altimeters, barometric altimeters, or GPS altimeters to measure altitude (a variability); then, a subfamily of these aircraft

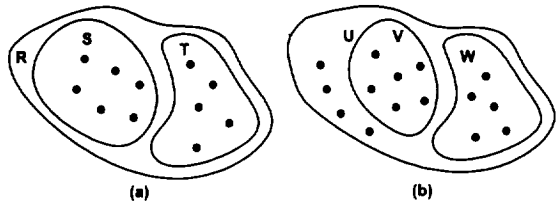

Figure 4.4: Abstract versus non-abstract families

could state as a commonality that all aircraft in that subfamily have only barometric altimeters. Additional commonalities can also be added which are unrelated to the parent family. For example, it is likely that the family of helicopters will need different commonalities than the family of fixed-wing aircraft. Finally, it is possible to add additional variabilities.

The two cases of hierarchical decomposition are shown in Figure 4.4. Part (a) of the figure demonstrates that the family R need not have any members that only exist in R. In a sense, R is an *abstract family*, because any member of R must be either a member of S or a member of T. This is similar to our FGS example from earlier, where all family members are either helicopters or fixed-wing aircraft and it does not make sense to talk about member which are only of the parent family. However, this need not be the case, as Figure 4.4(b) demonstrates. In the mobile robotics domain (see Section 4.5), we will have a basic robotic platform which will form the outer family member. This outer family will *not* be abstract because there are some robots which only conform to the minimum specification.

4.2.2 Intersection of Sub-Families

Another structure that can be represented using a set-theoretic approach is that of set intersection. The ability to represent a set intersection distinguishes this approach from the purely hierarchical structures which have been applied by others. This is

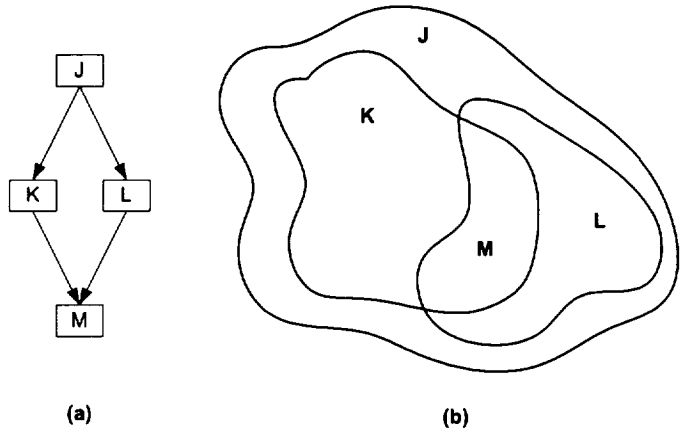

Figure 4.5: Set intersection and non-hierarchical structure

shown in Figure 4.5.

Consider a member, m_1, of M. By definition, m_1 is also a member of families K, L, and J. Thus, m_1 must have all the commonalities of both K and L. In addition, M is a subfamily of both families K and L (this is shown in the figure). The constraints on any family M which is a subset of families K and L are as follows:

- M must include all the commonalities of both K and L.
- M must include all the variabilities of both K and L; however, it may restrict those variabilities as above for subsets.
- M may introduce additional commonalities which are not present in either K or L so long as those commonalities do not conflict with the commonalities or variabilities in K or L.
- M may introduce additional variabilities which are not present in either K or L so long as those variabilities do not conflict with the above.

These structures can be used to describe product families which are both *n*-

dimensional and *hierarchical.* Representing hierarchy is done primarily by using the subset concept. Representing a dimension requires a bit more thought.

Dimensions represent alternate views of the product family based on some particular aspect of the commonalities and variabilities. For example, commonalities and variabilities in the hardware platforms may be viewed as one dimension, and the functionality of the family members viewed as another dimension. As mentioned previously, our notion of dimensions is similar to the notion of dimensions in extended design spaces [6, 7]; however, we would advocate the choice of several primary dimensions defined over cohesive aspects of the system and not make every variability a dimension. Possible dimensions may be hardware platform, required behavior, fault tolerance capabilities, etc. Some examples of dimensions are provided in Sections 4.3 and 4.5.

When a family or subfamily has been decomposed into several dimensions, we expect to have to make a choice in each one of those dimensions in order to have a valid family member. That is, we will have to instantiate the variabilities and select, for example, both a hardware platform as well as the desired functions for a family member.

When a family has been decomposed into several dimensions, we expect to have to make a choice in each one of those dimensions in order to have a valid family member.

4.2.3 Addressing Existing Issues

As we mentioned previously, current attempts at scoping the requirements for product lines are thwarted by near-commonalities and complex dependencies among the variability choices. Before we move on to the more detailed examples of the dissertation, we will take a moment to give an overview of how our structuring technique of

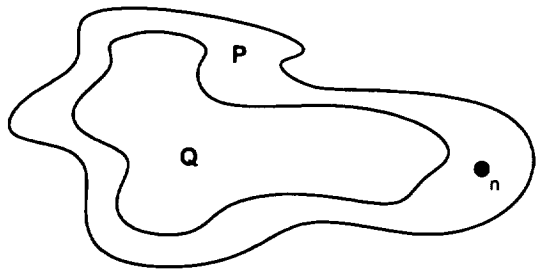

Figure 4.6: Set representation of a near-commonality

n-dimensional and hierarchical product lines will help to solve the problems presented by near-commonalities and complex variability dependencies.

Near-commonalities: A near-commonality (NC) is a commonality which is true for almost all (e.g., all except one) member of the product family. Lutz states that in her experience near commonalities "frequently had to be modeled" [69]. One solution for near commonalities is to model them as variabilities; however, this is, in some sense, a misrepresentation of their basic properties. The solution that Lutz advises is to model it as a constrained commonality of the form "If not member n then NC_1." However, a complex domain might contain numerous constrained commonalities with conditions significantly more complex than the example just mentioned.

Figure 4.6 shows how a near-commonality is represented in our approach. The near commonality, NC_1, would simply be a property of family Q (and not of P). Thus, the commonality naturally does not apply to n, a member of only P but does apply to any member of Q. This has several advantages. First, NC_1 is now a pure commonality of Q. Second, if another member of the family is introduced with reduced functionality [69] it need only be added as a member of P and Q may remain untouched. Finally, the subset structure can act as a guide in determining that certain components in the eventual application engineering environment will not be needed

for n.

Dependencies among options: In [69], Lutz cites modeling dependencies among options as one issues that must be addressed in product family engineering effort. A dependency is typically a constraint among the variabilities, for example, if variability V_1 has value B then variability V_2 must have option C. Ardis recommends treating this constraint as a commonality. However, in our experience, without some additional structuring, the domain could become littered with such commonalities; in addition, it may not be clear given a set of constraints whether or not a particular variability is viable.

In an approach where the commonalities and variabilities are qualified as above, the subfamily has no explicit description and its definition is essentially *distributed* across all the commonalities and variabilities that it has. If the family has many such dependencies with complex interactions, it will rapidly become difficult to visualize the structure of the domain. Furthermore, this distribution of the domain's structure to each variability and commonality makes changing the structure difficult and error prone (in order to defined a sub-family, you must change every commonality and variability that belongs to that sub-family).

In our approach, we can also represent constraints like these as commonalities. However, we isolate them into logical groups by forming different subfamilies so that their numbers do not become overwhelming. In the abstract example given above, a subfamily would be defined where "V_1 has option B" and "V_2 has option C" are both commonalities. This subfamily can be named and described in the requirements. Furthermore, the relationship of one sub-family to another, i.e., the structure of the domain, can be factored out making it easier to visualize and maintain in the future.

4.3 Flight Guidance System

In this section, we will discuss a small but illustrative subset of the FGS. Due to concerns about proprietary information, the full FGS commonality analysis cannot be given in this dissertation. Nevertheless, we can illustrate some key concepts with the FGS that are in the public domain and can be published.

As discussed in Chapter 3, the FGS is responsible for deciding which lateral and vertical modes of the aircraft are active. The lateral and vertical modes of the aircraft determine a number of important properties about the aircraft's operation, for example, whether the system is looking to find a navigation source or not, whether the plane is ascending or descending to a selected or flight plan altitude, and so on. The following commonalities describe the overall structure of the FGS.

C1 Every FGS has a lateral axis and a vertical axis, each of which has one or more modes.

C2 On every FGS, along each axis, exactly one mode shall be active at one time.

C3 Every FGS designates one mode for each axis (lateral and vertical) that shall be made active in the event that no other mode on that axis is active. This is called the *default* mode for that axis.

The ways in which each FGS differs is primarily in the number and type of modes that are present on the various aircraft. The engineers think of the modes as pluggable features; however, in reality there are dependencies among the choices of which modes the aircraft contains (e.g., every aircraft with mode x also has mode y) as well as subfamilies of the aircraft that contain the same collection of modes (e.g., every aircraft a in the sub-family A contains modes m_1, m_2, $\ldots m_n$).

V1 The set of modes along each axis varies from aircraft to aircraft

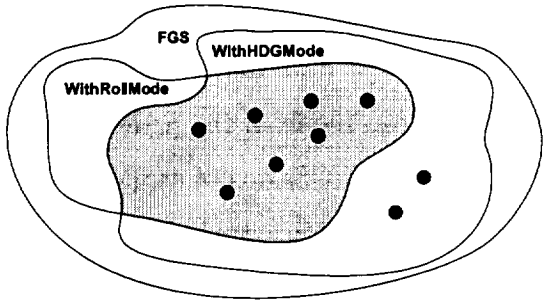

Figure 4.7: Example of sub-families of FGS

V2 The mode which each FGS designates as the default varies from aircraft to aircraft

V3 The FGS may or may not select the default mode for a particular axis upon transfer of flight guidance computations

Each mode, for example, Roll Mode can be viewed as defining a sub-family of FGSs that contain that mode. Thus, an FGS that contained Roll mode, Pitch Mode, and Heading mode would exist at the intersection of these three sub-families.

$C_{Roll}1E$ very Roll-subfamily FGS has a Roll Mode

$C_{Roll}2E$ very Roll-subfamily FGS uses a roll reference

$C_{Roll}3T$ he roll reference is synchronized when the SYNC switch is pressed with the Flight Director on

$V_{Roll}1$ There may or may not be a roll knob to adjust the roll reference.

$V_{Roll}1.1$ The roll knob may have a detent angle of 0, 5, or 6 degrees.

$V_{Roll}2$ The Roll/Heading transition angle can assume values of 5 or 6 degrees

In this way, each mode can be specified. Figure 4.7 shows a simple example of the intersection between the Roll and Heading modes. Although this example is purely hypothetical, we can see from the figure that most FGSs support both Roll and Heading mode while two FGSs do not support Roll mode. Notice that if only one FGS did not support Roll mode this would be a *near commonality*; in our structuring technique a near commonality is simply a special case of one sub-family having many less members than another sub-family.

If two modes must occur together in every FGS, then they can be specified in the same sub-family and that constraint can be noted as a commonality. This occurs between several lateral and vertical modes that must synchronize with each other. One example is the lateral and vertical go around modes.

$C_{GA}1E$ very GA-subfamily FGS has both a lateral and a vertical Go Around mode

$C_{GA}2T$ he lateral and vertical go around modes are always either both active or both cleared

$V_{GA}1$ The number and type of cockpit-located switched used to select go around mode varies from aircraft to aircraft.

We have tried with the FGS to give a partial picture of what the complexity of an industrial sized family might be like. We have discussed several modes of the FGS and hypothesized how the interaction between the modes of the FGS might be represented using our product family structuring approach. While it is unfortunate, that the entire FGS cannot be included here, it is over 100 pages long and, therefore, does not make a very illustrative example. Better illustrative examples are the ASW and the Mobile Robots, which are discussed in the next two sections.

4.4 Altitude Switch (ASW)

Recall that the ASW family is a simple collection of devices that all perform some action in response to changes in the altitude of the aircraft. The ASW is a much simpler example than the FGS, but it illustrates many concepts about the product family structuring approach. Furthermore, the ASW has the advantage that it may be presented here in full, and not abbreviated as the FGS.

4.4.1 Commonalities and Variabilities for the ASW

The ASW family consists of systems on board the aircraft that use the values from the various altimeters on board to make a choice among various options for actions (one of which being to do nothing) and perform the chosen action. Therefore, some high-level commonalities and variabilities are the following:

C1 All ASW systems will have a method of measuring the altitude of the aircraft

- C1.1 The ASW system will use the information about the aircraft's altitude to make a decision as to what action the ASW system shall perform

V1 The actions that the ASW takes in response to the altitude and the criteria to perform those actions varies from aircraft to aircraft

At this point, we have defined the ASW to be essentially a family of systems that process the altitude and then can perform some action based on the altitude that is measured. Of course, the ASW exists on board and aircraft of some kind and that aircraft will have a specified number and type of altimeters. This is noted in the following two variabilities.

V2 The number and type of Altimeters, devices that measure altitude, on board each aircraft may vary.

V2.1 Some altimeters provide a numeric measure of the altitude (digital altimeters) whereas some altimeters simply indicate whether or not the altitude is above or below a constant threshold which is determined when the altimeter is installed (analog altimeters).

Different manufacturers and/or different situations may dictate using different algorithms to process and threshold the altitude. This is noted in the following variabilities.

V3 In family members where there is more than one altimeter, a variety of smoothing and/or thresholding algorithms may be used on the *valid* altitudes [C2.1] to determine the estimated value for the true altitude or estimated value of whether or not the aircraft is truly above or below a certain threshold.

V3.1 Methods for choosing numeric altitude from several numeric sources will be mean, median, smallest, largest

V3.2 Methods for choosing whether or not the aircraft is above or below a certain threshold from a variety of altimeters which are either thresholded or numeric are any one above/below, all above/below, and majority above/below.

All the altimeters that are used on-board the aircraft are required to provide a measure of the validity of the measure. Furthermore, if the ASW cannot get a valid (or high enough precision) estimate of the altitude, it should declare that the system has failed. Therefore, we would like to record that fact as a commonality for the ASW family.

C2 All Altimeters will provide an indication of whether or not the supplied altitude is valid or not

C2.1 An altitude which is denoted to be *invalid* shall not be used in a computation to determine the action to be performed by the ASW

C2.2 If no altitude can be determined (i.e., all altimeters report invalid altitudes) for a specified period of time, then the ASW will declare that the system has failed. This period of time shall be constant for each family member (i.e., determined at specification time).

V4 The period of time that the altitude must be invalid before the ASW will declare a failure may vary between 2 seconds and 10 seconds from family member to family member.

In order for other devices on board the aircraft to know that the ASW has failed, the ASW must provide some kind of failure indication. Usually, this is done by having the system in question cease to strobe a watchdog output. If the watchdog is not present, then other devices on board the aircraft know that that piece of the system is no long functioning for some reason.

C3 All ASW systems will provide a failure indication to the environment.

C3.1 The indication that the ASW has failed will be the fact that the ASW has not strobed a watchdog timer within a specified amount of time. This period of time shall be a constant for each family member (i.e., known at specification time).

V5 The time interval with which the ASW must strobe the watchdog timer varies from aircraft to aircraft.

The ASW also accepts an inhibit and a reset signal. The inhibit signal should prevent the ASW from performing any action other than declaring a failure. The reset signal should return the ASW to its initial state.

C4 The ASW shall accept an inhibit signal. While inhibited, the ASW shall not attempt to perform any action other than declaring a failure.

C5 The ASW shall except a reset signal. When the reset signal is received, the ASW shall return to its initial state.

The ASW has several operating modes in addition to the normal one described above. The ASW should wait until receiving at least 5 seconds of valid altitude before performing any action.

C6 The ASW shall receive at least 5 seconds of valid altitude upon startup before entering normal operation.

Finally, the ASW has a reduced functionality mode that is activated when two episodes of invalid altitude lasting at least one second occur within a minute of each other. In the reduced functionality mode, if the ASW detects that an action should be performed, it shall wait for a minimum of two seconds before checking the conditions for action again. If, after that minimum delay, the conditions for action are still satisfied, then it will perform the action. However, if after six seconds the conditions are not satisfied then the ASW will discard that action and go back to waiting for the aircraft to cross the threshold.

C7 The ASW shall enter reduced functionality mode when two episodes of invalid altitude lasting at least one second occur within one minute of each other

C7.1 While in reduced functionality mode, the ASW will delay performing any action by a minimum delay period (2 seconds) at which time if the conditions for action are still satisfied the ASW will perform the action

C7.2 While in reduced functionality mode, the ASW will not wait to perform an action longer than the maximum delay time (6 seconds).

C7.3 The ASW shall exit the reduced functionality mode upon receipt of one minute of valid altitude data

As defined, the ASW system currently allows for almost any action to be performed as a result of the estimated altitude. A subfamily of the broad ASW family would be the class of ASW devices responsible for turning on or off a particular Device of Interest (DOI) on board the aircraft.

$C_{DOI}1T$ he ASW shall change the status (turn on or off) a Device of Interest (DOI) when it crosses a certain threshold

$V_{DOI}1T$ he threshold for the ASW varies from 0 to 8024 feet from aircraft to aircraft

$V_{DOI}2W$ hether the ASW turns on/off the DOI when passing above/below the threshold is a variability with nine possible choices (all combinations of do nothing, turn on, and turn off in the above and below directions).

To deal with noisy data, or the aircraft flying near to the threshold altitude, the DOI controlling ASW needs to have a certain hysteresis factor that is used to determine how much the altitude of the plane must change in order to have the DOI powered on or off again. The commonalities and variabilities that govern the hysteresis function of the ASW are given below.

$C_{DOI}2T$ he ASW shall employ a hysteresis factor to ensure that when the aircraft is flying at approximately the threshold altitude noisy data from the altimeters or slight variations in altitude do not cause the ASW to turn on/off the DOI in rapid succession

$V_{DOI}3T$ he hysteresis factor may vary from aircraft to aircraft between 50 ft and 500 ft.

$V_{DOI}4T$ he hysteresis factor may vary depending whether or not the aircraft is going above or below the threshold.

$C_{DOI}3Both$ the hysteresis factor for going above and the hysteresis factor for going below shall be a constant for each particular aircraft (i.e., known at specification time).

Finally, the ASW will received updates from the DOI whenever the status of the DOI changes. This is important to confirm whether or not the DOI is responding to the commands issued by the ASW as well as fulfill the requirement denoted by the final commonality.

$C_{DOI}4T$ he DOI shall give the ASW an indication of its status (on or off) whenever that status changes

$C_{DOI}5W$ henever the ASW submits a command to the DOI, it shall wait for a specified period of time for the status of the DOI to change to reflect the command. If the status does not change within the specified period of time, then the ASW shall declare a failure. The period of time will be a constant for each family member.

$V_{DOI}5T$ he period of time that the ASW will wait after issuing a command to the DOI before indicating a failure if the DOI does not change status shall vary between 1 second and 5 seconds from DOI to DOI.

$C_{DOI}6T$ he ASW shall not attempt to power on the DOI if the DOI is already on or attempt to power off the DOI if the DOI is already off.

As we have presented the commonalities and variabilities for the ASW, some of the structure of the ASW family is certainly visible. Nevertheless, the advantage of separating the structure from the commonalities and variabilities is primarily that the structure may be visualized independently. Some possible visualizations of the ASW family structure are presented in the next section.

4.4.2 Structure and Members of the ASW Family

Even for a family as small and simple as the ASW, we can identify elements of structure in the family. This identification is useful because it helps us to understand the family and it is invaluable if, in the future, we would like to refactor the family or incorporate the family as a part of a larger family. For example, we might like to have one family that encompasses all the avionics devices built (not just the ASW).

Dimensions of the family are used as a visualization technique to separate out the major choices of the family. Dividing a family into dimensions does not necessarily mean partitioning *all* the commonalities and variabilities of the family. For the ASW, we decided to concentrate on two primary dimensions when visualizing the structure: (1) the choice of the altitude smoothing and/or thresholding algorithm and (2) the major choice of functionality for the DOI. This decomposition is show in Figure 4.8. Of course, there are more dimensions to the ASW family, for example, the various types of altimeters might be considered a dimension.

Figure 4.8 depicts the various possible members of the ASW family. A notable property of the figure is that there are no family members currently that use the numeric altitude methods which we discussed in the commonalities and variabilities. This is because we have only looked at a small sub-family of the possible behaviors of the ASW family. In the future, we can envision adding all sorts of behaviors some of which might use the numeric methods.

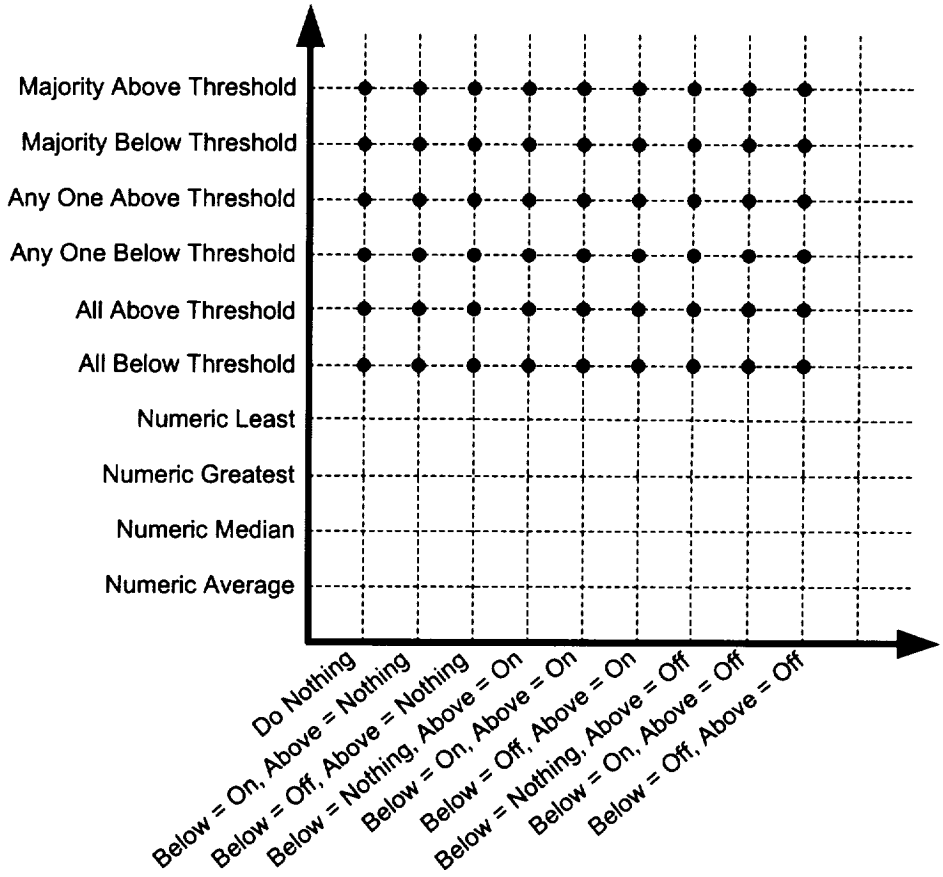

Figure 4.8: The ASW family structure visualized in 2 dimensions

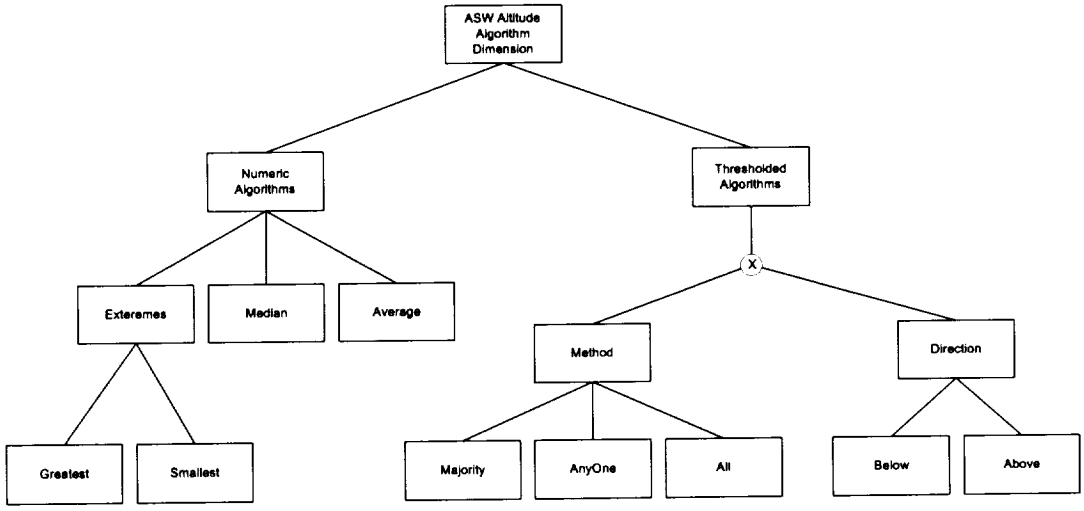

Figure 4.9: The structure of the Altitude Dimension for the ASW

The reader might note that the dimension of the family that shows the choice of smoothing or thresholding algorithms has some structure. That is, either the algorithm will have a numeric result and be a smoothing algorithm or it will have a boolean result and be a thresholding algorithm. This structure is visualized in Figure 4.9.

Visualizing the structure of the family in this way can be useful in developing a better understanding of the system. It may be that some commonalities should be made into more general statements and moved to the top-level family. Alternatively, you may discover that certain commonalities and variabilities may be closely tied to the current way of doing things and, thus, likely to change. These commonalities and variabilities may be isolated by placing them in a subfamily. This is a benefit of our structuring technique.

The decision model represents a recording of which choices for all the possible variabilities result in current family members. Obviously, the more complex the structure of the family, the more complex the decision model will be.

One way that the decision model can be written down is by simply noting which choices are made for each family member. For the ASW family, we have done that below for several ASW family members.

- **CS-123**: This aircraft has one analog and one digital altimeter, turns on the DOI when at least one altimeter is below 2000 feet, will not turn the DOI back on until going 200 ft above the threshold, has a timeout of 4 seconds for altitude staleness and 2 seconds for the DOI.

- **CS-134**: This aircraft has one analog and two digital altimeter, turns on the DOI when at least one altimeter is below 2000 feet, will not turn the DOI back on until going 200 ft above the threshold, has a timeout of 4 seconds for altitude staleness and 2 seconds for the DOI.

- **DD-123**: This aircraft has one analog and one digital altimeter, turns on the DOI when at least one altimeter is below 2000 feet, will not turn the DOI back on until going 250 ft above the threshold, has a timeout of 2 seconds for altitude staleness and 2 seconds for the DOI.

Even so, there are a number of disadvantages to listing the family member configurations in this way. First, it is difficult to tell whether all required variabilities have been given values. Second, it is difficult to see family members that have the same choices for the variability values. A tabular format is often used to represent the decision model. A tabular decision model for the ASW family members that we will consider in this methodology is presented in Figure 4.10.

In the next section, we describe the mobile robotics example, which better illustrates some the n-dimensional and hierarchical structuring of the product families.

Variability	CS-123	CS-134	DD-123	DD-134	EF-155
# of Analog Alt.	1	1	1	1	2
# of Digital Alt.	1	2	1	2	3
Threshold Algo.	Any	Any	Any	Majority	Majority
Invalid Alt. Failure	4 s	2 s	2 s	2 s	2 s
Threshold	2000 ft	2000 ft	2000 ft	2000 ft	1500 ft
Go Above Action	None	None	None	None	Turn Off
Go Below Action	Turn On	Turn On	Turn On	Turn On	Turn On
Go Above Hyst.	200 ft	200 ft	250 ft	200 ft	200 ft
Go Below Hyst.	NA	NA	NA	NA	200 ft
DOI timeout	2 s	2 s	2 s	2 s	2 s

Figure 4.10: A tabular representation of the ASW family decision model

4.5 Mobile Robotics

As mentioned in Chapter 3, the mobile robotics domain breaks down along two clear dimensions: the hardware platform and the desired behavior. This section describes in detail the commonalities and variabilities associated with these two dimensions and then presents an overview of how the structuring technique works on the family as a whole.

4.5.1 Hardware Dimension

Along the hardware dimension, we will consider a limited subset of the robot domain containing three families of hardware. The actual domain is much more complex; it includes many more types of sensors and different actuators, for example, a gripper or robotic arm that can be used to pickup and move objects in the environment. We

will consider the following three classes of mobile robotic hardware in this section.

1. A basic robot with forward and backward motion capabilities, a range sensor that give distance to the nearest obstacle and whether or not the obstacle is on the right or on the left, and a forward collision detection mechanism.
2. The basic robot with the ability to distinguish between obstacles that are straight ahead versus only the right or left (i.e., better granularity in the estimation of the obstacle's position).
3. The basic robot with the ability to distinguish the color of objects in its environment.

Basic platform: A basic feature of our robotic platform will be that it can move around its environment in some fashion. Thus a common feature of the robots is the following:

$C_H1.1$ Each platform will provide a basic means of locomotion; it will have the ability to turn a specified number of degrees from the initial heading, move forward, move backward, and stop.

Nevertheless, the robotic platforms that we will consider differ greatly. Some platforms are commercially built whereas others are built in-house, for example, out of Lego building blocks and small motors. The following variabilities capture these ideas:

$V_H1.3$ The hardware comprising the robotic platform varies

$V_H1.3a$ The means of locomotion may vary (e.g. treads, wheels, legs, etc.)

$V_H1.3b$ The maximum speed of the robot varies.

V_H1.3c The control of locomotion varies. The locomotion system may provide simple on/off values or real or digital valued representation of speed and direction.

V_H1.3d The type of input expected by the locomotion system varies. It may expect boolean, real, or digital values indicating speed and direction of the platform.

V_H1.3e The size of the platform varies. This will dictate the amount of room needed to turn or avoid an obstacle.

In order to avoid running into obstacles in the environment, the robot must have some kind of range finder. The platform must also be able to tell whether or not the obstacle is on the right or left so that it can take actions to avoid hitting the obstacle. However, range finders vary significantly in the type and quality of information they provide. For example, a sonar sensor provides a wide field of detection but is noisy and inaccurate. A laser range finder, on the other hand, will provide distance with high accuracy and can detect even small obstacles.

C_H1.2 All platforms will have at least one range finder that will provide input to the system regarding the detection of an obstacle.

C_H1.2a The range finder will provide an indication of the distance to the obstacle.

C_H1.2b The range finder will provide an indication of the location (right or left) of the obstacle in relation to the robot.

V_H1.1 The number and type of devices used for range finding is likely to vary. The type of output generated by the range finder varies. Different range finders

may provide output as a real-valued estimate, a digital estimate, or a boolean indication of obstacle detection.

Finally, because the mobile robots operate with such noisy and inaccurate sensors it is a certainty that they will occasionally have collisions. Thus, platforms must have a method of detecting collisions so that they can perform recovery actions in the behaviors. This could be implemented in a variety of different ways, for example, buy installing bumpers on the robot or by detecting that the motors that drive the wheels have stalled.

$C_H1.3$ All platforms will have at least one mechanism for detecting collisions.

$V_H1.2$ The number and type of collision sensors(s) varies and the type of output generated by the collision sensor varies.

Enhanced obstacle detection: Some platforms may have more advanced sensors to detect obstacles. For example, a robot with an array of sonar sensors arranged in an arc can get much more information about potential obstacles than merely whether they are on the right or on the left. For enhanced obstacle detection, the robotic platform should be able to detect whether or not it has an obstacle in front of it in addition to obstacles on the right and left.

$C_H2.1$ Platforms will have the ability to distinguish whether an obstacle exists directly in front of them as well as whether it is on the right or on the left. See related [$C_H1.2b$]

$V_H2.1$ The granularity of obstacle position detected will vary. For example, some platforms may provide an enumerated indication of left, right, or front for the obstacle whereas some may provide an estimated degrees to the obstacle.

This sensing capability allows the robot to perform more complex behaviors, for example, maneuvering closer to obstacles or going through doors.

Environmental vision: Some robots may be equipped with a camera or other sensing device that can give them information about the color objects in their environment. The type and quality of robotic vision systems varies greatly; however, most can distinguish between primary colors.

C_H3.1 Platforms will have a sensor capable of determining the color of objects in their environment; for example, the sensor should be able to distinguish between red objects and blue objects.

4.5.2 Behavioral Dimension

The behavioral dimension defines *what* the robot does. Of course, the behavior of the robot is highly related to the hardware dimension, which constrains what the robot *can* do and what information about the environment is available. Nevertheless, to a large extent the behaviors can and should be reused across different hardware platforms. The spectrum of behaviors possible, even with the limited hardware classes that we have defined, is large. For the purposes of this report, we only have space to discuss a few of them. Thus, along the behavioral dimension, we will consider the following classes of behavior.

1. Random exploration, where the robot moves around its environment attempting to avoid obstacles.
2. Random exploration with the ability to negotiate doors.
3. Random exploration with the ability to signal when it encounters objects of a particular color.

Random exploration: Rodney Brooks [14] recommends a layered architecture of robotic behaviors with a simple reactive behavior being on the lowest level and higher-level behaviors built on top of this. When the robot encounters a problem, for example, a collision, in a higher-level behavior, then the higher-level behavior is suspended by a lower-level behavior designed to correct the problem. Our approach to modeling the behavioral dimension is similar in that our basic behavior is a random environmental exploration and more complex behaviors are built on top of it. Note, however, that we have just chosen Brooks' subsumption architecture as an example and that we could have easily chosen another method of structuring the behavioral method. The real point is that the two dimensions of the mobile robotics system should be able to be structured independently.

Our basic behavior is a random exploration; while exploring, the robot should attempt to avoid obstacles in the environment.

$C_B1.1$ The robot shall attempt to avoid colliding with obstacles in its environment using its sensors to detect obstacle(s) and changing its course or speed to avoid the obstacle.

$V_B1.1$ Although detected by the robot's sensors, an object may or may not be considered an obstacle depending on the robot's mode of operation. See, for example, [$C_B2.1a$]

As mentioned previously, because of the robot's noisy and inaccurate sensors it is likely that the robot will sometimes collide with an obstacle. When this occurs, the robot should attempt to recover from the collision and continue exploration.

$C_B1.2$ If the robot collides with an obstacle, it shall attempt to recover from the collision.

V1.2 Successive collisions (i.e., a collision during the recovery from a previous collision) may result in the robot shutting down all activity and declaring failure. The number collisions in a chain that the robot can tolerate varies.

