Technical Report

CMU/SEI-90-TR-5
ESD-TR-90-206

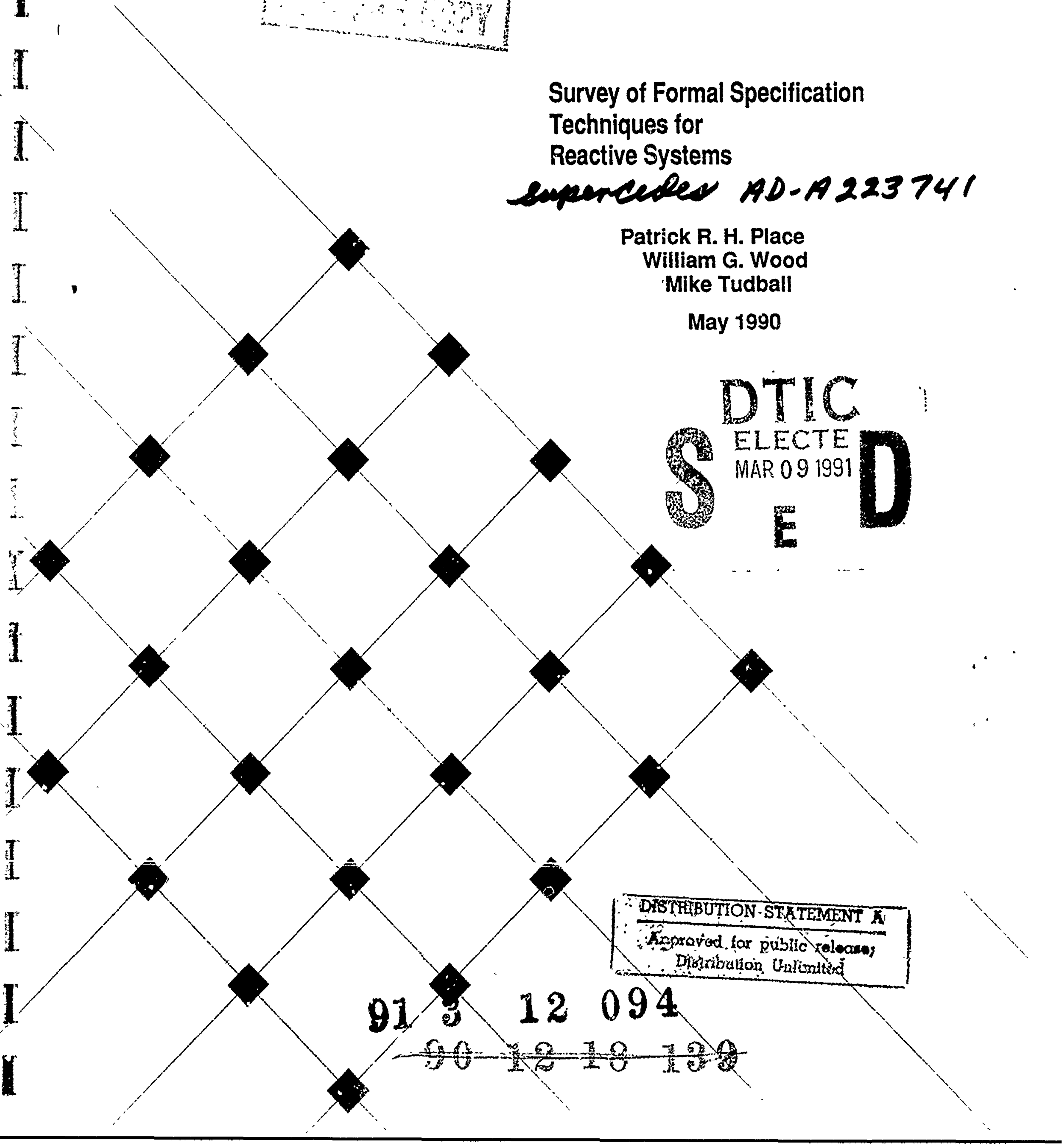
Carnegie-Mellon University
Software Engineering Institute

Survey of Formal Specification Techniques for Reactive Systems

supercedes AD-A223741

Patrick R. H. Place
William G. Wood
Mike Tudball

May 1990



The following statement of assurance is more than a statement required to comply with the federal law This is a sincere statement by the university to assure that all people are included in the diversity which makes Carnegie Mellon an exciting place Carnegie Mellon wishes to include people without regard to race, color, national origin, sex, handicap, religion, creed, ancestry, belief, age, veteran status or sexual orientation.

Carnegie Mellon University does not discriminate and Carnegie Mellon University is required not to discriminate in admissions and employment on the basis of race, color, national origin, sex or handicap in violation of Title VI of the Civil Rights Act of 1964, Title IX of the Educational Amendments of 1972 and Section 504 of the Rehabilitation Act of 1973 or other federal, state, or local laws or executive orders. In addition, Carnegie Mellon does not discriminate in admissions and employment on the basis of religion, creed, ancestry, belief, age, veteran status or sexual orientation in violation of any federal, state, or local laws or executive orders Inquiries concerning application of this policy should be directed to the Provost, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh PA 15213 telephone (412) 268 6684 or the Vice President for Enrollment, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268 2053

Technical Report
CMU/SEI-90-TR-5
ESD-TR-90-206
May 1990

Survey of Formal Specification Techniques for Reactive Systems

Patrick R. H. Place
William G. Wood
Specification and Design Methods
and Tools Project

Mike Tudball
Ferranti Computer Systems Ltd.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Contents

Supercedes AD A223 741 per telecon
Barbara Zayas. Carnegie Mellon University
Software Engineering Institute. Pittsburgh
Pa. 15213.
VHG 2/19/91

List of Figures

List of Tables

Chapter 1

Executive Summary

Abstract: Formal methods are being considered for the description of many systems including systems with real-time constraints and multiple concurrently executing processes. This report develops a set of evaluation criteria and evaluates Communicating Sequential Processes (CSP), the Vienna Development Method (VDM), and temporal logic. The evaluation is based on specifications, written with each of the techniques, of an example avionics system.

In software-intensive, reactive systems, many of the errors detected in the integration, testing, and installation stages of a system's development can be traced to ambiguities, inconsistencies, and incompleteness in the requirements specification. The removal of these errors often requires a considerable amount of backtracking of the system to discover the source of the error, followed by changes at many levels of the system's description to eliminate the cause of the error. For this reason it is advantageous to have a more precise, consistent, and complete specification of the system. The formal specification techniques evaluated in this report are one way of developing specifications with the desired characteristics. We have restricted our attention to formal methods for the specification of concurrent systems, since there are already adequate comparisons of formal sequential techniques. We have further chosen to look at only three of the currently popular formal concurrent techniques to ensure that our approach is valid before committing further resources to the task. Each of the three techniques was applied by a different project team member to specify the same avionics problem.

In order to gain confidence in the validity of formal methods, we visited a number of organizations involved in formal software development. These organizations were variously developing formal methods, or tools supporting such methods, or were using the methods and tools either on real projects or pilot project case studies. Our visits were largely to organizations in the United Kingdom, since this is where most formal specification methods development and usage is taking place. Further, the Ministry of Defense (MOD) in the UK has recently dictated the use of such methods for safety-critical systems. Our conclusions from these trips were that formal methods are being taken seriously, though most usages are pilot projects and are still at the paper, pencil, and wastebasket level, and that useful automated tools are slowly being developed.

The comparison and evaluation of each formal method were performed in the following five stages.

1. We developed some evaluation criteria, based on existing taxonomies of methods and tools.
2. We chose an example avionics problem to serve as a basis for evaluating the various techniques.
3. We then specified the problem using Hoare's communicating sequential processes (CSP) technique. During this specification, we made a number of assumptions about system operation that were not adequately described in the requirements document for the example problem.
4. Using the CSP specification as our base, we re-specified the system using temporal logic and an extended Vienna development method (VDM).
5. We evaluated each method individually against our previously selected criteria, created a summary of our evaluations, and drew some conclusions from the evaluations.

The results of this evaluation must be taken cautiously, because one important criterion in evaluating a method is the ease with which a design and implementation can be derived from the specification created by using that method. We developed neither designs nor implementations from the specifications. Given this limitation, we preferred CSP, largely because of its ease in composing primitive processes into larger grained processes. We found temporal logic to be a close second, and were impressed with the ability to use model checking to verify the consistency of the temporal formulas. We rated VDM as a rather distant third, mainly because of the awkwardness of expressing post-conditions on the larger grained composite processes. Again, our method of mimicking the CSP specification may have handicapped VDM's performance. A radically different approach may have yielded a better specification.

Chapter 2

Introduction

This report describes an evaluation of formal techniques for specifying reactive systems. The term *reactive systems*[1] is used in this report to describe systems that do not terminate but instead maintain continuing interaction with the environment. Examples of reactive systems are operating systems and industrial control systems.

The report classifies each specification technique using criteria based on the characteristics and development of reactive systems. We also provide an evaluation of the applicability of the techniques to the specification of reactive systems. The report also includes a summary of our discussions with developers and practitioners of formal specification techniques.

2.1 Purpose of the Survey

The purpose of this report is threefold: First, to report on the current practice in applying formal methods to developing software systems; second, to present an objective comparison and evaluation of three formal methods; and third, to provide the basis for further work.

We report on the state of the practice for two reasons. It gave us a picture of the state of the industry in using formal techniques both in terms of the techniques being used and the tools used to support the formalisms. It helps us to understand what is needed to make formal methods viablc in industry.

[1] Coined by Pnueli [15] in 1985.

The intent of this work was to provide an objective comparison and evaluation of a number of currently popular formal specification techniques. This comparison could subsequently be used as a guide to choosing the most appropriate technique to be applied to the specification of a system. This report also provides some examples of specification using the various techniques; these examples may be used as models and, we believe, may help readers understand how to specify their own systems.

The sample problem and classification criteria reported here can be used to classify and evaluate other specification techniques. The greater the number of techniques that can be classified and evaluated, the greater the utility of the work started in this report. A developer wishing to use the report as a base for selecting a formal specification technique to apply to some system will have a wide range of techniques from which to choose. Further, a reader interested in formal specifications would have a number of model specifications to read.

2.2 Background

Before presenting the results of the survey, it is useful to explain why another survey is of value when others have already been published [11, 17].

The Software Productivity Consortium (SPC) report [11], which publishes a survey of formal methods, discusses aspects of development other than specification. A narrower focus may provide more insight into the nature of the specification techniques. Further, each specification presented by the SPC is of a different software system, making it harder to compare the specification techniques in an objective manner. The Sannella report [17] discusses specification of sequential software systems which, although such systems play an important role, does not address the harder problems of concurrency found in typical reactive systems. This survey fills a gap left by the other two.

This report acts as an aid to making a choice of specification technique. We should be able to examine a system, classify it according to our criteria, and choose the technique that most closely matches the problem.

Rather than considering techniques applicable to every aspect of a system's development, we focus on requirements specification alone. This enables us to narrow the list of available formal techniques, so that these techniques can be treated in greater depth than they could in a study which covered all techniques applicable at every stage of the development cycle.

It should be noted that this report's restriction in scope to only formal methods does not imply that formal techniques applied to other stages of system development are unimportant. Rather, we consider requirements specification to be the basis of the software development process and that subsequent effort must investigate the role of formalism in the other stages of the development life cycle. We would also like to note that we consider writing a formal specification, even without doing subsequent formal development, to have advantages over omitting such a step. Specifically, writing the specification implies that the requirements become well understood and many problems of conflicting, confusing, and ambiguous requirements may be resolved early rather than late in the development of a system.

This report, as stated in the title, is a survey of specification techniques. We do not attempt to provide a tutorial on any of the specification techniques and recommend that a reader looking for such peruse the appropriate literature. We also do not justify our belief in the value of formal specification. Such justifications have already been treated at length in the literature, for example by Turski and Maibaum [18].

A complete survey would cover specification techniques that are currently popular or are, in our opinion, of particular importance. Techniques such as Communicating Sequential Processes (CSP) [7], temporal logic [3], an extended Vienna Development Method [10], Calculus of Communicating Systems [13], Petri nets [16], LOTOS [8], and many more should be covered. This report covers the initial work in this survey effort where CSP, temporal logic, and an extended VDM have been classified and evaluated.

As a final note on the choice of techniques, it is obvious that we cannot cover the entire field of specification techniques and we are happy to act as a "clearinghouse" for other method developers or users who believe that a particular technique has been unjustly omitted and wish to provide a solution to our problem. We would then be in a position to periodically update this report by adding new techniques.

2.3 Organization of This Report

This report is organized according to the order of the activities we carried out while performing our survey.

In Chapter 3 we present the results of a number of visits that we made to both developers and users of formal specification techniques. To preserve anonymity, we have omitted the names of the people visited and present a summary of the visits.

In Chapter 4 we describe in detail the classification criteria used on the formal specifications. This chapter defines the terms used in the classification and the reasoning behind the organization of the criteria.

The sample problem is presented in Chapter 5. This problem is taken from the original text of a requirements document of an example avionics system, with sections of the original text removed. The original text was cut down in order to create a problem of a manageable size. We have not altered any of the text of the original document in order to make the work of creating the specifications simpler. This section of the report also discusses the manner in which we generated the problem specifications and our aims in this mode of work.

Successive chapters present the formal specifications of the problem. With each specification are associated comments, both on the specification and on the problem as determined by the particular specification. We also present with each formal specification the classification of the technique employed and the justification for our classifications. The classifications are based primarily upon the specifications we developed, though additional data from the literature is included.

In Chapter 9 we summarize the technique classifications and also present some generic statements about the problem. These statements formulate issues that we believe would need resolution before undertaking development, in that they demonstrate potentially unexpected behaviors of the system.

Finally, in Chapter 10 we present our conclusions on this work, discussing the value of the work as well as the current state of the practice. This chapter also includes discussion of a number of possible research directions that could be taken based on the work of this report. We list those that seem of interest and value.