The random exploration behavior coexists with all the other possible behaviors that we might define. In the absence of any obstacle or collision, the robot will potentially be performing some other functions which are defined by a subfamily. However, this family is *not* abstract; thus, if no other behaviors are specified the robot will move forward at full speed.

V_B1.3 In the absence of an obstacle or collision, the behavior of the robot may be further specified by a sub-family

C_B1.3 In the absence of an obstacle, collision, or any other specified behavior, the robot will move forward at maximum speed.

Door navigation: Maneuvering through a doorway is difficult for a mobile robot. Often, obstacle detection sensors provide little information about the environment; thus, doorways are often not seen as viable passageways. Furthermore, it is difficult for the robot to find doorways in the first place given the noisy sensor data it receives.

C_B2.1 The robot shall attempt to locate doors in its environment

C_B2.1a Once the robot has found what it believes to be a door, it shall not consider the sides of the door to be obstacles as the door is navigated. See [C_B1.1], [V_B1.1].

V_B2.1 The width of the door which can be navigated by the robot will vary according to the width of the robotic platform and the quality of the on-board sensors.

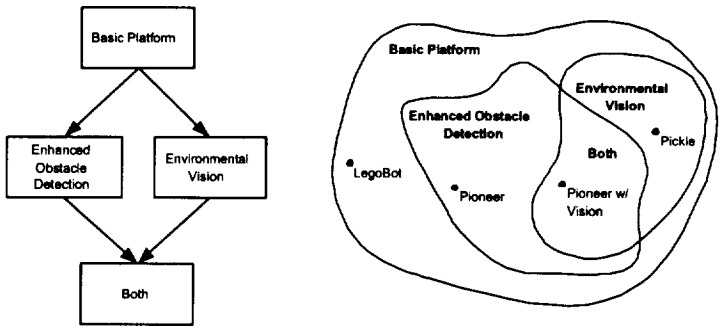

Figure 4.11: The mobile robot family along the hardware dimension

Environmental interpretation: This behavior allows the robot to signal when it encounters a particular object in the environment. That object or objects will be identified by a particular color.

C_B3.1 The robot will signal when it has detected an object in its environment of the desired color.

V_B3.1 The color of the object(s) to be detected will vary and may be configurable at run time.

4.5.3 The Whole Family

The real mobile robotics domain is significantly more complex than space allows us to present in this dissertation. For example, we have not discussed whether or not the robot can move objects in its environment (with a gripper, for example). Nevertheless, we can illustrate some interesting properties of the domain even with this limited example.

There are several ways of visualizing the mobile robot product family. First, we will examine the mobile robot family along the hardware dimension (Figure 4.11). Notice that family members can fall into one of four different categories. The robot

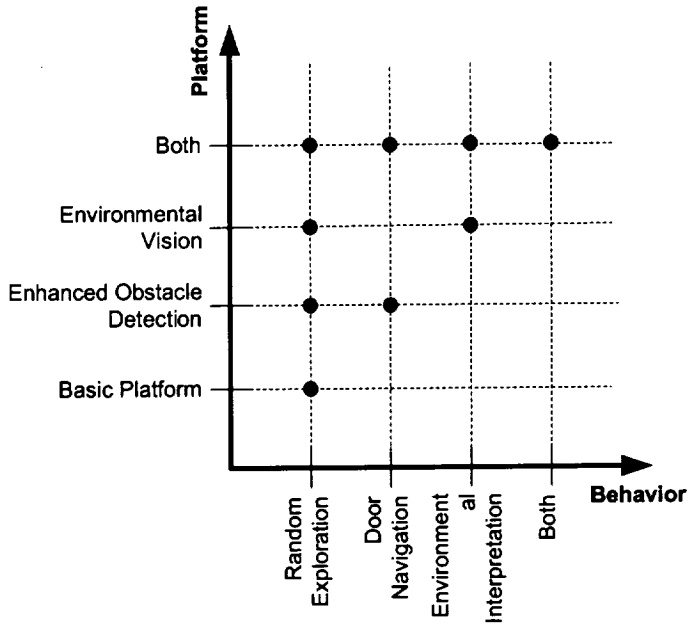

Figure 4.12: A possible 2-dimensional view of the robot product-line

may have only the basic capability, in which case it exists only for the family Basic Platform. This is the case for LegoBot. The robot may have either one or the other of the additional hardware capabilities specified by the Enhanced Obstacle Detection or Environmental Vision. Finally, the robot may posses both the additional capabilities of Enhanced Obstacle Detection and Environmental Vision; therefore, it lies in the intersection of those two subfamilies. This is only one slice of the system, however, and if we were to look at the mobile robot family along the behavioral dimension we would see a similar picture. A somewhat more effective means of viewing 2-dimensional product family is in a 2-dimensional grid as shown in Figure 4.12.

The representation is symmetrical in this case because of the one-to-one mapping between behavioral subfamilies and hardware subfamilies. The full mobile robotic domain, however, is not symmetrical. In the full domain, behaviors may be composed

and combined to form a composite behavior. For example, we might envision a behavior which includes the door navigation, combined with a mapping function, a wall following behavior, and a high-level planner. The mapping and high-level planning behaviors will need to communicate with the lower level random exploration, door navigation, and wall following to direct the robot towards high-level goals. However, if the robot collides with an obstacle, then the lower level behavior will take over and recover from the collision. Thus the structuring of the behavioral dimension is much more complex and resembles Brooks' subsumptive architecture [14]. Furthermore, defining the behaviors independent of the hardware allows us to focus on only the behaviors and their interactions (a significant problem in an of itself).

These combinations of behaviors might require several different sets in the hardware domain, which will have sub-families that define, for example, robots with grippers, robots with bumpers, robots that have radio communications devices, and so forth. Thus, it is generally not the case in the full domain that a behavior will require exactly one subset in the hardware dimension or that the behavior and hardware dimensions have the same structure. By defining the intersection of the hardware dimension with the behavioral dimension, we define which family members are viable and which are not.

The division of the system into behavioral and hardware dimensions is a classical one which; however, these are not the only two dimensions possible. For instance, performance, for example, battery life, might be modeled as a separate dimension of the system.

4.6 Evaluation and Summary

The structuring technique presented results in the creation of more families within the domain than with a traditional approach. However, these sub-families are more

cohesive and simpler than would be the case if we created just one top level-family. We believe that this provides several benefits. First, the top-level family can now be much broader than was previously possible. Second, the overall family can be expanded and contracted by adding and subtracting sub-families. Finally, these techniques will allow a family to be more easily refactored as the definition of the family evolves over time.

The ability to draw a larger product family was an essential requirement for the structuring technique. This grows out of our own experiences with mobile robotics [22, 106], where we had difficulty in applying the product family approach. This difficulty stems from the fact that the mobile robotics domain is both *n-dimensional* and *hierarchical.*

The mobile robotics domain breaks down along two clear dimensions: the hardware platform and the desired behavior. Each hardware platform conforms to a basic specification: it can move forward and backward, turn left and right, sense whether or not an object is in front of it. The hardware platform may also be equipped with a variety of sensors and actuators that give it additional capabilities; and, the various sensors differ greatly in the speed and accuracy with which they provide information. Thus, on the hardware side, there are many different configurations that must be modeled.

On the behavior side, we can imagine that a basic behavior might be a random exploration where the primary goal of the robot is collision avoidance and recovery. More complex behaviors can be added, for example, wall following, going through doors, and finding particular objects. Furthermore, those behaviors may be composed and combined to form a composite behavior. We might envision a behavior which includes the door navigation, a wall following behavior, and a high-level planner. The high-level planning behavior needs to communicate with the random ex-

ploration, door navigation, and wall following to direct the robot towards high-level goals. However, if the robot collides with an obstacle, then the lower level behavior will take over and recover from the collision. Thus structure of the behavioral dimension is much different from the hardware dimension and resembles Brooks' subsumptive architecture [14].

Certainly, a domain such as mobile robotics which absolutely requires n-dimensional and hierarchical product families will necessarily be more complex than a domain that does not require these techniques. Nevertheless, any domain can benefit from reuse of the artifacts at the top of the family hierarchy and a more traditional cost-benefit will exist towards the leaves of the family (along each particular dimension). Even in a domain such as the ASW or the FGS, the requirements benefit from the ability to clearly separate the concerns of the various modes and denote constraints specifically to when two modes occur together.

Another benefit of the technique is the ability to expand and contract the family as necessary. For example, suppose that we discover that we have a new kind of DOI which has three states instead of two (e.g., Off, Low, and High). Clearly, our commonalities and variabilities are oriented towards a DOI that is either On or Off; nevertheless, this new DOI will share much in common with the two state DOI. To accommodate this change, commonality [$C_{DOI}1$] and variability [$V_{DOI}2$] will need to be updated to reflect the larger DOI family. Then, we may define two subfamilies of the larger DOI family – one for two-state DOI and one for three-state DOI; or, we may choose to model an n-state DOI. In any event, the vast majority of the ASW family specification will be isolated by the structure that we have chosen for the family.

This ability to redraw and rework pieces of the commonality analysis while being confident of not affecting other parts of it is essential because it allows a more

incremental development of product-lines than is facilitated by current approaches. Furthermore, it facilitates *family refactoring*; that is, the family can be redefined more easily as the product line evolves over time. Thus, this structuring technique has much potential to increase the usefulness of the product family approach.

One of the barriers to traditional product family approaches is that the whole organization must change to accommodate product-line oriented development. Many resources are required to develop the domain engineering support for the entire product line while at the same time continuing to produce products for existing customers. Our approach allows an organization to start out with a high-level product family and reuse just a few key pieces between the major product areas. As the payoff from this reuse makes more organizations resources available, the organization can then afford to make the family more rich (by refactoring and/or adding sub-families) and thus achieving more payoff from the effort.

Of course, these benefits do not come for free. The broader and more flexible view of product families allowed by our techniques will result in families which are more complex than traditional families. In addition, because of this broader view, it may be more difficult to determine what constitutes a viable family under our approach. Almost anything is related in some fashion or other and it may be difficult for organizations to decide when to define an encompassing family for a particular group of subfamilies. Nevertheless, we feel that these techniques hold promise and may serve to advance the frontiers of product-line engineering.

The cost-benefit analysis of our product-line engineering approach is more difficult because one must not only consider the cost of developing domain engineering support of the particular sub-family in which the member resides, but also all sub-families above that one in the product family hierarchy. Suppose that we wanted to build a family of FGS systems for both fixed-wing aircraft and helicopters. The cost-

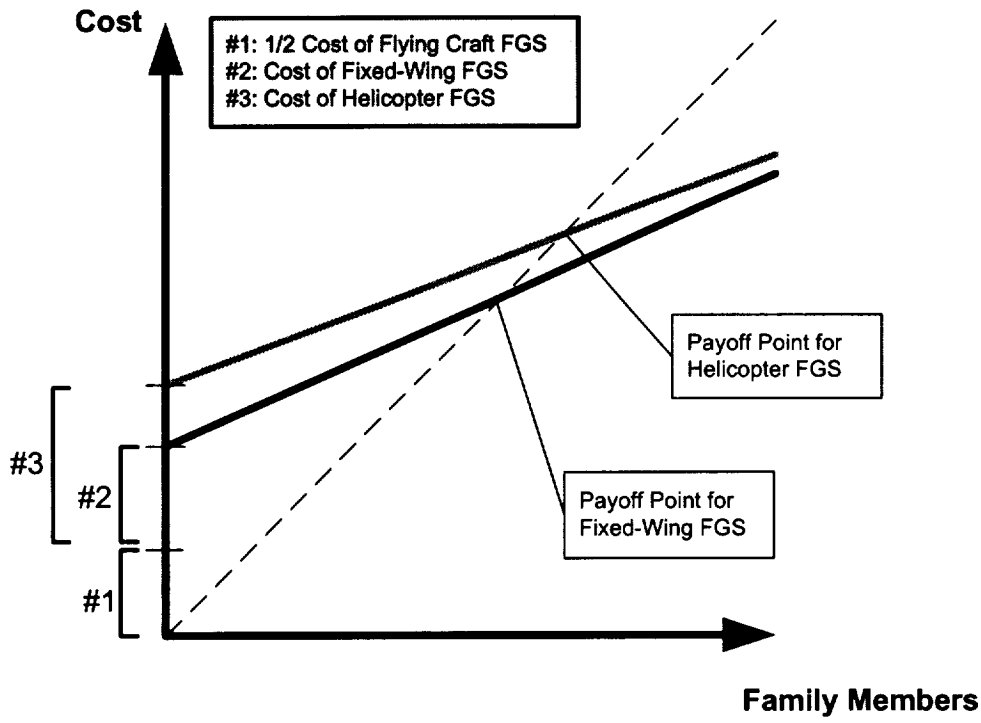

Figure 4.13: Cost-benefit of the FGS Family

benefit analysis for this family is shown in Figure 4.13. To build either a fixed-wing aircraft or a helicopter, we must have built the assets in the Flying Craft FGS family; therefore, we can amortize the cost of the Flying Craft FGS over both the fixed-wing and helicopter families. This is the cost #1 in the figure. Next, if we want to build the assets for the fixed wing family, we must spend some additional amount over an above the shared cost for the Flying Craft FGS. This is noted by the cost #2 in the figure. Once we know what both of these costs are, we can determine how many fixed-wing FGS systems we must build in order for the family development effort to be justified. However, the cost of building the helicopter assets may well be *different* from the cost of building the fixed-wind assets (this is cost #3 on the figure). Therefore, if the helicopter assets are more expensive to construct we will have to build more of the helicopter FGS members to justify the costs. As the structure of the family becomes more complex, for example, through the creation of a deeper hierarchies and/or the use of multiple dimensions with constraints between them, this relationship will become more complex. This dissertation does not address how to perform a cost-benefit analysis in the most complex scenarios, but it is a topic that should be addressed in the future.

In this chapter, we have taken a look at the structures that are present in many product families and given our own approach to representing that structure in a usable way, illustrated with examples from the ASW, FGS, and mobile robotics families. This chapter provided one of the major building blocks for the methodology. The next chapter will discuss the other major building block for the methodology, the work that was done on specification-based prototyping. These two building blocks will be tied together in Chapter 6.

Chapter 5

Methodology Foundations

The goal of this chapter is to provide a framework for the methodology that is described in Chapter 6. Much of the work presented in here is based on our work with specification-based prototyping [109, 110, 108] and the NIMBUS environment [105, 104]. This chapter is also based on Miller's extended four-variable model [83], which was developed in collaboration with the work at the University of Minnesota.

As discussed in Chapter 2, the system requirements should always be expressed in terms of the physical process. These requirements are determined by the need to change the world in which the system operates and are represented by the REQ relation. The IN and OUT relations are determined by the sensors and actuators used in the system. For example, to measure the altitude we may use a radio altimeter providing the measured altitude as an integer value. Similarly, to turn on a device, a certain code may have to be transmitted over a serial line. Armed with the REQ, IN, and OUT relations we can derive the SOFT relation.

All of these relations are likely to change over the lifetime of the controller. Furthermore, the sensors and actuators are likely to change independently of the requirements as new hardware becomes available or the software is used in subtly different operating environments. If any one of the REQ, IN, or OUT relations changes, the SOFT relation must be modified. What is needed is to provide a smooth transition from system requirements (REQ) to software requirements (SOFT) and to isolate the impact of requirements, sensor, and actuator changes.

The question is, how shall we do this and how shall we structure the SOFT relation? Our results in this area are presented in the next section, where we discuss how to structure the SOFT relation, and in the section after that, where we discuss the overall process to be used in refining the requirements. Finally, we explain how formal languages and tools greatly benefit this system model and describe briefly describe results that we have achieved with the research toolset at the University of Minnesota.

5.1 The FORM$_{PCS}$ System Model

This section introduces the FORM$_{PCS}$ system model, which is essentially an extended version of the four variable model that was presented in Chapter 2.

There are several variations of the four-variable model that one could imagine might be useful on occasion. For example, it may be helpful to layer the IN and OUT relations into levels much like the ISO Reference Model for communication protocols. Another variation is to "glue" the controlled variables of one or more models to the monitored variables of another model to create a larger system specification or to split a large model up into several smaller models (although care must be taken not to fall into the trap of introducing implementation bias).

The traditional four variable model leaves the software developer with the question of how to structure an implementation of SOFT, i.e., how to design the software. One appealing approach is to "stretch" the SOFT into the relations IN', REQ', and OUT' as shown in Figure 5.1 [83]. IN' takes the measured input and reconstructs an estimate of the physical quantities in MON. The OUT' relation maps the internal representation of the controlled variables to the output needed for the actuators to manipulate the actual controlled variables. Given the IN' and OUT' relations, the REQ' relation will now be essentially isomorphic to the REQ relation and, thus, be

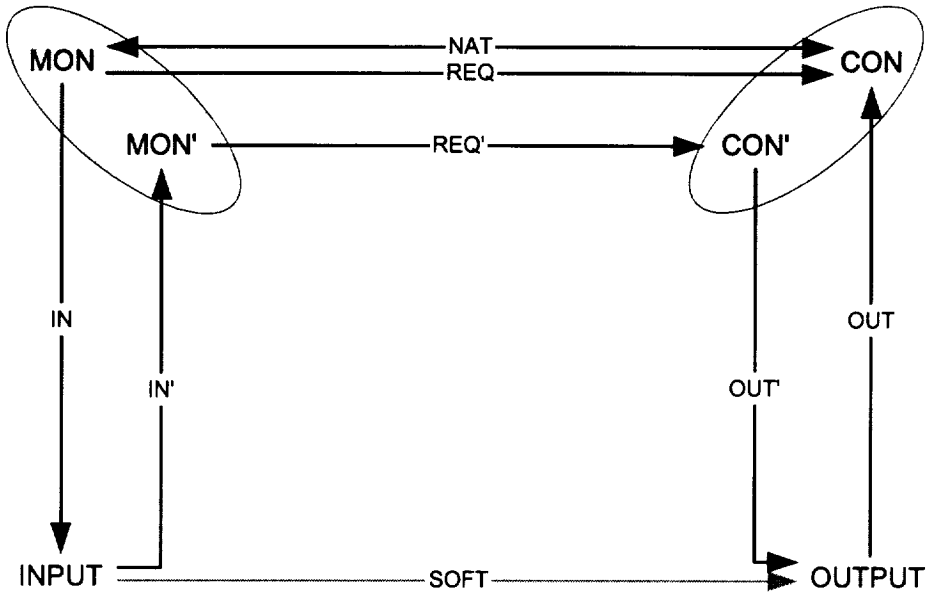

Figure 5.1: The FORM$_{PCS}$ system model adapted from [83, 109]

robust in the face of likely changes to the IN and OUT relations (sensor and actuator changes). This conceptual view creates a virtual image of the MON and the CON variables in software, an approach often advocated in object-oriented design methods.

Decomposing the software in this way has several benefits. First, if MON and CON are chosen correctly, the portion of the software specified by IN' will change only as the input hardware changes. Likewise, the portion of the software specified by OUT' will change only as the output hardware changes. In a similar fashion, the portion of the software specified by REQ' will be isolated from hardware changes and will change only in response to changes in REQ, the system requirements. Since customer driven changes and hardware driven changes arise for different reasons, this helps to make the software more robust in the face of change. It also greatly simplifies tracing the system requirements to the software requirements.

Of course, it is important to note that MON' and CON' are not the same as the system level variables represented by MON and CON. This is highlighted in the figure. Small differences in value are introduced both by the hardware and the software. Differences in timing are introduced when sensing and setting the input and output variables. For example, the value of an aircraft's altitude created in software is always going to lag behind and differ somewhat from the aircraft's true altitude. In safety-critical applications, the existence of these differences must be taken into account. However, if they are well within the tolerances of the system, the paradigm of Figure 5.1 provides a natural conceptual model relating the system and the software requirements. This directly addresses the issue of integrating systems and software engineering.

Nevertheless, even armed with this technique for structuring the SOFT relation, it is not clear how exactly to proceed. This topic of how the refinement process should be organized is discussed in the next section.

5.2 The FORM$_{PCS}$ Process Framework

The previous section described an overall structure for the SOFT relation. This section describes a process for deriving the SOFT relation given the REQ relation. The specification starts as a high-level model of the *system requirements* (i.e., the REQ relation). This model is is then iteratively refined, adding more detail as the system becomes better understood. During each iteration, if a formal, executable specification language is used, the specification is executable and can therefore be used as the prototype of the proposed system. Eventually, the system requirements will be well-defined and the system engineer must allocate requirements to particular hardware and software components within the system. At that point, the *system requirements* can be refined to the *software requirements* by adding descriptions pertaining to the

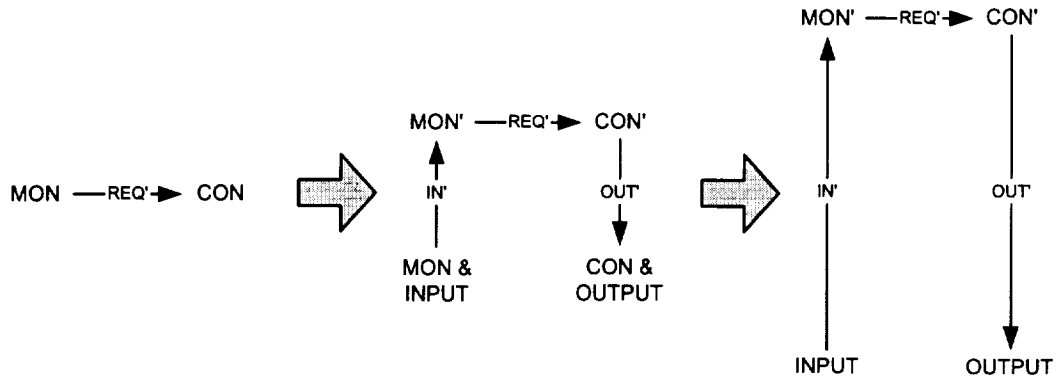

Figure 5.2: Refining REQ to SOFT

actual hardware with which the software must interact.

From the start of the modeling effort, we know that we will not be able to directly access the monitored and controlled variables—we must use sensors and actuators. At this early stage, we may not know exactly what hardware will be used for sensors and actuators; but, we do know that we must use something and we may as well prepare for it. By simply encapsulating the monitored and controlled variables we can get a model that is essentially isomorphic to the requirements model; the only difference is that this model is more suited for the refinement steps that will follow as the surrounding system is completed.

The method of this encapsulation differs depending on the language used. If the language does not have a modularity construct, then extra variables or functions can be introduced in the specification to isolate the REQ' behavior from the hardware specification. If the language does have a modularity construct, the specifier may choose to define a module that computes the REQ' relation and then the module's interface naturally provides the encapsulation.

As the hardware components of the system are defined (either developed in house

or procured), the IN and OUT relations can be rigorously specified. Figure 5.2 shows a high-level view of the refinement process. At the far left of the figure, we start the process with just a notion of the REQ' relation and evaluate REQ' with the monitored and controlled variables (we basically assume that the sensors and actuators are perfect–there are no delays or noise). Next, we move into an intermediate stage as we add more and more detail to the IN' and OUT' relations. During this stage, the specifications for some sensors and actuators might be completely finished while the specifications of others are under development; this is the reason that both MON and INPUT are noted as the sources for the IN' relation (and similarly for the OUT' relation). Finally, we will arrive at a complete specification of both the IN' and OUT' relations, shown at the far right of the figure.

We have shown in the abstract how the SOFT relation should be structured and our conception of the process that should be used to refine the REQ relation to the SOFT relation. In the next sections, we illustrate this approach by applying it to the ASW and the Mobile Robotics examples.

5.2.1 The ASW Example

This section describes the overall process of how the ASW specification was developed, using examples taken from the specification. In the next chapter, we go into more detail about exactly how this was done as we discuss the methodology itself.

We identified the aircraft altitude as one monitored variable and the commands that the ASW sends to the device of interest as a controlled variable. Both are clearly concepts in the physical world, and thus suitable candidates as monitored and controlled variables for the requirements model. In our case, using a function, *MeasuredAltitude*(), instead of the monitored variable *Altitude* will shield the specification from possible changes in how the altitude measure is delivered to the software.

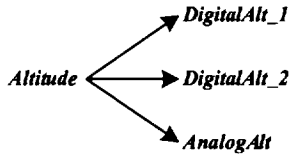

Figure 5.3: The true altitude is mapped to three software inputs.

By performing this encapsulation for all monitored and controlled variables we refine REQ to REQ', a mapping from estimates of the monitored variables to an internal representation of the controlled variables.

The IN and OUT models represent our assumptions about how the sensors and actuators operate. In this version of the altitude switch we will use one analog and two digital altimeters. Thus, we will map the true altitude in the physical world to three software inputs (Figure 5.3).

In the case of the digital altimeter, the altitude will be reported over an ARINC-429 low speed bus as a signed floating point value that represents the altitude as a fraction of 8,192 ft. If we ignore inaccuracies introduced in the altitude measure and problems caused by the limited resolution of the ARINC-429 word, the transfer function for the digital altitude measures can be defined as

$$DigitalAlt = \frac{Altitude}{8192}$$

The analog altimeter operates in a completely different way. Due to considerations of cost and simplicity of construction, the analog altimeter does not provide an actual altitude value, only a Boolean indication if the measured altitude is above or below a hardwired threshold (defined to be the same as the one required in the altitude switch). Assuming again an ideal measure of the true altitude, the transfer function for the analog altimeter could be modeled as

$$AnalogAlt = \begin{cases} Above & \textbf{if } Altitude > \text{Threshold} \\ Below & \textbf{if } Altitude \leq \text{Threshold} \end{cases}$$

In addition, all three altimeters provide an indication regarding the quality of the altitude measures.

With the information about the sensor (IN) and actuator (OUT) relations, we can start refining the REQ' relation towards SOFT. In our case we must model, among other things, the three sources of altitude information and fuse them to one estimate whether we are above or below the threshold altitude. To achieve this, we refine the IN' relation in our model.

In the refined model, internal models of the perceived state of the sensors have been included in the state machine as a representation of the IN' relation for Altitude. Instead of the idealistic true altitude used when evaluating REQ, the specification now takes two digital altitude measures and one analog estimate of the altitude as input. This *BelowThreshold*() macro is shown in Figure 5.4. Note that the figure uses a simple RSML^{-e} construct for expressing Boolean conditions in disjunctive normal form. A much more detailed introduction to RSML^{-e} is provided later in Chapter 7. Thanks to the structuring of the SOFT relation, this refinement could be done with minimal changes to the REQ' relation As the components in the environment are developed, this process will be repeated for all inputs and outputs until a detailed definition of the SOFT relation is derived.

5.2.2 The Mobile Robotics Example

The mobile robotics domain's dimensions coincide with the breakdown of the SOFT relation into IN', OUT', and REQ' – those sections of the relation dealing with the platform are confined to IN' and OUT' while those sections dealing with the behavior

Macro

BelowThreshold

Parameters: NONE

Condition:

MeasuredDigitalAlt(DigitalAlt1) <= AltitudeThreshold	T	•	•
DigitalAltimeter1_Ok()	T	•	•
MeasuredDigitalAlt(DigitalAlt2) <= AltitudeThreshold	•	T	•
DigitalAltimeter2_Ok()	•	T	•
AnalogAltitudeMeasure() = Below	•	•	T
AnalogAltimeter_Ok()	•	•	T

Figure 5.4: Macro modified to handle the tree inputs instead of the true altitude as it did in the REQ model.

are primarily confined to REQ. Therefore, in the mobile robotics domain we will have a number of different REQ relations, one for each behavior, and an IN' and OUT' relations, one set per platform. In this section we will consider only the random exploration behavior and concentrate on the differences between the platforms.

We begin with a short discussion of the REQ relation. To capture the desired behavior we must discover monitored and controlled variables in the environment that will allow us to build the formal model. In addition, while evaluating candidates for monitored and controlled variables we must keep in mind that the REQ model shall apply to all members of the product family.

We identified a robot's *speed* and *heading* as controlled variables. *Speed* ranges from 0 to 100 and can be mapped into a speed for each family member using the maximum speed of the particular robot. *Heading* ranges from -180 to 180 and indicates the number of degrees that the robot may have to turn to avoid an obstacle.

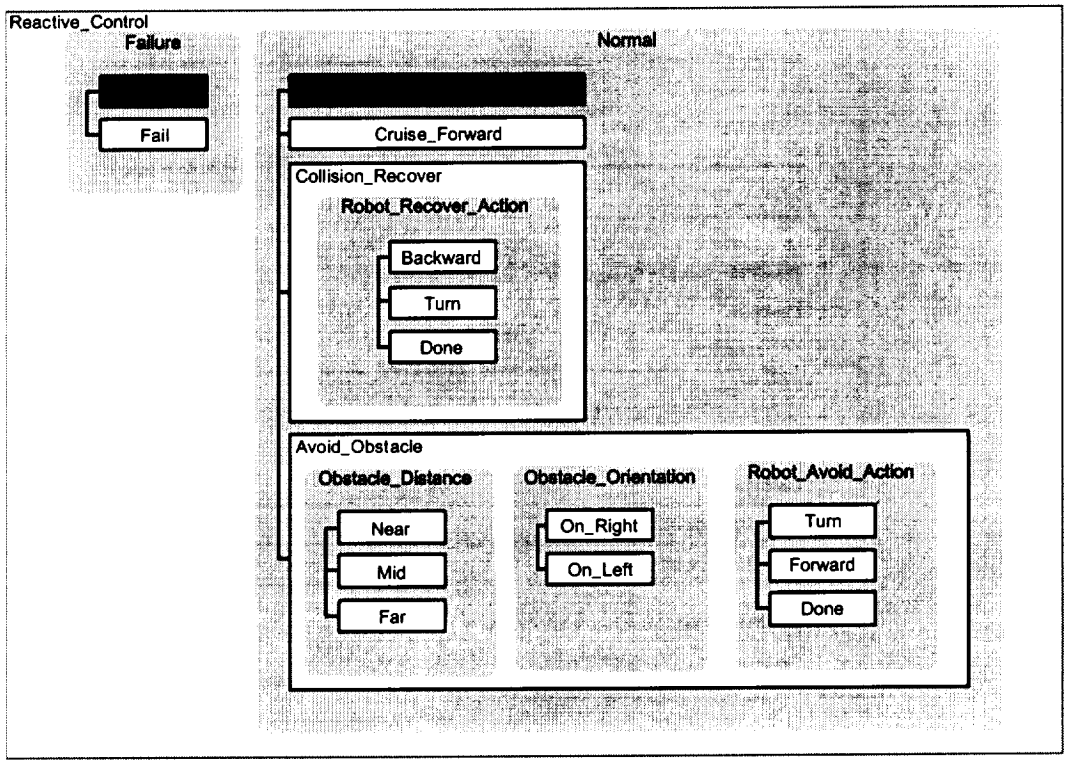

Figure 5.5: Mobile Robotics platform Random Exploration REQ relation

We identified *Collision*, *Range*, and *ObstacleOrientation* as monitored variables. The *Collision* variable is simply a Boolean value which is true when there is a collision and false otherwise. The *Range* variable is the distance from the robot to the nearest obstacle and the *ObstacleOrientation* denotes whether the obstacle is on the right or left of the robot. These variables clearly reside in the system domain and are sufficient to model the desired behavior. If the monitored and controlled variables are chosen appropriately, the specification of the REQ relation will be focused on the issues which are *central* to the requirements on the system.

Figure 5.5 shows that the REQ relation definition at the top level is split between two state variables: *Failure* and *Normal*. The *Failure* state variable encapsulates the

State Variable

Normal

Location: Reactive_Control

Transition: Startup→Cruise_Forward

Condition:

TIME > 2 s	T
..Failure IN_STATE Ok	T

Transition: Cruise_Forward→Collision_Recover

Condition:

CollisionDetectedMacro() = TRUE	T
..Failure IN_STATE Ok	T

Transition: Cruise_Forward →Avoid_Obstacle

Condition:

ObstacleDetectedMacro() = TRUE	T
CollisionDetectedMacro() = FALSE	T
..Failure IN_STATE Ok	T

Transition: Collision_Recover →Cruise_Forward

Condition:

Prev_Step(..Robot_Recover_Action IN_STATE Done)	T
..Failure IN_STATE Ok	T

Transition: Avoid_Obstacle →Cruise_Forward

Condition:

Prev_Step(..Robot_Avoid_Action IN_STATE Done)	T
..Failure IN_STATE Ok	T

Transition: Avoid_Obstacle →Collision_Recover

Condition:

CollisionDetectedMacro() = TRUE	T
..Failure IN_STATE Ok	T

Figure 5.6: The definition of the *Normal* state variable

failure conditions of the REQ relation, whereas the *Normal* state variable describes the how the robot transitions between the various high-level behaviors discussed at the introduction to this section (obstacle avoidance, collision recovery, etc.). For the remainder of our discussion of REQ, we will focus on the *Normal* state variable where this aspect of the requirements is captured (Figure 5.6).

The *Normal* variable defaults to the *Startup* value. This allows the specification to perform various initialization tasks and checks before the main behavior takes over. The first transition in Figure 5.6 states that after two seconds, the specification will enter the *Cruise_Forward* state.

The next two transitions govern the way that the *Normal* state variable can change from the *Cruise_Forward* value. If a collision is detected, then the state variable changes to the *Collision_Recover* state. If an obstacle is detected, then the specification will enter the *Avoid_Obstacle* state. Otherwise, the value of the *Normal* state variable will remain unchanged.

The *Cruise_Forward* behavior becomes active after the avoidance/recovery action has been completed. We accomplished this in the mobile robotics specification by providing a "done" state in each of the sub-behaviors. This is illustrated by the fifth and sixth transitions in Figure 5.6.

Finally, it is also possible to transition from *Avoid_Obstacle* directly to *Collision_Recover* if, for example, the robot hits an undetected obstacle; this case is covered by the final transition in Figure 5.6.

Given this definition of the REQ relation high-level behaviors, the definitions of the sub-behaviors can be constructed in a similar and straightforward manner. For example, if the robot hits an obstacle, it will attempt to back up, turn, and then proceed forward again. This behavior is specified in the *Robot_Recover_Action* state variable by having the variable cycle though the values *Backward*, *Turn*, and finally

Done.

When refining the specification from REQ to SOFT, we select the sensors and actuators that will supply the software with information about the environment, that is, we must select the hardware and define the IN and OUT relations for each platform. Consequently, we will also need to define the IN' and OUT' for each platform.

5.2.3 Process Summary

The structuring techniques above can be applied to virtually any specification language. Nevertheless, to realize the full potential of a prototyping-style development process, it is necessary to allow the analyst to fully evaluate the specification after each iteration (i.e., perform a risk assessment). Therefore, a language which has a formal semantics and can be analyzed and executed during each iteration will have significant benefits over others that do not. This process, called specification-based prototyping, is discussed in detail in the next section.

5.3 Languages and Tools to Support FORM$_{PCS}$

The capability to dynamically analyze, or execute, the description of a software system early in a project has many advantages: it helps the analyst to evaluate and address poorly understood aspects of a design, improves communication between the different parties involved in development, allows empirical evaluation of design alternatives, and is one of the more feasible ways of validating a system's behavior.

Thus, in order to fully realize the benefits from specification-based prototyping two important criteria must be satisfied. First, one must have a language suitable for expressing the high-level system requirements and refinement to the detailed software requirements. Second, such a language must have a formal semantics and be supported by tools which allow static analysis and execution. Furthermore, in our

work, we discovered that sophisticated execution capabilities (e.g., hardware-in-the-loop simulation) can have great benefits. These execution capabilities are provided by an environment that we call NIMBUS [104, 105, 109] that was developed as a testbed for specification-based prototyping. This section describes these tools and the rational behind their construction.

Specification-based prototyping combines the advantages of traditional formal specifications (e.g., preciseness and analyzability) with the advantages of rapid prototyping (e.g., risk management and early end-user involvement). The approach lets us refine a formal executable model of the *system requirements* to a detailed model of the *software requirements*. Throughout this refinement process, the specification is used as an early prototype of the proposed software. By using the specification as the prototype, most of the problems that plague traditional code-based prototyping disappear. First, the formal specification will always be consistent with the behavior of the prototype (excluding real-time response) and the specification is, by definition, updated as the prototype evolves. Second, the risk of evolving the prototype into a poorly designed production system is largely eliminated. Finally, the dynamic evaluation of the prototype can be augmented with formal analysis.

When starting to develop NIMBUS, we identified the following fundamental properties such an environment must possess. First, it must support the execution of the specification while interacting with accurate models of the components in the surrounding environment, be they RSML^{-e} specifications, numerical simulations, statistical models, or physical hardware. Second, the environment must allow an analyst to easily modify and interchange the models of the components. Third, as the specification is being refined to a design and finally production code, there should not be any large conceptual leaps in the way in which the control software communicates with the environment.

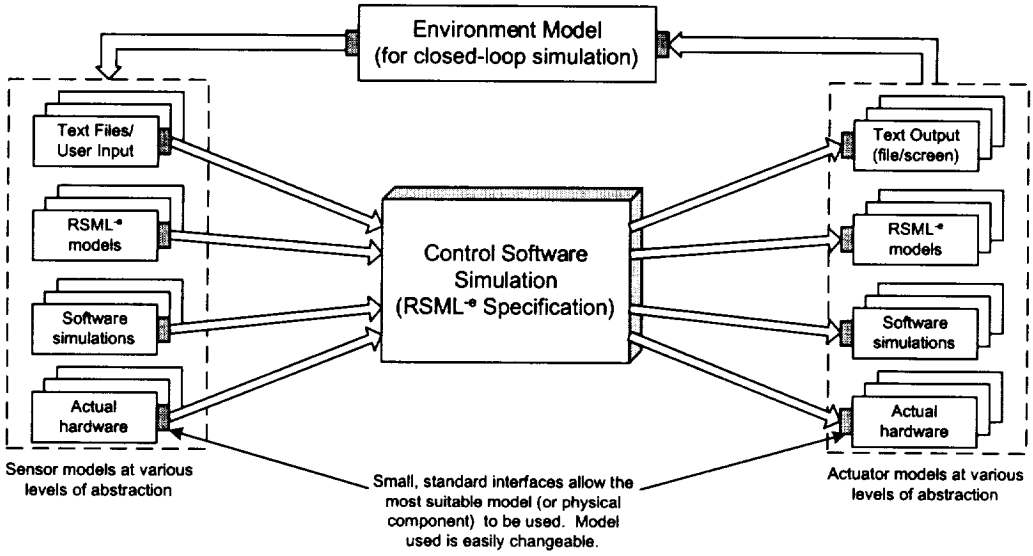

Figure 5.7: The NIMBUS Environment

In the initial stages of the project, we want the executions to take their input from simple models, for example, text files or user input. As the specification is refined, the analyst can add more detailed models of the sensors and actuators, for example, additional RSML^{-e} specifications or software simulations. In order to have a closed loop simulation, a model of the environment can be added between the sensor and actuator models. Finally, when the specification has been refined to the point of defining the hardware interfaces, the analyst can execute it directly with the hardware. This hardware-in-the-loop simulation closes the gap between the prototype and the actual hardware. These ideas are illustrated in Figure 5.7.

In NIMBUS, the interfaces are connected via *channels*. A channel (in this context) simply means the way in which data is passed from an interface in one component to the interface in another component. Before going into the specific details about channels, it will be helpful to get an overview of the major participants in the environment

and how they interact.

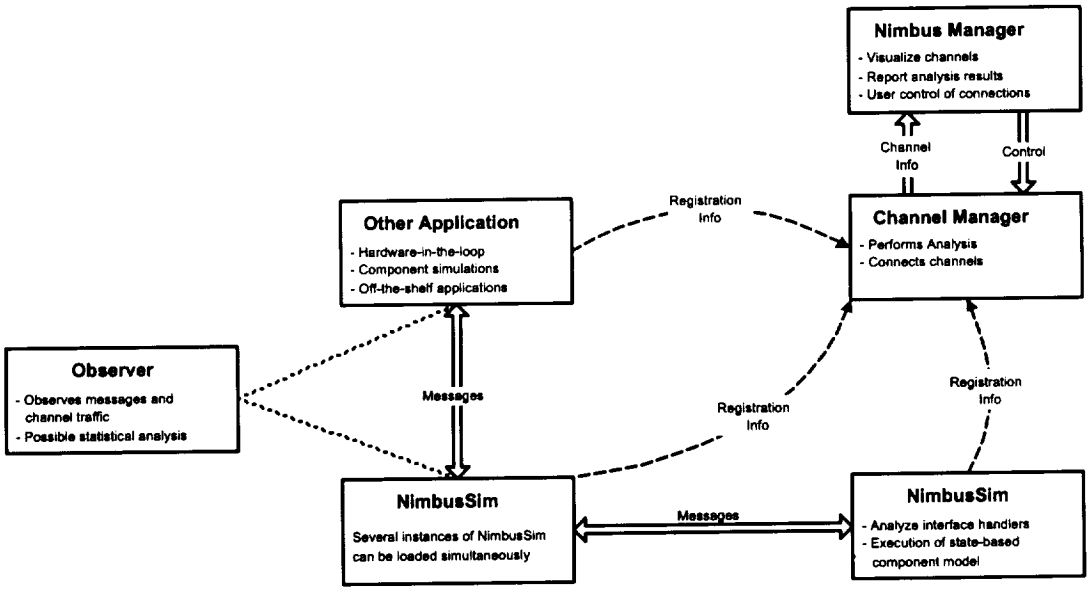

Figure 5.8: The architecture of the NIMBUS environment

Figure 5.8 shows the various components of the NIMBUS environment and gives examples of how they interact. NIMBUSSim is the simulation and analysis tool for RSML^{-e}. Multiple instances of NIMBUSSim can run concurrently so that different specifications can simulate different components of the system. Other applications can also be added into the environment. Although it is not shown in Figure 5.8, any application in the NIMBUS environment can exchange messages with any other application.

The NIMBUS Manager allows the user to dynamically control the connections between the various components which are registered with the NIMBUS system. Figure 5.9 shows the main window of the NIMBUS Manager. In the left-hand column, all the channels currently registered in the system are listed. When the user clicks on a channel name, the detailed information about that channel is displayed in the

area to the right of the channel list.

For each channel, the NIMBUS Manager displays the channel name (in Figure 5.9, AltitudeChannel), the type (Send-Receive) and three lists (sources, destinations, and observers). In the lists are the components that interact with that particular channel. For example, Figure 5.9 shows that Altitude channel currently has two sources and two destinations. The sources are "MS Excel ASW System" and "Simple MFC Client app."

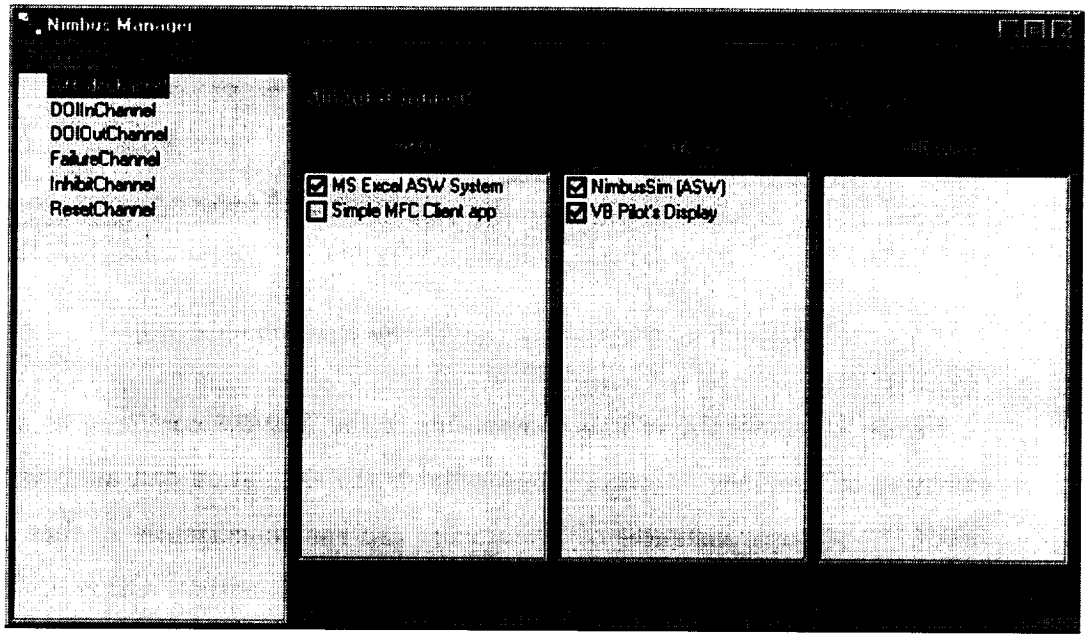

Figure 5.9: The main window of the NIMBUS Manager

The check marks beside the component names in Figure 5.9 indicate whether or not that component is currently active on the channel or not. A component that is not active cannot participate in the message processing (sending, receiving, publishing, etc) on that channel. Thus, Figure 5.9 shows that the "MS Excel ASW System" component is active, whereas the "Simple MFC Client app" is not. Notice

in the figure that both "NIMBUSSim (ASW)" and "VB Pilot's Display" are active destinations on the AltitudeChannel. The NIMBUS environment allows multicast communication to allow multiple displays of the data. Also allowed are observers on a channel (no observers are pictured in Figure 5.9).

The NIMBUS environment does not allow multiple active sources on a channel. Thus, if the user where to click on the checkbox next to "Simple MFC Client app," then it would become the active source on the AltitudeChannel and "MS Excel ASW System" would become inactive. For convenience, the first registered source and destination on a channel are made active by default.

The NIMBUS environment allows us to execute and simulate this model using input data representing the monitored variables and collect output representing the controlled variables. Input data could come from several sources. The simplest option for input is, of course, to have the user specify the values (either interactively, or by putting the values into a text file ahead of time). This scenario is illustrated for the ASW in Figure 5.10(a). Unfortunately, it is often difficult to create appropriate input values since the physical characteristics of the environment enforce constraints and interrelationships over the monitored variables. Thus, to create a valid (i.e., physically realistic) input sequence, the analyst must have a model of the environment. Initially, this model may be an informal mental model of how the environment operates. As the evaluation process progresses, however, a more detailed model is most likely needed. Therefore, in this stage of the modeling we may develop a simulation of the physical environment. The NIMBUS architecture lets us easily replace the inputs read from text files with a software simulation emulating the environment. This refinement can be done without any modifications to the REQ model.

NIMBUS allows the user to visualize the system in many ways. The visualizations constructed using powerful user interface construction tools, for example, Visual Basic

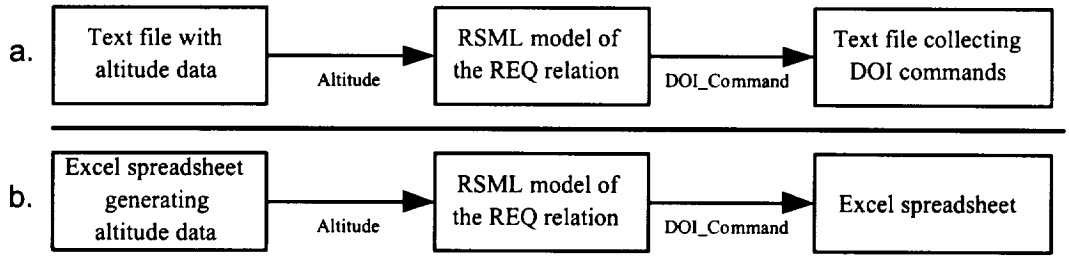

Figure 5.10: The REQ relation can be evaluated using text files or user input (a) or interacting with a simulation of the environment (b).

and can utilize the many third party ActiveX controls that are on the market. This makes it possible to construct rich visualizations, without expending large amounts of money. Thus, more development dollars can be used to ensure the quality of the specification.

When evaluating RSML^{-e} specifications in NIMBUS, the analyst has great freedom in how he or she models the environment. When we evaluated the REQ model, we used text files or a software simulation of the physical process to provide the RSML^{-e} model with monitored variables and to evaluate the controlled variables. As the IN' and OUT' relations are added to the RSML^{-e} model, the data provided (and consumed) by the model of the environment must also be refined to reflect the software inputs and outputs (INPUT and OUTPUT) instead of the monitored and controlled variables. This can be achieved in two ways; (1) refine the model of the physical process to produce INPUT and consume OUTPUT, or (2) add explicit models of the sensors and actuators to the simulation. In reality, the refinement of the environmental model and the SOFT relation progress in parallel and is an iterative process. The sensor and actuator models may be added one at a time and the interaction with different components may merit different refinement strategies. NIMBUS naturally allows any combination of the approaches mentioned above to be

used.

As the refinement of the SOFT relation and the models of the environment progresses, we may at some point desire to perform hardware-in-the-loop simulation. This not only provides a powerful evaluation of the proposed software system, we can also use NIMBUS to evaluate the physical system itself. For instance, by forcing the RSML^{-e} model of the software requirements into unexpected and/or hazardous states, we can inject simulated software failures into the hardware system.

5.3.1 Simulations of the ASW

For the ASW, we created a spreadsheet in Microsoft Excel to emulate the behavior of the aircraft (Recall Figure 5.10(b)). The graphical interface for the Excel model is shown in Figure 5.11. This simple environmental model allows us to interactively modify the ascent and descent rates of the aircraft, and easily explore many possible scenarios.

Figure 5.12 shows a mockup of a portion of the Pilot's Display that we developed to show this concept. Mockups like this, which are available while REQ is being developed, could be used to evaluate the potential operator interface early in the development life cycle. This can allow the specifiers to catch many errors early and also evaluate the REQ relation for, e.g., potential mode confusion. In fact, the Rockwell Advanced Technology Center has been very successful at developing high-fidelity mockups of their control panels and displays for the Flight Guidance System and using them to explore operator issues such as mode confusion.

In the case of the Altitude Switch, to simulate the refined SOFT relation we modified our Excel model of the physical environment to produce digital and analog altitude measures (Figure 5.13(a)). The refinement was achieved by simply making Excel provide the three altitudes and applying the two transfer functions defined

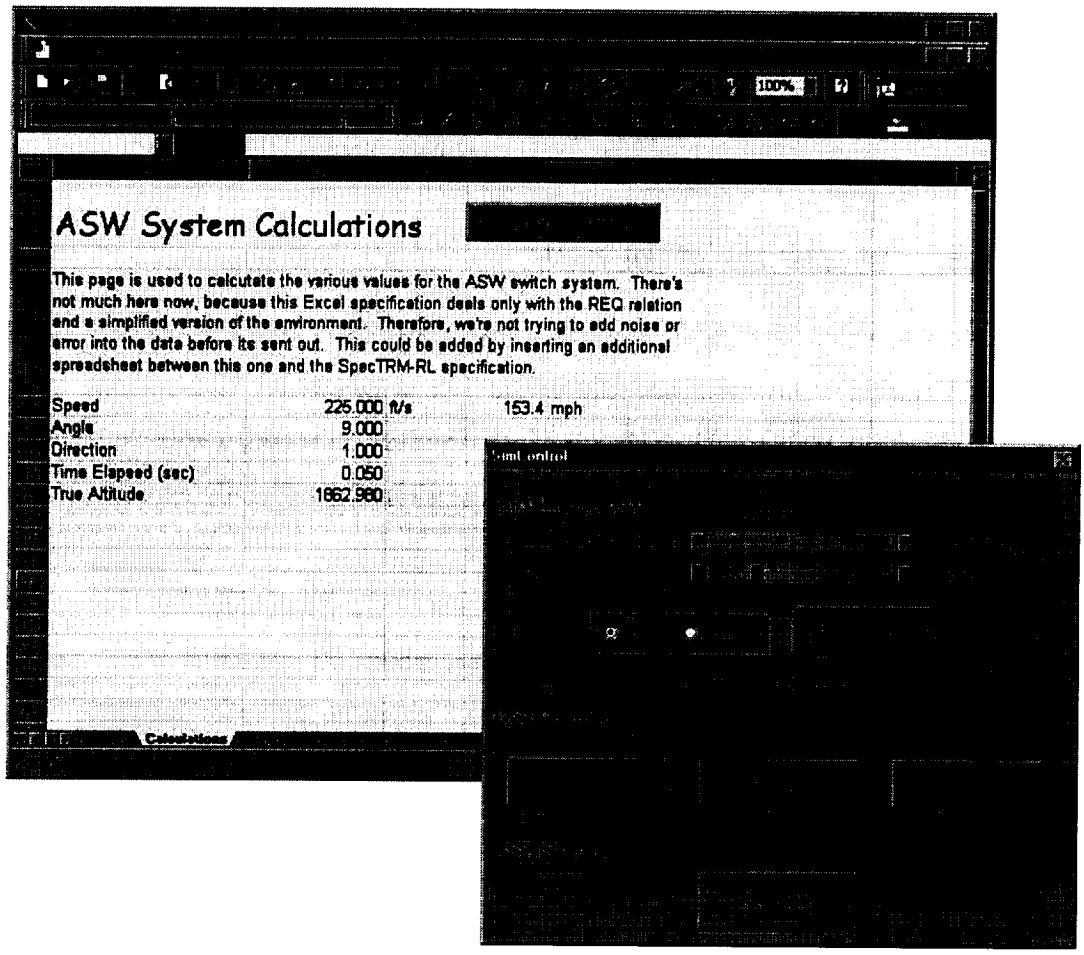

Figure 5.11: The ASW Excel REQ environment

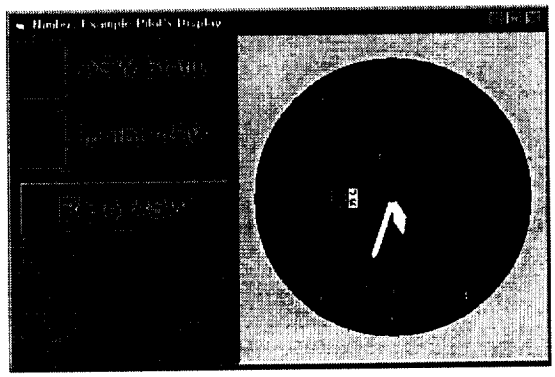

Figure 5.12: A mockup of the Pilot's display for the REQ model

earlier before the output was sent to the RSML^{-e} model. Adding measurement errors to the sensor models can further refine the simulation of the ASW. For instance, by modifying the computation of the digital altimeter outputs to

$$DigitalAlt = \frac{Altitude}{8192} + \varepsilon$$

where ε is some normally distributed random error (easily modeled using standard functions in Excel), we can provide a more realistic simulation that includes the natural noise in the data from the altimeters.

As an alternative to refining the Excel model to include the altimeter models, we can explicitly add altimeter models to the simulation (Figure 5.13(b)). In our case, we added altimeter models expressed in RSML^{-e}. By adding explicit models of the sensors and actuators, we can easily explore how the software controller reacts to simulated sensor and actuator failures. Note that the integration of various different models with the RSML^{-e} simulation of the control software does not require any modifications of the RSML^{-e} model, the channel architecture of NIMBUS allows the analyst to easily interchange the component models comprising the environment.

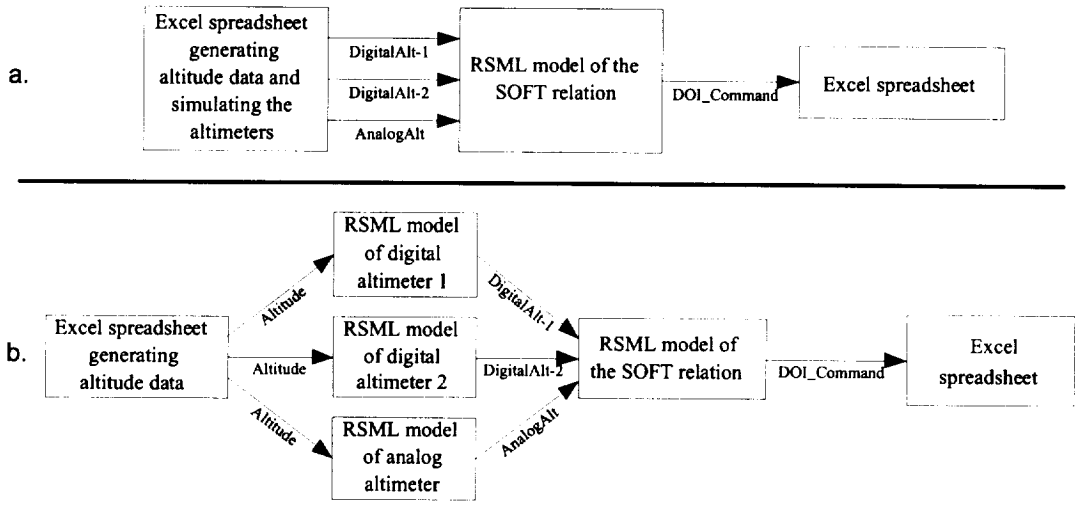

Figure 5.13: Refined models of the environment; (a) using Excel to simulate the physical process as well as the sensors and (b) using Excel to simulate the physical process and RSML^{-e} models to model the sensors.

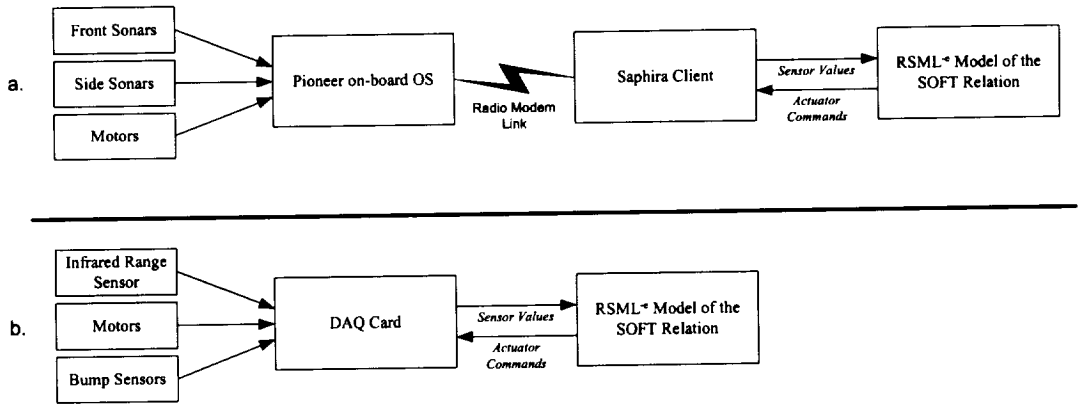

Figure 5.14: Summary of the hardware-in-the-loop simulations performed with the mobile robotics platforms

5.3.2 Simulations of the Mobile Robotics

Figure 5.14 shows the configuration that we used for the hardware-in-the-loop simulations of the mobile robots. The Pioneer is shown in part a. of the figure and the Lego-bot is shown in part b. The Pioneer has a high-level interface called Saphira. The Pioneer has a small CPU on board which manages the actual hardware. The sensor input and actuator commands are managed by the Pioneer OS which runs on the Pioneer's CPU. Part of the Pioneer OS manages the communication over the radio modem. On the other end of the radio modem, Saphira translates the inputs from the Pioneer OS into values the can be accessed by the user using Saphira function calls.

In the case of the Pioneer, there is quite a bit of software between the RSML^{-e} simulation and the actual hardware. This situation is common in real world situations since there probably already exists driver software for the hardware that you wish to control. It is natural to use this existing software rather than encode all the details in the specification.

NIMBUS also allows the analyst to bypass any software drivers and model all aspects of the control in RSML^{-e}. In the case of the lego-bot, the RSML^{-e} specification gives low-level direction to the data acquisition (DAQ) card. The card then supplies the voltages to run the Lego-bot through a tether cable. Some aspects of the Lego-bots behavior had to be manually calibrated. For example, it is difficult to predict in advance how fast the robot will move given a particular voltage supplied to the motors. This depends on the strength of the motors and the gearing of the wheels. Instead of trying to calculate speeds using the gear sizes, etc., we calibrated the IN and IN' relations based on experimentation.

In the mobile robotics domain, it is virtually impossible to develop a realistic simulation of the robot and its environment. The ability of NIMBUS to do hardware-in-the-loop simulation allows us to extend our formal modeling approaches to systems where experimentation is an integral part of the development process, for example, mobile robotics.

5.4 Summary

In this chapter, we have illustrated the FORM$_{PCS}$ system model as well as the process framework of specification-based prototyping. We have also illustrated this process on the ASW and Mobile Robotics examples, including how these two examples can be simulated in the NIMBUS environment. This chapter provides a basis for understanding the FORM$_{PCS}$ methodology. Even so, there are a number of questions that remain open. First, how do we arrive at the REQ relation? How is the REQ relation structured and how can we find the monitored and controlled variables? And, how can all of this be tied to the product family work that was discussed in Chapter 4. These questions are all addressed in the methodology itself, which is the topic of the next Chapter.

Chapter 6

Methodology Overview

This chapter gives a high-level overview our Family Oriented Requirements Method for Process Control Systems (FORM$_{PCS}$) concentrating on the original contributions of the dissertation. The methodology builds on both the preceding chapter and Chapter 4 to present an integrated view of the overall process from high-level product family requirements to a detailed specification of the SOFT relation. This process and the activities in each phase also borrow from related work, especially CoRE [99] and work done by Steve Miller [83, 82]. In this chapter, we will hightlight the original components of the methodology while giving a broad overview of the process and activities.

We begin by discussing FORM$_{PCS}$ in an idealized setting where the specifier always has all the information necessary to make correct decisions at each stage of the process. Often, however, this is not the case [91]. Thus, the idealized process is not necessarily a realistic one. The iteration that one would expect to find in the methodology is found in the section after the idealized process. Finally, we end this chapter with a section on what languages are suitable for use with FORM$_{PCS}$ and what the various tradeoffs between those languages might be.

6.1 FORM$_{PCS}$ Process Phases

This section describes the idealized FORM$_{PCS}$ process phases. Each subsection below describes a phase of the methodology, beginning with the commonality analysis and ending with the specification related to the sensors and actuators in the final, physical system. Along the way, FORM$_{PCS}$ guides practitioners in defining environmental quantities and operator set points, developing an overall structure for the requirements and then developing a draft specification, refining the draft specification (adding, for example, error handling and fault recovery behaviors), and finally adding details about the sensors and actuators.

6.1.1 Commonality Analysis

The commonality analysis, the first phase of the methodology, begins with a short (i.e., one to 5 paragraphs) high-level description of the intended family. This description is then refined until the analyst can begin to identify the *commonalities*, i.e., those features which are present in all family members, and the *variabilities*, i.e, those features which vary across members of the family. This initial set of commonalities and variabilities forms the basis for the rest of the process.

As we discussed in Chapter 4, we allow a family to be broken down along different dimensions, for example, a hardware dimension and a behavioral dimension. In addition, we allow a family to be broken into several sub-families, for example, a general family of flying craft might be broken down into fixed-wing aircraft and helicopters. This family-level structuring occurs as a result of discovering additional commonalities and variabilities during the commonality analysis. Finally, we will examine the commonalities and variabilities in terms of whether they apply to the REQ relation or whether they apply to the IN' or OUT' relations.

Define the Top-Level Family: The first activity of FORM$_{PCS}$ is to defining the top-level family which will form the basis for the specification(s) developed in the later phases. This important activity ends when a short description of the family has been generated. We gave the high-level description the Altitude Switch family in Chapter 4 and it is reproduced below for reference:

> *The ASW family consists of systems on board the aircraft that utilize the values from the various altimeters on board to make a choice among various options for actions (one of which being to do nothing) and perform the chosen action.*

From this high-level description, the initial commonalities and variabilities may be stated.

Initial Commonalities and Variabilities: The initial commonalities and variabilities are found by examining the system description and the high-level description that was written in the previous activity. Much has been written about elicitation and recording of the commonalities and variabilities for a product family [117,12 , 53] and FORM$_{PCS}$ includes some guidelines as well as pointers to these references.

FORM$_{PCS}$ advocates a slightly modified approach starting with high-level commonalities and variabilities and working towards a more refined description of the family. The highest level commonalities define the boundaries of the broadest possible product family. As more commonalities are added, the definition of the family becomes more refined. We assert that it is useful to preserve which commonalities define the outermost scope of the family – these are the least likely to change in the future and, thus, should depend on the essential purpose of the system, i.e., the most basic reasons for the system's existence.

In addition, because FORM_{PCS} deals with process-control systems in particular, we needed to provide a connection between the commonalities and variabilities and the REQ, IN', OUT' relations. Therefore, we advocate noting which relation a commonality or variability applies to and partitioning the commonalities and variabilities based on that information. This separation is useful because we can then first concentrate first on the REQ relation and before moving on to the IN' and OUT' relations (as outlined in Chapter 5).

In Chapter 4 a number of the initial commonalities for the ASW were listed. The ASW was first defined as a broad family, allowing for all possible members of the ASW. These high-level commonalities and variabilities are listed again for reference.

C1 All ASW systems will have a method of measuring the altitude of the aircraft

C1.1 The ASW system will use the information about the aircraft's altitude to make a decision as to what action the ASW system shall perform

V1 The actions that the ASW takes in response to the altitude and the criteria to perform those actions varies from aircraft to aircraft

Then, we added more commonalities and variabilities that further defined the scope and purpose of the product family. We will not duplicate all the commonalities and variabilities here, but we will mention the following is an example of the original text of [V4].

V4 The period of time that the altitude must be invalid before the ASW will declare a failure may vary.

Note that this is an earlier, less refined version of [V4] than what was presented in Chapter 4; the refined version is presented below under the elaboration of the commonalities and variabilities.

Identify Family Structure: Even for a family as small and simple as the ASW, we can identify elements of structure in the family. This identification is useful because it helps us to understand the family and it is invaluable if, in the future, we would like to re-factor the family or incorporate the family as a part of a larger family. For example, we might like to have one family that encompasses all the avionics devices (not just the ASW). The techniques for representing the family structure that were developed as a part of this research were discussed in Chapter 4 and this activity of the methodology presents these techniques in a form suitable for practitioners.

One of the contributions of this activity is a greater explaining of how to identify sub-families. We believe that a good clue to the existence of a sub-family is commonalities that start with the word "if," for example, in the case of the ASW we could have written all the DOI commonalities as "If the action to be performed is turning on or off a DOI, then ..." This activity also involves the visualization of the family structure as was discussed in Chapter 4

In Chapter 4 we presented several visualizations of the structure of the ASW family (Figures 4.8 and 4.9).

Elaborate Variabilities and Commonalities: In the next activity in developing the family description, the commonalities and (especially) the variabilities are refined so that they contain actual quantities, or choices for the variations. This activity is a precursor to developing the product family decision model (next), as well as later when the tolerances and bounds of variables in the specification must be given.

When [V4] was first defined, we did not specify the tolerances on the period of time that the ASW shall wait before declaring a failure. As we progressed in the definition of the ASW family, that information was added so that variability four appears as it was printed in Chapter 4.

V4 The period of time that the altitude must be invalid before the ASW will declare a failure may vary between 2 seconds and 10 seconds from family member to family member.

Define the Decision Model: The decision model represents a recording of which choices for all the possible variabilities result in valid family members. Obviously, the more complex the structure of the family, the more complex the decision model will be.

Building the decision model can often help to identify commonalities or variabilities that may have been forgotten in the initial draft of the family requirements. This is because engineers, familiar with the products, may recall items that must be specified about a particular family members that they did not recall when attempting to generalize to all family members.

One way that the decision model may be written down is by simply noting which choices are made for each family member. However, the most common technique in the literature is a simple tabular representation similar to what was given for the ASW in Chapter 4 and is reproduced in Figure 6.1. In a family with a more complex structure, a hierarchical series of tables might be used with one table for each sub-family.

At the end of the commonality analysis, the requirements document will contain a description of the family including all the sub-families and dimensions involved; and, the analyst will have identified a subset of the commonalities and variabilities that he/she will use to specify the REQ relation in the next stages.

Variability	CS-123	CS-134	DD-123	DD-134	EF-155
# of Analog Alt.	1	1	1	1	2
# of Digital Alt.	1	2	1	2	3
Threshold Algo.	Any	Any	Any	Majority	Majority
Invalid Alt. Failure	4 s	2 s	2 s	2 s	2 s
Threshold	2000 ft	2000 ft	2000 ft	2000 ft	1500 ft
Go Above Action	None	None	None	None	Turn Off
Go Below Action	Turn On	Turn On	Turn On	Turn On	Turn On
Go Above Hyst.	200 ft	200 ft	250 ft	200 ft	200 ft
Go Below Hyst.	NA	NA	NA	NA	200 ft
DOI timeout	2 s	2 s	2 s	2 s	2 s

Figure 6.1: A tabular representation of the ASW family decision model

6.1.2 Environmental Variables

In the environmental variables phase, the goal is to identify quantities in the environment that are important to the specification. Earlier, we discussed several models of viewing the system's interaction with the environment. Many environmental quantities are mentioned in the commonalities and variabilities that were created in the previous phase. This phase of the methodology provides concrete guidance on how to choose monitored and controlled quantities. In addition, this section will demonstrate the characteristics of the various types of environmental variables.