Chapter 3

State of the Industry

In order to substantiate the validity of our efforts, we visited a dozen sites involved in formal software development. Although most of the sites we visited were in Britain, where considerable effort is being put into the use of formal techniques, we did try to ensure that the visits covered a representative sample of industry and academia.

The purpose of the visits was twofold: first, to discover the current state of industrial application of formal techniques and second, to discuss and refine our classification criteria. In addition to method users, we also visited method developers so that we could obtain their views on how industrial users were applying their methods.

For various commercial reasons, we were asked by a number of people we visited to keep their comments anonymous, which we have done. However, we would like to acknowledge their contributions here and thank them for the time they spent talking to us and the advice given to us concerning our project.

The three sections that follow summarize our visits. The first section indicates the sizes of the projects where formal techniques were used and how the method users felt about the techniques they used. The second section lists the key points that were made by the people interviewed. The sentences in bold summarize the points. The associated paragraphs provide the justification of the point. It should be noted that some of the points were made to us by a number of people and that no importance is to be attached to the order in which the points are listed. The final section presents our conclusions concerning the points made to us on the trip.

3.1 Project Information

The sizes of the projects on which formalisms were used varied enormously from a 1 year (2 people for 6 months) project to a project of approximately 100 person years. The projects covered a wide range of applications, including medical systems where correctness was of paramount importance, project management tools, the interface to a layer of a project support environment, a number of applications used by the security people, and a transaction processing system. All of the people we spoke to were using formalisms in applications that formed part of their respective company's product line. These applications were important to the general business of the company.

All of the people we spoke to felt that using a formal specification had improved the quality of their code and, in a number of cases, had reduced the time between product inception and release. In one case, we were told that the company believed that the product could not have been released on time had it not been for the use of a formal specification of the system. Generally, the users had only subjective rather than objective views. One project was keeping data relating the bugs found in the formally specified release of their product to those found in previous product releases; however, at the time we visited, these figures were not available. The subjective view was that they expected to see an improvement with fewer bugs being reported than for the previous releases.

3.2 Key Points

As has been stated, the points presented in this section are in no particular order.

- **Formalism aids in the understanding of the system being built.** One of the main advantages of using a formal technique is that the developer is encouraged to think abstractly about the system rather than thinking about how the system will be implemented. This conceptualization enabled the developers we interviewed to gain insight into the system that would otherwise have been lost in the details of the implementation. One group, about to construct a system similar to an existing system, took the object-oriented design of the existing system and created a more abstract, formal specification which they subsequently used as the basis for a new object-oriented design for the new product. They stated that they found the formal specification to be invaluable, that it enabled them to produce a better design for the new product than if they had started by modifying the existing design.

- **The manner in which formalisms are introduced is of great importance.** A number of people stressed that for greatest success, formal techniques must be introduced carefully into any new group. Most important is that the group should be carefully selected. Specifically, they should already be using some method, they should recognize the deficiencies in their current method, and they should be interested in adopting a better approach. Such a group is receptive to formalism and by a process of training and consulting may be gradually introduced to the use of formal techniques. At first, they will need a great deal of assistance; this effort will decrease as the group becomes more familiar with, and confident in, their ability to apply the technique.

- **There is much activity in mixing and matching techniques.** The people we visited are using combinations of techniques for industrial-scale applications more often than they are using a single technique. There are two main reasons for this mixing and matching of techniques.

  1. The introduction of a completely unfamiliar method requires re-training of development staff and is costly in terms of the time required for the developers to gain the ability to use the new method effectively. The use of formalism to overcome specific deficiencies in the existing method reduces the time needed for the development staff to start using formalism.
  2. There has been a gap in the development of formal techniques in that some techniques have concentrated on sequential systems and others have concentrated on concurrency. Until recently, no technique has looked at both sequential and concurrent aspects of a system. Most industrial systems contain both concurrent and sequential components and a combination of two formal techniques is required for complete systems specification. There is interest in using the recently standardized technique, LOTOS, which may be used to specify both concurrent and sequential systems.

  For these two reasons, people are mixing and matching formal sequential specification techniques with both formal and informal concurrent specification techniques.

- **Tools are of less importance in practical use of formalism than is usually assumed.** Although a number of practitioners are busy writing tools of various types, from syntax-checking editors to complicated proof assistants, evidence suggests that simple tools are sufficient as long as developers are more concerned with the construction of the specification than with manipulation and examination of that specification. Many of the people to whom we spoke felt that for their projects, the greatest benefit was in writing down the formal specification, and that they had only limited interest in performing a full formal proof of development. Their reason was that the formal specification could be developed at little cost and was very useful, but that formal proof of the development was too expensive and had limited gain in most cases. It was generally felt that tools become much more important to a project when a formal development is going to take place from the specification. Then tools such as proof assistants become important to stop the development time from growing rapidly. The communities concerned with safety-critical and security systems are investing more effort into developing tools than other parts of the computing community.

- **The most important part of developing a formal specification technique is the development of the semantics.** A problem arises in the development of a formal specification technique in terms of the different needs that people will have of the technique. There are three different requirements of the technique: the ability to reason with it, its expressive power, and the tools available to support the technique. All of the people we spoke to agreed that the semantics of the formal specification technique were of fundamental importance to the successful application of the technique. The semantics of the technique have to be such that specifications can be reasoned about using the technique. The other interests, the expressive power and tool availability, were of lesser importance, with some people stating that the availability of tools was of no importance.

- **There are now many small examples of formal specifications.** Interest in formal techniques has spread from the universities into industry and there are now a large number of examples of formal specifications of relatively small systems that have been developed. Although these specifications are small, they provide confidence to the specifiers and are leading to the development of larger specifications. Further, in a number of cases, formal specification has moved out of the research departments of large corporations and into active use by developers throughout those companies.

- **Specification execution is becoming an important aspect of formal specification.** Many people consider specification animation as a part of the specification phase of a system. It is easy to make statements with a formal specification technique that are complete and internally consistent. However, it is important to ensure that the formal specification delineates the system described in the requirements document. The technique currently receiving the most attention for this form of validation is specification execution in which the specification is, in some sense, executed as though it were a high-level program implementing the system. This execution allows the specifier to determine that the specification describes a system that exhibits the desired behavior. There are a number of forms of animation such as direct execution, some form of transformation followed by execution, or even transformation followed by proof. However, no matter what the animation technique employed, the concept of animation is important both for checking the specification against the requirements and demonstrating unexpected properties of the specification.

- **Formal methods can be taught at the undergraduate level.** The ability to teach formal techniques was discussed a number of times. The consensus was that there is nothing particularly difficult about introducing formal methods at an early educational level; indeed, they can be taught from the very start of an introduction to programming course. This introduction would give the students a reading knowledge of the techniques and later courses would teach the students how to write specifications. We were told that although many universities are not teaching detailed courses on formal methods, the general awareness of such techniques is higher in new graduates than in long-time practicing engineers. This has meant that it is easier to teach formal methods to the new graduates than to the other engineers in an industrial setting.

- **There is increasing interest in hardware verification.** More and more formal specifications of system hardware components of a system are being produced. This is happening for two reasons.

  1. Hardware specification and verification should be simpler than software specification and therefore offers a proving ground for the techniques. This enables the developers to improve the quality of the techniques and also demonstrates the validity of those techniques.

  2. Because of the cost of chip development and testing, it is becoming more important to know in advance that the implementation of a chip will meet its requirements.

There is a lot of interest in the hardware specification language ELLA, though there have also been hardware specifications written in Z.

- **Z is becoming as popular as VDM.** Evidence, in the form of tools being developed and sold and specifications being written, suggests that Z is becoming at least as popular as VDM. Tool vendors are concentrating on Z tools, finding that they have a wider market for these tools than for equivalent VDM tools. Also, many of the specifications alluded to during our visits were in Z rather than VDM. It was stated, on the other hand, that the VDM standardization effort may be retarding the process of introducing VDM, since tool developers are waiting for the standard to be published before committing effort to VDM tools.
- **MOD (Ministry of Defense) 00-55 will have an effect on British defense contractors.** MOD 00-55 is a draft standard developed by the British Ministry of Defense in conjunction with some academic and industrial groups. It requires that all safety-critical portions of a system should be formally specified and verified. The standard has recommendations about the formal techniques to use, as well as code constructs that must be avoided. A related document (MOD 00-56) discusses techniques for identifying the safety-critical portions of the system. Opinions on MOD 00-55 were mixed, with most people stating that it would not affect them directly, though they were convinced that in the long run, either MOD 00-55 or a similar standard will be enforced and will affect their business. Two groups thought that it was premature and could not be enforced until at least the mid-1990's, though another group suggested that the time was now right for the introduction of the standard and that it could be enforced immediately. Finally, one group we spoke to is opposed to the standard and believes that it can never be enforced.

3.3 Summary

It should be stressed that this summary is our view, based on our discussions with the method developers and users, concerning the state of practice of application of formal specification techniques and thus differs from the preceding section of this chapter, which represented the views of the people we visited.

The existence of MOD 00-55 will make a difference to software development in Britain. We expect that contractors and researchers will pay more attention to formal developments of large-scale systems. The researchers will look for ways in which the process of formal development may be made simpler, while the contractors will be gaining an increasing awareness of formal techniques and will attempt pilot projects. It is questionable as tc whether MOD 00-55 will ever be an enforceable standard; however, the existence of the draft standard and the support for that standard by the MOD will mean that developers will position themselves to be ready to employ formal specification techniques if required. Also, rather than insisting on complete formality, we expect that there will be more emphasis on rigorous development. Although MOD 00-55 may never be an enforceable standard as it currently exists, it may well lead to a standard which is enforceable and acceptable to the contractors, and which requires some level of formality in software development.

Formal specification and development techniques are beginning to be taught at all levels in Britain. There used to be a concern that formalism was difficult to learn and that teaching these techniques was best left to optional courses at the graduate level. However, it has since been recognized that formalism is easier to learn than was previously thought and it can be taught to undergraduates in introductory programming courses.

There are now a number of companies applying formalism and although this has been happening primarily in research departments, it is now beginning to move into other branches of the companies so that developers are beginning to use these techniques on live projects.

There are now businesses which base a significant portion of their business on the use of formalism. These companies are promoting formalism through education and tool sets. Further, established companies are losing business to software houses with such skills, precisely because of the software houses' ability to apply formalism, which is becoming increasingly popular in requirements.

Chapter 4

Classification Scheme

This chapter describes the criteria we use to classify and evaluate specification techniques. The following sections introduce the terms we use. In some cases, we have defined a well known term, for example, *transformation*, in a narrower sense than is often used and have introduced additional terms, in this case *elaboration*, to cover meaning traditionally ascribed to transformation but that is not covered in our definition.

A classification scheme for specification techniques requires a set of criteria upon which the specification techniques may be judged. Earlier work of our project [5] divided criteria appropriate for specification into three categories:

1. Representation — the concepts of a system that can be described using the technique.
2. Derivation — the methods for producing one specification from another.
3. Examination — the properties of a system that may be determined using the specification technique.

We have adopted the same categories within the work described in this report.

There is a further category, miscellaneous criteria, based on subjective rather than objective judgements. Although we recognize the value of such an evaluation, we do not intend to do more than describe the aspects of this category that, in our opinion, a good specification notation should exhibit. Judgements based upon these criteria are only valid when a set of metrics exists against which the specifications can be measured.

The miscellaneous category also includes some additional criteria which are important to the techniques from a technical point of view and which should, perhaps, be included as part of our survey. As stated in the introduction, we restricted our investigation to specification alone, and we consider that a number of these additional criteria are not applicable to the formal techniques when only considering specification. An investigation considering the role of the formal techniques throughout the entire development life cycle may well need to investigate a number of these additional criteria.

4.1 Representation

This category of classifying specification techniques consists of the concepts of a system that a specification technique could be used to describe. It is not an exhaustive list of concepts, but rather is a list of concepts that we consider to be the most important. We consider each of these properties to be orthogonal, though a specification technique not addressing concurrency is unlikely to be able to specify communication between processes.

Style — A specification technique will usually either specify an object in the system by its *behavior* or by its *function*. While there are techniques that can specify a system either behaviorally or functionally, the manner in which the techniques are used leads the specifier to develop either a functional or behavioral specification and not a combination of the two. A behavioral specification describes the sequences of events that a correctly behaving system object can exhibit, and otherwise treats the object as a black box, giving no hint of its internal operation. A functional specification describes the outputs of an object as a mathematical function of the object's inputs (and possibly includes some notion of the state of the system). It is possible to derive the function from a behavioral specification and the behavior from a functional specification.

A second aspect of style is the manner in which the objects are described; this is generally either *operational* or *declarative*. In the operational style, a recipe is provided to describe the system in a mechanistic way. These specifications are programs, and some or all of them may be executable. In the declarative style, relationships between the objects are described with no thought as to the order of execution of the statements.

Concurrency — For many systems, especially systems that are highly reactive, there may be a need to specify the system as a number of concurrently executing processes. Although it is possible to specify the behavior (or function) of a concurrent system at an abstract level using a notation that does not represent concurrency, we are more interested in whether the specification notation can describe concurrency directly.

Communication — We are interested in systems in which the concurrently executing processes communicate. Without some form of communication, each of the processes could be independently specified with no interactions from other processes. The form of communication that the specification technique can describe affects the meaning of the specification, so we are interested in whether or not the communication is *synchronous* or *asynchronous* (or both) and whether it is through *shared data* or *communication channels*.

Non-Determinism — A system is said to be non-deterministic if an observer of the system cannot predict, given the current state and appropriate inputs, the next state of the system.

An example of a non-deterministic system is a system comprising two concurrently executing processes. If we assume that at all times one or other of the processes is executing, then we can never predict whether the currently executing process will in fact perform its next transition, or if some underlying arbiter will intervene and restart the other process.

Another example of non-deterministic behavior is that of a system which in some state may make a transition into one of a number of states, with no observable reason dictating the decision. This is an example of non-deterministic behavior, since each time the system arrives in this state, an observer cannot predict what the next state will be.

Other examples that cause non-deterministic behaviors in systems are unpredictable events, such as events caused by the passing of time.