Identifying Controlled Variables: The focus of this activity is to identify the quantities that are under the system's control. We categorize the controlled variables into several classifications:

- **Environmental Quantities:** These are variables in the environment that system changes in order to achieve the requirements. These should not be tied to any particular actuators, but should represent in general the effects that the system may introduce in the environment.

- **User Displays:** These are variables that need to be displayed to the user. This type of controlled variables often represent indicator lights, gauges, etc. that are present in the physical system. Their purpose is to help the operator develop a mental model about the state of the system being controlled; thus, indications of the state of the controller are also often included.

- **Values for Another Subsystem:** These are variables that go to another subsystem. This type of controlled variables is common when specifying one piece out of a system or subsystem and there are certain details that must be abstracted away.

This classification scheme is unique to FORM_{PCS} and provides more guidance in this area than what is currently available in either CoRE [99] or REVEAL [94].

In the ASW, one controlled variable is the DOI status, which we know from [$C_{DOI}1$] is changed by the ASW. As it happens, the DOI is an interesting case, because the state of the DOI is both controlled *and* monitored by the ASW. This is because other systems on the aircraft can turn the DOI on and off. In terms of our categories of controlled variables, the DOI fits best as an environmental variable because the DOI is a device that will exist on the aircraft presumably whether or not the ASW is on board.

Another controlled variable is the failure indication of the ASW. The ASW is required to supply an indication of whether or not it is operating correctly [C3]. Therefor, a controlled variable is required to support this indication. In terms of

the categories, failure indication fits best as a user display, but could also be viewed as a subsystem interface because it may be used by other components on board the aircraft.

Identifying Monitored Variables: This activity compliments the identification of controlled variables by identifying the monitored variables. Monitored quantities, similar to controlled quantities, are broken down into several different types that help in identifying them.

- **Environmental Quantities:** Variables or conditions that exist in the environment, are observable, and can be used to compute the values of controlled variables.
- **User set-points:** Variables that are specified by the user (operator) of the system. These variables change the way that the controlled quantities are computed.
- **Abstracted quantities:** Variables that are received from another subsystem that are introduced because the specifier desires to concentrate on the current subsystem.
- **Quality Indications:** These are variables which indicate the quality or ability to observe of other monitored variables. These variables are often Boolean, for example, indicating that the altitude can or cannot be observed.

Certainly, the most obvious monitored quantity in the ASW is that of the Altitude. This is clearly an environmental quantity, because the aircraft will have some altitude whether or not the ASW is present. In addition, we know that eventually we will have some kind of sensors in the system that actually measure the altitude.

Because it is always possible for sensors to fail, it is possible that there will times when the altitude will not be measurable. Therefor, we require a quality indication for the Altitude, the Altitude_Quality variable.

Define the Variables: The monitored and controlled variables represent the interface of the system requirements, the REQ relation, to the environment. It is important to capture the essential information about each variable. In this activity, FORM$_{PCS}$ provides a template for defining the monitored and controlled variables based on [99] and also in [52].

In accordance with the guidelines in CoRE, we advocate that for controlled variables a short description of the conditions under which the variable can take on its various values is given. This activity helps in later stages as the informal description of each controlled variable is refined into a formal description of the REQ relation (and later the SOFT relation). In addition, as the conditions under which each variable takes on its various values are defined, often previously overlooked errors can be found.

An example of an initial controlled variable definition is shown in Figure 6.2.

Define Relationships Among Variables: In this activity, the relationships between the monitored and controlled variables that exist as part of the environment (and in the absence of the proposed system) are noted. Thus, in this activity we are encoding the NAT relation.

Jackson *et al.* provides good references on expression the system context and the NAT relation, some of which is duplicated in FORM$_{PCS}$ [49, 47, 50, 51, 48, 32]. The primary method of visualizing the system's interaction with its environment is a context diagram.

A system context diagram is a picture that shows each input and output to the

STATE VARIABLE

CON_DOI_P2

Parent: NONE
Possible Values: On, Off, Uncommanded
Initial Value: UNDEFINED
Classified as: Controlled
Purpose: This variable represents the ASW's commanded status of the Device of Interest (DOI).

Interpretation:

On: Indicates that the DOI is commanded to be On. The DOI is commanded to be on when the aircraft enters the target region for turning the DOI on, the DOI is not already on, and the ASW is not inhibited.

Off: Indicates that the DOI is commanded to be Off. The DOI is commanded to be off when the aircraft leaves the target region and after a certain period of time has passed. If this time is UNDEFINED, then the ASW will never turn the DOI Off.

Uncommanded: Indicates that the DOI is not commanded by the ASW. This CON_DOI variable will be equal to Uncommanded in any step were the ASW does not issue a command to the device of interest.

Issues:

- If the aircraft leaves the target area and the DOI is on, but was *not* commanded to be on by the ASW, should the ASW turn it off?

Figure 6.2: The CON_DOI variable in Phase 2 of the methodology

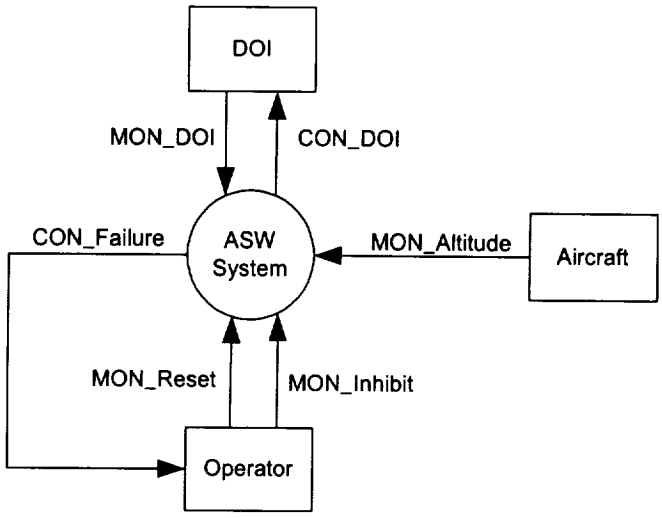

Figure 6.3: The System Context Diagram for the ASW in this Phase

system. The key in capturing the NAT relation is to begin to think about how the rectangular boxes (i.e, the monitored and controlled variable sources) interact with one another in the environment. An example system context diagram for the ASW is shown in Figure 6.3.

At the end of this phase, the specification will have a list of all the monitored and controlled variables used in the system cataloged according to their type. This will form the boundaries of the REQ relation. Furthermore, there will also be a statement of the NAT relation, i.e., a statement of the constraints that are imposed upon the environmental variables in the absence of the proposed system(s).

6.1.3 Initial Structure

In the initial structure phase, the environmental variable descriptions developed in the previous phase along with the product family structure identified in the first phase are used to develop an initial structure of the REQ relation. In languages

that support a module construct, specification entities may be grouped together into pieces that can be reused across the product family. In languages that do not support a module construct, specification pieces can be formed by textual delimitation (e.g., using comments) and physical grouping. Component reuse can then be accomplished by cut-and-paste.

In this and later sections, we have use the modularized version of RSML^{-e} proposed in the next chapter; thus, Chapters 6 and 7 are dependent upon one another. Because this chapter provides the *motivation* for the addition of the modules, we have chosen to present the methodology overview first. All that is really necessary to understand the examples here is that a module consists of a number of state variables that are imported, a number that are exported, and a number that are encapsulated (hidden).

Define Dependency Relationships: In this activity, the monitored variables and modes are necessary for the computation of each controlled variable are identified. The goal of this activity is not to produce a detailed dependency graph. Rather, the goal is to formulate a solid idea of the *order* in which entities in the system must be computed so that there are no circular dependencies between the various variables.

The first step is to make a list in each controlled variable definition of which other controlled variables, monitored variables, and mode machines it depends upon. Then it is possible to examine each variable and attempt to identify circular dependencies.

We advocate viewing a large specification as a series of functional blocks. The different blocks can then be drilled down into in a functional-decomposition type style. This helps to sequence the computation in broad strokes, and then it is easier to avoid circular dependencies within the block. Nevertheless, circular dependencies can be difficult to see, which is why a tool supported language can be invaluable.

Define Modules and Interfaces: In this activity, the dependency relationships created previously are used to start to group pieces of the computation together to form modules.

Parnas [88] defined a criteria to be used in decomposing a system into modules called information hiding. Using this philosophy, every module in the system should be chosen so as to encapsulate a decision or several decisions about the system, for example, a module may encapsulate a variability or group of variabilities. The interface of such a module exposes only the essential information that the rest of the specification requires. It has been suggested in CoRE [99] that a method for determining which decisions should be grouped together should be whether they are expected to change together. Still another way to view a module is as an addition to the vocabulary that is used to express the requirements. This is the reasoning that lies behind the standard modules used in functional declaration style in RSML^{-e}. A module may allow the specifier to map a construct in the physical domain to a single construct in the specification.

One building block that is useful for the ASW is a module that exports the thresholded altitude taking into consideration the hysteresis factor that is required. The interface for this module is defined in Figure 6.4.

The outcome of the this phase phase will be that the REQ relation is divided into a series of manageable pieces each of which will be specified in detail in the next phase of the methodology.

6.1.4 Draft Specification

In this phase, a preliminary behavioral specification of the system requirements is developed by refining the module definitions developed in the previous stage into working pieces of the specification. This first version of the specification will deal

```
MODULE ThresholdedAltitude_P3 :

  INTERFACE :

    IMPORT Altitude_P3 : Integer
      UNITS : ft
      EXPECTED_MIN : 0
      EXPECTED_MAX : 50000
    END IMPORT

    IMPORT CONSTANT Threshold_P3 : Integer
      UNITS : ft
      EXPECTED_MIN : 0
      EXPECTED_MAX : 8024
    END IMPORT

    IMPORT CONSTANT Hysteresis_P3 : Integer
      UNITS : ft
      EXPECTED_MIN : 50
      EXPECTED_MAX : 500
    END IMPORT

    IMPORT CONSTANT Direction_P3 : UpDownType

      Purpose : &*L This parameter tells the thresholding algorithm
      which direction we are interested in applying the hysteresis
      to.  If the direction is specified as Down, then we will have to
      go above threshold altitude by the hysteresis amount before we
      can declare that we are above (and, thus, be allowed to declare
      below again). L*&

    END IMPORT

    EXPORT AboveOrBelow : AboveBelowType

      Purpose : &*L this export reports whether or not the altitude is
      above or below the threshold given the hysteresis factor L*&

    END EXPORT

  END INTERFACE

  DEFINITION :
  END DEFINITION

END MODULE
```

Figure 6.4: The ThresholdedAltitude Interface in Phase 3

```
EXPORT CON_Failure_P4 :
  PARENT : NONE
  DEFAULT_VALUE : False

  EQUALS TRUE IF
    TABLE
      DURATION(AttemptingOn() , 0 S, Clock) > DOI_Timeout_P4      : T * * * ;
      DURATION(AttemptingOff( ), 0 S, Clock) > DOI_Timeout_P4     : * T * * ;
      DURATION(MON_Altitude_Q uality_P4 = Invalid, 0 S, Clock)    : * * T * ;
      PRE(CON_Failure_P4) = False                                 : * * * T ;
    END TABLE

  EQUALS FALSE IF
    TABLE
      DURATION(AttemptingOn() , 0 S, Clock) > DOI_Timeout_P4      : F ;
      DURATION(AttemptingOff( ), 0 S, Clock) > DOI_Timeout_P4     : F ;
      DURATION(MON_Altitude_Q uality_P4 = Invalid, 0 S, Clock)    : F ;
      PRE(CON_Failure_P4) = False                                 : F ;
    END TABLE

END EXPORT
```

Figure 6.5: The CON_Failure variable in Phase 4 of FORM$_{PCS}$

primarily with the intended, normal case behavior. While failure modes and fault tolerance must be kept in mind, these characteristics will be added to the specification in later stages.

Specify Each Controlled Variable: In this activity, how each controlled variable assumes its various values is specified. This activity involves not only thinking about what values are necessary to compute the controlled variables, but exactly how those variables contribute to the controlled values. Much of the information on specifying controlled variables was adapted from CoRE; however, we have added a distinction between two styles of specification: equivalence-style and transitional-style.

Equivalence-style specification of a state variable is, perhaps, the most straightforward. In this style, the specifier states explicitly in a series of cases what value

the state variable assumes. The value of the variable is, thus, always defined unless explicitly noted otherwise by the specifier or unless it is a child underneath another state variable. An example of an equivalence-style specification for the Failure state variable is shown in Figure 6.5.

For any computation of the specification, it is expected that one and only one case of the variable will be true; the state variable then assumes the value specified by the one unique case. If the state variable does not have a case which evaluates to true in some step, then we say that the variable definition is *incomplete* because for the particular sequence of inputs events leading up to this step the variable does not have a defined value. If the state variable has more than one case which is true then we say that the specification is *inconsistent*; how can we know which case is the one that was intended by the specifier?

On the other hand, sometimes we are not so interested in what values a variable should have in each step but, rather, it is desirable to specify when the variable should *change* values. A transitional-style specification consists of a series of transitions, each with a source state, a destination state, and a condition. When the condition is true and the variable has the value specified by the source state, then the variable will become the value specified by the destination state.

Transitional-style specifications have the same notion of consistency as equivalence-style specifications. In contrast, a transitional-style specification is usually expected to retain its current value in the absence of any need to change. Therefore, transitional-style specifications often do not make use of the notion of completeness because it is expected that there will be some steps (probably many steps) in which the none of the transitions may be taken. A transitional-style specification may be made complete by adding transitions from a state back to itself that cover the conditions underwhich the state variable shall retain that particular value. An example of a transitional-style

```
EXPORT CON_DOI_P4 :
  PARENT : NONE
  DEFAULT_VALUE : Uncommanded

  TRANSITION Uncommanded TO On IF
    TABLE
       DOI_Action_Ok(On)                                    : T T ;
       WHEN(ThresholdedAlt_P4. Result_P4 = Below, False)    : T * ;
       GoBelowAction = TurnOn                               : T * ;
       WHEN(ThresholdedAlt_P4. Result_P4 = Above, False)    : * T ;
       GoAboveAction = TurnOn                               : * T ;
    END TABLE

  TRANSITION Uncommanded TO Off IF
    TABLE
       DOI_Action_Ok(Off)                                   : T T ;
       WHEN(ThresholdedAlt_P4. Result_P4 = Below, False)    : T * ;
       GoBelowAction = TurnOff                              : T * ;
       WHEN(ThresholdedAlt_P4. Result_P4 = Above, False)    : * T ;
       GoAboveAction = TurnOff                              : * T ;
    END TABLE

  TRANSITION On TO Uncommanded IF WHEN(MON_DOI_P4 = On, False)

  TRANSITION Off TO Uncommanded IF WHEN(MON_DOI_P4 = Off, False)

END EXPORT
```

Figure 6.6: The CON_DOI variable in Phase 4

specification is shown in Figure 6.6.

Identify Potential Modes: In general, modes of the system are points of discontinuity in the functions of the controlled variables. For example, a controlled variable might depend on a specific series of user inputs and events before it can take on a particular value; thus, we will require a mode machine of some kind which will record for us *where* in the sequence of actions we are and what input we expect to occur next.

A concrete example is that of a weapons firing interlock. It is usually true that a number of conditions must become true before pressing the 'fire' button will cause the weapon to fire, for example, perhaps the airplane must be traveling at a particular speed, or at least a certain altitude. Furthermore, it is usually *not* desirable to have the depression of the firing button precede these events: what if the firing button is stuck down and we cross a threshold altitude which makes the preconditions true? We probably do not want to fire in that case. To model this type of behavior, the specification must store internal state information so that it can track where in the sequence it is.

Modes partition the functionality of the system. When a mode variable has one value, the system behaves in one way and when the mode has a different value, the system behaves in a different way. The above example of a sequence of values is not the only time when this occurs. Modes may represent some alternate or reduced operation of the system. For example, many systems have a startup or shutdown mode in addition to the normal operation mode. Another example is when a system has some reduced functionality modes; for example, when the values of some environmental quantities are not available, the system may only be allowed to perform a subset of the available actions.

Finally, modes may be introduced to represent to the environment or controller

what the system is doing. For example, in an aircraft, the various systems can be on autopilot, or in landing or take-off modes. If the system being built is responsible for implementing one or more of these modes, then it will be useful to represent them explicitly in the requirements because they are the language in which the customer will be most comfortable to communicate. In addition, it be common to state properties about these modes, for example, "the system shall not raise the landing gear while in landing mode."

There are a number of examples of variables which might be considered modes in the ASW specification. One example is ASW_System_Mode variable. This variable controls the overall functioning of the ASW. In phase four, the ASW_System_Mode variable has only two values: Operating and Rest. Nevertheless, this same structure could be used to represent a startup and shutdown mode, or it could be used to represent different modes of reduced functionality simply by adding values to the ASW_System_Mode variable and then defining appropriate behavior for those modes. Using the module construct (or cut and paste) it is possible to allow modes to share functionality while still differing significantly in some areas.

Use Tools to Visualize the Preliminary Behavioral Specification: Many formal languages are supported by tools, including RSML^{-e}, which is supported by the NIMBUS tools. Simulation of the specification was discussed in Chapter 5.

The outcome of the draft specification is a document which can be reviewed so that all interested parties can agree on the *essential* behavior of the REQ relation without getting bogged down in details about particular sensors and actuators, or about complex failure modes and error handling. Using RSML^{-e} with the NIMBUS environment, it is possible to simulate the high-level behavior at this point; therefore, everyone involved on the specification effort can get a good idea of the behavior that

was specified.

6.1.5 Detailed Requirements

When producing the Detailed Requirements, the analyst will begin to add to the REQ relation all things that were initially left out of the preliminary behavioral specification. In this phase, we will consider the fault tolerance of the specification, error conditions which may arise due to the fact that we are using sensors and actuators, and so forth. Also, hear is where we need to consider in more detail the startup and shutdown behavior of the system.

As these new behaviors are added, we may find it necessary to revisit decisions which were made about the preliminary specification as well as about the requirements structure. Thus, it is natural to iterate between these phases.

At this point, the analysts should begin to think about completeness and consistency of the REQ specification. Therefore, if analysis tools are available, the REQ specification should be run through these tools and any errors which are found should be corrected.

Specify Initialization and Shutdown Activities: Most controllers have (or should have) a different operational profile immediately after they are turned on and just before they are about to turn off. The reason for this is that the environment in which the controller operates is a system of its own right; it exists with or without the presence or operation of the controller. Certainly, there are two different systems: one with the controller turned on and one with the controller turned off. And, these systems behave differently from one another.

The ASW's startup mode is very simple: it just has to receive five seconds of valid altitude in order to transfer to normal operation. Thus, it can be represented

```
EXPORT CON_Failure_P5 :
  PARENT :  NONE
  DEFAULT_VALUE : False

  TRANSITION False TO True IF
    TABLE
      ASW_System_Mode_P5 = NormalOperating    : T * ;
      ASW_Operating_Mode_P5.C ON_Failure_P5   : T * ;
      ASW_System_Mode_P5 = Degraded           : * T ;
      ASW_Operating_Mode_P5.C ON_Failure_P5   : * T ;
    END TABLE

  TRANSITION True TO False IF ASW_System_Mode_P5 = Reset

END EXPORT
```

Figure 6.7: The CON_Failure variable in Phase Five

with only a single transition and does not need other behavior. In other systems, the controller may need to wait until it develops a certain confidence in the estimates of the monitored quantities before it issues any commands to the environment.

Specify Error Handling: The first thing to do in specifying the error handling behavior of the specification is to create a list of potential error conditions. Note that all of the possible error conditions may not be known at this time; some error conditions may only come to light when information about the sensors and actuators is added. Nevertheless, many possible error conditions will be known during our development of the REQ relation and those error conditions should be handled.

A useful technique is to have a global failure mode that encapsulates the failure mode of the system. High-level failure conditions cause this mode to transition between its various values (i.e., "Ok" and "Failed"). The global failure mode can then be supported by having each module below the main module also export a failure indication that covers failures local to that module. Then, the global failure mode

checks each of these local failure indications and, if they are true, may decide declare a failure or to enter some reduced functionality mode as is described in the next activity. This technique of structuring the failure mode computation is unique to FORM$_{PCS}$. The ASW's global failure mode is shown in Figure 6.7.

Degraded Modes of Functionality: Often, we wish to have a system which has some behavior under ideal conditions, i.e., good knowledge about the environment, but which will continue to function in a safe manner event if conditions are not ideal (for example, with failed sensors or actuators). It is possible to plan ahead and establish several different modes of functionality ranging from fully operational where all information is known to an acceptable confidence to a shutting down mode where the system will turn itself off and leave the process in a safe state.

This sort of system is difficult to construct because, in a sense, many different systems are being specified – one for each degraded functionality mode. Nevertheless, it may be that the system behavior is more or less the same in these various modes. In that case, the modes may be able to be treated as a family of sorts. This view of degraded modes of functionality as a family and the structuring of them was first introduced in FORM$_{PCS}$.

The various modes of the ASW and how the ASW switches between them are shown below. We have simply added additional states to the undeveloped ASW_-System_Mode from the previous phase. In Figure 6.8, we have added an overall failure mode to deal with system failures and also a value for the started and degraded functionality modes.

In order to enter the degraded functionality mode, we must know whether two episodes of invalid altitude lasting at least one second have occurred within one minute of each other. This requires state information, so we have introduced the EpisodeMonitor_P5 variable to track the occurrence of episodes and inform the ASW_-

```
STATE_VARIABLE ASW_System_Mode_P5 :
  VALUES : {Startup, NormalOperating, Degraded, Failed, Reset}
  PARENT : NONE

  Purpose : &*L This is the top-level mode of the ASW.  If the ASW
  were to have a startup mode, etc., we could put those modes as
  children of this controlling mode.  Currently, we have only two
  states, the reset mode which is used for when the reset signal
  is received and the operating mode that handles the main
  behavior. L*&

  DEFAULT_VALUE : Startup

  TRANSITION NormalOperating TO Reset IF MON_Reset_P5

  TRANSITION Degraded TO Reset IF MON_Reset_P5

  TRANSITION NormalOperating TO Degraded IF
    EpisodeMonitor_P5 = QualifyingEpisode

  TRANSITION Degraded TO NormalOperating IF
    DURATION (MON_Altitude_Quality_P5 = Valid, 0 S, Clock) > 1 M

  TRANSITION Reset TO NormalOperating IF
    DURATION(PRE(ASW_System_Mode_P5), 0 s, Clock) >= 0 S

END STATE_VARIABLE
```

Figure 6.8: The ASW_System_Mode variable in Phase 5

```
STATE_VARIABLE EpisodeMonitor_P5 :
  VALUES : {NoEpisode, FirstEpisode, QualifyingEpisode}
  PARENT : NONE

  Purpose : &*L This simple state variable tracks whether or not
  we have met the conditions for being in degraded functionality
  mode.  Namely, whether or not we have seen two periods of
  invalid altitude lasting 1 second or more within 1 minute. L*&

   DEFAULT_VALUE : NoEpisode

   TRANSITION NoEpisode TO FirstEpisode IF
     DURATION(MON_Altitude_Q uality_P5 = Invalid, 0 S, Clock) > 1 S

   TRANSITION FirstEpisode TO QualifyingEpisode IF
     TABLE
       DURATION(MON_Altitude_Q uality_P5 = Invalid, 0 S, Clock) > 1 S : T ;
       DURATION(PRE(EpisodeMon itor_P5) = FirstEpisode) > 1 S         : T ;
     END TRANSITION

   TRANSITION FirstEpisode TO NoEpisode IF
     DURATION(PRE(EpisodeMon itor_P5) = FirstEpisode) >= 1 M

   TRANSITION QualifyingEpiso de TO NoEpisode IF
     DURATION(MON_Altitude_Q uality_P5 = Valid, 0 S, Clock) >= 2 M

END STATE_VARIABLE
```

Figure 6.9: The EpisodeMonitor variable in Phase 5

System_Mode variable when a qualifying episode as occurred and it is necessary to enter degraded functionality mode. This is shown in figure 6.9.

Specify Tolerances and Handle Violations: In the ideal world of the REQ specification, the value of each controlled variable is known with exact precision. Nevertheless, we know that eventually a physical implementation of the system will be built and that in that implementation we cannot know the values for certain or to an infinite accuracy.

In many cases, the tolerance of a controlled variable is constant throughout the entire specification. In that case, the tolerance may be specified in much the same way as the precision was specified for monitored variables. In other cases, the tolerance of a controlled variable may be a function of one or more modes of the system. In the methodology, we give several examples of when this can be the case.

The outcome of this phase is a completed specification of REQ. This specification can then be analyzed using whatever formal analysis techniques are supported by language/toolset used in the specification effort.

6.1.6 Sensors and Actuators

Phases two through five have illustrated how to move from the commonality analysis in phase one to a completed REQ specification in phase five. In this final phase, the process will be repeated for the IN' and OUT' relations. In discussing this phase, we point out which parts of the process are generalizable and what information needs to be considered specifically for the hardware.

Identify and Describe the Sensors and Actuators: The first step in adding the IN' and OUT' relations is to identify and describe the sensors and actuators

involved in the system. After that, the input and output variables for the software can be identified. This activity is analogous to phase two for the REQ relation.

For the ASW, each aircraft as a number of altimeters that measure the altitude, a status indication from the DOI, a reset signal, and an inhibit signal. All inputs except for the altimeters can be mapped directly to the existing monitored variables. Therefore, on the input side we will concentrate in refining the IN' relation for the Altitude monitored quantity.

The commonality analysis from phase one tells us that we will have a varying number and type of altimeters for each aircraft that we wish to build. Furthermore, we know that the different types of altimeters yield different information: analog altimeters give only above or below whereas digital altimeters yield a numeric altitude.

The DOI command indication and the failure output are the controlled variables of concern for the OUT' relation. Only the failure output needs significant changes to specify the output relation.

For the failure indication, the ASW must produce a pulse on a watchdog timer at least every 200 MS or else the other devices on board the aircraft will believe that the ASW has failed. This is the opposite from the way that the REQ relation works, where we only produce an indication if there *was* a failure. Thus, we need a small state machine that will produce a pulse if there is not a failure.

Outline the IN' and OUT' Relations: Just as for the REQ relation, the first step in specifying IN' and OUT' relations is to outline the computation. In this activity, depending on the complexity of the IN'/OUT' relation, a data dependency graph might be developed, modes identified, and so forth. Furthermore, if the IN' and OUT' relations contain sufficient structure, modules may be introduced. Indeed, in systems with noisy sensors/actuators, or sensors/actuators with complex IN relations, the IN' and OUT' relations may represent the majority of the complexity of the

software. Thus, the stepwise refinement process defined as the foundation of the methodology combined with the activities of this phase will be essential to achieving a correct specification of the SOFT relation.

In the ASW family, each aircraft differs in the number and type of altimeters and in the algorithm used to determine whether the aircraft is above or below the threshold from the various altimeters. The first thing to notice is that the specification of REQ from phase five expects a numeric altitude input. For compatibility, we will change the input to REQ to be a thresholded value and move the thresholding of the digital altimeters into the IN' relation.

Thus, the overall structure of the IN' relation for Altitude is given by the following module definition:

```
MODULE Altimeters_IN_P6 :

  INTERFACE :

    IMPORT CONSTANT NumDigitalAlt_P6 : INTEGER
      UNITS : NA
      EXPECTED_MIN : 0
      EXPECTED_MAX : 10
    END IMPORT

    IMPORT CONSTANT NumAnalogAlt_P6 : INTEGER
      UNITS : NA
      EXPECTED_MIN : 0
      EXPECTED_MAX : 10
    END IMPORT

    IMPORT DigialAlt_P6 : [1 TO NumDigitalAlt] OF INTEGER
      UNITS : ft
      EXPECTED_MIN : 0
      EXPECTED_MAX : 50000
    END IMPORT

    IMPORT CONSTANT Threshold_P6 : INTEGER
    END IMPORT

    IMPORT CONSTANT GoAboveHyst_P6 : INTEGER
      UNITS : ft
```

```
    EXPECTED_MIN : 50
    EXPECTED_MAX : 500

    Purpose : &*L This defines the hysteresis factor for going above
    the threshold altitude. L*&

  END IMPORT

  IMPORT CONSTANT GoBelowHyst_P6 : INTEGER
    UNITS : ft
    EXPECTED_MIN : 50
    EXPECTED_MAX : 500

    Purpose : &*L This defines the hysteresis factor for going above
    the threshold altitude. L*&

  END IMPORT

  IMPORT AnalogAlt_P6 : [1 TO NumAnalogAlt] OF AboveBelowType
  END IMPORT

  IMPORT DigitalQuality_P6 : [1 TO NumDigitalAlt] OF AltitudeQualityType
  END IMPORT

  IMPORT AnalogQuality_P6 : [1 TO NumAnalogAlt] OF AltitudeQualityType
  END IMPORT

  IMPORT INTERFACE AltitudeVoter_P6 :
  END IMPORT

  EXPORT Altitude_P6 : AboveBelowType
  END EXPORT

  EXPORT AltitudeQuality_P6 : AltitudeQualityType
  END EXPORT

END INTERFACE

DEFINITION :
END DEFINITION

END MODULE
```

The interface AltitudeVoter will be used by all the various implementations of the altitude voting algorithm. The specification for each aircraft will decide how many

altimeters and which algorithm to use. This is discussed in greater detail in the next chapter.

Specify the Normal-Case: For this activity, we need to fill in the actual behavior of the IN' and OUT' modules that we have declared. At the end of this activity, it is possible to simulate the entire SOFT relation by connecting the IN', OUT' and REQ relations together. The definition of the Altimeters_IN module is given in Figure 6.10.

We also need to specify the various altitude voting algorithms, but for space concerns these will not be duplicated here. They can be found in Appendix C.

At this point, it is possible to simulate the entire SOFT relation by connecting the IN' OUT' and REQ relations together.

Specify Detailed SOFT Relation: With the preliminary version of the IN' and OUT' relations completed, it is possible to move on and consider the startup, shutdown, and degraded functionality modes of the IN' and OUT' relations.

All the analyses that were done on the REQ relation are also applicable to the IN' and OUT' relations. They should be consistent and (ideally) complete just as the REQ relation was refined to be. In addition, analysis to determine the timing properties of the SOFT relation, and the deviation of the output under noisy data should be performed.

The outcome of this phase is the completed behavioral specification of the SOFT relation.

6.1.7 Iteration Among the Phases

As in CoRE, we felt it important to provide guidance on how the specifier would expect to iterate among the various phases of the idealized process.

```
DEFINITION :

  MODULE_INSTANCE ThresholdedDigital_P6 :
    [1 TO NumDigitalAlt] OF ThresholdedAltitude_P6
    PARENT : NONE
    ASSIGNMENT
      Altitude_P6         := DigitalAlt_P6,
      Threshold_P6        := EXTEND Threshold_P6
                             TO [1 TO NumDigitalAlt] OF INTEGER,
      AboveHysteresis_P6  := EXTEND GoAboveHyst_P6
                             TO [1 TO NumDigitalAlt] OF INTEGER,
      BelowHysteresis_P6  := EXTEND GoBelowHyst_P6
                             TO [1 TO NumDigitalAlt] OF INTEGER
    END ASSIGNMENT
  END MODULE_INSTANCE

  SLOT_INSTANCE AltitudeVoter_P6 :
    ASSIGNMENT
      Num_of_Alt    := NumDigitalAlt_P6 + NumAnalogAlt_P6,
      Altitudes     := ThresholdedDigital_P6.Result_P6 | AnalogAlt_P6,
      Qualities     := DigitialQuality_P6 | AnalogQuality_P6
    END ASSIGNMENT
  END SLOT_INSTANCE

  EXPORT Altitude_P6 :
    PARENT : NONE
    DEFAULT_VALUE : AltitudeVoter_P6.Altitude_P6
    EQUALS AltitudeVoter_P6.Altitude_P6
  END EXPORT

  EXPORT AltitudeQuality_P6 :
    PARENT : NONE
    DEFAULT_VALUE : AltitudeVoter_P6.AltitudeQuality_P6
    EQUALS AltitudeVoter_P6.AltitudeQuality_P6
  END EXPORT

END DEFINITION
```

Figure 6.10: The Definition of the Altimeters_IN module

Constructing Partial Specifications: In the specification process, it is common to have one portion of the specification more refined than another portion. This is sometimes a conscious choice – focusing on some aspects of the system while ignoring others. For example, during the Sensors and Actuators phase, some of the specification will still be at the detailed requirements phase while the effort concentrates on refining the specification of a particular sensor or group of sensors and actuators as described above.

Another situation is when abstracting away certain portions of the computation. For example, in avionics systems sometimes there are complex conditions that must be satisfied for certain mode transitions. These often depend on control laws, continuous functions, etc. that might not included in this stage in the modeling. Furthermore, the way in which these conditions are met is often well understood. Thus, it may be beneficial to delay defining exactly how these conditions are satisfied until later in the specification effort. When this information is added, the new parts of the specification will need to go through the phases of just like the other parts of the specification.

Monitored and Controlled quantities: Sometimes, new monitored and controlled quantities will emerge as the preliminary behavior specification is constructed. There are several reasons why this might be the case. First, additional information from the environment may be needed to be able to compute the values of the controlled quantities. Second, as the system is studied in more detail, it may become clear that there are more controlled variables. Finally, to make it possible to make clear and concise statements about the domain it is sometimes easier to adjust the particular choices of monitored and controlled variables rather than express a very complex relationship between the ones that are already defined.

Draft Requirements and Requirements Structure: It is natural to switch back and forth between the structuring activities and the development of the behavioral specification. Specification of the behavior in more detail may lead to the discovery of modules or pieces of the computation that may be reused across different sections. In addition, it may be desirable to reorganize the computation, or refine the interfaces of the modules. Similarly, as the module structure is developed, new information about how the computation is to be performed may come to light.

The iteration between these activities is similar to the iteration that one would normally see in an object oriented development between the creation of the class diagrams and the creation of sequence diagrams. That is, the creation of a sequence diagram may inspire the creation of a new class (or classes) and the creation of a new class inspires sequence diagrams that use that class.

Detailed Requirements and Prior Phases: When adding the information in the detailed requirements phase, the structures that have been chosen for the requirements may not be conducive to adding fault tolerance, etc. Thus, the requirements may have to be restructured to support these additional behaviors. In general, it is necessary to keep these issues in mind from the beginning of the specification effort, but beneficial to not get bogged down in the details when first understanding the system. This is a delicate balance which becomes easier with experience in specification.

Of course, it is impossible to give a detailed overview of all the possible ways that one might iterate between the various phases of the methodology. We have en devoured here to point out the most common sources of iteration so that a specifier may proceed through the method with an "eye to the future" in the earlier phases.

6.2 Languages for FORM$_{PCS}$

This section describes some of the languages that are suitable for use with the FORM$_{PCS}$ process. The goal of this section is not to be a comprehensive survey of all specification languages; that would consume far to much space in the dissertation. Rather, the goal is to illustrate that FORM$_{PCS}$ is applicable to a wide variety of languages, which was one of the goals in developing the methodology.

Of course, languages are not equally suitable for use with FORM$_{PCS}$. In particular, there is a noticeable difference between languages that support some built-in notion of modularity and reuse and those languages that do not. This is understandably due to the heavy emphasis on reuse and product families in FORM$_{PCS}$.

Statecharts [36, 37, 38] is based around a named-event driven execution model. Events are produced by taking transitions in the state diagram and those events can cause other events, and so forth. These events are *globally visible* and, as a result, it can be very difficult to know how changes to one part of the system can effect other parts.

During the specification of TCAS II (Traffic Alert and Collision Avoidance II) using a Statechart variant (the original version of RSML), Leveson *et al.* discovered that a major source of errors in the specification were due to the event mechanism of Statecharts [65]. In addition, this global event visibility makes it very difficult to reuse pieces of a Statecharts specification or to "simulate" the effect of modules.

SCR (Software Cost Reduction) [45, 46] is a tabular, state-based specification language designed with a formal semantics. The tabular nature of SCR results in a data-flow type specification language where a change in an input variable results immediately in a change in the tables that depend on that input variable. This in turn results in changes in the tables that depend on those tables and so forth until table(s) defining the values of the output variable(s) change. Circular dependencies

are clearly not allowed in SCR, because they would result in potentially infinite recursion. The data flow, non-circular structure of SCR makes is relatively easy to achieve a separation of concerns among the pieces of a computation. Thus, it would be possible in SCR to cut and paste groups of variable definitions to simulate the effect of a modularity construct (this would work similarly for the current version of RSML^{-e}).

The CoRE methodology uses a modified version of SCR as its example. In the modified version, SRC is augmented with "classes" that are essentially groups of elements in the data dependency diagram. Nevertheless, CoRE classes are static and cannot be "instantiated" multiple times within the specification. For example, we could not defined a CoRE class that voted on the altitude and then reuse that class definition n times, once for each altimeter input.

Statecharts, SCR, and RSML^{-e} have all been used successfully on moderately sized process-control systems projects. Nevertheless, it will be challenging to use these languages taking full advantage of the FORM$_{PCS}$ process because their lack of module support. Clearly, if these languages had a module construct they would be the top choice for developing requirements for process-control systems.

Most programming languages like C++, Ada, Java, etc. have excellent modularity features. However, it has been shown that programming languages are simply not suitable for expressing requirements. They contain too many constructs that may be abused during the requirements phase to introduce design and implementation detail.

ADLs were discussed in Chapter 2 in regards to their capabilities in the development of product families. ADLs have rich modularity and module-interconnection concepts – these properties are fundamental to ADLs. However, like programming languages, architecture description languages are oriented specifically towards software design and not software requirements.

The specification languages Z [101, 119], VDM [24, 31] and other logic-style specification languages use predicate and/or propositional logic to record system specifications. Such languages would be well suited to use with FORM$_{PCS}$, because specifying details at an appropriate level of abstraction would not be a problem. In addition, it would be difficult to introduce design and implementation details in such a specification. Furthermore, Z and VDM both have good modularity support and would be able to implement the product family and structuring features of FORM$_{PCS}$.

The drawback of logic-style specification languages such as Z and VDM has proven to be that some stake holders in the project have difficulty understanding the mathematical symbols and concepts used. Of course, engineers and scientists would have no problem with these techniques, but higher-level managers and regulatory agencies may have (and have had) problems with these notations.

The synchronous languages Esterel [8] and Lustre [35, 34, 73, 74] are several languages both supported by the same French commercial company, Esterel Technologies. These languages were developed as specialized programming languages for the process-control community; both are supported by graphical toolsets. In addition, both languages were designed from the beginning to have a simple and elegant syntax and semantics. Therefore, these languages have been at the forefront of the static analysis and code generation for commercial tools. Finally, both languages also have good, albeit simple, module support which would enable them to be used with relative ease in the FORM$_{PCS}$ process.

The primary drawback of Esterel and Lustre is that they were designed to be programming language replacements. Therefore, many features desirable for the requirements level are not in the language or tools (nicely formatted documents, the ability to add descriptive comments, and so forth).

This section has presented which languages FORM$_{PCS}$ might be applied to. Cer-

tainly, we would like FORM$_{PCS}$ to be able to be applied to many languages so that users can choose the language that best suits their needs. Nevertheless, not all languages are created equal. Languages that do not have a builtin modularity construct have difficulties representing the product family and structuring concepts in FORM$_{PCS}$, while languages which do have a module construct have a tendency to be not understandable by all stake holders in the project or they are at too low a level of abstraction. In the next chapter we will address this issue by illustrating how a module construct can be added to RSML^{-e}.

6.3 Summary

This chapter has presented an overview of FORM$_{PCS}$. FORM$_{PCS}$ represents the integration of the product family structuring techniques that were presented in Chapter 4, the overall specification structure and process structure presented in Chapter 5, as well as many techniques that were presented by others in previous research. In addition, in this chapter we have presented a number of ad-hoc structuring techniques that were introduced in the methodology.

Chapter 7

Module Construct for RSML^{-e}

This chapter presents the final contribution of the dissertation, a module construct for the specification language RSML^{-e}. As we discussed at the end of the preceding chapter, reuse and product family structuring in RSML^{-e} must be accomplished via error prone and tedious "cut-and-paste" techniques. Nevertheless, RSML^{-e} is otherwise a good notation for expressing requirements, having many desirable features such as understandability by all stakeholders, simulation capabilities, and formal analysis support. Therefore, there is strong motivation to create a module construct for RSML^{-e}.

First, we will give a high-level overview of the chapter so that the reader may understand the various pieces of the module proposal and how they contribute to achieving the goals that were established for the modules. Next, we move into the actual description of the module construct itself starting with the general syntax and usage, and moving on to placing modules within the state-hierarchy, the functional module reference syntax, initial values for modules, and module interfaces as imports. Finally, we present a summary of the module proposal at the end of the chapter.

7.1 Overview

The incorporation of a module construct in RSML^{-e} allows for full support of the FORM$_{PCS}$ process. In addition, modularity construct has the potential to signifi-

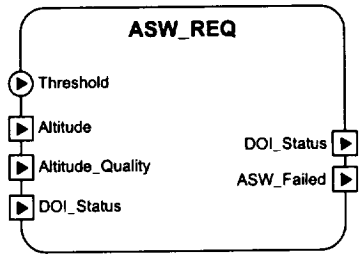

Figure 7.1: The ASW_REQ module, interface diagram

cantly shrink the size of large specifications, and opens up the possibility of performing verification of parts of systems rather than entire systems, potentially making analysis and conceptual understanding of specifications much simpler. Thus, the addition if a module construct to RSML^{-e} has many advantages.

When adding a module construct to RSML^{-e}, several goals must be achieved. First, the module will be the unit of reuse within a specification. Second, the module construct must enable (or be amenable) to the kinds of reuse that will enable analysts to develop specifications for product families and other structuring techniques in FORM$_{PCS}$. Third, the module must support current and future analysis methods. Finally, we would also like, as much as possible, to keep the language *simple* — to minimize the number of concepts necessary to create and explain an RSML^{-e} specification.

In order to be a unit of reuse, a strict module interface must be established and the requirements on the environment of the module be stated explicitly. A graphical representation of a slightly simplified ASW REQ interface is shown in Figure 7.1. If a state variable that exists *inside* of the ASW_REQ module may reference state variables *outside* of the ASW_REQ module then that breaks the encapsulation of the module. Such a module would not be able to be reused in a new context unless the new context provided the same state variables that were referenced outside the

ASW_REQ module as the original context. Therefore, we restrict the state variables inside of a module to referencing only other entities inside of the module and elements of the module's interface. This allows a module to be moved from one context (or specification) to another without fear of breaking the functionality and is key to enabling reuse.

Similarly, state variables *outside* of the module's borders may not access entities inside of the ASW_REQ module. If they did, then it would be impossible to replace the module's implementation with an alternate implementation. The capability to do just that is important for product families where we might like to establish an interface for an algorithm or behavior and have each family member choose *which* algorithm to use. The ASW family does just that with the altitude voting algorithms (discussed in Section 7.6, see Figure 7.3).

As we discussed in the previous chapter, sometimes it is useful to have different behaviors underneath various degraded functionality modes. The modules provide a mechanism for providing a clear grouping of functionality for a particular mode. Furthermore, a module might be parameterized to function with different tolerances and thereby be capable of acting as several different degraded modes depending on its instantiation. Thus, the capability of a module instance to have a parent variable is essential in implementing some of the structuring techniques in FORM$_{PCS}$. We discuss module instances as children of state variables more thoroughly in Section 7.3 and 7.4.

Finally, we would like to have a simple semantics for the language. The modules provide a convenient means to create building blocks of functionality that would otherwise be required to be included in the language definition itself. To make these smaller, building block-style modules easier to use we have introduced the functional module syntax of Section 7.5.

7.2 General Usage

As discussed in Chapter 3, an RSML^{-e} specification can be thought of as a *relation* from the inputs (as set by the input interfaces) to the outputs (those variables send out by the output interfaces). Each state variable in an RSML^{-e} specification contains an assignment relation that represents a piece of the overall relation computed by the specification.

Similarly, a module may be thought of as a relation between the imported variables and the exported variables. Nevertheless, simply providing a relation from the imports to the exports does not allow a module to be computed: we must provide an assignment relation for the imports. Thus, when a module is used, or *instantiated*, within the context of an RSML^{-e} specification then (1) we must provide an assignment relation for all the module's imports, and (2) the module instance provides a piece of the overall relation represented by the specification. In a sense, a module instance may be thought of as just another piece of the relation, albeit a larger aggregate than a state variable.

Modules in RSML^{-e} consist of several parts:

- The **interface** part of the module defines which values are imported into the module and which values are visible to the externally (or exported).
- The **definition** part of the module defines the encapsulated functionality, or the "secret," of the module. The definition part includes the assignment relation of all the exported state variables and it also may contain other state variables which are used in computing the module's exports.
- The **instance** part of the module defines how the module is used within the specification. The instance determines how the imports to the module are to

be provided by the instantiating scope (i.e., the assignment relations of the imports).

We make a distinction between these three parts for good reason. The interface of a module may be shared by several module definitions; this is done for the ASW altitude voting algorithms later in this chapter. This is similar to abstract interface inheritance in object-oriented terms. The module definition part may be instantiated multiple times, with each time providing different assignment relations for the imported variables (thus, a module definition is a kind of "template" for a module instance). This is similar to instantiating a class in object-oriented terms.

As mentioned above, modules define a scope. Within the module definition, the only variables that can be accessible are the imported variables of the module, the exported variables of the module and other variables defined within the module. Similarly, at the scope in which the module is instantiated, the only variables which will be accessible are imported variables (which must be provided by the instantiating scope) and the exported variables. The same is true for macros and functions; those macros and functions that are declared within the module are accessible only within the module and those that are declared outside the module are not accessible inside it. Finally, the simulation time (currently given special treatment) will now be the same as any other variable and will thus have to be imported into each module that desires to use it.

The module construct for RSML^{-e} will have a global scope for all types, module interface declarations, and module definitions. However, the user will be able to declare generic types, similar to templates in C++, within the modules. A generic type mechanism is necessary to allow RSML^{-e} be able to define modules that will adequately replace PREV_VALUE, PREV_ASSIGN, etc. This is discussed in more detail in Section 7.5.

The simplest usage of modules is when the interface is used as an anonymous component of the module definition. For example, the module encapsulating the REQ relation of the ASW might look like the following.

```
TYPE_DEF AltitudeQualityType { Good, Bad }
TYPE_DEF OnOffType { On, Off }

MODULE ASW_REQ :

  INTERFACE :

    IMPORT MON_Altitude : INTEGER
      EXPECTED_MIN : -2000
      EXPECTED_MAX : 50000
      UNITS : ft
    END IMPORT

    IMPORT MON_Altitude_Quality : AltitudeQualityType
    END IMPORT

    IMPORT MON_DOI_Status : OnOffType
    END IMPORT

    EXPORT CON_DOI_Status : OnOffType
    END EXPORT

    EXPORT CON_ASW_Failed : Boolean
    END EXPORT

    IMPORT_CONSTANT Threshold : INTEGER
      EXPECTED_MIN : 0
      EXPECTED_MAX : 8192
      UNITS : ft
    END IMPORT_CONSTANT

  END INTERFACE

  DEFINITION :

      /*
      In here, we have the REQ part of the ASW spec.
      Here, we can only reference IMPORTs, EXPORTs and
      other variables, macros, and functions declared
      within the definition.

      However, we can access the AltitudeQualityType,
```

```
      and other types declared outside of the module.
      */

  END DEFINITION

END MODULE
```

Now that the definition of the module is created, we can use it as many times as desired in the specification. In this case, we probably only want one ASW_REQ, but in other cases, we will use the module definition to "stamp out" many copies of a particular module definition. Each of these module instances must say how all the imports are assigned by the scope in which the instance is declared. For the ASW_REQ module, the module instance declaration is given below.

```
MODULE_INSTANCE ASW_REQ_Instance : ASW_REQ
  PARENT : NONE

  MON_Altitude := IN_Altitude
  MON_Altitude_Quality := IN_Altitude_Quality
  MON_DOI_Status := IN_DOI_Status
  Threshold := 2000

END MODULE_INSTANCE
```

These expressions defined the assignment relation for the imported variables. One view the expressions as simply an assignment relation with one clause, $i = e$ if TRUE, where ii s the imported variable and e is the expression

As a second example, in the FGS specification there is a left and a right FGS. One way to model this situation might be to define an FGS module that would then be instantiated several times. For example,

```
MODULE FGS :

  INTERFACE :
     /*
     FGS imports and exports are defined here
     */
  END INTERFACE
```

```
  DEFINITION :
     /*
     State variables, Export definitions, and other entities
     defining the behavior of the FGS are defined here
     */
  END DEFINITION

END MODULE

MODULE_INSTANCE LeftFGS : FGS
   PARENT : NONE

   /*
   This module instance defines that assignment relation for
   the imports of the left side FGS
   */

END MODULE_INSTANCE

MODULE_INSTANCE RightFGS : FGS
   PARENT : NONE

   /*
   This module instance defines the assignment relation for
   the imports of the right side FGS.
   */

END MODULE_INSTANCE
```

Note that FGS module may be instantiated any number of times. For example, if we wanted a center FGS in addition to the Left and Right, we could very easily do this by simply adding another MODULE_INSTANCE block to the above.

We can see from the examples above that the module instance definition includes a PARENT clause. This allows a module instance to be placed as a child underneath a state variable in the instantiating scope. Exactly how this is done and what it means for a module to be placed under a state variable is explained in the next section.

7.3 Module Instances Within the Hierarchy

As mentioned earlier, each state variable and module instance in an RSML^{-e} specification is a piece of the overall relation that is computed by the specification. When we say that a state variable is a *child* of another state variable in RSML^{-e} that is a statement that the child variable is only *relevant* when the parent variable has a particular value; that is, we do not care what the value of the child variable is if the parent does not have the appropriate value and the RSML^{-e} semantics state that the value of such a variable is UNDEFINED.

This view extends naturally to module instances, which are really just a larger aggregate of the overall specification. For example, the FGS has a Flight_Director variable that governs whether or not the FGS shall produce outputs controlling the aircraft (this can be equal to 'On' or 'Off'). On way that the FGS might be modeled this way if the Flight_Director state machine was placed at the top level and the behavior for the FGS were place beneath the On state as in the example below.

```
STATE_VARIABLE Flight_Director : OnOffType

  DEFAULT_VALUE : Off
  EQUALS On IF OnButtonPressed()
  EQUALS Off IF OffButtonPressed()

END STATE_VARIABLE

MODULE_INSTANCE FGS : FGS_Functionality
    PARENT : Flight_Director.On

    /* Define how the imports to FGS_Functionality are
       provided. */

END MODULE_INSTANCE
```

If the FGS is in the Off state, then any values computed by FGS_Functionality should not matter; thus, the semantics defines them to be UNDEFINED. In more detailed terms, the declaration of a 'child' module instantiation at location X is the

same as directly declaring the module's top-level variables as child state variables of X.

As a second example, consider the degraded modes of functionality in the ASW. For each value of ASW_System_Mode a module is instantiated that represents the behavior of the ASW in that mode. Below we have reproduced the top-level mode that controls which module is active.

```
STATE_VARIABLE ASW_System_Mode_P5 :
  VALUES : {Startup, NormalOperating, Degraded, Failed, Reset}
  PARENT : NONE

  Purpose : &*L This is the top-level mode of the ASW.  If the ASW
  were to have a startup mode, etc., we could put those modes as
  children of this controlling mode.  Currently, we have only two
  states, the reset mode which is used for when the reset signal
  is received and the operating mode that handles the main
  behavior. L*&

  DEFAULT_VALUE : Startup

  TRANSITION NormalOperating TO Reset IF MON_Reset_P5

  TRANSITION Degraded TO Reset IF MON_Reset_P5

  TRANSITION NormalOperating TO Degraded IF
    EpisodeMonitor_P5 = QualifyingEpisode

  TRANSITION Degraded TO NormalOperating IF
    DURATION (MON_Altitude_Quality_P5 = Valid, 0 S, Clock) > 1 MIN

  TRANSITION Reset TO NormalOperating IF
    DURATION(PRE(ASW_System_Mode_P5), 0 s, Clock) >= 0 S

END STATE_VARIABLE
```

7.4 Initial Values

A state machine is usually defined with an initial configuration, a set of states, and a set of transitions. Not surprisingly, in RSML^{-e} an initial configuration is also

necessary. Currently, this initial configuration is supplied by the user in the form of `INITIAL_VALUE` clauses supplied for each input variable and state variable. The current initial configuration is static and specified completely by the user of the language.

This method of determining the initial configuration has proven to be troublesome. First, it makes it difficult to reorganize the state hierarchy. Consider the simple state hierarchy in Figure 7.2. At the top of the hierarchy, there is the state variable X, that can take on values 'a', 'b', or 'c'. X has two children, Y1 and Y2. Y1 exists under the value 'a' of X and Y2 exists under the value 'b'. Because the initial value of X is equal to 'a', we must choose an initial value for Y1 (which is 'd' in the figure). However, because the initial value of X is not equal to 'b,' the initial value of Y2 is *required* to be UNDEFINED. This applies transitively to Z1 and Z2.

A minor change in the organization of the state hierarchy can, thus, cause major changes in the initial configuration of the machine. Suppose, for example, that the initial value of X was changed from 'a' to 'b.' Then, we would be *required* to specify initial values for Y2, Z1, and Z2 whereas no initial values had been specified before. We may experience similar difficulties when moving state variable Y2 underneath a different parent variable.

This property presents particularly difficult issues for the modules. Recall from the previous section that a module instance may be placed under a state variable and that when this is done all the top-level state variables inside the module essentially become children of that variable. Unfortunately, there is no way to know what an encapsulated state variable's initial value should be. This is because whether or not the initial value should be UNDEFINED or one of the state variable's values is potentially dependent on the parent of the module instance. However, the initial value would be specified in the module *definition*; therefore, it is possible to have

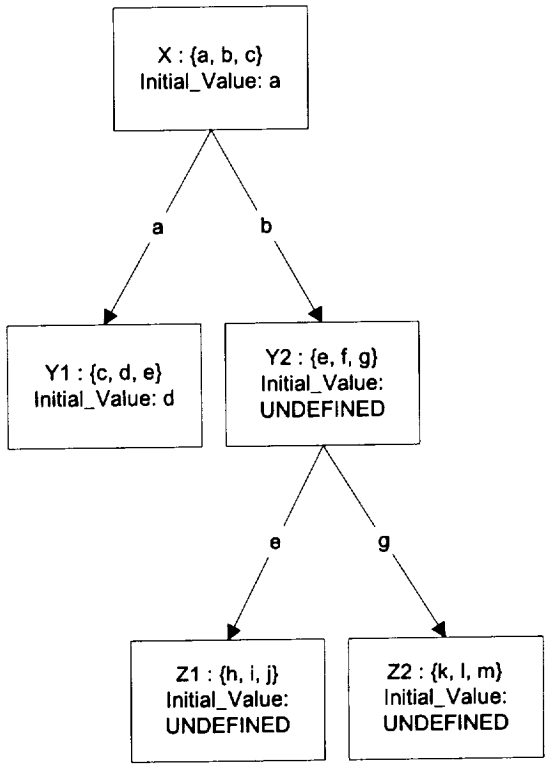

Figure 7.2: Initial Values of State Variable

conflicting demands as to what the initial value of an encapsulated state variable should be for module definitions that are instantiated several times under different parent state variables. Clearly, this is not a workable situation.

Nevertheless, we must have some way to determine an initial value for all the variables in the specification. And, this method must not break the encapsulation of the modules. Therefore, instead of an initial value, it would be better for state variables to have a notion of a default value. Input variables would retain an initial value but all state variables would be determined by evaluating their assignment relations as if they had been UNDEFINED in the previous step. This allows the initial configuration of the specification to be computed rather than statically specified and solves the above problems. The default value would not be dependent on context, so the state variable (or module instance) could function correctly at any position in the state variable hierarchy.

7.5 Functional Module Syntax

In the above examples, modules have been used to encapsulate relatively large portions of the RSML^{-e} specification. These large-scale building blocks can then be assembled to form the overall specification.

Modules can also be viewed as much smaller building blocks; this view is common in the data-flow language Lustre [34]. When used in this way, we can leverage the power of the module construct to replace some complex language features with module implementations and thereby simplify the underlying language semantics.

For example, the semantics of the PREV_VALUE and PREV_ASSIGN expressions are complex and it is difficult to prove properties about them. Currently, the formal semantics of RSML^{-e} states that the entire history of variable assignments is recorded, along with the time of assignment. RSML^{-e} provides three expressions to access

the variable history lists: PREV_STEP, PREV_VALUE, and PREV_ASSIGN. In practice, specifiers tend not to use PREV_VALUE and PREV_ASSIGN, finding them both too complex and too restrictive. Also, to determine the PREV_STEP value of a variable, it is necessary to check whether the variable was assigned in the current step.

Furthermore, any translations from RSML^{-e} to other languages must account for these complex expressions in the translation; expressions which many desired target languages, e.g., SMV [75, 87], do not have in their semantics. Using the modules, we can express the same concepts, but with a simpler semantics and translation.

The only problem is one of syntax: it is inconvenient to be required to declare a named module instance for each time that we desire to use one of these sorts of modules. Therefore, modules that do not import module interfaces and have only one export can be thought of as "functions with state." We provide an alternative syntax for these types of modules, similar to that provided by Lustre, that is in the style of a function call: ModuleBodyType(expr1, expr2, ..., exprN).

Although this syntax is only available to modules with one output, it is quite useful for defining expressions like: DURATION, PREV_VALUE, PREV_ASSIGN, etc. As we indicated above, this syntax allows us to simplify the language semantics significantly with respect to these constructs while simultaneously increasing the expressiveness.

In the next version of RSML^{-e}, the only values for variables which will be available are the value at the beginning of the step, or PRE(X), and the value at the end of the step (or current value) which will be referenced as it is now. Instead of recording histories, only the value of a variable from the previous step and its current value need be recorded. This approach is similar to many other languages such as SCR [43], Lustre [34], Esterel [8], Z [102], SMV [75, 87], and many more.

Using just these two constructs, we can define modules to sample the value stream of a variable at arbitrary points. This mechanism provides both a formally simpler and more flexible scheme for recording variable histories. For example, the following illustrates the construction of a PREV_VALUE module.

```
MODULE PREV_VALUE :
  INTERFACE :

    /* Since the PREV_VALUE module should be able to operate on
       many different types of variables, it has a declared
       generic type.  Values of generic type in RSML-e can be
       compared for equality and inequality, but arithmetic
       operations are not allowed
     */

    GENERIC_TYPE G

    /* The import T is the variable that we want to sample */

    IMPORT Variable : G
    END IMPORT

    /* The import InitialValue is the initial value of the
       result of the PREV_VALUE module */

    IMPORT_CONSTANT InitialValue : G
    END IMPORT_CONSTANT

    /* The export PreviousValue gives the result of the
       PREV_VALUE module */

    EXPORT PreviousValue : G
    END EXPORT

  END INTERFACE

  DEFINITION :

    /* This defines the assignment relation for the Previous-
       Value exported variable.  We can see that it is initially
       set equal to the InitialValue import.  Then, if the
       value changes, the value of PreviousValue is updated.
     */

    EXPORT PreviousValue :
```

```
    DEFAULT : InitialValue
    EQUALS PRE(PreviousValue) IF PRE(PreviousValue) = Variable
    EQUALS PRE(Variable) IF PRE(PreviousValue) != Variable
  END EXPORT

 END DEFINITION
END MODULE
```

This implementation of PREV_VALUE provides the same functionality as in the current production version of the tools. Suppose that we had a variable, X of which we wanted the previous value. We could simply write `PREV_VALUE(X, UNDEFINED)`.

Currently, times are integral in the current formal semantics of RSML^{-e}. Times are recorded along with each variable assignment and are used for computing all of the PREV expressions. Of course, the times are useful for their own sake; they can be used to determine how long a variable has held a particular value, for instance.

Nevertheless, this integration of times into the variable history is a mistake for the following reasons: (1) it is unnecessary for many variables and (2) the time expressions that are derivable from this scheme are not very flexible. Furthermore, with modules being able to describe PREV_VALUE and PREV_ASSIGN expressions better than the current variable histories, there is little reason to keep variable histories around.

Appendix A gives the definition of the other standard modules that we have defined and are meant to be used with any RSML^{-e} specification under the new version of the language.

7.6 Module Interfaces as Imports

Each module will have one and only one interface that defines the boundaries of the module. A module's interface may be declared separately as a named entity (like a type) and, therefore, several modules may share the same interface. This capability might be used, for example, to define a number of different tracking, hysteresis, or

error correcting behaviors. Furthermore, an interface may import other interfaces. An imported interface acts like a "slot;" any module which supports the imported interface may be plugged in by the instantiating scope. This makes it possible, for example, to encapsulate algorithms and leave the choice of *which* particular algorithm to use to the enclosing scope of the module.

Why might we like to do this? Consider the ASW example defined earlier. Now, we wish to refine our REQ model of the ASW to REQ' so that we can add the IN' and OUT' relations. However, the first problem that we encounter is that for our family of ASW we have both analog altimeters which only give above and below the threshold and digital altimeters which give a number for the altitude. Furthermore, suppose that the different aircraft in our product family have a different number of altimeters. For example, a commercial jet may have two analog and two digital altimeters whereas a personal aviation craft may only have one of each.

To complicate matters even more, different customers demand different algorithms for determining, from a variety of altitude data, whether or not we are above or below the threshold value. One customer wants the altitude to be considered "below" when at least one altimeter reads below; another customer wants all of the altimeters to be below before it is declared; finally, a third customer wants a majority of the altimeters to be below before the altitude is declared below.

We can deal with this kind of complexity by refining the REQ relation so that the ASW_REQ' module imports the AltitudeVoter interface. Then, the enclosing scope may choose which implementation of AltitudeVoter satisfies the particular customer's requirements. This is given in textual format below and graphically in Figure 7.3.

```
MODULE ASW_REQ_Prime :

  INTERFACE
    IMPORT INTERFACE AltitudeVoter
    END IMPORT
```

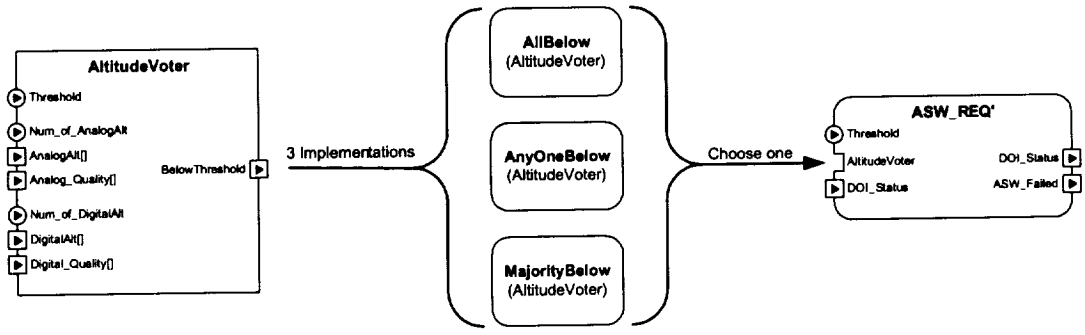

Figure 7.3: The ASW_REQ' model illustrating the utility of nested interface definitions

```
    /* Other imports and exports of ASW_REQ here */

  END INTERFACE

  DEFINITION :

    SLOT_INSTANCE VotedAltitude : AltitudeVoter

      /* Inside of ASW_REQ, we instantiate the module
         that was plugged in so that we can use the
         imports and exports of the voting algorithm
         in the REQ relation
       */

    END SLOT_INSTANCE

  END DEFINITION
END MODULE
```

The full specification of the individual altitude voting algorithms may be found in Appendix C.

7.7 Conclusion

In this chapter, we have presented the essentials of the module additions to the next version of RSML^{-e}. This module construct will provide RSML^{-e}with the ability to

fully support the FORM_{PCS} process as well as providing specifiers with many more options in organizing specifications in RSML^{-e}, a cleaner RSML^{-e} semantics, and the possibility of new analysis techniques based on the module structure.

Chapter 8

Conclusion and Future Directions

This dissertation has visited a number of different topics, from structuring of product families in Chapter 4 to a methodology for safety-critical process-control systems in Chapters 5 and 6, to the addition of a module construct to RSML^{-e} in Chapter 7. In this final chapter, we take a step back and revisit how the different pieces of the work connect with one another. We also look to the future to see in which directions the work presented in the dissertation is expected to progress.

8.1 Conclusions

The goal of the work presented here was to reduce some major barriers to industrial acceptance of formal specification techniques. Of course, achieving industrial acceptance of formal methods is not something that may be addressed within the scope of one doctoral dissertation. But specifically, we wanted provide guidance on how to construct formal specifications and to make it easier to develop a formal specification for a family of products so as to facilitate reuse.

The dissertation has three main contributions: (1) the development of structuring techniques for formal requirements specifications, including family structuring techniques, (2) the addition of the module construct to RSML^{-e}, and (3) the integration of this work with existing work to form a comprehensive methodology for developing formal specifications of process-control systems.

We presented how the the contributions of this work, while they may seem to be from somewhat unrelated areas, fit into an overall framework (first shown in Figure 1.1 and now given again in Figure 8.1). The framework helped to organize the dissertation and we will use it here to review the information that was covered.

We began in Chapter 4 by discussing the most general and fundamental contributions of the dissertation: those in product family structuring. As we began to look at structuring the specifications for product families, it became clear that current work in product families did not have a clear separation between the *requirements* for a product line and the design and implementation of the product line. The structuring technique in Chapter 4 allows for this separation – this work first appeared at the Fifth International Requirements Engineering conference [106]. Furthermore, it is important to point out that this work is applicable to all software systems, not just the safety-critical process-control systems of interest to this dissertation.

In the next chapter, we started to build the foundation for what would eventually become our methodology for process-control systems. This chapter primarily contains the early work on process and structure that was presented in [109, 104]. Then, in Chapter 6, we have given the practitioner's view of the methodology, detailing the various phases and activities.

Chapter 6 also contains a number of different smaller contributions that were made as the methodology was developed. For example, the classification scheme for the different types of commonly encountered monitored and control variables, and the structuring method to deal with failure modes and degraded modes of functionality. The methodological work presented in Chapters 5 and 6 is applicable to many different specification languages and we gave examples of a number of them at the end of Chapter 6.

Finally, in Chapter 7 we have given a proposal for the next version of RSML^{-e},

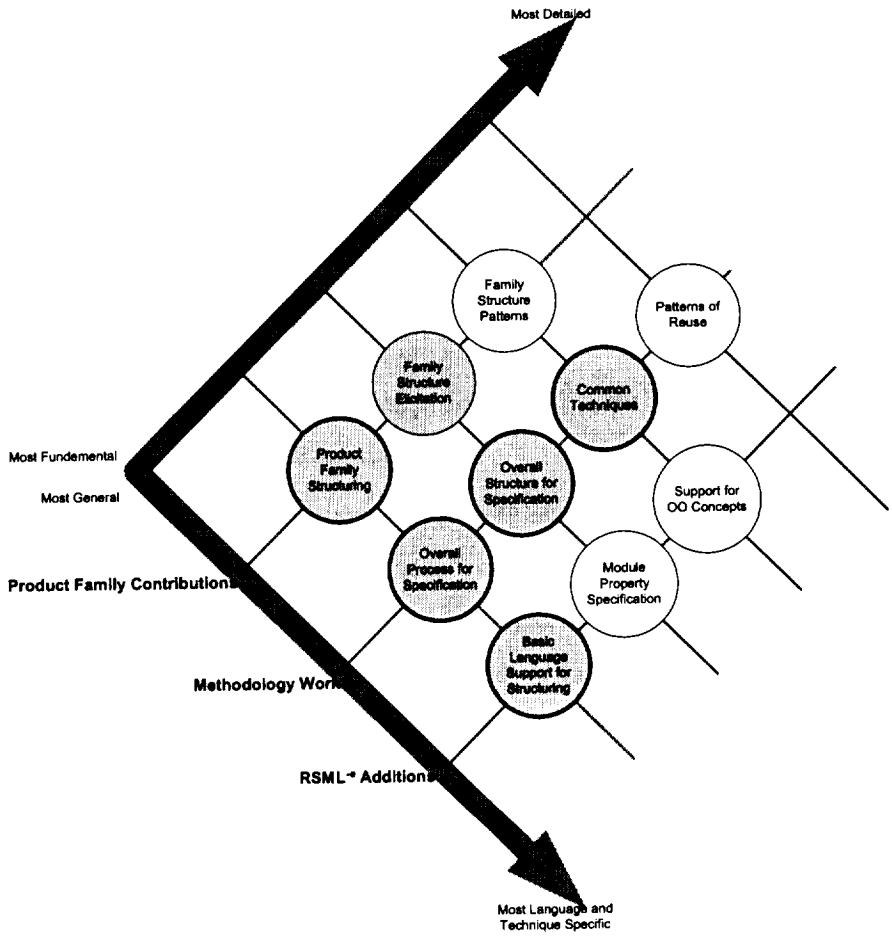

Figure 8.1: Framework of Contributions. Bubbles with a Bold outline indicate areas of contribution by this dissertation; bubbles with a grey background indicate areas where significant research results have been achieved.

including how to add a module construct to RSML^{-e} that will better support the FORM$_{PCS}$ methodology.