Fairness — A system is fair if, whenever some non-deterministic choice is to be made, no choice is postponed forever. This means that if the system keeps arriving at the point where the non-deterministic choice is to be made, eventually each of the paths will have been taken. Note that the concept of fairness should not be confused with the concept of fair scheduling of processes.

We note that any concept of fairness in a specification exists due to the specification technique and not due to any wishes of the specifier. A specification technique may have no concept of fairness, which simply means that a specifier can make no assumptions about fairness of non-deterministic events. It is possible for a specifier to place fairness constraints on the input data (or sequences of environment events — those out of the control of the system); however, such fairness constraints are placed on the input to the system and do not affect the fairness of the non-deterministic events of the system.

Francez [6] describes three notions of fairness:

1. *Unconditional fairness* refers to the concept that for each behavior, each event occurs infinitely often. An example of this fairness may be found in a multi-processing system where each process is independent of the others, in which case, unconditional fairness means that no process will be permanently blocked.
2. *Weak fairness* refers to the concept that a process that is continuously enabled cannot be postponed indefinitely. For example, a process waiting to enter a critical region will eventually enter that region (so long as the condition for its waiting does not alter).
3. *Strong fairness* refers to the concept that an infinitely enabled process will eventually proceed. The difference between this and weak fairness is that the waiting condition need not be continuously enabled, simply enabled (and disabled) infinitely often.

Modularity — Many systems are large and complex. Specifications of such systems also have a tendency to be large and complex. It is important for the specification technique to be able to construct specifications in a modular way since it is easier to read small modular pieces of a specification than it is to read the entire specification as a single unit.

Time — An important aspect of a system is the behavior of its components with respect to time. We may consider timed events to divide into two categories, *periodic* and *sporadic* events, a periodic event being some event that happens at regular intervals and a sporadic event being one that happens at apparently random times. For example, a clock interrupt is a periodic event, whereas a device interrupt would be sporadic.

It is useful to specify that an event does not occur before a certain time (either absolute or relative). This may be specified by requiring that some clock event occurs before the event of interest. Then, the problem becomes one of being able to specify absolute and relative times. To do this, it is convenient if the specification technique can be used to model both local and global clocks. Within the specifications, we are interested in both periodic and sporadic events, so the notation should be able to specify that an event occurs every, say, 1 second, as well as being able to specify that an event occurs within, say, 100 ms. of another event. The latter example shows the typical use of sporadic events, that is, to snecify hard deadlines.

Data — Many large software systems contain complex data structures to store information reflecting the overall state of the system. The values of this stored information may affect the behavior of the system.

Techniques with only limited ability to describe data will almost certainly affect the way in which a specification is structured if the system to be specified contains complex data structures.

For example, a specification language with no ability to describe data would not be an appropriate choice for the specification of, say, a compiler symbol table. Further, a specification language with the ability to specify, say, only lists will treat all problems of data in this fashion, and all specifications will be biased toward lists.

User Presentation — As well as being able to describe the function of the system, the specification technique is sometimes required to describe the manner in which information is presented to the user. This is both in terms of a *user dialogue*, in which the communications between the user and the system are described, and also in terms of the position of information on the user's display, that is, in terms of what the user sees when employing the system.

4.2 Derivation

In addition to the importance of being able to represent different aspects of the system being specified, the way in which the specification technique permits one specification to bc derived from another is also important. This category, derivation, includes three criteria we examine to classify and evaluate each technique's capacity for deriving specifications from other specifications.

Transformation is the process whereby one specification is transformed into another specification by means of manipulating the symbols of the notation using the appropriate transformation rules.

Proof that one specification describes the same system as another specification is often performed by applying the transformation rules of the technique successively until the two specifications are textually identical.

Transformation, as we have defined it, adds no new information, but permits the specifier to describe the system in a different manner, one which may lead to the design of a more efficient system. Generally, the term *transformation* encompasses the addition of information; however, we have found it convenient to differentiate between change and addition, calling the latter elaboration.

Elaboration is also known as *reification* and is the process of deriving one specification from another specification by describing more details of the system. This process uses an abstract representation to construct a more concrete representation of the system. Elaboration may occur in the following ways.

1. By describing details of the system that had been omitted for the purpose of generating a clear, understandable description of the system.
2. By describing the abstract representations of data structures of the system in more concrete terms. For example, describing a set (abstract representation) in terms of an array (more concrete representation).

Proof that one specification is an elaboration of another is performed by demonstrating that all behaviors (or states) that exist in the abstract specification also exist in the elaborated specification. However, the elaborated specification may describe more behaviors (or states) than existed previously.

The reverse of elaboration is *abstraction*, a process in which details of a system are successively removed in order to produce a less and less concrete description of the system.

Composition is the term used for combining a number of specifications to form the specification of a larger system. This is the inverse of *decomposition*, which is the term used to describe splitting a specification of a system into a number of specifications of cooperating subsystems.

Typically, composition of specifications is used as a derivation technique in large systems that may be broken down into a number of smaller subsystems, each of which may then, in turn, be further broken down forming a hierarchy of cooperating specifications, the combination of which describes the entire system.

These three forms of derivation are all closely connected; often both transformation and elaboration take place simultaneously. It should be noted that a complete formal derivation of a system is rarely performed, due to the cost of such a derivation. Often, the literature will speak of *rigorous derivation*, meaning that proof is not performed, but that it could be performed if necessary.

4.3 Examination

Throughout the specification process, the specifier may wish to examine the specification to determine whether the system described has certain desirable properties. There is a danger of confusing the things that we can examine in a specification technique with the things that can be represented. The types of examination that can be carried out may well be limited by the choice of specification technique. For example, it might be possible to specify a system using a particular technique but it is not possible to examine the system to demonstrate absence of deadlock. In our evaluation, we use the specification of a system as a model of it, and thus behaviors (or function) of the specified system are assumed to be properties of the system. We have attempted to ensure that each of these examinations may be tested, in some way, whether by proof or by observation.

Properties of a system that a specifier may wish to examine are: equivalence, consistency, safety, liveness, determinacy, and correctness.

Equivalence is a concept that is closely related to that of the derivation. We have discussed the three types of derivation that may be performed on a specification. Each time such a derivation takes place, the developer is obligated to ensure that the resulting specification equates, at some level, to the original specification; that is, the developer must demonstrate that the two specifications are equivalent.

Many different forms of equivalence have been identified, ranging from a very strict notion, such as one specification may be transformed into the other using the transformation rules of the technique, to other, weaker notions. It is obvious, for example, that if a derivation by elaboration is performed and many new events are introduced, the original and derived specifications cannot be identical in terms of, say, their event histories. However, a suitable notion of equivalence may be that all of the behaviors (functions) exhibited by the original specification are also exhibited by the derived specification.

We stress that for a formal derivation to be complete, the developer must have shown, using an appropriate notion of equivalence, that the derived specification is equivalent to the original.

Consistency, as we use it, encompasses a number of concepts. First, it is most important that the specification is internally consistent. For example, a specification that states that a property is true but elsewhere in the specification states that the property is false is inconsistent. Such a specification is impossible to meet.

We also use this classification to consider the matter of under- and over-specification. We need to examine the specification to determine whether enough has been stated about the system being built and when more than necessary has been stated. *Under-specification* means that the specifier has failed to describe the actions of the system in some way; this can be caused either by missing some part of the requirements, or, perhaps, by failing to describe the system in enough detail so that undesired behaviors (or functions) may be present in the specification.

The following are two examples of over-specification.

1. The specifier makes statements about the system that can be derived from other statements in the specification. As long as the specification is consistent, then this type of over-specification is not harmful, and may even be helpful. The disadvantages are that there is more specification to read (and check); the advantage is that the additional statements might make the specification clearer to the reader.
2. Too much detail has been specified, forcing the specifier's view upon the developer. For example, this sort of over-specification occurs when the specification specifies additional properties not required by the system. An example of this is modeling a set with an array; the array has additional properties, such as ordering of elements, that are not properties of the system (in this case a set). An implementor looking at the specification may not be able to determine if the properties stated are actually required or if they are artifacts of the specification.

Safety and Liveness are, perhaps, the most commonly checked properties of a specification.

Safety encompasses a set of properties indicating that "nothing bad will happen." The method of examination is to construct a statement in the notation of the "bad thing," say two processes using a monitor at the same time, and then to demonstrate by proof, or otherwise, that the system implies that the statement cannot be achieved. Another technique for demonstrating safety is to construct a statement describing the "good" states of the system, and then to demonstrate that all of the operations of the system preserve the statement.

Liveness encompasses a set of properties indicating that "something good will happen." The specified system is examined to demonstrate that it will reach a given state (or set of states). In a system intended to execute forever, liveness examination requires that the system cannot get "stuck" in a particular state for the rest of its execution. Further, as is the case for fairness, any path of the system that is enabled an infinite number of times will be taken an infinite number of times.

Determinacy is a property related to non-determinism, but it should not be confused with it. Determinacy is the property that no matter how the execution is performed, the same result is produced. For example, we would like a sorting algorithm to be determinate; that is, every sort of the data yields the same results. Non-determinism means that we cannot predict the next action that the system will perform — whether it will execute the next step in one process or whether some other process will preempt the exiting one and start executing.

There are cases where we would like our systems to be non-determinate. For example, we would like a random number generator to generate random numbers, that is, be non-determinate, so no matter what the input to the generator, random values would be produced.

Correctness of a specification is something that can only be examined with respect to some other object. As for derivation, correctness is the process of identifying some notion of equivalence between a specification and some other object which is intended as a restatement of the specification, whether that specification be a requirements document and the restatement of a formal specification or whether the specification is formal and the restatement is in some programming language.

There are two important forms of correctness.

1. Correctness with respect to the requirements document. The process of examining a specification for this form of correctness is often referred to as *validation.* In a large project, the first document describing the system will be a customer requirements document; from this the system specification is constructed. The correctness examination that must be performed is to ensure that the specification restates the requirements document. This examination is generally performed by constructing test scenarios and then demonstrating that the specification behaves as expected in those scenarios. This demonstration may be by animation if possible, or by mathematical derivation. It should be noted that the entire process is an informal process, since the purpose of validation is to ensure that the specification conforms to the intent of the requirements which are described informally.

2. Correctness of the implementation. The process of examining a specification for this form of correctness is known as *verification*, and the examination may be performed formally, unlike validation, which depends on the perceptions of the correctness examiner. Once the specification has been accepted as describing the desired system, it is important to ensure that the subsequent development generates designs and implementations that satisfy the specification. Thus, we wish to examine the design with respect to the specification to ensure that the design restates the specification. We may even wish to examine the implementation to ensure that it satisfies the specification.

4.4 Other Criteria

As stated at the beginning of this section, there is a further category that comprises subjective rather than objective properties of a system. We discuss the properties that we consider of interest here; however, we do not specifically address these properties for each specification technique, since the properties cannot be suitably quantified. It should be noted that although we do not consider these properties further, we do not dismiss them as unimportant. We also include important aspects of the formal techniques that may not be of immediate importance in the specification of a system, but are important when a fully formal development is being performed.

It seems that for the following categories, there are many parallels between specifications and programs. Some of the properties discussed are more properties of the specification than of the specification technique, though we consider that they may be affected by the technique — hence their inclusion here. Since for the following criteria we consider specifications to be similar to programs, it is possible that metrics applied to programs may also be applicable to specifications.

Understandability is perhaps the most important subjective aspect of a specification. It is affected by the specification technique used.

While most specification techniques can be used to represent (to varying degrees) most aspects of a system, if the technique cannot directly represent some aspect, then the specifier will have to perform this representation explicitly, usually by modeling the aspect in terms of the aspects that the technique can represent. This leads to more complex and less easily understood specifications.

The notation used by the specification technique also affects the understandability of the specification. However, it should be noted that different readers look for different types of notation. For example, a reader with a mathematical background might look for a terse mathematics-like notation, whereas someone from another background might consider a pictorial representation to be more understandable.

Complexity is an important aspect of understandability. It is affected by the specification technique, though it is not obvious how specification techniques may be altered to reduce the complexity of the specifications. It is an issue related to the expressive capabilities of the technique.

A complicated specification is less likely to be understood by other readers than is a simple specification of the same system. This is not to say that a specification of a complicated object is, of necessity, a complicated specification. However, a specification that misses the appropriate abstractions is more likely to be complicated than a specification that uses such abstractions. Similarly, a specification attempting to represent some feature of the system in a notation not designed for such a representation is more likely to be complex than if the system were specified in an appropriate notation. For example, specifying a concurrent system in a sequentially based specification technique is more likely to be complicated than an equivalent specification written in a concurrency based specification technique.

Unfortunately, reliable measures of complexity are still unknown, and to date no research has been performed on the complexity of specifications.

Maintainability is obviously dependent upon understandability. If a specifier finds it hard to understand the specification, then the specifier is going to find it hard to maintain that specification. This point aside, there are aspects of the specification technique that affect the maintainability of the specification.

1. A specification technique with no notion of modularity will obviously be harder to maintain than one with such a notion. This follows from the fact that large systems tend to have large specifications, and if the maintainer cannot examine a small piece of the specification in isolation, then the effort required to maintain that specification will be greater since the entire specification has to be examined to discover the effects of a change, rather than some small piece of the specification.

2. An important aspect of maintainability is the process of altering the specification. Of particular interest is the effect of making a change in

the specification in terms of the effort required to ensure that the resulting specification has the same properties as the unchanged specification.

A formal software development process requires that all the examinations previously performed be redone in order to demonstrate that the specification still supports all properties of interest. Further, the design that followed from the specification component should be redesigned with respect to the changes in the specification.

Thus, if the specification technique makes the process of examination and design difficult, through use of obscure techniques or hard-to-use examinations, the specification technique will lead to less maintainable specifications than a technique with easy-to-use examinations and design processes.

Usability was, in a sense, touched upon under the heading of maintainability, insofar as the processes required to create, examine, and alter a specification affect the usability of the technique.