Work such as the research presented in this dissertation is inherently difficult to evaluate. No easy experiments can be run to validate the work; and, long term studies (i.e., lasting several years) cannot be done in the scope of doctoral work. Furthermore, corporations are not generally willing to have their own full-time employees adopt a new and unproven methodology or techniques so evaluation in an industrial setting proves to be a challenge. Nevertheless, it is important for some evaluation of the work to be done, so what kind of evaluation and validation was done on the work presented here?

One of the best ways to evaluate process and methodological work is to publish it so that others in the field can critique and review it. All of the early and foundational work of this dissertation has been published in top conferences and/or journals that are related to the topic. Feedback from these venues has been uniformly positive.

Second, we have been successful at evaluating these techniques in an industrial setting despite the difficulties involved in doing so. Through the Critical Systems Research Group's partnership with Rockwell-Collins, many of the techniques presented in this dissertation have been evaluated and used on their family of flight guidance systems. In addition, the author personally evaluated the techniques during a summer study at Medtronic, where the methodology and structuring techniques were applied to a family of implantible pacemakers and defibrillators. The results of both of these experiences have been positive and have validated the proposed techniques. Unfortunately, details of these evaluations cannot be made public due to the proprietary and sensitive nature of the systems developed within both Medtronic and Rockwell-Collins.

In summary, in this dissertation we have developed a number of structuring tech-

niques and an overall methodology for process-control systems that should allow these techniques to achieve a new level of acceptance in industry. Furthermore, we have added a module construct to the language RSML^{-e} that will allow RSML^{-e} to better support the aforementioned methodology and techniques. Finally, we are confident that these techniques are usable by industry and make significant contributions to the field due to their positive reviews from leaders in the field, successful publication in leading conferences and journals, and successful application to real-world industrial-sized problems in two separate organizations.

8.2 Future Directions

There are many avenues of future directions for the work presented in this dissertation. The most immediate are given in Figure 8.1. Generally, the future work may be broken down into the same directions as the dissertation; thus, future work is possible in product families, the methodology, and the additions RSML^{-e}.

In the product family area, there is work still to be done on establishing a separation of concerns between the product family requirements and the product family design and implementation. As the first white bubble in Figure 8.1 shows, additional work is needed in how to elicit the structure of a product family. Current techniques do not provide guidance on how to discover the dimensions of the product family and our work in this area is preliminary.

The second product family bubble listed in Figure 8.1 is family structure patterns. While we have provided a structuring technique for product lines, there must be patterns in product families that can be identified (similar and analogous to the work that has been done by the software architectures). These common patterns can then be used accross many similar product families. Furthermore, this work will have a strong relationship to the research being done in architectural patterns.

Along the methodology aspect of Figure 8.1, additional ad-hoc structuring techniques can be developed ad-nauseum as more patterns of specification language use become known. Also, as shown in the last white bubble along the methodology aspect, we may like to add a section to the methodology that specifically mentions patterns of reuse in the specification domain. Finally, the methodology should be further refined on more industrial sized case studies and then published as a textbook so that it can gain wide audience.

The final aspect of contribution shown in Figure 8.1 is the future work with RSML^{-e}.

In this dissertation, we have provided a basic module construct that meets the needs of FORM$_{PCS}$. In the future, it will probably be desirable to add some kind of property specification language to the module interfaces (the first white bubble in the figure). This would allow the user to specify end-to-end properties on the module that could then be verified formally. Furthermore, properties already verified about sub-modules could be used in the verification of properties on the enclosing module.

The second white bubble in the figure along the RSML^{-e} aspect shows that it would be nice if the module construct supported the object oriented notion in a more full-featured way. Currently, we allow interfaces as imports, but it would be ideal to allow for a notion of inheritance among modules in RSML^{-e}.

Finally, we have a specification of the array concept for RSML^{-e} that is not yet fully completed that would complement our work with the modules. This array construct needs to be added to the language so that we can complete the work in elminating complex expressions from the language and so that it is easier to specify large systems that are likely to make use of the array construct.

In summary, there is much that could be done in the future based on the work presented in this dissertation. All aspects of the research – product families, the

methodology, and RSML^{-e}– contain areas for future research.

Bibliography

[1] Activmedia robotics website. Makers of the Pioneer robot. http://www.activrobots.com/.

[2] R. Allen. *A Formal Approach to Software Architecture.* PhD thesis, Carnegie Mellon University, May 1997.

[3] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[4] Mark Ardis, Nigel Daley, Daniel Hoffman, and Harvey Siy. Software product lines: A case study. *Software Practice and Experience*, 2000. To Appear.

[5] Mark A. Ardis and David M. Weiss. Defining families: The commonality analysis. In *Nineteenth International Conference on Software Engineering (ICSE'97)*, pages 649–650, 1997.

[6] L. Baum, M. Becker, L. Geyer, and G. Molter. Mapping requirements to reusable components using design spaces. In *The Fourth International Conference on Requirements Engineering (ICRE'00)*, June 2000.

[7] Lothar Baum, Lars Geyer, Georg Molter, Steffen Rothkugel, and Peter Sturm. Architecture-centric software development based on extended design spaces. In *Development and Evolution of Software Architectures for Product Families: The Second International Workshop on Development and Evolution of Software Architectures for Product Familes (ARES)*, number 1429 in Lecture Notes in Computer Science, pages 197–204. Springer, February 1998.

[8] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[9] P. Binns, M. Engelhart, M. Jackson, and S. Vestal. Domain-specific software architectures for guidance, navigation, and control. *International Journal of Software Engineering and Knowledge Engineering*, 6(2), 1996.

[10] B. Boehm. *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[11] Barry Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, May 1988.

[12] Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach.* Addison-Wesley, 2000.

[13] K.H. Britton, R.A. Parker, and D.L. Parnas. A procedure for designing abstract interfaces for device interface modules. In *Fifth International Conference on Software Engineering*, 1981.

[14] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, March 1986.

[15] Lisa Brownsword and Paul Clements. A case study in successful product line development. Technical Report CMU/SEI-96-TR-016, Software Engineering Institute, Carnegie-Mellon University, October 1996.

[16] Grady H. Jr. Campbell, Stuart R. Faulk, and David M. Weiss. Introduction to synthesis. Technical Report INTRO-SYNTHESIS-PROCESS-90019-N, Software Productivity Consortium, Herdon, VA, 1990.

[17] Paul C. Clements. A survey of architecture description languages. In *Proceedings of the Eighth International Workshop on Software Specification and Design*, March 1996.

[18] James Coplien, Daniel Hoffman, and David Weiss. Commonality and variability on software engineering. *IEEE Software*, 15(6):37–, November/December 1998.

[19] David A. Cuka and David M. Weiss. Specifying executable commands: An example of FAST domain engineering. Technical report, Lucent Techonologies, unknown. Submitted to Transactions on Software Engineering.

[20] David Dikel, David Kane, Steve Ornburn, William Loftus, and Jim Wilson. Applying software product-line architecture. *IEEE Computer*, 30(8):49–55, August 1997.

[21] Tom Dolan, Ruud Weterings, and J.C. Wortman. Stakeholders in software-system family architectures. In Frank van der Linden, editor, *Development and Evolution of Software Architectures for Product Families: Second International ESPRIT ARES Workshop*, number 1429 in Lecture Notes in Computer Science, pages 172–187. Springer, February 1998.

[22] Debra M. Erickson. Structuring formal requirements specifications for reuse: A mobile robotics case study. Masters Project, University of Minnesota, April 2000.

[23] Stuart R. Faulk. Product-line requirements specification (PRS): An approach and case study. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering (RE'01)*, pages 48–55, August 2001.

[24] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems: Practical Tools and Techniques in Software Development.* Cambridge University Press, 1998.

[25] D. Garlan, R. Allen, and J. Ockerbloom. Exploting style in architectural design environments. In *Proceedings SIGSOFT'94: Foundations on Software Engineering*, pages 175–188, December 1994.

[26] David Garlan. A introduction to the Aesop system, July 1995. http://www.cs.cmu.edu/afs/cs/project/able/www/aesop/html/aesop-overview.ps.

[27] Nancy G.Leveson. Intent specifications:an approach to building human-centered specifications.

[28] Hassan Gomaa. Object oriented analysis and modeling for families of systems with uml. In *The Sixth International Conference on Software Reuse (ICSR)*, number 1844 in Lecture Notes in Computer Science, pages 89–99. June, June 2000.

[29] M. Gorlick and A. Quilici. Visual programming in the large verses visual programming in the small. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 137–144, October 1994.

[30] M. Gorlick and R. Razouk. Using Weaves for software construction and analysis. In *Proceedings of the Thirteenth International Conference on Software Engineering (ICSE'91)*, pages 23–34, May 1991.

[31] The VDM Tool Group. The IFAD VDM^{++} toolbox user manual. Technical Report, IFAD-VDM-43. Available from IFAD, Forskerparken 10, 5230 Odense M, Denmark, September 1997.

[32] Carl A. Gunter, Elsa L. Gunter, Michael Jackson, and Pamela Zave. A reference model for requirements and specifications. *IEEE Software*, 17(3):37–43, May/June 2000.

[33] Neeraj K. Gupta, Lalita J. Jagadeesan, Eleftherios E. Koutsofios, and David M. Weiss. Auditdraw: Generating audits the FAST way. In *Third IEEE International Symposium on Requirements Engineering (RE'97)*, pages 188–197, 1997.

[34] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[35] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems.* Klower Academic Press, 1993.

[36] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[37] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[38] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Transactions of Software Engineering and Methodology*, 5(4):293 – 333, October 1996.

[39] Mats P. E. Heimdahl and Nancy G. Leveson. Completeness and consistency in hierarchical state-base requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.

[40] Mats P.E. Heimdahl, Jeffrey M. Thompson, and Barbara J. Czerny. Specification and analysis of intercomponent communication. *IEEE Computer*, pages 47–54, April 1998.

[41] Mats P.E. Heimdahl, Jeffrey M. Thompson, and Steven P. Miller. Product families, formality, and reuse: A guide to the FORM$_{PCS}$ method. Technical report, University of Minnesota, 2002.

[42] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance, COMPASS 95*, 1995.

[43] C. L. Heitmeyer, B. L. Labaw, and D. Kiskis. Consistency checking of SCR-style requirements specifications. In *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, March 1995.

[44] C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

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

[46] K.L. Heninger, J.W. Kallander, J.E. Shore, and D.L. Parnas. Software Requirements for the A-7e Aircraft. Technical Report 3876, Naval Research Laboratory, Washington, D.C., November 1978.

[47] Michael Jackson. *Software Requirements and Specifications.* ACM Press and Addison-Wesley, 1995.

[48] Michael Jackson. The world and the machine. In *Proceedings of the 1995 Internation Conference on Software Engineering*, pages 283–292, 1995.

[49] Michael Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems.* ACM Press and Addison-Wesley, 2001.

[50] Michael Jackson and Pamela Zave. Domain descriptions. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 56–64, 1992.

[51] Michael Jackson and Pamela Zave. Deriving specifications from requirements: An example. In *Proceedings of the Seventeenth International Conference on Software Engineering (ICSE'95)*, pages 15–24, May 1995.

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

[53] Mehdi Jazayeri, Alexander Ran, and Frank van der Linden. *Software Architecture for Product Families: Principles and Practice.* Addison-Wesley, 2000.

[54] G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28(4es), December 1996.

[55] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the Eleventh European Conference on Object-Oriented Programming (ECOOP'97)*, number 1241 in Lecture Notes in Computer Science, pages 220–242. Springer-Verlag, June 1997.

[56] Grego Kiczales, Erik Hilsdale, Jim Hungunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the Fifteenth European Conference on Object-Oriented Programming*, number 2072 in Lecture Notes in Computer Science. Springer-Verlag, June 2001.

[57] Juha Kuusela and Juha Savolainen. Requirements engineering for product families. In *Proceedings of the Twenty-Second International Conference on Software Engineering (ICSE'00)*, pages 60–68, June 2000.

[58] W. Lam. Achieving requirements reuse: A domain-specific approach from avionics. *Journal of Systems and Software*, 38(3):197–209, 1997.

[59] W. Lam. Creating reusable architectures: Initial experience report. *ACM SIGSOFT Software Engineering Notes*, 22(4):39–43, 1997.

[60] W. Lam. Developing component-based tools for requirements reuse: A process guide. In *Eighth International Workshop on Software Technology and Engineering Practice (STEP'97)*, pages 473–483, 1997.

[61] W. Lam, J.A. McDermid, and A.J. Vickers. Ten steps towards systematics requiremens reuse. *Requirements Engineering*, 2(2):120–113, 1997.

[62] W. Lam and B.R. Whittle. A taxonomy of domain-specific reuse problems and thier resolutions - version 1.0. *ACM SIGSOFT Software Engineering Notes*, 21(5):72–77, September 1996.

[63] Thomas G. Lane. Studying software architecture through design spaces and rules. Technical Report CMU/SEI-90-TR-18, Software Engineering Institute, Carnegie Mellon University, November 1990.

[64] Nancy G. Leveson. Sample tcas intent specification.

[65] Nancy G. Leveson, Mats P.E. Heimdahl, and Jon Damon Reese. Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, volume 1687 of *LNCS*, pages 127–145, September 1999.

[66] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering*, 20(9):684–706, September 1994.

[67] Robyn R. Lutz. Safety analysis of requirements for a product family. In *1998 International Conference on Requirements Engineering (ICRE'98)*, 1998.

[68] Robyn R. Lutz. Toward safe reuse of product family specifications. In *Symposium on Software Reusability (SSR'99)*, 1999.

[69] Robyn R. Lutz. Extenting the product family approach to support safe reuse. *Journal of Systems and Software*, 53:207–217, 2000.

[70] J. Magee, N. Dulay, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the Fifth European Software Engineering Conference (ESEC'95)*, pages 137–153, September 1995.

[71] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of the ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering*, pages 3–14, October 1996.

[72] Jeff Magee, Jeff Kramer, and Dimitra Giannakopoulou. Behaviour analysis of software architectures. In *First Working IFIP Conference on Software Architecture (WISCSA1)*, Feburary 1999.

[73] F. Maraninchi and Y. Rémond. Applying formal methods to industrial cases: The language approach (the production-cell and mode-automata). In *Proc. 5th International Workshop on Formal Methods for Industrial Critical Systems*, April 2000.

[74] Florence Maraninchi and Yann Rémond. Mode-automata: About modes and states for reactive systems. In *Proc. European Symposium on Programming*, 1998.

[75] Kenneth L. McMillan. Symbolic Model Verifer (SMV) - Cadence Berkeley Laboratories Version. Available at http://www-cad.eecs.berkeley.edu/~kenmcmil/smv.

[76] N. Medvidovic, P. Oreizy, J.E. Robbins, and R.N. Taylor. Using object-oriented typing to support architectural design in the C2 style. In *Proceedings of the ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering*, pages 24–32, October 1996.

[77] N. Medvidovic, D.S. Rosenblum, and R.N. Taylor. A language and environment for architecture-based software development and evolution. In *Proceedings of the Twenty-first International Conference on Software Engineering (ICSE'99)*, pages 44–53, Los Angeles, CA, May 1999.

[78] Nenad Medvidovic and David S. Rosenblum. Domains of concern in software architectures and architecture description languages. In *Proceedings of the 1997 USENIX Conference on Domain-Specific Languages*, October 1997.

[79] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

[80] Sandra K. Miller. Aspect-oriented programming takes aim at software complexity. *IEEE Computer*, 34(4):18–21, April 2001.

[81] Steven P. Miller. Specifying the mode logic of a flight guidance system in CoRE and SCR. In *Proceedings of the Second Workshop on Formal Methods in Software Practice*, pages 44–53, 1998.

[82] Steven P. Miller. Modeling software requirements for embedded systems. Technical report, Advanced Technology Center, Rockwell Collins, Inc., 1999. In Progress.

[83] Steven P. Miller and Alan C. Tribble. Extending the four-variable model to bridge the system-software gap. In *Proceedings of the Twentith IEEE/AIAA Digital Avionics Systems Conference (DASC'01)*, October 2001.

[84] M. Moriconi, X. Qian, and R.A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, April 1995.

[85] M. Moriconi and R.A. Riemenschneider. Introduction to SADL 1.0: A language for specifying software architecture hierarchies. Technical Report SRI-CSL-97-01, Carnegie Mellon University, March 1997.

[86] Gleb Naumovich, George S. Avrunin, Lori A. Clarke, and Leon J. Osterweil. Applying static analysis to software architectures. In *Proceedings of the Sixth European Software Engineering Conference (ESEC'97)*, number 1301 in Lecture Notes in Computer Science, pages 77–93. Springer-Verlag, 1997.

[87] NuSMV: A New Symbolic Model Checking. Available at http://http://nusmv.irst.itc.it/.

[88] D.L. Parnas. On the criteria to be used in decomposing a system into modules. *Communications of the ACM*, 15:1053–1058, December 1972.

[89] D.L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(1):1–9, March 1976.

[90] D.L. Parnas. Designing software for ease of extension and contraction. In *Third International Conference on Software Engineering*, 1978.

[91] D.L. Parnas and P.C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, 1986.

[92] D.L. Parnas, P.C. Clements, and D.M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, 11(3):256–266, 1985.

[93] D.L. Parnas and J. Madey. Functional documentation for computer systems engineering. *Science of Computer Programming*, 25(1):41–61, 1991.

[94] Praxis Critical Systems Limited. *REVEAL: A Keystone of Modern Systems Engineering*, issue 1.1 edition, July 2000.

[95] R. Prieto-Diaz. Domain analysis for reusability. In *Proceedings of COMPSAC'87*, pages 23–29, 1987.

[96] R. Prieto-Diaz. Domain analysis: An introduction. *ACM SIGSOFT Software Engineering Notes*, 15(2):47–54, 1990.

[97] W.W. Royce. Managing the development of large software systems: Concepts and techniques. In *Proceedings of WESCON*, August 1970.

[98] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.

[99] Software Productivity Consortium. *Consortium Requirements Engineering Handbook*, 1993. SPC-92060-CMC.

[100] *Software Productivity Consortium Reuse Adoption Guidebook*, version 01.00.03 edition, November 1992. SPC-92051-CMC.

[101] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 1992.

[102] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 1992.

[103] A. Terry, R. London, G. Papanagopoulos, and M. Devito. The ARDEC/Teknowledge architecture description language (ArTek). Technical report, Teknowledge Federal Syst. and U.S. Army Armament Research, Development, and Eng. Center, July 1995. Version 4.0.

[104] Jeffrey M. Thompson. NIMBUS: A framework for static analysis and simulation of system-level inter-component communication. Master's thesis, University of Minnesota, December 1999.

[105] Jeffrey M. Thompson and Mats P.E. Heimdahl. An integrated development environment prototyping safety critical systems. In *Tenth IEEE International Workshop on Rapid System Prototyping (RSP) 99*, pages 172–177, June 1999.

[106] Jeffrey M. Thompson and Mats P.E. Heimdahl. Extending the product family approach to support n-dimensional and hierarchical product lines. In *The Fifth IEEE International Symposium on Requirements Engineering*, August 2001.

[107] Jeffrey M. Thompson and Mats P.E. Heimdahl. Structuring product family requirements for n-dimensional and hierarchical product lines. *Requirements Engineering Journal*, 2002. (Submitted).

[108] Jeffrey M. Thompson, Mats P.E. Heimdahl, and Debra M. Erickson. Structuring formal control systems specifications for reuse: Surviving hardware changes. In *Proceedings of the Fifth NASA Langley Formal Methods Conference (Lfm2000)*, 2000.

[109] Jeffrey M. Thompson, Mats P.E. Heimdahl, and Steven P. Miller. Specification based prototyping for embedded systems. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, number 1687 in LNCS, pages 163–179, September 1999.

[110] Jeffrey M. Thompson, Michael W. Whalen, and Mats P.E. Heimdahl. Requirements capture and evaluation in NIMBUS: The light-control case study. *Journal of Universal Computer Science*, 6(7):731–757, July 2000.

[111] W. Tracz. LILEANNA: A parameterized programming language. In *Proceedings of the Second International Workshop on Software Reuse*, pages 66–78, Lucca, Italy, March 1993.

[112] W. Tracz. Dssa (domain specific software architecture) pedagogical example. *ACM SIGSOFT Software Engineering Notes*, 20(3):49–62, 1995.

[113] W. Tracz, L. Coglianese, and P. Young. A domain specific software architecture engineering process outline. *ACM SIGSOFT Software Engineering Notes*, 18(2):40–49, 1993.

[114] S. Vestal. MetaH programmer's manual. Technical report, Honeywell Technology Center, Minneapolis, MN, April 1996. Version 1.09.

[115] Steve Vestal. Metah programmer's manual. Technical Report 1.1.4, Honeywell Technology Center, 3660 Technology Drive, Mpls, MN 55418, 1993.

[116] David M. Weiss. Defining families: The commonality analysis. Technical report, Lucent Technologies Bell Laboratories, 1000 E. Warrenville Rd, Naperville, IL 60566, 1997.

[117] David M. Weiss and Chi Tau Robert Lai. *Software Product Line Engineering: A Family-Based Software Development Process.* Addison-Wesley, 1999.

[118] Michael W. Whalen. A formal semantics for RSML^{-e}. Master's thesis, University of Minnesota, May 2000.

[119] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof.* Prentice-Hall, 1996.

[120] Pamela Zave. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–29, January 1997.

Appendix A

Standard Modules for RSML^{-e}

In the new tool, the user will be able to specify files to be included (much like the include directives in C++ or Java). This will allow a large and complex specification to be divided into modules and allow these modules to be stored in separate files. This will give a much finer grained version control than what we currently have with a single monolithic file and it will also make it much easier to reuse and recombine the module definitions.

This section defines the standard module include file to be used with the new version of RSML^{-e}. This file will be included at the bottom of most RSML^{-e} specifications and the users will have the opportunity to use all of the predefined module definitions found here.

```
MODULE PREV :
  INTERFACE :
    GENERIC_TYPE G

    IMPORT val : G
    END IMPORT

    IMPORT CONSTANT InitialValue : G
    END IMPORT

    IMPORT cond : Boolean
    END IMPORT

    IMPORT CONSTANT size : integer
       UNITS : NA
       EXPECTED_MIN : 1
       EXPECTED_MAX : UNDEFINED
    END IMPORT
```

```
    EXPORT previousValue : G
    END EXPORT

  END INTERFACE

  DEFINITION :

      STATE_VARIABLE internal_array : [1 TO size] OF G
         PARENT : NONE

         ASSIGNMENT [1] :
            DEFAULT_VALUE : InitialValue
            EQUALS val IF cond
            EQUALS PRE(internal_array[1]) IF NOT (cond)
         END ASSIGNMENT

         ASSIGNMENT [2 TO size] :
            EQUALS internal_array[this-1] IF cond
            EQUALS pre(internal_array[this]) IF NOT (cond)
         END ASSIGNMENT

      END STATE_VARIABLE

      EXPORT previousValue :
         PARENT : NONE

         EQUALS internal_array[size] IF TRUE
      END EXPORT

   END DEFINITION
END MODULE

MODULE PREV_VALUE :

  INTERFACE :
    GENERIC_TYPE G

    IMPORT Variable : G
    END IMPORT

    IMPORT CONSTANT InitialValue : G
    END IMPORT

    EXPORT PreviousValue : G
    END EXPORT
```

```
  END INTERFACE

  DEFINITION :

    EXPORT PreviousValue :
      PARENT : NONE
      DEFAULT_VALUE : InitialValue
      EQUALS PRE(PreviousValue) IF PRE(PreviousValue) = Variable
      EQUALS PRE(Variable) IF PRE(PreviousValue) != Variable
    END EXPORT

  END DEFINITION

END MODULE


MODULE VALUE_AT_TIME :

  INTERFACE :
    GENERIC_TYPE G

    IMPORT SpecifiedTime : TIME
    END IMPORT

    IMPORT CurrentValue : G
    END IMPORT

    IMPORT CONSTANT InitialValue : G
    END IMPORT

    IMPORT Clock : TIME
    END IMPORT

    EXPORT Val : G
    END EXPORT

  END INTERFACE

  DEFINITION :

    EXPORT Val :
      PARENT : NONE
      DEFAULT_VALUE : InitialValue
      EQUALS Pre(Val) IF Clock != SpecifiedTime
      EQUALS CurrentValue IF Clock = SpecifiedTime
    END EXPORT
```

```
  END DEFINITION

END MODULE


INTERFACE BooleanMonitor:
  IMPORT Expr : BOOLEAN
  END IMPORT

  IMPORT CONSTANT InitialValue : BOOLEAN
  END IMPORT

  EXPORT Result : BOOLEAN
  END EXPORT
END INTERFACE


MODULE WHEN : BooleanMonitor
  DEFINITION :

    EXPORT Result :
      PARENT : NONE
      DEFAULT_VALUE : InitialValue
      EQUALS True IF
        TABLE
          PRE(Expr) : F ;
          Expr      : T ;
        END TABLE
      EQUALS False IF
        TABLE
          PRE(Expr) : T * ;
          Expr      : * F ;
        END TABLE
    END EXPORT

  END DEFINITION
END MODULE

MODULE WHEN_NOT : BooleanMonitor
  DEFINITION :

    EXPORT Result :
      PARENT : NONE
      DEFAULT_VALUE : InitialValue
      EQUALS True IF
        TABLE
```

```
          PRE(Expr) : T ;
          Expr      : F ;
        END TABLE
      EQUALS False IF
        TABLE
          PRE(Expr) : F * ;
          Expr      : * T ;
        END TABLE
    END EXPORT

  END DEFINITION
END MODULE

INTERFACE GenericMonitor :

  GENERIC_TYPE G

  IMPORT Expr : G
  END IMPORT

  EXPORT Result : BOOLEAN
  END EXPORT

END INTERFACE


MODULE CHANGED : GenericMonitor
  DEFINITION :

    EXPORT Result :
      PARENT : NONE
      DEFAULT_VALUE : False
      EQUALS Pre(Expr) != Expr
    END EXPORT

  END DEFINITION
END MODULE

MODULE UNCHANGED : GenericMonitor
  DEFINITION:

    EXPORT Result :
      PARENT : NONE
      DEFAULT_VALUE : False
      EQUALS PRE(Expr) = Expr
    END EXPORT
```

```
  END DEFINITION
END MODULE

INTERFACE TimeMonitor :
  IMPORT Expr : BOOLEAN
  END IMPORT

  IMPORT CONSTANT InitialValue : TIME
  END IMPORT

  IMPORT Clock : TIME
  END IMPORT

  EXPORT Result : TIME
  END EXPORT
END INTERFACE

MODULE TIME_CHANGED : TimeMonitor
  DEFINITION :

    EXPORT Result :
      PARENT : NONE
      DEFAULT_VALUE : InitialValue
      EQUALS Clock IF PRE(Expr) != Expr
      EQUALS PRE(Result) IF PRE(Expr) = Expr
    END EXPORT

  END DEFINITION
END MODULE

MODULE TIME_WHEN : TimeMonitor
  DEFINITION :

    EXPORT Result :
      PARENT : NONE
      DEFAULT_VALUE : InitialValue
      EQUALS Clock IF
        TABLE
          PRE(Expr) : F ;
          Expr      : T ;
        END TABLE
      EQUALS PRE(Result) IF
        TABLE
          PRE(Expr) : T * ;
          Expr      : * F ;
        END TABLE
    END EXPORT
```

```
  END DEFINITION
END MODULE

MODULE TIME_WHEN_NOT : TimeMonitor
  DEFINITION :

    EXPORT Result :
      PARENT : NONE
      DEFAULT_VALUE : InitialValue
      EQUALS Clock IF
        TABLE
          PRE(Expr) : T ;
          Expr      : F ;
        END TABLE
      EQUALS PRE(Result) IF
        TABLE
          PRE(Expr) : F * ;
          Expr      : * T ;
        END TABLE
    END EXPORT

  END DEFINITION
END MODULE

MODULE DURATION : TimeMonitor
  DEFINITION :

    STATE_VARIABLE InitialTime : TIME
      PARENT : NONE
      DEFAULT_VALUE : UNDEFINED
      EQUALS Clock IF
        TABLE
          DEFINED(PRE(InitialTime )) : F ;
          PRE(Expr)                  : F ;
          Expr                       : T ;
        END TABLE
      EQUALS PRE(InitialTime) IF
        TABLE
          DEFINED(PRE(InitialTime )) : T * ;
          PRE(Expr)                  : * T ;
        END TABLE
      EQUALS UNDEFINED IF Expr = False
    END STATE_VARIABLE

    EXPORT Result :
      PARENT : NONE
```

```
      DEFAULT_VALUE : InitialValue
      EQUALS Clock - InitialTime IF DEFINED(InitialTime)
      EQUALS InitialValue IF NOT (DEFINED(InitialTime))
    END EXPORT

  END DEFINITION
END MODULE


INTERFACE BooleanResultArrayAggregate :

  IMPORT conditions : [1 TO size] OF BOOLEAN
  END IMPORT

  IMPORT CONSTANT size : INTEGER
  END IMPORT

  EXPORT result : BOOLEAN
  END EXPORT

END INTERFACE

MODULE FORALL : BooleanResultArrayAggregate
  DEFINITION :

     STATE_VARIABLE internal_array : [1 TO size] OF BOOLEAN
        PARENT : NONE
        UNITS : NA
        EXPECTED_MIN : 1
        EXPECTED_MAX : size

        ASSIGNMENT [1] :
           EQUALS conditions[this]
        END ASSIGNMENT

        ASSIGNMENT [2 TO size] :
           EQUALS True IF
             TABLE
               internal_array[this-1]   : T ;
               conditions[this]           : T ;
             END TABLE
          EQUALS False IF
             TABLE
               internal_array[this-1]   : F * ;
               conditions[this]           : * F ;
             END TABLE
         END ASSIGNMENT
```

```
      END STATE_VARIABLE

      EXPORT result :
         PARENT : NONE
         EQUALS internal_array[size]
      END EXPORT

  END DEFINITION
END MODULE

MODULE EXISTS : BooleanResultArrayAggregate
  DEFINITION :

      STATE_VARIABLE internal_array : [1 TO size] OF BOOLEAN
         PARENT : NONE
         UNITS : NA
         EXPECTED_MIN : 1
         EXPECTED_MAX : size

         ASSIGNMENT [1] :
            EQUALS conditions[this]
         END ASSIGNMENT

         ASSIGNMENT [2 TO size] :
            EQUALS True IF
              TABLE
                internal_array[this-1]   : T * ;
                conditions[this]            : * T ;
              END TABLE
           EQUALS False IF
              TABLE
                internal_array[this-1]   : F ;
                conditions[this]            : F ;
              END TABLE
          END ASSIGNMENT

      END STATE_VARIABLE

      EXPORT result :
         PARENT : NONE
         EQUALS internal_array[size]
      END EXPORT

  END DEFINITION
END MODULE
```

```
INTERFACE IntegerResultArrayAggregate :

  IMPORT conditions : [1 TO size] OF BOOLEAN
  END IMPORT

  IMPORT CONSTANT size : INTEGER
  END IMPORT

  EXPORT result : BOOLEAN
  END EXPORT

END INTERFACE

MODULE FIRST_INDEX : IntegerResultArrayAggregate
  DEFINITION :

     STATE_VARIABLE internal_array : [1 TO size] OF INTEGER
        PARENT : NONE
        UNITS : NA
        EXPECTED_MIN : 1
        EXPECTED_MAX : size

        ASSIGNMENT [1] :
           EQUALS conditions[this] IF conditions[this]
           EQUALS UNDEFINED IF NOT (conditions[this])
        END ASSIGNMENT

        ASSIGNMENT [2 TO size] :
           EQUALS this IF
             TABLE
               DEFINED(internal_array[this-1])      : F ;
               conditions[this]                     : T ;
             END TABLE
           EQUALS internal_array[this-1] IF
             TABLE
               DEFINED(internal_array[this-1])      : T * ;
               conditions[this]                     : * T ;
             END TABLE
        END ASSIGNMENT

     END STATE_VARIABLE

     EXPORT result :
        PARENT : NONE
        EQUALS internal_array[size]
     END EXPORT
```

```
  END DEFINITION
END MODULE

MODULE LAST_INDEX : IntegerResultArrayAggregate
  DEFINITION :

      STATE_VARIABLE internal_array : [1 TO size] OF INTEGER
         PARENT : NONE
         UNITS : NA
         EXPECTED_MIN : 1
         EXPECTED_MAX : size

         ASSIGNMENT [size] :
            EQUALS conditions[this] IF conditions[this]
            EQUALS UNDEFINED IF NOT (conditions[this])
         END ASSIGNMENT

         ASSIGNMENT [1 TO size-1] :
            EQUALS this IF
              TABLE
                DEFINED(internal_array[ this+1])    : F ;
                conditions[this]                    : T ;
              END TABLE
            EQUALS internal_array[t his+1] IF
              TABLE
                DEFINED(internal_array[ this+1])    : T * ;
                conditions[this]                    : * T ;
              END TABLE
         END ASSIGNMENT

      END STATE_VARIABLE

      EXPORT result :
         PARENT : NONE
         EQUALS internal_array[1]
      END EXPORT

  END DEFINITION
END MODULE

MODULE COUNT : IntegerResultArrayAggregate
  DEFINITION :

      STATE_VARIABLE internal_array : [1 TO size] OF INTEGER
         PARENT : NONE
         UNITS : NA
```

```
        EXPECTED_MIN : 1
        EXPECTED_MAX : size

        ASSIGNMENT [1] :
           EQUALS 1 IF conditions[this]
           EQUALS 0 IF NOT (conditions[this])
        END ASSIGNMENT

        ASSIGNMENT [2 TO size] :
           EQUALS internal_array[this-1] + 1 IF conditions[this]
           EQUALS internal_array[this-1]     IF NOT (conditions[this])
        END ASSIGNMENT

      END STATE_VARIABLE

      EXPORT result :
         PARENT : NONE
         EQUALS internal_array[size]
      END EXPORT

  END DEFINITION
END MODULE



INTERFACE IntegerMathArrayAggregate :

    IMPORT vals : [1 TO size] of INTEGER
      UNITS : NA
      EXPECTED_MIN : UNDEFINED
      EXPECTED_MAX : UNDEFINED
    END IMPORT

    IMPORT CONSTANT size : INTEGER
      UNITS : NA
      EXPECTED_MIN : 1
      EXPECTED_MAX : UNDEFINED
    END IMPORT

    EXPORT result : INTEGER
      UNITS : NA
      EXPECTED_MIN : UNDEFINED
      EXPECTED_MAX : UNDEFINED
    END EXPORT

END INTERFACE
```

```
MODULE SUM : IntegerMathArrayAggregate
  DEFINITION :

      STATE_VARIABLE internal_array : [1 TO size] OF INTEGER
         PARENT : NONE
         UNITS : NA
         EXPECTED_MIN : 1
         EXPECTED_MAX : UNDEFINED

         ASSIGNMENT [1] :
            EQUALS vals[this]
         END ASSIGNMENT

         ASSIGNMENT [2 TO size] :
            EQUALS vals[this] + internal_array[this-1]
         END ASSIGNMENT

      END STATE_VARIABLE

      EXPORT result :
         PARENT : NONE
         EQUALS internal_array[size]
      END EXPORT

  END DEFINITION
END MODULE

MODULE AVERAGE : IntegerMathArrayAggregate
  DEFINITION :

      EXPORT sumValue :
         PARENT : NONE
         EQUALS SUM(vals, size)/size
      END EXPORT

  END DEFINITION
END MODULE

MODULE MAXIMUM : IntegerMathArrayAggregate
  DEFINITION :

      STATE_VARIABLE internal_array : [1 TO size] OF INTEGER
         PARENT : NONE
         UNITS : NA
         EXPECTED_MIN : 1
         EXPECTED_MAX : UNDEFINED
```

```
      ASSIGNMENT [1] :
         EQUALS vals[this]
      END ASSIGNMENT

      ASSIGNMENT [2 TO size] :
         EQUALS vals[this] IF vals[this] > internal_array[this-1]
         EQUALS internal_array[this-1] IF vals[this] <= internal_array[this-1]
      END ASSIGNMENT

   END STATE_VARIABLE

   EXPORT result :
      PARENT : NONE
      EQUALS internal_array[size]
   END EXPORT

  END DEFINITION
END MODULE

MODULE MINIMUM : IntegerMathArrayAggregate
  DEFINITION :

   STATE_VARIABLE internal_array : [1 TO size] OF INTEGER
      PARENT : NONE
      UNITS : NA
      EXPECTED_MIN : 1
      EXPECTED_MAX : UNDEFINED

      ASSIGNMENT [1] :
         EQUALS vals[this]
      END ASSIGNMENT

      ASSIGNMENT [2 TO size] :
         EQUALS vals[this] IF vals[this] < internal_array[this-1]
         EQUALS internal_array[this-1] IF vals[this] >= internal_array[this-1]
      END ASSIGNMENT

   END STATE_VARIABLE

   EXPORT result :
      PARENT : NONE
      EQUALS internal_array[size]
   END EXPORT

  END DEFINITION
END MODULE
```

```
INTERFACE RealMathArrayAggregate :

    IMPORT vals : [1 TO size] of REAL
      UNITS : NA
      EXPECTED_MIN : UNDEFINED
      EXPECTED_MAX : UNDEFINED
    END IMPORT

    IMPORT CONSTANT size : REAL
      UNITS : NA
      EXPECTED_MIN : 1
      EXPECTED_MAX : UNDEFINED
    END IMPORT

    EXPORT result : REAL
      UNITS : NA
      EXPECTED_MIN : UNDEFINED
      EXPECTED_MAX : UNDEFINED
    END EXPORT

END INTERFACE

MODULE SUM_REAL : RealMathArrayAggregate
  DEFINITION :

      STATE_VARIABLE internal_array : [1 TO size] OF REAL
         PARENT : NONE
         UNITS : NA
         EXPECTED_MIN : 1
         EXPECTED_MAX : UNDEFINED

         ASSIGNMENT [1] :
            EQUALS vals[this]
         END ASSIGNMENT

         ASSIGNMENT [2 TO size] :
            EQUALS vals[this] + internal_array[this-1]
         END ASSIGNMENT

      END STATE_VARIABLE

      EXPORT result :
         PARENT : NONE
         EQUALS internal_array[size]
      END EXPORT
```

```
  END DEFINITION
END MODULE

MODULE AVERAGE_REAL : RealMathArrayAggregate
  DEFINITION :

      EXPORT sumValue :
         PARENT : NONE
         EQUALS SUM(vals, size)/size
      END EXPORT

  END DEFINITION
END MODULE

MODULE MAXIMUM_REAL : RealMathArrayAggregate
  DEFINITION :

      STATE_VARIABLE internal_array : [1 TO size] OF REAL
         PARENT : NONE
         UNITS : NA
         EXPECTED_MIN : 1
         EXPECTED_MAX : UNDEFINED

         ASSIGNMENT [1] :
            EQUALS vals[this]
         END ASSIGNMENT

         ASSIGNMENT [2 TO size] :
            EQUALS vals[this] IF vals[this] > internal_array[this-1]
            EQUALS internal_array[this-1] IF vals[this] <= internal_array[this-1]
         END ASSIGNMENT

      END STATE_VARIABLE

      EXPORT result :
         PARENT : NONE
         EQUALS internal_array[size]
      END EXPORT

  END DEFINITION
END MODULE

MODULE MINIMUM_REAL : RealMathArrayAggregate
  DEFINITION :

      STATE_VARIABLE internal_array : [1 TO size] OF REAL
```

```
      PARENT : NONE
      UNITS : NA
      EXPECTED_MIN : 1
      EXPECTED_MAX : UNDEFINED

      ASSIGNMENT [1] :
         EQUALS vals[this]
      END ASSIGNMENT

      ASSIGNMENT [2 TO size] :
         EQUALS vals[this] IF vals[this] < internal_array[this-1]
         EQUALS internal_array[this-1] IF vals[this] >= internal_array[this-1]
      END ASSIGNMENT

   END STATE_VARIABLE

   EXPORT result :
      PARENT : NONE
      EQUALS internal_array[size]
   END EXPORT

  END DEFINITION
END MODULE
```