A major factor affecting the usability of a technique is the amount of computer support for the technique. There are three aspects to such support: the ease of automation, the availability of tool support, and the ease of use of the tool. A technique for which no computer assistance is possible will be less usable than a technique for which there are many tools supporting the notation, derivation, and examinations. Generally, if all aspects of a technique are supported by tools, the technique will be easier to use than one where only some of the technique is supported. Finally, the ease of tool use is a factor; after all, it is possible to create computer tools that are harder to use than manual methods.

The following are further criteria which we have not examined within this survey, but which are important, though not directly in the derivation of a specification. The manner in which a formal technique measures up to these criteria may well affect the ease with which the examinations may be performed. A fuller survey would examine these criteria, as well as those listed in the first three sections of this chapter.

Whether or not the technique admits subsets of use. There are a number of important subsets that may be required, for example, a deterministic subset. It is easier to reason about a deterministic system than a non-deterministic system, and if a system may be described using only a deterministic subset of the specification technique, the examinations should be simpler than a technique that does not admit a deterministic subset. Another important subset is the specification of time; again, reasoning about a system that has no time constraints is simpler than reasoning about a system that includes time constraints. If the technique does not have a non-timed subset, then the examinations will have to take time into account, even in systems with no time constraints.

Whether or not the technique is more than just a notation. Not all formal techniques have a method for the creation and manipulation of specifications. Specifically, it is desirable for a technique to encompass a method for the derivation of a specification, for performing the examinations and, important to a subsequent formal development, a method of proceeding towards an implementation from the specification. This latter will involve the notion of the steps to take when performing either a transformation or an elaboration, and the obligations placed upon the specifier when such steps are taken. Without a method, there are the dangers of missing key parts of the system when deriving the specification and of producing an erroneous implementation.

Whether or not the technique would discuss the reasoning ability (if any) used by the technique. If, for example, a technique had no calculus, then none of the examinations described can be carried out in a formal manner. and the technique does not provide anything more than a recasting of the initial requirements. The ability to reason about the specification, in addition to enabling the examinations, also permits the specifier to predict the behavior of the system in a given set of circumstances; this enables the specifier to detect undesirable behaviors based on the specification rather than on the implementation.

Chapter 5

Application of the Classification Scheme

In this chapter, we discuss two issues relating to the classification of the specification techniques. First, we discuss the approach we took, describing our intentions and our expectations of the value of this approach, as well as problems that may arise from the approach. Second, we present the text of the sample problem, which we used as the starting point for our formal specifications.

5.1 Approach to Creating Comparable Specifications

We wanted each of the formal specifications to describe the same system, minimizing (preferably eliminating) any differences between descriptions. In order to achieve this, the first specification written was used as the arbiter when ambiguities in the requirements had to be resolved for subsequent specifications. Specifically, where there was ambiguity in the problem requirements, the first specification became the arbiter. The first specification technique used was CSP. We consider that our approach, however, would have been as successful no matter which formal specification technique was the first to be chosen.

We met to discuss all of the specifications and agree that the system described in the specifications met the requirements as stated in the sample problem. More importantly, we discussed each of the specifications in detail in order to ensure that the systems being described were, in fact, identical. On the basis of these discussions, the specifications were modified. This process was repeated until we became satisfied with each of the formal specifications. It should be noted that, due to lack of resources, we performed this validation by inspection and discussion rather than by formal arguments.

Finally, we discussed the classifications and evaluations of the formal specification techniques. Using the experience we gained from performing the example specification, in addition to available literature, we evaluated the techniques.

5.1.1 Advantages of the Cooperative Approach

One of our early decisions was that each of the specifications should be as close to the others as possible. We consider that this makes each specification easier to compare in terms of the classification, especially where there is some element of subjective judgement.

Because we generated the problem before starting to write any specification, we avoided omitting pieces of the system simply because they were awkward to specify. Had we omitted parts of the system for such reasons we believe that our evaluations of the specification techniques would have been seriously weakened, as we would be presenting an optimistic rather than an accurate evaluation of the techniques.

A further advantage of the approach was the elimination of personalities. We removed the possibility of the different specifiers interpreting the original problem statements in ways that were most advantageous to the notations they were using at the time. Also, the discussions and agreement on the classifications reduced the level of personal bias introduced into the classifications.

5.1.2 Disadvantages of the Cooperative Approach

Using the CSP specification as a basis for other specifications biases subsequently created formal specifications. Specifically, they are limited to only the parts of the system that were specified in CSP (which does not handle some components well). This bias did occur in our work, in that the VDM specification could have been used to describe more of the details of the values computed by the navigation unit. We somewhat reduced the effects of this bias through changes late in the development of the specifications. Specifically, sections of the CSP specification were altered after they had originally been written in order to suit the approaches taken in the other specifications.

The second form of bias introduced is that of seeing a problem solution and being unable to think of the problem in a way other than that of the solution already presented. An attempt to reduce this bias has been made by use of the best approach possible throughout the specifications, using the CSP solution as a description of the problem rather than as a guide to the solution.

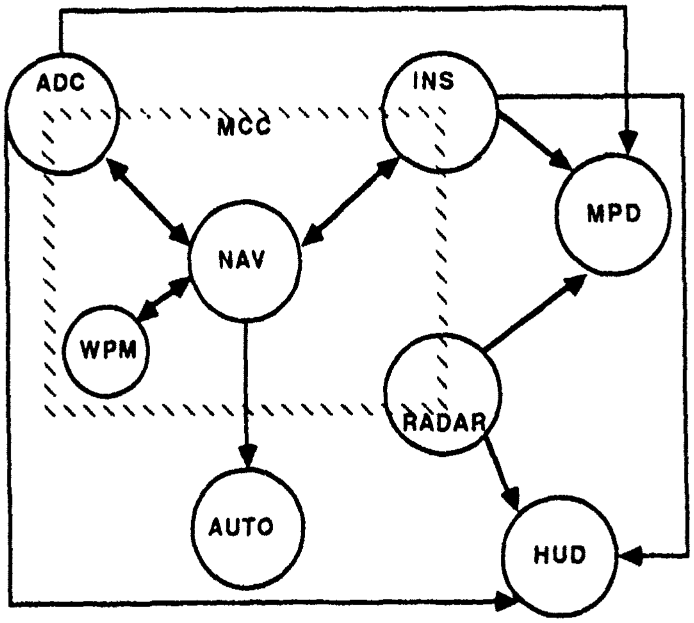
5.2 Problem Statement

The following sample problem is taken from the text of an example system provided to the SEI [12] — a specification of a generalized avionics system. Rather than specify the entire avionics system, we have chosen the parts of it. **The text that follows (Subsections 5.2.1 and 5.2.2) is a direct copy of the requirements document, we have not altered the text in any way.** We have, however, omitted some devices that were part of the original requirements document in order to make the problem a manageable size. We have not simplified in any way the components that we chose to maintain as part of our problem.

Figure 5.1 is an informal, pictorial representation of the major components and information flows of the system that we have produced as an aid to understanding the avionics system. The arrows in the diagram represent information flow and the shaded box represents the function of the mission control computer. As can be seen, the mission control computer can, at times, perform functions of the physical devices.

5.2.1 Navigation

The mission control computer (MCC) shall utilize the Air Data Computer (ADC) and the Inertial Navigation System (INS) equipment to determine the aircraft's position at periodic intervals throughout the flight. The required precision of this position will be 100 feet at all times. When the INS is operating, the accuracy shall be at least 100 feet, which shall be further maintained in the event of the loss of the INS, assuming that the computed wind velocity does not change when the INS is not operating. The MCC shall compute the aircraft position based on air mass dead reckoning when the INS is inoperative, using the last computed wind velocity. In addition to position, the navigation function shall compute the following parameters for use by any other functions requiring them.



ADC	Air Data Computer
AUTO	Autopilot
HUD	Heads Up Display
INS	Inertial Navigation System
MCC	Mission Control Computer
MPD	Multi-Purpose Display
NAV	Navigation Function
RADAR	Radar Device
WPM	WayPoint Manager

Figure 5.1: Avionics System to Be Specified

- Ground track angle (i.e., True course)
- Ground speed
- True heading
- True airspeed
- Altitude
- Magnetic variation
- Lat/Long
- Aircraft-target geometry
- Wind course and speed.

To achieve this accuracy, the position must be computed at least every 59 milliseconds, based on an upper aircraft velocity bound of 1000 kt.

Associated with the navigation requirement is the management of waypoints, which are pre-planned ground locations defining the route for the flight for the current mission. Up to 20 waypoints, assumed to have been inserted into the MCC during mission initialization, are managed by the MCC, which provides steering commands to the pilot and autopilot sufficient to permit the aircraft to cross the waypoint within a radius of 200 feet. Certain of the waypoints may have been designated as requiring avoidance; the pilot and autopilot steering commands, while flying between the other waypoints, shall cause the pilot to avoid these by at least 30 miles. The pilot steering commands must be updated on the Pilot's Horizontal Situation Indicator (HSI) and to the autopilot at least every 100ms.

In addition, the current status of the navigation equipment, including both the ADC and the INS, shall be continuously monitored, and shall be updated on the pilot's Multi-Purpose Display (MPD) at a 1hz. rate, within .25

sec. When there is a change in status, an indicator on the Heads Up Display (HUD) shall be caused to blink until the next switch is depressed by the pilot, with a minimum of 2 seconds.

5.2.2 Radar Control

The Pilot shall use the keyset to start and stop the radar display on the MPD and HUD. Radar target information shall be updated whenever it is reported by the radar, and shall be maintained by the MCC when not reported by the radar, assuming that its last computed course and speed has not changed. Once a velocity has been established, the position of each target shall be updated at least every 200 ms. by the MCC.

Chapter 6

Communicating Sequential Processes

This chapter of the survey discusses a specification of the previously described avionics system using Communicating Sequential Processes (CSP), as defined by Hoare in his book of that title [7].

The system components were specified separately both in terms of constraints on the observable events (traces) in the component's behavior and in terms of a CSP model that exhibits the desired behavior. For a number of the components, we have modeled more than one level of the component, elaborating on the initial specification to derive a more detailed specification of the system.

CSP operators

We use bold font in the following descriptions when using an English word that has a specific meaning in CSP. This meaning may often be the same as that of English.

The usual logical operators **and** ($\wedge$), **or** ($\vee$), **equivalence** ($\equiv$), **implication** ($\Rightarrow$) and **universal quantification** ($\forall$) are used throughout the specification.

Set operations used in the specification are **membership** ($\in$) and **distributed union** ($\bigcup$).

The following are the CSP trace operations used in the specification.

$< ... >$ — The trace comprising the named events.

$s\hat{}t$ — The **concatenation** of two traces, that is, the events of s followed by the events of t.

s^n — The trace s **repeated** n times. This repetition is in the form of s, followed by s, and so on.

$s \uparrow A$ — The trace s **restricted** to the elements of the set A.

$s \leq t$ — The trace s is a **prefix** of the trace t.

$\#s$ — The **length** of the trace s.

$s[i]$ — The i^{th} element of the trace s.

The following are the CSP process operations used in this specification.

$a \rightarrow P$ — The event a **then** process P.

$a \rightarrow P \mid b \rightarrow Q$ — Means (a then P) **choice** (b then Q) as long as $a \neq b$.

$\mu X : A.F(X)$ — The process X which has alphabet A where $X = F(X)$.

$P \parallel Q$ — Process P in **parallel** with process Q.

$l : P$ — **Relabels** the events of process P by prefixing each event name with l and a point (.).

$P \sqcap Q$ — Process P **non-deterministic or** process Q.

$P \setminus C$ — Process P **without** the events of C. This is the method of **hiding** events.

$P; Q$ — Process Q follows process P's successful completion.

6.1 Specification

We have not attempted to keep a uniform style of use of CSP throughout our specification of the problem example, but have instead used this as an opportunity to experiment with specification style. Obviously, in a real development project, there would be some pressure to maintain a consistency of style.

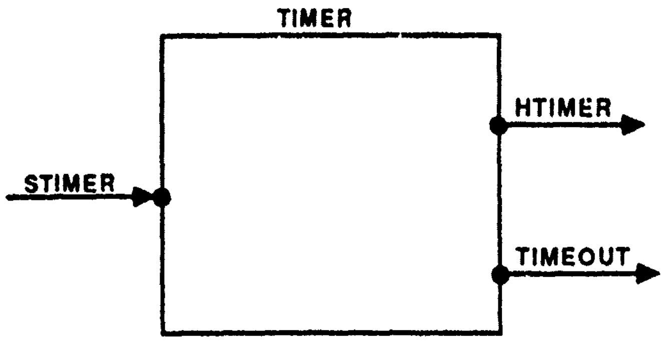


Figure 6.1: Timer Process

6.1.1 Handling Time

One of the important, recurring themes of this specification is that of a timer. The timer acts as a "guard" around some event. The intent is that either the event will occur within a given period of time, or a special timer expiry event will occur within that same period of time. For example, we might send a message to some process and set a timer such that either the process responds or the timer expires.

We will define a generalized TIMER process that is initialized and then is either explicitly terminated or produces some timeout event. Figure 6.1 is a picture of our timer process.

Therefore, the alphabet of TIMER is:

stimer — The event that explicitly starts the timer.

htimer — The event that explicitly stops the timer.

timeout — The event indicating that the timer is expiring.

Therefore, with our definition of the timer and the alphabet, we can constrain the possible sequences of events in which the timer engages by the following. If s is any trace of TIMER, then

$$\forall i. i > 0 \Rightarrow (s[i] = \mathit{htimer} \vee s[i] = \mathit{timeout}) \equiv s[i-1] = \mathit{stimer}$$

This specification states that if either an *htimer* or *timeout* event occurs, it was preceded by a *stimer* event, and that every other event of the timer will be the *stimer* event.

The following specification of a generalized timer can be shown to satisfy the above constraint.

$$TIMER = stimer \rightarrow (htimer \rightarrow TIMER \mid timeout \rightarrow TIMER)$$

We will use the TIMER process throughout the specification and will distinguish between different timers by labeling the process.