Appendix B

The ASW REQ Model (Phase 5)

```
INCLUDE "asw-alltypes.nimbus"

MODULE ASW_REQ_P5 :

  INTERFACE :

    EXPORT CON_DOI_P5 : DOIControlledType
    Purpose : &*L  This variable represents the ASW's
    commanded status of the Device of Interest (DOI). L*&

    Interpretation : &*L
      \begin{quote}
      \begin{mydescription}
      \item[On:] Indicates that the DOI is commanded to be On.  The DOI
      is commanded to be on when the aircraft enters the target region
      for turning the DOI on, the DOI is not already on,
      and the ASW is not inhibitied.
      \item[Off:] Indicates that the DOI is commanded to be Off.  The
      DOI is commanded to be off when the aircraft leaves the target
      region and after a certain period of time has passed.  If this
      time is \RUndefined, then the ASW will never turn the DOI Off.
      \item[Uncommanded:] Indicates that the DOI is not commanded by the
      ASW.  This CON\_DOI variable will be equal to Uncommanded in any
      step were the ASW does not issue a command to the device of interest.
      \end{mydescription}
      \end{quote}
      L*&

    Issues : &*L
      \begin{myitemize}
      \item If the aircraft leaves the target area and the DOI is on,
      but was {\em not} commanded to be on by the ASW, should the ASW
      turn it off?
      \end{myitemize}
      L*&

    END EXPORT
```

```
EXPORT CON_Failure_P5 : Boolean
 Purpose : &*L This variable represents the ASW's indication of
 whether or not it has failed to the external world.  It is
 potentially displayed to the pilot and/or used by other subsystems
 on board the aircraft. L*&

 Interpretation : &*L
  \begin{quote}
  \begin{mydescription}
  \item[True:] Indicates that the ASW has failed.  The ASW is
  considered to be failed if it attempts to turn on the DOI, but the
  DOI does not turn on after a certain timeout period.
  \item[False:] Indicates that the ASW has not failed.  The ASW is
  considered to be operating normally if none of the failure
  conditions are true.
  \end{mydescription}
  \end{quote}
  L*&
END EXPORT

IMPORT MON_Altitude_P5 : INTEGER
  UNITS : ft
  EXPECTED_MIN : 0
  EXPECTED_MAX : 50000
  CLASSIFICATION : Monitored

  Purpose : &*L This variable represents the ASW's idea of what the
  altitude of the aircraft is.  It is related to the Altitude\_Quality
  variable. L*&

  Interpretation : &*L
    \begin{quote}
    \begin{mydescription}
    \item[Precision:] We will know the altitude to within $\pm 10$ ft.
    \end{mydescription}
    \end{quote}
    L*&

END IMPORT

IMPORT MON_Altitude_Qua lity_P5 : AltitudeQualityType
  CLASSIFICATION : Monitored

  Purpose : &*L This variable represents the quality of the
  Altitude of the aircraft is. L*&
END IMPORT
```

```
IMPORT MON_DOI_P5 : OnOffType_P5
  Purpose : &*L This variable indicates the monitored status of the
  DOI.  The DOI can be turned on or off by other devices/systems on
  board the aircraft, so the ASW needs an accurate accounting of the
  status of the DOI L*&

  Interpretation : &*L
    \begin{quote}
    \begin{mydescription}
    \item[On:] Indicates that the DOI is currently on.
    \item[Off:] Indicates that the DOI is currently off.
    \end{mydescription}
    \end{quote}
    L*&

 END IMPORT

IMPORT MON_Reset_P5 : Boolean

  Purpose : &*L This variable indicates the whether the ASW should be
  reset or not.  In a step where the ASW is reset, this variable will
  have the value true.  In all others, this variable will have the
  value false. L*&

  Interpretation : &*L
    \begin{quote}
    \begin{mydescription}
    \item[True:] Indicates that the ASW as been reset.
    \item[False:] Indicates that the ASW has not been reset.
    \end{mydescription}
    \end{quote}
    L*&

END IMPORT

IMPORT MON_Inhibit_P5 : Boolean

   Purpose : &*L This variable is true when the ASW is inhibited and
   false otherwise.  The value is determined by the user and/or other
   systems on board the aircraft. L*&

  Interpretation : &*L
    \begin{quote}
    \begin{mydescription}
    \item[True:] Indicates that the operation of the ASW has been
    inhibited; the ASW shall not attempt to change the status of the
```

```
    DOI.
    \item[False:] Indicates that the ASW has not been inhibited; the
    ASW will behave as specified by other requirements.
    \end{mydescription}
    \end{quote}
    L*&

END IMPORT

IMPORT CONSTANT Threshold_P5 : INTEGER
  UNITS : ft
  EXPECTED_MIN : 0
  EXPECTED_MAX : 8024

  Purpose : &*L This constant will be defined by each family
  member when the REQ module is instantiated.  It is the altitude
  at which the ASW is required to turn on or off the ASW. L*&

END IMPORT

IMPORT CONSTANT Invalid_Alt_Failure_P5 : Time
  UNITS : NA
  EXPECTED_MIN : 2 s
  EXPECTED_MAX : 10 s

  Purpose : &*L This constant will be defined by each family
  member.  It is the length of time after which the ASW will
  declare a failure if there is not valid altitude. L*&

END IMPORT

IMPORT CONSTANT DOI_Timeout_P5 : Time
  UNITS : NA
  EXPECTED_MIN : 1 s
  EXPECTED_MAX : 5 s

  Purpose : &*L This constant will be defined by each member of
  the ASW family to represent the amount of time before the ASW
  declares a failure if the DOI does not respond to a command. L*&

END IMPORT

IMPORT CONSTANT GoAboveAction_P5 : ActionType

  Purpose : &*L This constant specifies the action that the ASW
  will perform when it crosses the Threshold going up. It is
  specified by the decision model for each family member. L*&
```

```
  END IMPORT

  IMPORT CONSTANT GoBelowAction_P5 : ActionType

    Purpose : &*L This constant specifies the action that the ASW
    will perform when it crosses the Threshold going down. It is
    specified by the decision model for each family member. L*&

  END IMPORT

  IMPORT CONSTANT GoAboveHyst_P5 : INTEGER
    UNITS : ft
    EXPECTED_MIN : 50
    EXPECTED_MAX : 500

    Purpose : &*L This defines the hysteresis factor for going above
    the threshold altitude. L*&

  END IMPORT

  IMPORT CONSTANT GoBelowHyst_P5 : INTEGER
    UNITS : ft
    EXPECTED_MIN : 50
    EXPECTED_MAX : 500

    Purpose : &*L This defines the hysteresis factor for going above
    the threshold altitude. L*&

  END IMPORT

END INTERFACE

DEFINITION :

  STATE_VARIABLE ASW_System_Mode_P5 :
    VALUES : {Startup, NormalOperating, Degraded, Failed, Reset}
    PARENT : NONE

    Purpose : &*L This is the top-level mode of the ASW.  If the ASW
    were to have a startup mode, etc., we could put those modes as
    children of this controlling mode.  Currently, we have only two
    states, the reset mode which is used for when the reset signal
    is received and the operating mode that handles the main
    behavior. L*&

    DEFAULT_VALUE : Startup
```

```
  TRANSITION NormalOperating TO Reset IF MON_Reset_P5

  TRANSITION Degraded TO Reset IF MON_Reset_P5

  TRANSITION NormalOperating TO Degraded IF
    EpisodeMonitor_P5 = QualifyingEpisode

  TRANSITION Degraded TO NormalOperating IF
    DURATION (MON_Altitude_Qua lity_P5 = Valid, 0 S, Clock) > 1 MIN

  TRANSITION Reset TO NormalOperating IF
    DURATION(PRE(ASW_System _Mode_P5), 0 s, Clock) >= 0 S

END STATE_VARIABLE

STATE_VARIABLE EpisodeMonitor_P5 :
  VALUES : {NoEpisode, FirstEpisode, QualifyingEpisode}
  PARENT : NONE

  Purpose : &*L This simple state variable tracks whether or not
  we have met the conditions for being in degraded functionality
  mode.  Namely, whether or not we have seen two periods of
  invalid altitude lasting 1 second or more within 1 minute. L*&

   DEFAULT_VALUE : NoEpisode

   TRANSITION NoEpisode TO FirstEpisode IF
     DURATION(MON_Altitude_Q uality_P5 = Invalid, 0 S, Clock) > 1 S

   TRANSITION FirstEpisode TO QualifyingEpisode IF
     TABLE
       DURATION(MON_Altitude_Q uality_P5 = Invalid, 0 S, Clock) > 1 S : T ;
       DURATION(PRE(EpisodeMon itor_P5) = FirstEpisode) > 1 S         : T ;
     END TABLE

   TRANSITION FirstEpisode TO NoEpisode IF
     DURATION(PRE(EpisodeMon itor_P5) = FirstEpisode) >= 1 MIN

   TRANSITION QualifyingEpiso de TO NoEpisode IF
     DURATION(MON_Altitude_Q uality_P5 = Valid, 0 S, Clock) >= 2 MIN

END STATE_VARIABLE

MODULE_INSTANCE ASW_Operating_Mode_P5 : ASW_Operating_Mode_Def_P5
  PARENT :  ASW_System_Mode_ P5.NormalOperating
  ASSIGNMENT
```

```
      MON_Altitude_P5          := MON_Altitude_P5,
      MON_Altitude_Quality_P5  := MON_Altidue_Quality_P5,
      MON_DOI_P5               := MON_DOI_P5,
      MON_Inhibit_P5           := MON_Inhibit_P5,
      Threshold_P5             := Threshold_P5,
      Invalid_Alt_Failure_P5   := Invalid_Alt_Failure_P5,
      DOI_Timeout_P5           := DOI_Timeout_P5,
      GoAboveAction_P5         := GoAboveAction_P5,
      GoBelowAction_P5         := GoBelowAction_P5,
      GoAboveHyst_P5           := GoAboveHyst_P5,
      GoBelowHyst_P5           := GoBelowHyst_P5,
      DOI_Delay_P5             := 0 S
  END ASSIGNMENT
END MODULE_INSTANCE

MODULE_INSTANCE ASW_Degraded_Mode_P5 : ASW_Operating_Mode_Def_P5
  PARENT :  ASW_System_Mode_P5.Degraded
  ASSIGNMENT
      MON_Altitude_P5          := MON_Altitude_P5,
      MON_Altitude_Quality_P5  := MON_Altidue_Quality_P5,
      MON_DOI_P5               := MON_DOI_P5,
      MON_Inhibit_P5           := MON_Inhibit_P5,
      Threshold_P5             := Threshold_P5,
      Invalid_Alt_Failure_P5   := Invalid_Alt_Failure_P5,
      DOI_Timeout_P5           := DOI_Timeout_P5,
      GoAboveAction_P5         := GoAboveAction_P5,
      GoBelowAction_P5         := GoBelowAction_P5,
      GoAboveHyst_P5           := GoAboveHyst_P5,
      GoBelowHyst_P5           := GoBelowHyst_P5,
      DOI_MinDelay_P5          := 2 S,
      DOI_MaxDelay_P5          := 6 S
  END ASSIGNMENT
END MODULE_INSTANCE

EXPORT CON_DOI_P5 :
  PARENT :  NONE
  DEFAULT_VALUE : Uncontrolled

  EQUALS ASW_Operating_Mode_P5.CON_DOI_P5
    IF ASW_System_Mode_P5 = NormalOperating

  EQUALS ASW_Degraded_Mode_P5.CON_DOI_P5
    IF ASW_System_Mode_P5 = Degraded

  EQUALS Uncontrolled IF
    TABLE
      ASW_System_Mode_P5 = Failed : T * ;
```

```
        ASW_System_Mode_P5 = Reset  : * T ;
      END TABLE

  END EXPORT

  EXPORT CON_Failure_P5 :
    PARENT :  NONE
    DEFAULT_VALUE : False

    TRANSITION False TO True IF
      TABLE
        ASW_System_Mode_P5 = NormalOperating    : T * ;
        ASW_Operating_Mode_P5.C ON_Failure_P5   : T * ;
        ASW_System_Mode_P5 = Degraded           : * T ;
        ASW_Operating_Mode_P5.C ON_Failure_P5   : * T ;
      END TABLE

    TRANSITION True TO False IF ASW_System_Mode_P5 = Reset

  END EXPORT

 END DEFINITION

END MODULE


MODULE ASW_OperatingMode_D ef_P5 :

 INTERFACE :

  EXPORT CON_DOI_P5 : DOIControlledType
  END EXPORT

  EXPORT CON_Failure_P5 : Boolean
  END EXPORT

  IMPORT MON_Altitude_P5 : INTEGER
  END IMPORT

  IMPORT MON_Altitude_Qua lity_P5 : AltitudeQualityType
  END IMPORT

  IMPORT MON_DOI_P5 : OnOffType_P5
  END IMPORT

  IMPORT MON_Inhibit_P5 : Boolean
  END IMPORT
```

```
IMPORT CONSTANT Threshold_P5 : INTEGER
  UNITS : ft
  EXPECTED_MIN : 0
  EXPECTED_MAX : 8024
END IMPORT

IMPORT CONSTANT Invalid_Alt_Failure_P5 : Time
  UNITS : NA
  EXPECTED_MIN : 2 s
  EXPECTED_MAX : 10 s
END IMPORT

IMPORT CONSTANT DOI_Timeout_P5 : Time
  UNITS : NA
  EXPECTED_MIN : 1 s
  EXPECTED_MAX : 5 s
END IMPORT

IMPORT CONSTANT GoAboveAction_P5 : ActionType
END IMPORT

IMPORT CONSTANT GoBelowAction_P5 : ActionType
END IMPORT

IMPORT CONSTANT GoAboveHyst_P5 : INTEGER
  UNITS : ft
  EXPECTED_MIN : 50
  EXPECTED_MAX : 500
END IMPORT

IMPORT CONSTANT GoBelowHyst_P5 : INTEGER
  UNITS : ft
  EXPECTED_MIN : 50
  EXPECTED_MAX : 500
END IMPORT

IMPORT DOI_MinDelay_P5 : TIME

  Purpose : &*L This parameter to the ASW operating module
  determines whether or not we will wait to turn the DOI on. If it
  is greater than zero, then we will wait.  It represents the
  minium waiting time L*&

END IMPORT

IMPORT DOI_MaxDelay_P5 : TIME
```

```
    Purpose : &*L This parameter to the ASW operating module
    determines the maximum waiting time that we will stay in a
    Delayed action state before giving up and returning to NoAction
    L*&

  END IMPORT

END INTERFACE

DEFINITION :

  EXPORT CON_DOI_P5 :
    PARENT : NONE
    DEFAULT_VALUE : Uncommanded

    TRANSITION Uncommanded TO On IF
      TABLE
         GoBelowAction = TurnOn              : T * ;
         ActionBelow_P5.PerformA ction_P5    : T * ;
         GoAboveAction = TurnOn              : * T ;
         ActionAbove_P5.PerformA ction_P5    : * T ;
      END TABLE

    TRANSITION Uncommanded TO Off IF
      TABLE
         GoBelowAction = TurnOff             : T * ;
         ActionBelow_P5.PerformA ction_P5    : T * ;
         GoAboveAction = TurnOff             : * T ;
         ActionAbove_P5.PerformA ction_P5    : * T ;
      END TABLE

    TRANSITION On TO Uncommanded IF WHEN(MON_DOI_P5 = On, False)

    TRANSITION Off TO Uncommanded IF WHEN(MON_DOI_P5 = Off, False)

  END EXPORT

  MODULE_INSTANCE ActionBelow_P5 : DOI_Action_P5
    PARENT : NONE
    ASSIGNMENT
      Direction_P5           := Down,
      ThresholdedAltitude_P5 := ThresholdedAlt_P5.Result_P5,
      MinDelay_P5            := DOI_MinDelay_P5,
      MaxDelay_P5            := DOI_MaxDelay_P5,
      AltitudeQuality_P5     := MON_AlitudeQuality_P5,
      ActionOK_P5            := DOI_Action_Ok_P5(),
```

```
    Clock                  := Clock
  END ASSIGNMENT
END MODULE_INSTANCE

MODULE_INSTANCE ActionAbove_P5 : DOI_Action_P5
  PARENT : NONE
  ASSIGNMENT
    Direction_P5           := Up,
    ThresholdedAltitude_P5 := ThresholdedAlt_P5.Result_P5,
    MinDelay_P5            := DOI_MinDelay_P5,
    MaxDelay_P5            := DOI_MaxDelay_P5,
    AltitudeQuality_P5     := MON_AlitudeQuality_P5,
    ActionOK_P5            := DOI_Action_Ok_P5(),
    Clock                  := Clock
  END ASSIGNMENT
END MODULE_INSTANCE

MACRO DOI_Action_Ok_P5( act IS ActionType) :
  TABLE
    MON_Inhibit_P5      : F F ;
    CON_Failure_P5      : F F ;
    MON_DOI_P5 = On     : T * ;
    act = On            : F * ;
    MON_DOI_P5 = Off    : * T ;
    act = Off           : * F ;
  END TABLE
END MACRO

EXPORT CON_Failure_P5 :
  PARENT : NONE
  DEFAULT_VALUE : False

  EQUALS TRUE IF
    TABLE
      DURATION(AttemptingOn() , 0 S, Clock) > DOI_Timeout_P5     : T * * * ;
      DURATION(AttemptingOff( ), 0 S, Clock) > DOI_Timeout_P5    : * T * * ;
      DURATION(MON_Altitude_Q uality_P5 = Invalid, 0 S, Clock)   : * * T * ;
      PRE(CON_Failure_P5) = False                                : * * * T ;
    END TABLE

  EQUALS FALSE IF
    TABLE
      DURATION(AttemptingOn() , 0 S, Clock) > DOI_Timeout_P5     : F ;
      DURATION(AttemptingOff( ), 0 S, Clock) > DOI_Timeout_P5    : F ;
      DURATION(MON_Altitude_Q uality_P5 = Invalid, 0 S, Clock)   : F ;
      PRE(CON_Failure_P5) = False                                : F ;
    END TABLE
```

```
    END EXPORT

    MACRO AttemptingOn() :
      TABLE
        MON_DOI_P5 = Off    : T ;
        CON_DOI_P5 = On     : T ;
      END TABLE
    END MACRO

    MACRO AttemptingOff() :
      TABLE
        MON_DOI_P5 = On     : T ;
        CON_DOI_P5 = Off    : T ;
      END TABLE
    END MACRO

    MODULE_INSTANCE ThresholdedAlt_P5 : ThresholdedAltitude_P5
      PARENT : NONE
      ASSIGNMENT
        Altitude_P5 := MON_Altitude_P5,
        Threshold_P5 := Threshold_P5,
        BelowHysteresis_P5 := GoBelowHyst_P5,
        AboveHysteresis_P5 := GoBelowHyst_P5
      END ASSIGNMENT
    END MODULE_INSTANCE

  END DEFINITION

END MODULE


MODULE ThresholdedAltitude_P5 :

  INTERFACE :

    IMPORT Altitude_P5 : Integer
      UNITS : ft
      EXPECTED_MIN : 0
      EXPECTED_MAX : 50000
    END IMPORT

    IMPORT CONSTANT Threshold_P5 : Integer
      UNITS : ft
      EXPECTED_MIN : 0
      EXPECTED_MAX : 8024
    END IMPORT
```

```
  IMPORT CONSTANT AboveHysteresis_P5 : Integer
    UNITS : ft
    EXPECTED_MIN : 50
    EXPECTED_MAX : 500
  END IMPORT

  IMPORT CONSTANT BelowHysteresis_P5 : Integer
    UNITS : ft
    EXPECTED_MIN : 50
    EXPECTED_MAX : 500
  END IMPORT

  EXPORT Result_P5 : AboveBelowType

    Purpose : &*L this export reports whether or not the altitude is
    above or below the threshold given the hysteresis factor L*&

  END EXPORT

END INTERFACE

DEFINITION :

  EXPORT Result_P5 :
    PARENT : NONE

    DEFAULT_VALUE : Above IF
      TABLE
        DEFINED(Altitude_P5)         : T ;
        Altitude_P5 > Threshold_P5   : T ;
      END TABLE

    DEFAULT_VALUE : Below IF
      TABLE
        DEFINED(Altitude_P5)         : T ;
        Altitude_P5 <= Threshold_P5  : T ;
      END TABLE

    DEFAULT_VALUE : UNDEFINED IF NOT (DEFINED(Altitude_P5))

    EQUALS Above IF
      TABLE
        DEFINED(Altitude_P5)                 : T ;
        Altitude_P5 > EffectiveThreshold_P5  : T ;
      END TABLE
```

```
  EQUALS Below IF
    TABLE
      DEFINED(Altitude_P5)                  : T ;
      Altitude_P5 <= EffectiveThreshold_P5  : T ;
    END TABLE

  EQUALS UNDEFINED IF NOT (DEFINED(Altitude_P5))

END EXPORT

STATE_VARIABLE ApplyHisteresis_P5 :
  VALUES : {NoHyst, Above, Below}
  PARENT : NONE

  DEFAULT_VALUE : NoHyst

  TRANSITION NoHyst TO Above IF
    TABLE
      DEFINED(Altitude_P5)                      : T ;
      WHEN(Altitude_P5 < Threshold_P5, False)   : T ;
    END TABLE

  TRANSITION NoHyst TO Below IF
    TABLE
      DEFINED(Altitude_P5)                      : T ;
      WHEN(Altitude_P5 > Threshold_P5, False)   : T ;
    END TABLE

  TRANSITION Above TO NoHyst IF
    TABLE
      DEFINED(Altitude_P5)                                          : T T ;
      WHEN(Altitude_P5 < Threshold_P5 + AboveHysteresis_P5, False)  : T * ;
      WHEN(Altitude_P5 > Threshold_P5 - BelowHysteresis_P5, False)  : * T ;

    END TABLE

  TRANSITION Below TO NoHyst IF
    TABLE
      DEFINED(Altitude_P5)                                          : T T ;
      WHEN(Altitude_P5 > Threshold_P5 + AboveHysteresis_P5, False)  : T * ;
      WHEN(Altitude_P5 < Threshold_P5 - BelowHysteresis_P5, False)  : * T ;

    END TABLE
END STATE_VARIABLE

STATE_VARIABLE EffectiveThreshold_P5 : INTEGER
  PARENT : NONE
```

```
      UNITS : ft
      EXPECTED_MIN : Threshold_P5 - BelowHysteresis_P5
      EXPECTED_MAX : Threshold_P5 + AboveHysteresis_P5

      DEFAULT_VALUE : Threshold_P5

      EQUALS Threshold_P5 + AboveHysteresis_P5
        IF ApplyHysteresis_P5 = Above

      EQUALS Threshold_P5 - BelowHysteresis_P5
        IF ApplyHysteresis_P5 = Below

      EQUALS Threshold_P5
        IF ApplyHysteresis_P5 = NoHyst

    END STATE_VARIABLE

  END DEFINITION

END MODULE

MODULE DOI_Action_P5 :

  INTERFACE :

    IMPORT MinDelay_P5 : TIME
    END IMPORT

    IMPORT MaxDelay_P5 : TIME
    END IMPORT

    IMPORT CONSTANT Direction_P5 : UpDownType
    END IMPORT

    IMPORT ThresholdedAltitude_P5 : AboveBelowType
    END IMPORT

    IMPORT AltitudeQuality_P5 : AltitudeQualityType
    END IMPORT

    IMPORT ActionOK_P5 : Boolean
    END IMPORT

    IMPORT Clock : TIME
    END IMPORT

    EXPORT PerformAction_P5 : Boolean
```

```
  END EXPORT

END INTERFACE

DEFINITION :

  EXPORT PerformAction_P5 :
    PARENT : NONE
    DEFAULT_VALUE : False
    EQUALS WHEN(_internal = Perform)
  END EXPORT

  STATE_VARIABLE internal_P5 :
    VALUES : {NoAction, Delayed, Perform}
    PARENT : NONE

    DEFAULT_VALUE : NoAction

    TRANSITION NoAction TO Delayed IF
      TABLE
        MinDelay_P5 > 0 S                        : T T ;
        ActionOK_P5                              : T T ;
        WHEN(ThresholdedAltitud e_P5 = Below)    : T * ;
        Direction_P5 = Below                     : T * ;
        WHEN(ThresholdedAltitud e_P5 = Above)    : * T ;
        Direction_P5 = Above                     : * T ;
      END TABLE

    TRANSITION NoAction TO Peform IF
      TABLE
        MinDelay_P5 > 0 S                        : F F ;
        ActionOK_P5                              : T T ;
        WHEN(ThresholdedAltitud e_P5 = Below)    : T * ;
        Direction_P5 = Down                      : T * ;
        WHEN(ThresholdedAltitud e_P5 = Above)    : * T ;
        Direction_P5 = Up                        : * T ;
      END TABLE

    TRANSITION Delayed TO Perform IF
      TABLE
        DURATION(PRE(internal_P 5) IN_STATE Delayed,
                 0 S, Clock) >= MinDelay_P5      : T T ;
        ActionOK_P5                              : T T ;
        AltitudeQuality_P5 = Valid               : T T ;
        Direction_P5 = Down                      : T * ;
        ThresholdedAltitude_P5 = Below           : T * ;
        Direction_P5 = Up                        : * T ;
```

```
          ThresholdedAltitude_P5  = Above                    : * T ;
        END TABLE

      TRANSITION Delayed TO NoAction IF
        DURATION(PRE(internal_P 5) IN_STATE Delayed, 0 S, Clock) >= MaxDelay_P5

      TRANSITION Perform TO NoAction IF
        DURATION(PRE(internal_P 5) IN_STATE Perform, 0 S, Clock) >= 0 S

    END STATE_VARIABLE

  END DEFINITION

END MODULE

INCLUDE "standard-modules. nimbus"
```