The timer still lacks a notion of the length of time that should elapse between the *stimer* and *timeout* events (assuming no intervening *htimer* event). As a notational convenience, we will use the process label to indicate the length of time to wait before the timer process should generate an interrupt.

6.1.2 The ADC and the INS

On examination, both the air data computer (ADC) and the inertial navigation system (INS) have essentially the same interface to the outside world. They both provide data to the navigation function (NAV) and, should they fail, status information is displayed to the pilot. Because of this similarity, we will specify a generalized process that has the necessary interface and then label the process appropriately.

Thus, we will specify a physical device (PD) that responds to requests for data with appropriate data. Note that we will not be discussing the actual content of the data or even its structure, we are simply interested in its existence. Should PD fail, a software function in the mission control computer (MCC) will be invoked to estimate data values based upon certain assumptions. We have also assumed that PD requires an explicit initialization event, without which nothing will happen. This corresponds to, say, the pilot's information check where if on start-up a device is shown to have failed, the mission would be aborted.

Further, we only consider a single mission (which with the additional assumption that we do not care about the termination of the system, corresponds to the system's being switched off after landing). This assumption simply means that we treat each mission as identical, in terms of the actions of the software system, though of course many other factors will cause variances in the mission (for example. different flight plans or mission goals).

A Perfect Device

The perfect interface between PD and NAV would be one where every request from NAV is responded to with appropriate data by PD. Thus, we will specify this interface and assume that any subsequent specification will be either a refinement or an elaboration of this specification.

We now have the following alphabet for PD:

start — The explicit initialization event.

req — The request for data from some other process.

data — The data produced by PD.

PD, having been initialized, responds to every request for data with some data values. Once a request has been accepted, it will not accept any more requests until the data has been produced; similarly, it will not produce any data unless requested. The request/data cycle repeats indefinitely (in fact, until the system is terminated). Further, we would like to specify that PD always responds within a given time; to ensure this, we start a timer after the request has been accepted and require that the data be produced before the timer expires. We will, in fact, explicitly halt the timer.

Given the above assumptions, we can state that if s is any trace of PD, then

$$s \leq <start> ^\frown <req, max.stimer, data, max.htimer>^*$$

This states that all behaviors of PD begin with a *start* event and then continue with a repeating sequence of *req*, *max.stimer*, *data*, and *max.htimer* events. If the timer should expire, the system will be in error and will halt.

As well as the specification of the traces of events, we can write a process that engages in the same sequence of events.

$$\begin{aligned} TPD &= start \rightarrow PDOP \\ PDOP &= req \rightarrow max.stimer \rightarrow (data \rightarrow max.htimer \rightarrow PDOP \\ &\qquad\qquad\qquad\qquad\qquad\;\; \mid \; max.timeout \rightarrow STOP) \end{aligned}$$

This states that TPD engages in a *start* event and then behaves like the process PDOP. And that PDOP engages in a *req* event, starts a timer, and then, if data is received (the *data* event) the timer is halted. Otherwise, the timer will expire and TPD is itself halted since it has failed to meet its specification.

Finally, we may construct the perfect device by using

$$PD = (TPD \parallel max : TIMER) \setminus \{max.stimer, max.htimer, max.timeout\}$$

A Physical Device

The physical device is similar to the ideal device we specified previously, in that it accepts requests for data, and having done so returns the measured data values. However, we will consider the possibility that the device may fail, in which case a request may be accepted, but no data will be returned. A possibility we will not consider is that the device generates data without a prior request.

We have the following alphabet for the actual physical device, APD:

preq — The request for data.

pdata — The physical device returning data values.

Given the above alphabet and the description of the physical device, we may write a constraint on the possible behaviors of APD describing the desired set of behaviors. If s is any trace of APD, then

$$\forall i.i > 0 \wedge s[i] = pdata \Rightarrow s[i-1] = preq$$

This specification states that every occurrence of the *pdata* event is preceded by at least one *preq* event.

This leads us to the following process specification, which satisfies the above constraint.

$$APD = preq \rightarrow APD \sqcap preq \rightarrow pdata \rightarrow APD$$

The above process is simulating the situation where a device accepts a request for data (*preq*) and then either does nothing or returns the appropriate data (*pdata*). Having made this choice, the process repeats itself. Non-determinism is used to indicate that the behavior is affected by some external event (in this case, the device breaking).

A Reliable Device

The specification of PD is one in which every component works correctly all the time; that is, the physical device always responds to every request. We must also consider the more realistic situation where the physical device may accept requests and not produce any data, or where it will not even accept requests. In these latter cases, some software function will be invoked which will generate appropriate data for subsequent transmission to the requester. Figure 6.2 illustrates the completed reliable device, including the components of the physical device and the timer.

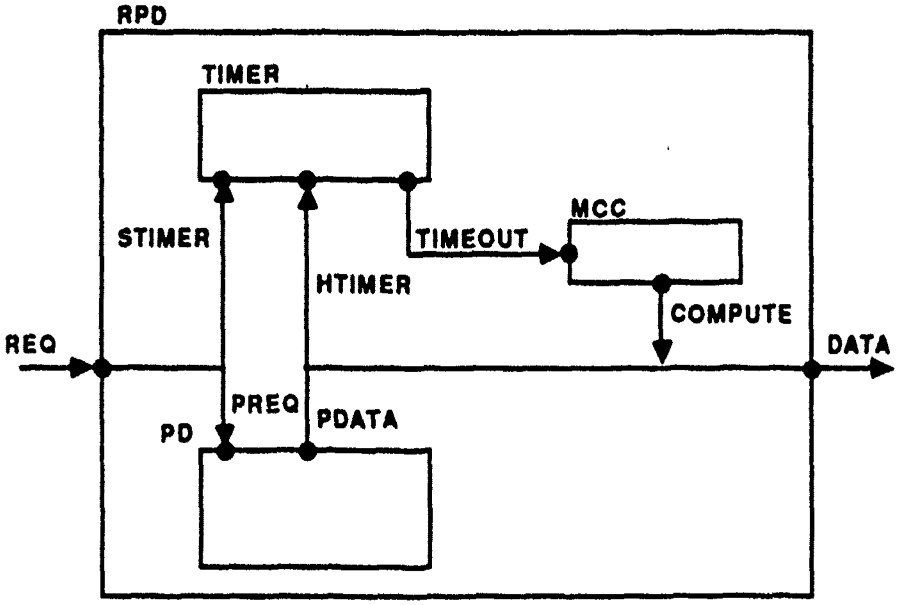


Figure 6.2: Reliable Device

The alphabet used for this process will be that of the ideal device, the timer labeled with a suitable timeout value, and the real physical device. We will also add the following additional value:

compute — The event of computing data when the physical device has failed. This event will also be the signal to the pilot's display indicating that the device failed.

We will also extend the meaning of the *pdata* event to indicate to the pilot's display that the device has behaved normally. We can now express the constraints on the behavior of th≈ reliable device, RD.

Before the timer is started, a request must have been made of the physical device for data (*preq*), and this request must have been preceded by an external request for data, where *mdev* is the label that will be applied to the TIMER process.

$$\forall i.i > 0 \wedge s[i] = mdev.stimer \Rightarrow s[i - i] = preq \wedge s[i - 2] = req$$

If the software has to compute some data values, then the physical device has failed to respond with some data and the timer has expired.

$$\forall i.i > 0 \wedge s[i] = compute \Rightarrow s[i - 1] = mdev.timeout$$

If the physical device returns data before the timer has expired, then the timer should be terminated.

$$\forall i.i > 0 \wedge s[i] = mdev.htimer \Rightarrow s[i-1] = pdata$$

After the timer has been started, the reliable physical device will do nothing until either the physical device returns some data, or the timer expires.

$$\forall i.i > 0 \wedge (s[i] = pdata \vee s[i] = mdev.timeout) \Rightarrow s[i-1] = mdev.stimer$$

If the reliable device produces data, then either the data was computed (since the physical device failed to respond) or the data returned is that of the physical device.

$$\forall i.i > 0 \wedge s[i] = data \Rightarrow s[i-1] = mdev.htimer \vee s[i-1] = compute$$

Given these constraints, which describe a reliable physical device, we can now construct a process that satisfies the requirements: that is, a combination of both software and hardware functions that, when combined, produce a result on demand, and such that if the hardware device should fail, will compute appropriate values.

$$\begin{aligned} RPD \;=\;& req \rightarrow preq \rightarrow mdev.stimer \rightarrow \\ & (pdata \rightarrow mdev.htimer \rightarrow data \rightarrow RPD \\ & \mid mdev.timeout \rightarrow compute \rightarrow data \rightarrow RPD) \end{aligned}$$

The above process, RPD, describes the behavior of the reliable physical device, which accepts a request for data that is passed on to the physical device. A timer is started so that if the physical device fails to respond, due either to its being slow or having failed, the entire system will not wait on the failed device. If the physical device responds in time, then the timer is halted and the data is returned to the requester. Otherwise, the timer will expire, at which point the software will compute values, based on characteristics of the device and previous values, and then this computed data will be returned to the requester.

We described a process, PDOP, in our specification of the perfect system that may be constructed from the processes we have just specified. A truly perfect device would not require any timing events, so our construction will hide such events.

$$\begin{aligned} PDOP \;\backslash\; & \{max.stimer, max.htimer, max.timeout\} \\ =\; & (RPD \parallel APD \parallel mdev : TIMER) \setminus \\ & \{mdev.stimer, mdev.htimer, mdev.timeout, compute, preq, pdata\} \end{aligned}$$

The above equation states that PDOP with no timer events behaves identically to the parallel combination of the RPD, APD, and TIMER (suitably relabeled) processes when the named list of events is hidden. In fact, the only events visible in each of the above processes are *req* and *data*.

Completed Devices

The final stage of the specification of the ADC and the INS is to use a process labeling of the generalized physical device specification to specify the different devices.

Therefore,

$$\begin{aligned} ADC &= (start \rightarrow adc : RPD \parallel mdev : TIMER) \\ &\setminus \{mdev.stimer, mdev.htimer, mdev.timeout\} \\ INS &= (start \rightarrow ins : RPD \parallel mdev : TIMER) \\ &\setminus \{mdev.stimer, mdev.htimer, mdev.timeout\} \end{aligned}$$

From the above, it can be seen that both INS and ADC are specifications of reliable devices, which engage in the same start event, and then perform as the reliable physical devices previously specified.

6.1.3 The Radar

We will consider the radar to be a simple device that provides data on request. The output of the radar will be data indicating the targets visible at the time. Unlike the devices ADC and INS, the radar will not be specified as a perfect device, and then mimicked by software. Rather, for the radar we will define a device that may respond to a request with data, but if it does not respond, it will not accept any further requests for data. We will use the description of the radar display to specify actions in the event of a failure.

Let the alphabet of the radar be:

rreq — A request for radar data.

rdata — The event of returning data on the targets currently visible to the radar.

With these events, we may state that if s is any trace of the radar (RAD) then

$$s \le <rreq, rdata>^*$$

Simply stated: the radar will engage in an alternating sequence of requests and responses, starting with a request for data.

The above constraint may be simply satisfied by the following process.

$$RAD = rreq \rightarrow rdata \rightarrow RAD$$

6.1.4 The MPD and HUD

In the specification of the ADC and the INS, we made statements concerning the fact that a change of status in either device would be displayed to the pilot. This section describes the heads up display (HUD) ɿnd the multi-purpose display (MPD) with regard to the device status information.

As we did for the ADC and INS, we will specify a generalized process that may be relabeled as appropriate.

Recall also that the pilot's displays also engage in the *compute* and *pdata* events, in that these are used as the continuous monitoring of the devices as described in the requirements.

Generic MPD Device Status Display

First, we will consider the MPD, which simply displays the status of the device.

Let the alphabet for MPD be:

compute — The previously described event of the MCC having to perform a computation due to the device's not responding to a request for data.

pdata — The previously described device event producing some data due to a request.

dbroken — An event associated with changing the MPD to indicate that the device is broken.

dwork — An event associated with changing the MPD to indicate that the device is working.

We cannot describe the actual display with CSP, so we will assume that the *dbroken* and *dwork* events display appropriate messages at appropriate locations on the MPD. We will also assume that these messages are persistent and do not need a command to indicate that they should be redisplayed.

From these assumptions, we can state that the change of MPD to indicate that the device is broken must be preceded by the event indicating that the device is broken.

$$\forall i. i > 0 \wedge s[i] = dbroken \Rightarrow s[i-1] = compute$$

Similarly, every time the MPD is changed to indicate that the device is working it must have been preceded by some data from the device.

$$\forall i. i > 0 \wedge s[i] = dwork \Rightarrow s[i-1] = pdata$$

The above constraints do not forbid the redisplay of, say, a "device broken" icon every time the *compute* event occurs. We could do this by requiring that if the i^{th} event were *dbroken* then the $(i-2)^{th}$ event be a *pdata* event. However, we do not consider it an error to attempt to redisplay the message so long as the correct message is being displayed. So we will not add further constraints.

A process that satisfies our constraints is

$$\begin{aligned} MPD &= dwork \rightarrow MPDWD \\ MPDWD &= pdata \rightarrow MPDWD \mid compute \rightarrow dbroken \rightarrow MPDBD \\ MPDBD &= compute \rightarrow MPDBD \mid pdata \rightarrow dwork \rightarrow MPDWD \end{aligned}$$

This states the assumption that the device starts in a correctly functioning state and that such a message is displayed on the MPD. Then as long as the device behaves correctly, the display does not change. If the device fails, as signaled by the MCC having to compute some data, the message is changed to indicate the device failure, and the system remains in this state for as long as the MCC has to continue to compute data. If the device should start operating again, the display is altered and our system reverts to its previous state.

MPD for INS and ADC

We may now specify the ADC and INS status displays on the MPD by:

$$\begin{aligned} MPDADC &= adc : MPD \\ MPDINS &= ins : MPD \end{aligned}$$

This labels the generalized MPD system appropriately. This labeling also matches that used by ADC and INS, thus ensuring that the *compute* and *pdata* communications occur appropriately.

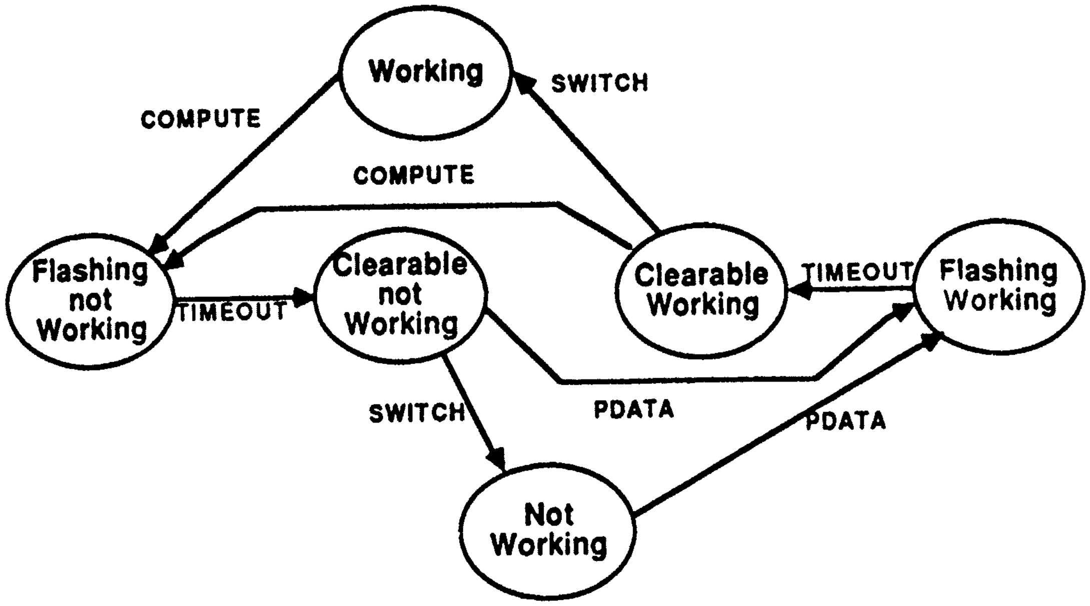


Figure 6.3: Behavior of the HUD

Generic HUD Device Status Display

The heads up display (HUD) should display the same sort of status information as the MPD; however, to ensure that the pilot sees the information, it should to flash for a period of time. Figure 6.3 illustrates a state machine-like representation of the behavior of the HUD when switched on. To avoid confusion, only the major transitions are displayed.

As with the MPD, the devices will signal their state by means of the *compute* and *pdata* events. The HUD will also use a timer to ensure that the flashing message stays on the screen for at least two seconds. In addition to these events, are the following events:

switch — A switch that the pilot may depress; it signals the pilot's intent to clear the HUD.

hclear — The action of clearing the HUD status information.

hbroken — The writing of the message indicating that the device is broken.

hwork — The writing of the message indicating that the device is working.

The following are constraints on the manner in which the HUD can behave.

If the HUD displays the message indicating that the device is working, then it must have previously received an indication that the device is working.

$$\forall i. i > 0 \wedge s[i] = hwork \Rightarrow s[i-1] = pdata$$

Similarly, if the HUD displays a message indicating that the device has failed, then it must have just received an indication that the device is broken.

$$\forall i. i > 0 \wedge s[i] = hbroken \Rightarrow s[i-1] = compute$$

The HUD display can only be cleared after a switch has been depressed.

$$\forall i. i > 0 \wedge s[i] = hclear \Rightarrow s[i-1] \; switch$$

If the HUD displays a message, then the timer should be started to enforce the requirement of messages flashing for at least 2 seconds.

$$\forall i. i > 0 \wedge s[i] = hudd.stimer \Rightarrow (s[i-1] = hbroken \vee s[i-1] = hwork)$$

The timer in use by the HUD is never stopped explicitly, but always generates a timeout.

$$\begin{aligned} traces(HUD) \uparrow \{hudd.stimer, hudd.htimer, hudd.timeout\} \\ = \bigcup hudd.stimer, hudd.timeout^{*} \end{aligned}$$

We can describe a collection of processes that describe the desired behavior of the HUD. As for the MPD, we cannot describe the form of the data on the display, merely the event of writing to the screen.

$$\begin{aligned}
HUDWO &= switch \rightarrow HUDWO \mid pdata \rightarrow HUDWO \\
&\mid compute \rightarrow hbroken \rightarrow hudd.stimer \rightarrow HUDBF \\
HUDBF &= compute \rightarrow HUDBF \mid pdata \rightarrow HUDBF \\
&\mid switch \rightarrow HUDBF \mid hudd.timeout \rightarrow HUDBC \\
HUDBC &= compute \rightarrow HUDBC \mid switch \rightarrow hclear \rightarrow HUDBR \\
&\mid pdata \rightarrow hwork \rightarrow hudd.stimer \rightarrow HUDWF \\
HUDBR &= switch \rightarrow HUDBR \mid compute \rightarrow HUDBR \\
&\mid pdata \rightarrow hwork \rightarrow hudd.stimer \rightarrow HUDWF \\
HUDWF &= compute \rightarrow HUDWF \mid pdata \rightarrow HUDWF \\
&\mid switch \rightarrow HUDWF \mid hudd.timeout \rightarrow HUDWC \\
HUDWC &= pdata \rightarrow HUDWC \mid switch \rightarrow hclear \rightarrow HUDWO \\
&\mid compute \rightarrow hbroken \rightarrow hudd.stimer \rightarrow HUDBF
\end{aligned}$$

The above sequence of processes describes a process which, on a change of state in the physical device, starts to flash an appropriate message on the HUD. The message is that the device is now working (if previously it was broken) or that it is now broken. This message will continue to flash until (1) a timer has expired and (2) the pilot presses some switch. There is one additional case where the message can change: when the HUD is displaying some flashing message and the timer has expired, but before the pilot has pressed an appropriate switch to clear the message, the device changes state. If this occurs, then the message will be changed from *broken* to *working* (or vice versa) and a new timer will be started. This sort of situation can occur if the device has a failure and immediately restarts itself, in which case the message will flash *broken* for two seconds, and then flash *working* for two seconds before the pilot can clear the HUD.

HUD for ADC and INS

As for the MPD, we may now describe the different messages displayed on the HUD by a relabeling of the processes above.

$$\begin{aligned} HUDADC &= adc : HUDWO \parallel hudd : TIMER \setminus \{timeout\} \\ HUDINS &= ins : HUDWO \parallel hudd : TIMER \setminus \{timeout\} \end{aligned}$$

These two processes describe the manner in which the messages about the status of the INS and ADC are displayed on the HUD.

A Special Clock Process

Let us assume that the computation of radar data (in the event that the radar is not working) takes less than c ms. We will introduce a new timing process that emits a pulse($p1$) $200 - c$ ms. after it has been initiated (event pn) and a second pulse($p2$) a further c ms. later, which is, of course, 200 ms. after being initiated. If s is any trace of the double pulse process, DTP, then

$$s \leq <pn, p1, p2>^{*}$$

A process that satisfies the above constraints may be constructed using two timers, as in the following description.

$$\begin{aligned} DT \;=\; & pn \rightarrow mnc.stimer \rightarrow mnc.timeout \rightarrow p1 \rightarrow mc.stimer \\ & \rightarrow mc.timeout \rightarrow p2 \rightarrow DT \end{aligned}$$

Then, to get the desired behavior, we use

$$\begin{aligned} DTP \;=\; & (DT \parallel mnc : TIMER \parallel mc : TIMER) \\ \setminus \; & \{mnc.stimer, mnc.timeout, mc.stimer, mc.timeout\} \end{aligned}$$

The Radar Display

The requirements presented in Chapter 4 state that the radar information will be displayed on both the MPD and the HUD and make no distinction between these two devices. We will therefore describe a general display process, and intend that both the HUD and the MPD may be described by a relabeling of this process. The difference between the two will be in the manner in which the information is displayed, which cannot be described in a convenient manner using CSP.

The pilot controls the radar display using some keyswitch, and we will assume that when the radar display is switched off that the display will be cleared. To ensure that everything happens in a timely manner, we will use the clock process just described.

In addition to the alphabets of the radar device previously described (*rreq* and *rdata*) and the special clock(*p1*, *p2*, and *pn*), these are the following events in the alphabet of the display:

ron — The pilot turning on the radar display.

roff — The pilot turning off the radar display.

rdisp — The act of displaying the current data on the device.

rclear — The act of clearing the displayed radar data when the display is turned off by the pilot.

rcomp — The computation of current data if the radar did not respond in time.

The pulse device is started after a request has been made for data from the radar.

$$\forall i.i > 0 \wedge s[i] = pn \Rightarrow s[i-1] = rreq$$

In order to ensure that data is displayed every 200 ms., the display will always wait for the second pulse before displaying the data (the double timer will be initiated at the start of the cycle). Note that we are assuming that it takes no time to display the data; if the time for display is significant, then the second timer pulse *p2* should occur sooner than 200 ms. so that the interval $p2$ plus the length of time to display data is a totaı of 200 ms. This change would be made by altering the intervals of the clock process *DTP*.

$$\forall i.i > 0 \land s[i] = rdisp \Rightarrow s[i-1] = p2$$

If the system has to compute the data, then the radar device has failed to produce data before the first pulse occurred.

$$\forall i.i > 0 \land s[i] = rcomp \Rightarrow s[i-1] = p1$$

Alternatively, the system may have received the second pulse directly after the first pulse, in which case data must have been returned from the radar device before the first pulse.

$$\forall i.i > 1 \land s[i] = p2 \land s[i-1] = p1 \Rightarrow s[i-2] = rdata$$

The following process describes the state when the system is switched off; the process will ignore every signal except the *on* signal, at which point it will behave as *RADON*.

$$\begin{aligned} RADOFF &= rdata \rightarrow RADOFF \mid p1 \rightarrow RADOFF \mid p2 \rightarrow RADOFF \\ &\mid \; ron \rightarrow RADON \mid roff \rightarrow RADOFF \end{aligned}$$

Finally, we may describe a process RADON which satisfies the above constraints.

$$\begin{aligned} RADON &= roff \rightarrow rclear \rightarrow RADOFF \\ &\mid \; rreq \rightarrow pn \rightarrow \\ &\quad\; (rdata \rightarrow p1 \rightarrow p2 \rightarrow rdisp \rightarrow RADON \\ &\quad\; \mid p1 \rightarrow rcomp \rightarrow p2 \rightarrow rdisp \rightarrow RADON) \end{aligned}$$

This leads to a final specification, assuming the radar is initially off, for the radar display device of

$$\begin{aligned} RADISP &= (RAD \parallel DTP \parallel RADOFF) \\ &\setminus \; \{p1, p2, pn, rreq, rdata, rcomp\} \end{aligned}$$

Final Description of HUD and MPD

We have now specified all of the components of the HUD and the MPD, in that we have descriptions of how the INS and ADC and radar interact with the display devices. Putting these specifications together, we arrive at the following specifications of HUD and MPD.

$$\begin{aligned} HUDISP &= HUDADC \parallel HUDINS \parallel hud : RADISP \\ MPDISP &= MPDADC \parallel MPDINS \parallel mpd : RADISP \end{aligned}$$

6.1.5 The Waypoint Manager

The waypoint manager maintains a number of waypoints which are used to guide the aircraft through a flight. Each waypoint has an associated indication as to whether the aircraft must arrive within 200 feet of the point or must avoid it by at least 30 miles. The initialization of the waypoint manager is of no concern to this specification, and we will therefore assume that it has been achieved at some stage during the initialization of the aircraft.

We will assume that each time the aircraft has successfully navigated towards a waypoint, it will ask for the next waypoint. As stated in the requirements (Chapter 4), there are a maximum of 20 waypoints that will be entered into the system. We will assume that within these 20 waypoints there is one that terminates the mission, which is to say that it is an appropriate runway on which the aircraft may land. Using this latter assumption, we will specify the waypoint manager with limited regard to the maximum number of waypoints.

We will use the following alphabet for the waypoint manager, WPM.

start — The event of initializing the waypoint manager.

wreq — A request for the next waypoint in the sequence.

wdata — The production of the pair of data values, the waypoint, and whether or not it must be avoided.

Given these assumptions and the listed events, we can specify the waypoint manager. Let us assume that s is any trace of WPM; then we have

$$s \leq <start> \frown <wreq, wdata>^{20}$$

This states that the any non-empty trace of the waypoint manager will begin with the *start* event, initializing the system, and will then continue with an alternating sequence of up to twenty requests, *wreq*, and responses with data, *wdata*.

A process that may be used to implement the above specification is

$$WPM = start \rightarrow \mu X : \{wreq, wdata\}.(wreq \rightarrow wdata \rightarrow X)$$

Although in general the process does not satisfy the specification, since the process describes behaviors with more than 20 iterations of the request/provide data cycle, the assumption that we made implies that there will be no more than 20 requests of the WPM process.

6.1.6 The Autopilot

For the purposes of this specification, we will assume that the autopilot is a simple device that accepts commands from the navigation unit and translates these into appropriate movements of the rudder, flaps, and other devices for steering the plane.

The autopilot will have the following alphabet:

course — The transfer of data from the navigation unit giving directions such as course, altitude, velocity, etc.

comms — The autopilot emitting appropriate commands to the physical devices for controlling the plane's course.

We will assume that the autopilot will always be in a position to accept commands, though it may not respond appropriately (by emitting control commands). However, we will state that the autopilot will not emit control commands unless it has been previously instructed to do so by the navigation unit. So, if s is any trace of the autopilot (AUTO), then:

$$\forall i.i > 0 \land s[i] = comms \Rightarrow s[i-1] = course$$

This simply states that every time controlling commands are emitted (*comms*), the event is preceded by at least one instruction on course (*course*).

A process that satisfies this constraint is

$$AUTO = course \rightarrow comms \rightarrow AUTO \sqcap course \rightarrow AUTO$$

6.1.7 The Navigation Function

The purpose of the navigation unit (NAV) is to control the other devices as appropriate and, using some function of navigation, emit commands to the pilot and the autopilot.