Appendix C

The ASW SOFT Model (Phase 6)

```
/*L

In this chapter, we add to the REQ specification for the ASW a
specification of the ASW's IN' and OUT' relations.  These relations
are developed in a similar way to the REQ relation, but starting out
at a high level and then refining the structure and computation,
finally taking into consideration completeness and error handling
constraints.

For this Phase, we will be defining a number of new modules.  The
Altimeters\_IN\_P6 module will transform the inputs from the digital altimeters

L*/

INCLUDE "asw-alltypes.nimbus"

MODULE Altimeters_IN_P6 :

  INTERFACE :

    IMPORT CONSTANT NumDigitalAlt_P6 : INTEGER
      UNITS : NA
      EXPECTED_MIN : 0
      EXPECTED_MAX : 10
    END IMPORT

    IMPORT CONSTANT NumAnalogAlt_P6 : INTEGER
      UNITS : NA
      EXPECTED_MIN : 0
      EXPECTED_MAX : 10
    END IMPORT

    IMPORT DigialAlt_P6 : [1 TO NumDigitalAlt] OF INTEGER
      UNITS : ft
      EXPECTED_MIN : 0
      EXPECTED_MAX : 50000
    END IMPORT
```

```
  IMPORT CONSTANT Threshold_P6 : INTEGER
  END IMPORT

  IMPORT CONSTANT GoAboveHyst_P6 : INTEGER
    UNITS : ft
    EXPECTED_MIN : 50
    EXPECTED_MAX : 500

    Purpose : &*L This defines the hysteresis factor for going above
    the threshold altitude. L*&

  END IMPORT

  IMPORT CONSTANT GoBelowHyst_P6 : INTEGER
    UNITS : ft
    EXPECTED_MIN : 50
    EXPECTED_MAX : 500

    Purpose : &*L This defines the hysteresis factor for going above
    the threshold altitude. L*&

  END IMPORT

  IMPORT AnalogAlt_P6 : [1 TO NumAnalogAlt] OF AboveBelowType
  END IMPORT

  IMPORT DigitalQuality_P6 : [1 TO NumDigitalAlt] OF AltitudeQualityType
  END IMPORT

  IMPORT AnalogQuality_P6 : [1 TO NumAnalogAlt] OF AltitudeQualityType
  END IMPORT

  IMPORT INTERFACE AltitudeVoter_P6 :
  END IMPORT

  EXPORT Altitude_P6 : AboveBelowType
  END EXPORT

  EXPORT AltitudeQuality_P6 : AltitudeQualityType
  END EXPORT

END INTERFACE

DEFINITION :

  MODULE_INSTANCE ThresholdedDigital_P6 :
```

```
      [1 TO NumDigitalAlt] OF ThresholdedAltitude_P6
      PARENT : NONE
      ASSIGNMENT
        Altitude_P6         := DigitalAlt_P6,
        Threshold_P6        := EXTEND Threshold_P6 TO
                               [ 1 TO NumDigitalAlt] OF INTEGER,
        AboveHysteresis_P6  := EXTEND GoAboveHyst_P6 TO
                               [ 1 TO NumDigitalAlt] OF INTEGER,
        BelowHysteresis_P6  := EXTEND GoBelowHyst_P6 TO
                               [1 TO NumDigitalAlt] OF INTEGER
      END ASSIGNMENT
    END MODULE_INSTANCE

    SLOT_INSTANCE AltitudeVoter_P6 :
      ASSIGNMENT
        Num_of_Alt    := NumDigitalAlt_P6 + NumAnalogAlt_P6,
        Altitudes     := ThresholdedDigital_P6.Result_P6 | AnalogAlt_P6,
        Qualities     := DigitialQuality_P6 | AnalogQuality_P6
      END ASSIGNMENT
    END SLOT_INSTANCE

    EXPORT Altitude_P6 :
      PARENT : NONE
      DEFAULT_VALUE : AltitudeVoter_P6.Altitude_P6
      EQUALS AltitudeVoter_P6.Altitude_P6
    END EXPORT

    EXPORT AltitudeQuality_P6 :
      PARENT : NONE
      DEFAULT_VALUE : AltitudeVoter_P6.AltitudeQuality_P6
      EQUALS AltitudeVoter_P6.AltitudeQuality_P6
    END EXPORT

  END DEFINITION

END MODULE


INTERFACE AltitudeVoter_P6 :

  IMPORT CONSTANT Num_of_Alt_P6 : INTEGER
    UNITS : NA
    EXPECTED_MIN : 0
    EXPECTED_MAX : 50
  END IMPORT

  IMPORT Altitudes_P6 : [1 TO Num_of_Alt_P6] OF AboveBelowType
```

```
  END IMPORT

  IMPORT Qualities_P6 : [1 TO Num_of_Alt_P6] OF AltitudeQualityType
  END IMPORT

  EXPORT Altitude_P6 : AboveBelowType
  END EXPORT

  EXPORT Quality_P6 : AltitudeQualityType
  END EXPORT

END INTERFACE

MODULE Alt_and_Quality_P6 :

  INTERFACE :

    IMPORT Altitude_P6 : AboveBelowType
    END IMPORT

    IMPORT Quality_P6 : AltitudeQualityType
    END IMPORT

    EXPORT Result : Alt_and_QualityType
    END EXPORT

  END INTERFACE

  DEFINITION :

    EXPORT Alt_and_QualityType :
      PARENT : NONE

      EQUALS Above IF
        TABLE
          Altitude_P6 = Above : T ;
          Quality_P6 = Valid  : T ;
        END TABLE

      EQUALS Below IF
        TABLE
          Altitude_P6 = Below : T ;
          Quality_P6 = Valid  : T ;
        END TABLE

      EQUALS Invaid IF Quality_P6 = Invalid
```

```
    END EXPORT

  END DEFINITION

END MODULE

MODULE Most_P6 : AltitudeVoter_P6

  DEFINITION :

    EXPORT Altitude_P6 :
      PARENT : NONE

      DEFAULT_VALUE : Below

      EQUALS Below IF
        COUNT(EXTEND Alt_and_Quality_P6(Altitudes_P6, Qualities_P6) TO
                [1 TO Num_of_Alt_P6] OF AltitudeQualityType =
              EXTEND Below TO
                [1 TO Num_of_Alt_P6] OF AltitudeQualityType, Num_of_Alt_P6)
        >
        COUNT(EXTEND Alt_and_Quality_P6(Altitudes_P6, Qualities_P6) TO
                [1 TO Num_of_Alt_P6] OF AltitudeQualityType =
              EXTEND Above TO
                [1 TO Num_of_Alt_P6] OF AltitudeQualityType, Num_of_Alt_P6)

      EQUALS Above IF
        COUNT(EXTEND Alt_and_Quality_P6(Altitudes_P6, Qualities_P6) TO
                 [1 TO Num_of_Alt_P6] OF AltitudeQualityType =
              EXTEND Below TO
                 [1 TO Num_of_Alt_P6] OF AltitudeQualityType, Num_of_Alt_P6)
         <=
        COUNT(EXTEND Alt_and_Quality_P6(Altitudes_P6, Qualities_P6) TO
                 [1 TO Num_of_Alt_P6] OF AltitudeQualityType =
              EXTEND Above TO
                 [1 TO Num_of_Alt_P6] OF AltitudeQualityType, Num_of_Alt_P6)

    END EXPORT

    EXPORT Quality_P6 :
      PARENT : NONE
      DEFAULT_VALUE : Valid

      EQUALS Valid IF
       EXISTS(Qualities_P6 = EXTEND Valid TO
           [1 TO Num_of_Alt_P6] OF AltitudeQualityType, Num_of_Alt_P6)
```

```
      EQUALS Invalid IF
       FORALL(Qualities_P6 = EXTEND Invalid TO
           [1 TO Num_of_Alt_P6] OF AltitudeQualityType, Num_of_Alt_P6)


    END EXPORT

  END DEFINITION

END MODULE

MODULE AnyCrossed_P6 : AltitudeVoter_P6

  DEFINITION :

    EXPORT Altitude_P6 :
      PARENT : NONE

      DEFAULT_VALUE : Below

      TRANSITION Below TO Above IF
          EXISTS(EXTEND Alt_and_Quality_P6(Altitudes_P6, Qualities_P6) TO
                    [1 TO Num_of_Alt_P6] OF AltitudeQualityType =
                  EXTEND Above TO
                    [1 TO Num_of_Alt_P6] OF AltitudeQualityType, Num_of_Alt_P6)

      TRANSITION Above TO Below IF
          EXISTS(EXTEND Alt_and_Quality_P6(Altitudes_P6, Qualities_P6) TO
                    [1 TO Num_of_Alt_P6] OF AltitudeQualityType =
              EXTEND Below TO
                    [1 TO Num_of_Alt_P6] OF AltitudeQualityType, Num_of_Alt_P6)

    END EXPORT

    EXPORT Quality_P6 :
      PARENT : NONE

      EQUALS Valid IF
       EXISTS(Qualities_P6 = EXTEND Valid TO
                [1 TO Num_of_Alt] OF AltitudeQualityType, Num_of_Alt)

      EQUALS Invalid IF
       FORALL(Qualities_P6 = EXTEND Invalid TO
                [1 TO Num_of_Alt] OF AltitudeQualityType, Num_of_Alt)

    END EXPORT
```

```
  END DEFINITION

END MODULE


MODULE AllCrossed_P6 : AltitideVoter_P6

  DEFINITION :

    EXPORT Altitude_P6 :
      PARENT : NONE

      DEFAULT_VALUE : Below

      TRANSITION Below TO Above IF
          FORALL(EXTEND Alt_and_Quality_P6(Altitudes_P6, Qualities_P6) TO
                  [1 TO Num_of_Alt_P6] OF AltitudeQualityType =
                 EXTEND Above TO
                  [1 TO Num_of_Alt_P6] OF AltitudeQualityType, Num_of_Alt_P6)

      TRANSITION Above TO Below IF
          FORALL(EXTEND Alt_and_Quality_P6(Altitudes_P6, Qualities_P6) TO
                  [1 TO Num_of_Alt_P6] OF AltitudeQualityType =
                 EXTEND Below TO
                  [1 TO Num_of_Alt_P6] OF AltitudeQualityType, Num_of_Alt_P6)

    END EXPORT

    EXPORT Quality_P6 :
      PARENT : NONE

      EQUALS Valid IF
       EXISTS(Qualities_P6 = EXTEND Valid TO
                [1 TO Num_of_Alt] OF AltitudeQualityType, Num_of_Alt)

      EQUALS Invalid IF
       FORALL(Qualities_P6 = EXTEND Invalid TO
                [1 TO Num_of_Alt] OF AltitudeQualityType, Num_of_Alt)

    END EXPORT

  END DEFINITION

END MODULE


MODULE Failure_OUT_P6 :
```

```
  INTERFACE :

     IMPORT Failure_P6 : Boolean
     END IMPORT

     IMPORT PulseInterval_P6 : TIME
     END IMPORT

     IMPORT Clock : TIME
     END IMPORT

     EXPORT Watchdog_Pulse_P6 : Boolean
     END EXPORT

  END INTERFACE

  DEFINITION :

    EXPORT Watchdog_Pulse_P 6 :
      PARENT : NONE

      DEFAULT_VALUE : false

      TRANSITION False TO True IF
        TABLE
          DURATION(PRE(Watchdog_P ulse_P6) IN_STATE False,
              0 S, Clock) >= PulseInterv al_P6          : T ;
          Failure_P6                                    : F ;
        END TABLE

      TRANSITION True TO False IF
        DURATION(PRE(Watchdog_P ulse_P6) IN_STATE True, 0 S, Clock) >= 0 S

    END EXPORT

  END DEFINITION

END MODULE



MODULE ASW_REQ_P6 :

  INTERFACE :
```

```
EXPORT CON_DOI_P6 : DOIControlledType
Purpose : &*L  This variable represents the ASW's
commanded status of the Device of Interest (DOI). L*&

Interpretation : &*L
  \begin{quote}
  \begin{mydescription}
  \item[On:] Indicates that the DOI is commanded to be On.  The DOI
  is commanded to be on when the aircraft enters the target region
  for turning the DOI on, the DOI is not already on,
  and the ASW is not inhibitied.
  \item[Off:] Indicates that the DOI is commanded to be Off.  The
  DOI is commanded to be off when the aircraft leaves the target
  region and after a certain period of time has passed.  If this
  time is \RUndefined, then the ASW will never turn the DOI Off.
  \item[Uncommanded:] Indicates that the DOI is not commanded by the
  ASW.  This CON\_DOI variable will be equal to Uncommanded in any
  step were the ASW does not issue a command to the device of interest.
  \end{mydescription}
  \end{quote}
  L*&

Issues : &*L
  \begin{myitemize}
  \item If the aircraft leaves the target area and the DOI is on,
  but was {\em not} commanded to be on by the ASW, should the ASW
  turn it off?
  \end{myitemize}
  L*&

END EXPORT

EXPORT CON_Failure_P6 : Boolean
 Purpose : &*L This variable represents the ASW's indication of
 whether or not it has failed to the external world.  It is
 potentially displayed to the pilot and/or used by other subsystems
 on board the aircraft. L*&

 Interpretation : &*L
  \begin{quote}
  \begin{mydescription}
  \item[True:] Indicates that the ASW has failed.  The ASW is
  considered to be failed if it attempts to turn on the DOI, but the
  DOI does not turn on after a certain timeout period.
  \item[False:] Indicates that the ASW has not failed.  The ASW is
  considered to be operating normally if none of the failure
```

```
  conditions are true.
  \end{mydescription}
  \end{quote}
  L*&
END EXPORT

IMPORT MON_Altitude_P6 : AboveBelowType
  CLASSIFICATION : Monitored

  Purpose : &*L This variable represents the ASW's idea of what the
  altitude of the aircraft is.  It is related to the Altitude\_Quality
  variable. L*&
END IMPORT

IMPORT MON_Altitude_Quality_P6 : AltitudeQualityType
  CLASSIFICATION : Monitored

  Purpose : &*L This variable represents the quality of the
  Altitude of the aircraft is. L*&
END IMPORT

IMPORT MON_DOI_P6 : OnOffType_P6
  Purpose : &*L This variable indicates the monitored status of the
  DOI.  The DOI can be turned on or off by other devices/systems on
  board the aircraft, so the ASW needs an accurate accounting of the
  status of the DOI L*&

  Interpretation : &*L
    \begin{quote}
    \begin{mydescription}
    \item[On:] Indicates that the DOI is currently on.
    \item[Off:] Indicates that the DOI is currently off.
    \end{mydescription}
    \end{quote}
    L*&

 END IMPORT

IMPORT MON_Reset_P6 : Boolean

  Purpose : &*L This variable indicates the whether the ASW should be
  reset or not.  In a step where the ASW is reset, this variable will
  have the value true.  In all others, this variable will have the
  value false. L*&

  Interpretation : &*L
    \begin{quote}
```

```
    \begin{mydescription}
    \item[True:] Indicates that the ASW as been reset.
    \item[False:] Indicates that the ASW has not been reset.
    \end{mydescription}
    \end{quote}
    L*&

END IMPORT

IMPORT MON_Inhibit_P6 : Boolean

   Purpose : &*L This variable is true when the ASW is inhibited and
   false otherwise.  The value is determined by the user and/or other
   systems on board the aircraft. L*&

  Interpretation : &*L
    \begin{quote}
    \begin{mydescription}
    \item[True:] Indicates that the operation of the ASW has been
    inhibited; the ASW shall not attempt to change the status of the
    DOI.
    \item[False:] Indicates that the ASW has not been inhibited; the
    ASW will behave as specified by other requirements.
    \end{mydescription}
    \end{quote}
    L*&

END IMPORT

IMPORT CONSTANT Threshold_P6 : INTEGER
  UNITS : ft
  EXPECTED_MIN : 0
  EXPECTED_MAX : 8024

  Purpose : &*L This constant will be defined by each family
  member when the REQ module is instantiated.  It is the altitude
  at which the ASW is required to turn on or off the ASW. L*&

END IMPORT

IMPORT CONSTANT Invalid_Alt_Failure_P6 : Time
  UNITS : NA
  EXPECTED_MIN : 2 s
  EXPECTED_MAX : 10 s

  Purpose : &*L This constant will be defined by each family
  member.  It is the length of time after which the ASW will
```

```
  declare a failure if there is not valid altitude. L*&

END IMPORT

IMPORT CONSTANT DOI_Timeout_P6 : Time
  UNITS : NA
  EXPECTED_MIN : 1 s
  EXPECTED_MAX : 5 s

  Purpose : &*L This constant will be defined by each member of
  the ASW family to represent the amount of time before the ASW
  declares a failure if the DOI does not respond to a command. L*&

END IMPORT

IMPORT CONSTANT GoAboveAction_P6 : ActionType

  Purpose : &*L This constant specifies the action that the ASW
  will perform when it crosses the Threshold going up. It is
  specified by the decision model for each family member. L*&

END IMPORT

IMPORT CONSTANT GoBelowAction_P6 : ActionType

  Purpose : &*L This constant specifies the action that the ASW
  will perform when it crosses the Threshold going down. It is
  specified by the decision model for each family member. L*&

END IMPORT

IMPORT CONSTANT GoAboveHyst_P6 : INTEGER
  UNITS : ft
  EXPECTED_MIN : 50
  EXPECTED_MAX : 500

  Purpose : &*L This defines the hysteresis factor for going above
  the threshold altitude. L*&

END IMPORT

IMPORT CONSTANT GoBelowHyst_P6 : INTEGER
  UNITS : ft
  EXPECTED_MIN : 50
  EXPECTED_MAX : 500

  Purpose : &*L This defines the hysteresis factor for going above
```

```
    the threshold altitude. L*&

  END IMPORT

END INTERFACE

DEFINITION :

  STATE_VARIABLE ASW_System_Mode_P6 :
    VALUES : {Startup, NormalOperating, Degraded, Failed, Reset}
    PARENT : NONE

    Purpose : &*L This is the top-level mode of the ASW.  If the ASW
    were to have a startup mode, etc., we could put those modes as
    children of this controlling mode.  Currently, we have only two
    states, the reset mode which is used for when the reset signal
    is received and the operating mode that handles the main
    behavior. L*&

    DEFAULT_VALUE : Startup

    TRANSITION NormalOperating TO Reset IF MON_Reset_P6

    TRANSITION Degraded TO Reset IF MON_Reset_P6

    TRANSITION NormalOperating TO Degraded IF
      EpisodeMonitor_P6 = QualifyingEpisode

    TRANSITION Degraded TO NormalOperating IF
      DURATION (MON_Altitude_Quality_P6 = Valid, 0 S, Clock) > 1 MIN

    TRANSITION Reset TO NormalOperating IF
      DURATION(PRE(ASW_System_Mode_P6), 0 s, Clock) >= 0 S

  END STATE_VARIABLE

  STATE_VARIABLE EpisodeMonitor_P6 :
    VALUES : {NoEpisode, FirstEpisode, QualifyingEpisode}
    PARENT : NONE

    Purpose : &*L This simple state variable tracks whether or not
    we have met the conditions for being in degraded functionality
    mode.  Namely, whether or not we have seen two periods of
    invalid altitude lasting 1 second or more within 1 minute. L*&

     DEFAULT_VALUE : NoEpisode
```

```
   TRANSITION NoEpisode TO FirstEpisode IF
     DURATION(MON_Altitude_Q uality_P6 = Invalid, 0 S, Clock) > 1 S

   TRANSITION FirstEpisode TO QualifyingEpisode IF
     TABLE
       DURATION(MON_Altitude_Q uality_P6 = Invalid, 0 S, Clock) > 1 S : T ;
       DURATION(PRE(EpisodeMon itor_P6) = FirstEpisode) > 1 S         : T ;
     END TABLE

   TRANSITION FirstEpisode TO NoEpisode IF
     DURATION(PRE(EpisodeMon itor_P6) = FirstEpisode) >= 1 MIN

   TRANSITION QualifyingEpiso de TO NoEpisode IF
     DURATION(MON_Altitude_Q uality_P6 = Valid, 0 S, Clock) >= 2 MIN

END STATE_VARIABLE

MODULE_INSTANCE ASW_Operating_Mode_P6 : ASW_Operating_Mode_Def_P6
  PARENT : ASW_System_Mode_ P6.NormalOperating
  ASSIGNMENT
     MON_Altitude_P6          := MON_Altitude_P6,
     MON_Altitude_Quality_P6  := MON_Altidue_Quality_P6,
     MON_DOI_P6               := MON_DOI_P6,
     MON_Inhibit_P6           := MON_Inhibit_P6,
     Threshold_P6             := Threshold_P6,
     Invalid_Alt_Failure_P6   := Invalid_Alt_Failure_P6,
     DOI_Timeout_P6           := DOI_Timeout_P6,
     GoAboveAction_P6         := GoAboveAction_P6,
     GoBelowAction_P6         := GoBelowAction_P6,
     GoAboveHyst_P6           := GoAboveHyst_P6,
     GoBelowHyst_P6           := GoBelowHyst_P6,
     DOI_Delay_P6             := 0 S
  END ASSIGNMENT
END MODULE_INSTANCE

MODULE_INSTANCE ASW_Degraded_Mode_P6 : ASW_Operating_Mode_Def_P6
  PARENT : ASW_System_Mode_ P6.Degraded
  ASSIGNMENT
     MON_Altitude_P6          := MON_Altitude_P6,
     MON_Altitude_Quality_P6  := MON_Altidue_Quality_P6,
     MON_DOI_P6               := MON_DOI_P6,
     MON_Inhibit_P6           := MON_Inhibit_P6,
     Threshold_P6             := Threshold_P6,
     Invalid_Alt_Failure_P6   := Invalid_Alt_Failure_P6,
     DOI_Timeout_P6           := DOI_Timeout_P6,
     GoAboveAction_P6         := GoAboveAction_P6,
     GoBelowAction_P6         := GoBelowAction_P6,
```

```
        GoAboveHyst_P6          := GoAboveHyst_P6,
        GoBelowHyst_P6          := GoBelowHyst_P6,
        DOI_MinDelay_P6         := 2 S,
        DOI_MaxDelay_P6         := 6 S
      END ASSIGNMENT
    END MODULE_INSTANCE

    EXPORT CON_DOI_P6 :
      PARENT :  NONE
      DEFAULT_VALUE : Uncontrolled

      EQUALS ASW_Operating_Mode_P6.CON_DOI_P6
        IF ASW_System_Mode_P6 = NormalOperating

      EQUALS ASW_Degraded_Mode_P6.CON_DOI_P6
        IF ASW_System_Mode_P6 = Degraded

      EQUALS Uncontrolled IF
        TABLE
          ASW_System_Mode_P6 = Failed : T * ;
          ASW_System_Mode_P6 = Reset  : * T ;
        END TABLE

    END EXPORT

    EXPORT CON_Failure_P6 :
      PARENT :  NONE
      DEFAULT_VALUE : False

      TRANSITION False TO True IF
        TABLE
          ASW_System_Mode_P6 = NormalOperating    : T * ;
          ASW_Operating_Mode_P6.CON_Failure_P6    : T * ;
          ASW_System_Mode_P6 = Degraded           : * T ;
          ASW_Operating_Mode_P6.CON_Failure_P6    : * T ;
        END TABLE

      TRANSITION True TO False IF ASW_System_Mode_P6 = Reset

    END EXPORT

  END DEFINITION

END MODULE


MODULE ASW_OperatingMode_Def_P6 :
```

```
INTERFACE :

  EXPORT CON_DOI_P6 : DOIControlledType
  END EXPORT

  EXPORT CON_Failure_P6 : Boolean
  END EXPORT

  IMPORT MON_Altitude_P6 : AboveBelowType
  END IMPORT

  IMPORT MON_Altitude_Qua lity_P6 : AltitudeQualityType
  END IMPORT

  IMPORT MON_DOI_P6 : OnOffType_P6
  END IMPORT

  IMPORT MON_Inhibit_P6 : Boolean
  END IMPORT

  IMPORT CONSTANT Threshold_P6 : INTEGER
    UNITS : ft
    EXPECTED_MIN : 0
    EXPECTED_MAX : 8024
  END IMPORT

  IMPORT CONSTANT Invalid_Alt_Failure_P6 : Time
    UNITS : NA
    EXPECTED_MIN : 2 s
    EXPECTED_MAX : 10 s
  END IMPORT

  IMPORT CONSTANT DOI_Timeout_P6 : Time
    UNITS : NA
    EXPECTED_MIN : 1 s
    EXPECTED_MAX : 5 s
  END IMPORT

  IMPORT CONSTANT GoAboveAction_P6 : ActionType
  END IMPORT

  IMPORT CONSTANT GoBelowAction_P6 : ActionType
  END IMPORT

  IMPORT CONSTANT GoAboveHyst_P6 : INTEGER
    UNITS : ft
```

```
    EXPECTED_MIN : 50
    EXPECTED_MAX : 500
  END IMPORT

  IMPORT CONSTANT GoBelowHyst_P6 : INTEGER
    UNITS : ft
    EXPECTED_MIN : 50
    EXPECTED_MAX : 500
  END IMPORT

  IMPORT DOI_MinDelay_P6 : TIME

    Purpose : &*L This parameter to the ASW operating module
    determines whether or not we will wait to turn the DOI on. If it
    is greater than zero, then we will wait.  It represents the
    minium waiting time L*&

  END IMPORT

  IMPORT DOI_MaxDelay_P6 : TIME

    Purpose : &*L This parameter to the ASW operating module
    determines the maximum waiting time that we will stay in a
    Delayed action state before giving up and returning to NoAction
    L*&

  END IMPORT

END INTERFACE

DEFINITION :

  EXPORT CON_DOI_P6 :
    PARENT : NONE
    DEFAULT_VALUE : Uncommanded

    TRANSITION Uncommanded TO On IF
      TABLE
         GoBelowAction = TurnOn              : T * ;
         ActionBelow_P6.PerformA ction_P6    : T * ;
         GoAboveAction = TurnOn              : * T ;
         ActionAbove_P6.PerformA ction_P6    : * T ;
      END TABLE

    TRANSITION Uncommanded TO Off IF
      TABLE
         GoBelowAction = TurnOff             : T * ;
```

```
        ActionBelow_P6.PerformA ction_P6     : T * ;
        GoAboveAction = TurnOff              : * T ;
        ActionAbove_P6.PerformA ction_P6     : * T ;
      END TABLE

    TRANSITION On TO Uncommanded IF WHEN(MON_DOI_P6 = On, False)

    TRANSITION Off TO Uncommanded IF WHEN(MON_DOI_P6 = Off, False)

  END EXPORT

  MODULE_INSTANCE ActionBelow_P6 : DOI_Action_P6
    PARENT : NONE
    ASSIGNMENT
      Direction_P6            := Down,
      ThresholdedAltitude_P6  := MON_Altitude_P6,
      MinDelay_P6             := DOI_MinDelay_P6,
      MaxDelay_P6             := DOI_MaxDelay_P6,
      AltitudeQuality_P6      := MON_AlitudeQuality_P6,
      ActionOK_P6             := DOI_Action_Ok_P6(),
      Clock                   := Clock
    END ASSIGNMENT
  END MODULE_INSTANCE

  MODULE_INSTANCE ActionAbove_P6 : DOI_Action_P6
    PARENT : NONE
    ASSIGNMENT
      Direction_P6            := Up,
      ThresholdedAltitude_P6  := MON_Altitude_P6,
      MinDelay_P6             := DOI_MinDelay_P6,
      MaxDelay_P6             := DOI_MaxDelay_P6,
      AltitudeQuality_P6      := MON_AlitudeQuality_P6,
      ActionOK_P6             := DOI_Action_Ok_P6(),
      Clock                   := Clock
    END ASSIGNMENT
  END MODULE_INSTANCE

  MACRO DOI_Action_Ok_P6( act IS ActionType) :
    TABLE
      MON_Inhibit_P6       : F F ;
      CON_Failure_P6       : F F ;
      MON_DOI_P6 = On      : T * ;
      act = On             : F * ;
      MON_DOI_P6 = Off     : * T ;
      act = Off            : * F ;
    END TABLE
  END MACRO
```

```
    EXPORT CON_Failure_P6 :
      PARENT : NONE
      DEFAULT_VALUE : False

      EQUALS TRUE IF
        TABLE
          DURATION(AttemptingOn() , 0 S, Clock) > DOI_Timeout_P6      : T * * * ;
          DURATION(AttemptingOff( ), 0 S, Clock) > DOI_Timeout_P6     : * T * * ;
          DURATION(MON_Altitude_Q uality_P6 = Invalid, 0 S, Clock)    : * * T * ;
          PRE(CON_Failure_P6) = False                                 : * * * T ;
        END TABLE

      EQUALS FALSE IF
        TABLE
          DURATION(AttemptingOn() , 0 S, Clock) > DOI_Timeout_P6      : F ;
          DURATION(AttemptingOff( ), 0 S, Clock) > DOI_Timeout_P6     : F ;
          DURATION(MON_Altitude_Q uality_P6 = Invalid, 0 S, Clock)    : F ;
          PRE(CON_Failure_P6) = False                                 : F ;
        END TABLE

    END EXPORT

    MACRO AttemptingOn() :
      TABLE
        MON_DOI_P6 = Off    : T ;
        CON_DOI_P6 = On     : T ;
      END TABLE
    END MACRO

    MACRO AttemptingOff() :
      TABLE
        MON_DOI_P6 = On     : T ;
        CON_DOI_P6 = Off    : T ;
      END TABLE
    END MACRO

  END DEFINITION

END MODULE


MODULE ThresholdedAltitude _P6 :

  INTERFACE :

    IMPORT Altitude_P6 : Integer
```

```
    UNITS : ft
    EXPECTED_MIN : 0
    EXPECTED_MAX : 50000
  END IMPORT

  IMPORT CONSTANT Threshold_P6 : Integer
    UNITS : ft
    EXPECTED_MIN : 0
    EXPECTED_MAX : 8024
  END IMPORT

  IMPORT CONSTANT AboveHysteresis_P6 : Integer
    UNITS : ft
    EXPECTED_MIN : 50
    EXPECTED_MAX : 500
  END IMPORT

  IMPORT CONSTANT BelowHysteresis_P6 : Integer
    UNITS : ft
    EXPECTED_MIN : 50
    EXPECTED_MAX : 500
  END IMPORT

  EXPORT Result_P6 : AboveBelowType

    Purpose : &*L this export reports whether or not the altitude is
    above or below the threshold given the hysteresis factor L*&

  END EXPORT

END INTERFACE

DEFINITION :

  EXPORT Result_P6 :
    PARENT : NONE

    DEFAULT_VALUE : Above IF
      TABLE
        DEFINED(Altitude_P6)         : T ;
        Altitude_P6 > Threshold_P6   : T ;
      END TABLE

    DEFAULT_VALUE : Below IF
      TABLE
        DEFINED(Altitude_P6)         : T ;
        Altitude_P6 <= Threshold_P6  : T ;
```

```
    END TABLE

  DEFAULT_VALUE : UNDEFINED IF NOT (DEFINED(Altitude_P6))

  EQUALS Above IF
    TABLE
      DEFINED(Altitude_P6)                : T ;
      Altitude_P6 > EffectiveThreshold_P6 : T ;
    END TABLE

  EQUALS Below IF
    TABLE
      DEFINED(Altitude_P6)                  : T ;
      Altitude_P6 <= EffectiveThreshold_P6 : T ;
    END TABLE

  EQUALS UNDEFINED IF NOT (DEFINED(Altitude_P6))

END EXPORT

STATE_VARIABLE ApplyHisteresis_P6 :
  VALUES : {NoHyst, Above, Below}
  PARENT : NONE

  DEFAULT_VALUE : NoHyst

  TRANSITION NoHyst TO Above IF
    TABLE
      DEFINED(Altitude_P6)                     : T ;
      WHEN(Altitude_P6 < Threshold_P6, False) : T ;
    END TABLE

  TRANSITION NoHyst TO Below IF
    TABLE
      DEFINED(Altitude_P6)                     : T ;
      WHEN(Altitude_P6 > Threshold_P6, False) : T ;
    END TABLE

  TRANSITION Above TO NoHyst IF
    TABLE
      DEFINED(Altitude_P6)                                          : T T ;
      WHEN(Altitude_P6 < Threshold_P6 + AboveHysteresis_P6, False) : T * ;
      WHEN(Altitude_P6 > Threshold_P6 - BelowHysteresis_P6, False) : * T ;

    END TABLE

  TRANSITION Below TO NoHyst IF
```

```
        TABLE
          DEFINED(Altitude_P6)                                          : T T ;
          WHEN(Altitude_P6 > Threshold_P6 + AboveHysteresis_P6, False) : T * ;
          WHEN(Altitude_P6 < Threshold_P6 - BelowHysteresis_P6, False) : * T ;

        END TABLE
    END STATE_VARIABLE

    STATE_VARIABLE EffectiveThreshold_P6 : INTEGER
      PARENT : NONE
      UNITS : ft
      EXPECTED_MIN : Threshold_P6 - BelowHysteresis_P6
      EXPECTED_MAX : Threshold_P6 + AboveHysteresis_P6

      DEFAULT_VALUE : Threshold_P6

      EQUALS Threshold_P6 + AboveHysteresis_P6
        IF ApplyHysteresis_P6 = Above

      EQUALS Threshold_P6 - BelowHysteresis_P6
        IF ApplyHysteresis_P6 = Below

      EQUALS Threshold_P6
        IF ApplyHysteresis_P6 = NoHyst

    END STATE_VARIABLE

  END DEFINITION

END MODULE

MODULE DOI_Action_P6 :

  INTERFACE :

    IMPORT MinDelay_P6 : TIME
    END IMPORT

    IMPORT MaxDelay_P6 : TIME
    END IMPORT

    IMPORT CONSTANT Direction_P6 : UpDownType
    END IMPORT

    IMPORT ThresholdedAltitude_P6 : AboveBelowType
    END IMPORT
```

```
  IMPORT AltitudeQuality_P6 : AltitudeQualityType
  END IMPORT

  IMPORT ActionOK_P6 : Boolean
  END IMPORT

  IMPORT Clock : TIME
  END IMPORT

  EXPORT PerformAction_P6 : Boolean
  END EXPORT

END INTERFACE

DEFINITION :

  EXPORT PerformAction_P6 :
    PARENT : NONE
    DEFAULT_VALUE : False
    EQUALS WHEN(_internal = Perform)
  END EXPORT

  STATE_VARIABLE internal_P6 :
    VALUES : {NoAction, Delayed, Perform}
    PARENT : NONE

    DEFAULT_VALUE : NoAction

    TRANSITION NoAction TO Delayed IF
      TABLE
        MinDelay_P6 > 0 S                        : T T ;
        ActionOK_P6                              : T T ;
        WHEN(ThresholdedAltitud e_P6 = Below)    : T * ;
        Direction_P6 = Below                     : T * ;
        WHEN(ThresholdedAltitud e_P6 = Above)    : * T ;
        Direction_P6 = Above                     : * T ;
      END TABLE

    TRANSITION NoAction TO Peform IF
      TABLE
        MinDelay_P6 > 0 S                        : F F ;
        ActionOK_P6                              : T T ;
        WHEN(ThresholdedAltitud e_P6 = Below)    : T * ;
        Direction_P6 = Down                      : T * ;
        WHEN(ThresholdedAltitud e_P6 = Above)    : * T ;
        Direction_P6 = Up                        : * T ;
      END TABLE
```

```
      TRANSITION Delayed TO Perform IF
        TABLE
          DURATION(PRE(internal_P 6) IN_STATE Delayed,
             0 S, Clock) >= MinDelay_P6                    : T T ;
          ActionOK_P6                                      : T T ;
          AltitudeQuality_P6 = Valid                       : T T ;
          Direction_P6 = Down                              : T * ;
          ThresholdedAltitude_P6 = Below                   : T * ;
          Direction_P6 = Up                                : * T ;
          ThresholdedAltitude_P6 = Above                   : * T ;
        END TABLE

      TRANSITION Delayed TO NoAction IF
        DURATION(PRE(internal_P 6) IN_STATE Delayed, 0 S, Clock) >= MaxDelay_P6

      TRANSITION Perform TO NoAction IF
        DURATION(PRE(internal_P 6) IN_STATE Perform, 0 S, Clock) >= 0 S

    END STATE_VARIABLE

  END DEFINITION

END MODULE

INCLUDE "standard-modules. nimbus"
```