Timing Control

The requirements state that the position of the aircraft should be updated at least every 59 ms. In order to do this, we introduce a clock that will emit a pulse at this frequency.

Let the alphabet of the clock include a timer and the following event:

pulse — A clock tick to be emitted every 59 ms.

The clock will use the timer by starting a 59 ms. timer and then waiting for the timer to expire, at which point our clock will emit its pulse event (*pulse*).

The clock will not be stopped at any time, so if s is a trace of the clock, GP, then

$$s \leq <m59.stimer, m59.timeout, pulse>^*$$

A process that satisfies this statement:

$$GP = m59.stimer \rightarrow m59.timeout \rightarrow pulse \rightarrow GP$$

However, we are only interested in the clock ticks. For this, we would like s, any trace of CONTROL, to satisfy

$$s \leq <pulse>^*$$

This may be constructed as follows:

$$CONTROL = (GP \parallel m59 : TIMER) \setminus \{m59.stimer, m59.timeout\}$$

Navigation function

The navigation function becomes a simple loop in which data is accepted from the devices and a computation is performed to instruct the pilot and autopilot; then the loop repeats itself. This may be constructed as follows: There is no requirement stating behavior in the event that the navigation function should fail, so we assume that the pilot will navigate by non-mechanical means — for example, steering by whatever landmarks are available.

The alphabet for NAV is:

atwayp — An event representing the determination as to whether the aircraft has reached its current waypoint and therefore needs to start flying towards the next waypoint in the predetermined flight plan.

ncomp — The event that performs some function based upon available data and computes the appropriate course commands.

course — The event that displays on the pilot display and to the autopilot the appropriately computed course.

pulse — The 59 ms. clock tick.

By the time a clock pulse *pulse* is emitted, the navigation function must have transmitted the course commands just computed.

$$\forall i.i > 1 \wedge s[i] = pulse \Rightarrow s[i-1] = course \wedge s[i-2] = ncomp$$

If the navigation unit detects that a waypoint has been reached, it will make a request for the next waypoint from the waypoint manager, which will respond with the appropriate data.

$$\forall i.i > 1 \wedge s[i] = wdata \Rightarrow s[i-1] = wreq \wedge s[i-2] = atwayp$$

To calculate the new course, NAV must have determined whether or not the aircraft has reached a waypoint, and if it has, a new waypoint must be selected before computation can occur.

$$\forall i.i > 0 \wedge s[i] = ncomp \Rightarrow s[i-1] = atwayp \vee s[i-1] = wdata$$

Before determining whether the aircraft has reached a waypoint, its position must be known. Further, the current air data should be known before attempting to compute the new course.

$$\begin{aligned} \forall i.i > 4 \wedge s[i] = atwayp \quad &\Rightarrow \quad \exists p.(\exists q.(\\ & \quad\quad p\,\hat{}\,q\,\hat{}\,<atwayp>\leq s \end{aligned}$$

$$\begin{aligned} \wedge\ & \#(p\hat{\ }q) = i \\ \wedge\ & \#q = 4 \\ \wedge\ & q \in traces(adc.req \rightarrow adc.data \rightarrow STOP \\ & \quad \| \ ins.req \rightarrow ins.data \rightarrow STOP))) \end{aligned}$$

Note that the length of $p\hat{\ }q$ is i since sequence indexing considers the first element to be indexed by 0 and # provides the length of the trace. An appropriate process description is:

$$\begin{aligned} NAV &= (adc.req \rightarrow adc.data \rightarrow STOP \ \| \ ins.req \rightarrow ins.data \rightarrow STOP) \\ &\ ; \ (atwayp \rightarrow wreq \rightarrow wdata \rightarrow DIR \\ &\ \sqcap \ atwayp \rightarrow DIR) \\ DIR &= ncomp \rightarrow course \rightarrow pulse \rightarrow NAV \end{aligned}$$

It should be noted that we have used a non-deterministic choice between *ncomp* events. What we are trying to indicate is that an observer of the system will not be able to determine why one path or the other is taken simply by looking at the events that occur. However, we believe that it is sufficiently clear that the choice will be made to engage in *wreq* and *wdata* events only when the aircraft has reached its current waypoint.

6.1.8 The System

Having described each of the components of the problem, we may now describe the entire system by composing the component process specifications.

Thus,

$$\begin{aligned} SYSTEM &= MPDISP \ \| \ HUDISP \ \| \ NAV \ \| \ CONTROL \\ &\ \| \ WPM \ \| \ ADC \ \| \ INS \ \| \ RAD \ \| \ AUTO \end{aligned}$$

6.1.9 An Alternate Waypoint Manager

During the development of the specification, we had a number of choices open to us. One such choice, examined briefly after the initial specification was written, was the introduction of an explicit termination event. We decided not to implement such an event, but as an experiment introduced explicit termination to the waypoint manager. The following specification is the result of introducing explicit termination.

We use the same alphabet as the previously described waypoint manager (WPM), but add the following event:

finish — An explicit event indicating that the flight is over and that there will be no more requests for waypoints from the manager.

Now, a correctly terminating flight will be initialized (*start*), will perform up to 20 sequences of request (*wreq*) and response (*wdata*), and will end with an explicit event (*finish*). Then, if s is any trace of WPM,

$$\exists i.i \leq 20 \wedge s \leq <start> \mathbin{\hat{}} <wreq, wdata>^{i} \mathbin{\hat{}} <finish>$$

We will now construct a process that describes the WPM. As in the previous description, the process that follows will not in general satisfy the above specification. However, based on the same assumption made in Section 6.1.5 concerning the existence of a waypoint which terminates the mission, we may describe the manager as

$$\begin{aligned} WPM = start \; \rightarrow \; & \mu X : \{wreq, wdata, finish\}. \\ & (wreq \rightarrow wdata \rightarrow X \mid finish \rightarrow STOP) \end{aligned}$$

6.1.10 More on Radar Displays

The radar display presented in the specification was not the first specification developed. In this section, we present the two alternative displays and give the reasons for their rejection.

Both of the radar displays use the same alphabet introduced for the radar display in the main specification. The first of the two alternative displays requires a special clock process, the specification of which is given here.

A Clock

The radar display on the MPD and HUD must be updated every 200 ms., and a convenient specification component is a clock that emits a pulse every 200 ms.

Let the alphabet of the clock include a timer and the following event.

pulse — A clock tick emitted every 200 ms.

Since this clock process is almost identical to the one given in the main body of the specification for the process CONTROL, we will not give the entire specification here, simply the process specification of the clock.

$$\begin{array}{rcl} CLOCK & = & \mu X : \{m200.stimer, m200.timeout, pulse\} \\ & . & (m200.stimer \rightarrow m200.timeout \\ & \rightarrow & pulse \rightarrow X) \\ & \backslash & \{m200.stimer, m200.timeout\} \end{array}$$

First Radar Display

The radar display clears only after the previous action has been the depression of the switch to turn off the display.

$$\forall i.i > 0 \wedge s[i] = rclear \Rightarrow s[i-1] = roff$$

Similarly, when the display is updated, either the data has been generated by the radar device or has been computed because the radar failed to respond in the appropriate time.

$$\forall i.i > 0 \wedge s[i] = rdisp \Rightarrow s[i-1] = rdata \vee s[i-1] = rcomp$$

Also, if the system has to compute the radar data, it means that the radar device has not responded to a request before the clock has emitted another pulse.

$$\forall i.i > 0 \wedge s[i] = rcomp \Rightarrow s[i-1] = pulse$$

The radar will not randomly produce data, so if the radar display receives some data, there must previously have been a request for that data.

$$\forall i.i > 0 \wedge s[i] = rdata \Rightarrow s[i-1] = rreq$$

Similarly, the radar display process only makes a request of the radar device after one of the pulses has been received.

$$\forall i.i > 0 \wedge s[i] = rreq \Rightarrow s[i-1] = pulse$$

When the radar is switched off, the only action of interest to the display process is the switching on of the radar, so the behavior of the radar in the off position may be described by the following process.

$$\begin{array}{rcl} RADOFF & = & rdata \rightarrow RADOFF \mid pulse \rightarrow RADOFF \\ & \mid & ron \rightarrow RADON \mid roff \rightarrow RADOFF \end{array}$$

This states that the system will do nothing until such time as the pilot turns on the radar display (*ron*), at which point the system will behave as the process RADON.

$$\begin{aligned} RADON &= roff \rightarrow rclear \rightarrow RADOFF \\ &\quad \mid pulse \rightarrow rreq \rightarrow \\ &\quad (rdata \rightarrow rdisp \rightarrow RADON \\ &\quad \mid pulse \rightarrow rcomp \rightarrow rdisp \rightarrow RADON) \end{aligned}$$

We may now describe the radar display device as

$$\begin{aligned} RADISP &= (CLOCK \parallel RAD \parallel RADOFF) \\ &\quad \setminus \ \{pulse, rreq, rdata, rcomp\} \end{aligned}$$

where RAD is the radar process description described in Section 6.1.3.

Unfortunately, this description does not exactly meet the stated requirements in that it is possible for the system to fail to update the display within the required 200 ms. Consider the following case. Let us assume that the radar takes r (where $r \leq 200$) ms. to respond. Also, let us assume that the computation takes c (where $c \leq 200$) ms. to respond. Then, if the radar is working, the display will be updated at r ms. after the pulse. Let us assume that the radar now fails. The system will start a request on the next pulse and will detect on the succeeding pulse that the radar has not responded, so the system will perform the computation and start displaying the computed data; this will be at $200 + c$ ms. after the initiating pulse. Let us now consider the sequence of events starting at the last pulse for which the radar was working; we will count this as time 0. Then the following table explains the sequence of events

Event Number	Relative Time	Event Explanation
1	0	pulse occurs, sequence starts
2	r	radar responds, data is displayed
3	200	pulse occurs, sequence restarts
4	400	timeout occurs, computation begins
5	$400 + c$	computation completes, data is displayed

Therefore, the total elapsed time between display updates will be $400 + c - r$ ms., since $r \leq 200$ ms. is greater than the required 200 ms.

Second Radar Display

As stated, our previous description of the radar did not meet the specification; thus, the following specification has been written so that it does redisplay the screen every 200 ms.

Let us assume that the label on the timer indicates a delay between starting the timer and the timeout of 200 − c ms., where c is the length of time it takes to compute the radar data. We will use *mnc.timer* to represent a timer that expires at this interval.

The timer is always started after a request has been made of the radar device for data.

$$\forall i.i > 0 \land s[i] = \mathit{mnc.stimer} \Rightarrow s[i-1] = \mathit{rreq}$$

If the display process has to compute data, then the radar device must have failed to produce data before the timer expired.

$$\forall i.i > 0 \land s[i] = \mathit{rcomp} \Rightarrow s[i-1] = \mathit{mnc.timeout}$$

The radar will be requested for more data whenever the display has just been updated, or the radar display has been switched on by the pilot.

$$\forall i.i > 0 \land s[i] = \mathit{rreq} \Rightarrow s[i-1] = \mathit{rdisp} \lor s[i-1] = \mathit{ron}$$

If the radar produces data in a timely fashion, then the timer should be halted.

$$\forall i.i > 0 \land s[i] = \mathit{mnc.htimer} \Rightarrow s[i-1] = \mathit{rdata}$$

Every time data is displayed, either the data has just been computed, or it was returned from the actual device (in which case the timer has just been halted).

$$\forall i.i > 0 \land s[i] = \mathit{rdisp} \Rightarrow s[i-1] = \mathit{rcomp} \lor s[i-1] = \mathit{mnc.htimer}$$

After the timer has been started, either the radar will return some data, or it will fail to do so and the timer will expire.

$$\forall i.i > 0 \land s[i] = \mathit{rdata} \lor s[i] = \mathit{mnc.timeout} \Rightarrow s[i-1] = \mathit{mnc.stimer}$$

We have the same constraint on the pilot switching off the radar, in that if the radar display is cleared, the pilot must have just operated the off switch.

$$\forall i.i > 0 \land s[i] = \mathit{rclear} \Rightarrow s[i-1] = \mathit{roff}$$

We will use a similar *RADOFF* process as that in the first specification; however, this *RADOFF* process will ignore any timer expiration events and will not receive any clock pulses.

$$\begin{aligned} RADOFF &= rdata \rightarrow RADOFF \mid mnc.timeout \rightarrow RADOFF \\ &\mid \; ron \rightarrow RADON \mid roff \rightarrow RADOFF \end{aligned}$$

With this specification of *RADOFF* and the above constraints, *RADON* becomes:

$$\begin{aligned} RADON &= roff \rightarrow rclear \rightarrow RADOFF \\ &\mid \; rreq \rightarrow mnc.stimer \rightarrow \\ &\quad (rdata \rightarrow mnc.htimer \rightarrow rdisp \rightarrow RADON \\ &\quad \mid mnc.timeout \rightarrow rcomp \rightarrow rdisp \rightarrow RADON) \end{aligned}$$

We may now describe the radar display device as:

$$\begin{aligned} RADISP &= (mnc : TIMER \parallel RAD \parallel RADOFF) \\ &\setminus \; \{rreq, rdata, rcomp, mnc.stimer, mnc.htimer, mnc.timeout\} \end{aligned}$$

The above specification satisfies the requirement that the radar display is updated at least every 200 ms. However, it should be noted that the display will in fact be updated more often than 200 ms. for as long as the radar is working. Every time the display has been updated a request will be sent immediately to the radar for more data and the display will be updated again. Again, on the assumption that the radar takes r (where $r < 200$) ms. to respond, the display will be updated every r ms., which satisfies the requirement but may not be the desired system.

6.2 Comments on the Specification

This section includes notes about the specification that are not covered by the classification criteria.

6.2.1 Uniformity of Style

Af first sight, the CSP specification appears confused, in that there are a number of occasions where two similar objects have been specified in different ways. This occurs both at the level of the specification (that is, using two different CSP constructs to specify similar processes) and at the level of the system where potentially similar components have been deliberately made into different components.

The reasons behind this lack of uniformity are that we wished to experiment with the notation and to provide numerous examples. The experimentation provided experience in using CSP on different types of components found on typical systems. Many examples of different types of use of CSP were desired to make the specification more interesting to read and to demonstrate the power of the notation.

We recognize that in a commercial development, our specification would be inappropriate and that there would be much greater emphasis on uniformity of style. We would expect that this uniformity would be achieved by concerted effort on behalf of the developers and by standards for specification style.

6.2.2 Interpretation of CSP

In order to write the specification and have it convey our intended meaning, we have used a slightly different interpretation of the meaning of the → construct of CSP. We believe that this change in interpretation does not affect the underlying formality of the technique, but simply the interpretation of what the system is expected to do. Specifically, a CSP process $\alpha \rightarrow X$ is able to perform action α and then proceed like the process X. However, the process need not do anything immediately, or even at all. Our change in interpretation is that a process able to perform an event will do so without delay. This change in interpretation permits the interpretation of the specification of the timer process to model the expected process — a component that, after some delay period, emits a *timeout* signal (the delay being caused by the length of the timer and not because the process chooses to ignore an event that it may perform). The change in interpretation does not affect the analyses that have to be performed, in that we still have to examine the possibilities of the *timeout* event's occurring before or after some other event of interest.

Further, with respect to timers, we wish the delay between the *stimer* and *timeout* events to be described by the specification assuming no intermediate *htimer* event. According to our statements above, the *timeout* event should happen immediately after the timer is started. This, however, would not suit our purpose, and instead we use a standard relabeling of the timer process to indicate the length of delay between the initiation and expiration of the timer.

There are a number of occasions where, in our specification, we have used the non-deterministic choice operator, ⊓. The semantics of CSP state that the system may make either choice, and we cannot restrict that choice. However, in our specification, we are using non-determinism to model certain behaviors of the avionics system where there is clearly a "right" choice. Typically, there is cne path that is to be chosen when the system is behaving normally and the other path is chosen only when the system is behaving abnormally, at which time the previously unchosen path becomes the "right" path. Thus, we are using non-determinism to model events over which we have no control, such as a device breaking, but our intent is for the model to be used to choose appropriate behaviors depending on whether or not the system is operating correctly.

We have often used an event as part of the history of a number of different processes. This event becomes a synchronization point for all of the processes, as opposed to simply the first two that could communicate. It may be seen that this extension can be simulated almost exactly by the use of a number of "sub-events" that synchronize the individual processes where a collection of sub-events forms the event in question and each sub-event synchronizes exactly two processes. However, we have used the single event as a total synchronization, since this leads to a more readable and understandable specification without, we believe, losing formality.

6.2.3 Modeling of Time

We have chosen to model the timer process in a very explicit way, and wherever we consider the interaction of a process with a timer we have modeled the combination of both processes. It might have been more appropriate to model the process of interest as though it would not fail and to model the timer as before. We would also need to model the sequence of events on the occurrence of the timer expiring (the *timeout* event). This latter sequence could be modeled as a separate process and then the entire system would be modeled by the combination of the "working" process interrupted by the "timeout" process.

Such an approach may well lead to specifications of the system that are easier to read and understand, though it might make reasoning about the specification harder.

6.3 Classification of CSP

We now present the classification of CSP according to our criteria. Each of the classification decisions is based both on the specification described in this chapter and on other published work on CSP.

6.3.1 Representation

Style — CSP specifies the objects of the systems by behavior rather than function. The specifications may be either declarative or operational, depending on the specifier's wishes. It is possible for any object to provide both the declarative and operational specifications and subsequently for the specifier to determine that the two specifications do indeed match. If the sequential process component of CSP is used, it is possible to specify processes that behave similarly to functions, in that they accept some input values, perform some computations and subsequently emit some output values. These are, however, processes and not functions.

Concurrency — CSP was designed for the specification of concurrent systems. Thus, the technique is able to express concepts of concurrent execution, though it does not specify the notion of two separate events occurring simultaneously. CSP may also be used in the specification of a single sequential process, though we have not shown this feature in any of our specifications.

Communication — CSP is modeled on the notion of unbuffered, synchronized communication between processes. This means that any process wishing to perform some communication will wait until the other process is also ready to communicate. Although we have not done so in our example, it is also possible to describe the data that is communicated between processes.

Non-Determinism — Non-determinism may be introduced if wished and the specifier may use one of the choice operators to indicate a non-deterministic choice between two processes. It should be noted that CSP does permit specifications of many concurrent systems without the introduction of non-determinism.

Fairness — CSP has no built-in notion of fairness and any notion that the user requires must be stated explicitly.

Modularity — CSP, with the ability to separately develop process specifications has, in some sense, a strong notion of modularity, in that components of a process that do not appear as part of the external interface may be hidden from other processes. However, at the level of a single sequential process, there is no concept of modular specification. Generally, the sequential processes are small and easily understood. The composition of such processes with appropriate use of hiding leads to a hierarchical structure of the system, with each level of the hierarchy comprising a number of processes with a limited number of interactions between them.

Time — CSP does not have any built-in notion of time, and if time should be included in the specification, it has to be modeled explicitly. However, it is possible to model time either as a global clock process emitting events as appropriate or, as we have done, by introducing a number of timer processes that provide an alarm at an appropriate point. It can also be used to model exceptional conditions such as failing components by introducing alternate behaviors depending on whether a clock timeout event occurs before or after a desired event. Using one or other of these approaches, both periodic and sporadic events may be represented.

Data — CSP has a limited notion of data, and the values passed between processes during the communication events have not been discussed at all. It is not possible to model complicated data structures using the CSP notation; rather some other notation must be used to formulate the appropriate structures.

User Presentation — As shown in the example specification, CSP is applicable to the specification of the interaction between the user and the system. However, it is not clear how CSP could be used to specify the format of the data that a user might see on a particular display.

6.3.2 Derivation

Transformation — CSP has many rules for transforming specifications. These rules maintain the semantics of the specification. The process of transforming the specification is guided by the choice of rule to apply. This is a matter for the specifier since there are no guidelines as a part of CSP. However, once chosen, a rule does, in the sense we have defined the word, transform a specification. We have not done this as part of our example; however, transformations could be used to show that, for example, $PDOP$ is modeled by the appropriate composition of RPD, APD, and a suitably labeled $TIMER$.

Elaboration — The specification may be elaborated according to the specifier's design principles. The specifier then has an obligation to demonstrate that the new specification correctly meets the intent of the original specification. If the elaborated specification simply has new events added without changing the meaning of the original events, then the elaborated specification may be shown to meet the original by a combination of event restriction and application of the transformation rules. If the elaboration involves refinement of an event to a number of sub-events, in the process removing the original event, there does noc appear to be a suitable mechanism for demonstrating that the elaborated specification meets the in ent of the original.

Composition — CSP has a powerful parallel combination operator which makes composition of processes a simple matter. Further, CSP has rules that determine the meaning of a process that is the combination of two processes whose meaning is known. One important feature of the parallel combination operator is that it need not introduce non-determinism. Events may be hidden, so internal communications of a process need not be visible to other processes.

6.3.3 Examination

Equivalence — In CSP, equivalence is shown by demonstrating that one process is a transformation of another process. This may be done by application of the transformation rules, or by demonstrating that the set of possible behaviors of one process is the same as the set of possible behaviors of the other process.

Consistency — Internal consistency of a specification is guaranteed in a sequential process, since a model-specified process has some meaning (though it may not be the desired meaning). In a system comprising a number of processes, internal consistency at the model level cannot be checked, other than determining whether the system violates certain other properties, for example safety and liveness properties.

At the level of specification using traces, it is possible to determine that two or more predicates over the traces of a process are consistent with each other. Each predicate makes requirements on the sequences of events that may occur in a correctly executing system. For example, a predicate may specify that the events *a*, *b*, and *c* occur in a specific order. If some other predicate requires that the events occur in a different order, then the second predicate is inconsistent with the first. Thus, a way to prove consistency is to pick a predicate and use it to describe the set of valid traces of a process satisfying the predicate. Then for each of the remaining predicates, remove those traces invalidated by the newly chosen predicate. If, after all predicates have been applied, the set of valid processes is empty, then the predicates are inconsistent with each other — there is no process that can satisfy all of the requirements expressed by the predicates.

There are no obvious ways of using CSP to check for under-specification. Checking for over-specification may be simpler in that at the level of specification of traces, it is possible to determine that a number of predicates imply another predicate, thus implying that the latter predicate is unnecessary.

Safety and Liveness — In order to check a system for safety, the desired property must be described in terms of some sequence of events that must not occur. Then if none of the traces of a process includes that sequence of events, the process preserves the safety property. The check is performed by examining all events preceding the current event in the trace.

The liveness properties that may be demonstrated are that the system is free of deadlock and livelock. Absence of deadlock may be proved by taking any trace of the system and showing that in every case there is at least one event which can extend the trace such that the extended sequence of events is still a trace of the system.

Determinancy — Because at this level of specification CSP does not discuss input or output values, but rather sequences of events, it is not possible to examine a system to see whether the results are determinate. It is, of course, possible to show that a system given the same sequence of events by the environment is determinate.

Correctness — CSP is able to discuss correctness, both with respect to the requirements as well as with respect to an implementation.

As shown in the specification, it was a simple task to predict behaviors of the system from the specification and therefore determine whether or not they met the user's requirements. CSP can be animated; the rules for animating a CSP specification have been described using Lisp as an implementation language [7].

Correctness with respect to the implementation may be determined at all levels for which a CSP specification exists. The requirement on the designer is to demonstrate that an implementation satisfies its specifications. CSP has algebraic laws concerning satisfaction that may be applied for a proof of correctness. Although no proof was performed in our example, it would have been possible, assuming the example is correct, to prove that the CSP process models satisfied the constraints on the traces.

Chapter 7

Vienna Development Method

This VDM specification of the avionics example is based on the CSP specification. Since the CSP specification has been used as the basis for the other specifications, this description of the VDM specification assumes that the reader is familiar with the overall problem, with the terminology of the problem domain, and with the expected behavior of the components.

The dialect of VDM used in this specification is based on the April 1989 draft of the British Standards Institute VDM standard [1], which is based on the syntax used by Jones [10]. The draft of the standard has been changed since then and features such as modules have been reduced to an annex as opposed to being part of the main text of the standard. It should be noted that since this specification is based on a draft standard that the VDM used may differ from the final standard.

VDM operators

VDM uses the usual logical operators of **and** ($\wedge$), **or** ($\vee$), **universal quantification** ($\forall x \in T \cdot E$) and **existential quantification** ($\exists x \in T \cdot E$).

VDM has operators for the specification of sequences (objects with type **seq** TYPE). The following are used in the specification.

$[]$ — The empty sequence.

$A \frown B$ — Concatenates the sequences A and B.

VDM uses decorated identifiers in the specification of post-conditions for operations and functions to indicate that the value being considered is the value **before** the operation or function is invoked. For example, if the post-condition of an operation were $x = \overleftarrow{x} + 1$, it would mean that the effect of invoking the operation would be to increment the value of x by one. Similarly, if the post-condition were $\overleftarrow{x} = x + 1$, it would mean that the operation decrements the value of x by one.

7.1 Specification

7.1.1 Handling Histories

VDM is a model-oriented specification method. The state is modeled in terms of well-understood mathematical structures, such as sets and sequences. Operations on the state are then specified as transformations on this model.

In the CSP version of the avionics problem, a number of events are described, some of which cause state transitions in the model (and therefore the system). These events may also have certain timing constraints. For example, a device must respond with some data within a given period of time, or a light must flash for at least two seconds before the pilot can clear the light. To model the state of the system, we must consider the events that occur within the operation of the system. There are cases where it is convenient to view the state of the system as a history of events, as in Pedersen and Klein [14]. In other cases, it is more convenient to view the system in terms of the history of state transitions caused by the occurrence of the events. Of course, it can be argued that a state transition is in itself an event. However, we are trying to maintain the distinction between events that are part of the problem statement and events arising from this specification. In any case, what is required is a theory of histories, independent of the subject of the histories. The most obvious approach is to use parameterized types, but VDM as presented in [10] does not admit parameterized types. Jones does use parameterized types informally.

At the time the specification was written, the standard [1] included module specifications. These are no longer part of the formal standard, but are used here in an obvious way.

A *history of X* is a time-stamped sequence of X:

$$Time\_Stamp = \mathbf{N}$$

$$\begin{array}{lll} Time\_Stamped(X) :: & type & : X \\ & time & : Time\_Stamp \end{array}$$

$History(X)$ :: **seq of** $Time\_Stamped(X)$

where

$inv\text{-}History(h) \;\triangleq\; \forall i,j \in \mathsf{dom}\, h \cdot i \leq j \;\Rightarrow\; h(i).time \leq h(j).time$

The invariant ensures that the sequence is time-ordered and this is analogous to the function Time_Stamps_Non_Decreasing in [14].

It is useful to know the most recent item in a history:

$last : History(X) \to Time\_Stamped(X)$

$last(h) \;\triangleq\; h(\mathsf{len}\, h)$

$\mathsf{pre}\; h \neq [\,]$

Where components of the specification have behaviors that are dependent upon timing constraints (for example the devices that have to respond within a given time), we will model the state of these components in terms of a history of events. The state of other components, such as the multi-purpose display, that have no timing requirements, will be modeled much more simply, in this particular case in terms of a single variable with two possible values.

The convention adopted is that a state ending in '_EH' is an event history, and a state ending in '_SH' is a state transition history.

7.1.2 The ADC and INS

A device is modeled by the sequence of time-stamped events (the history) in which it has participated. This model describes the device whether working or not. It is possible to deduce that a device has failed if the timestamp of the last event is significantly earlier than the current time. We are modeling the sequence of events corresponding to a request for data and the return of appropriate values as a VDM operation. Although the request is an event in terms of the system, it was found to be unnecessary to add a *request* event into the device history. In the system, the request is followed by data being returned from the device which could be modeled in our specification. However, our specification (following the CSP specification) is concerned with the sequence of events that occur in the system as opposed to the actual content (or even structure) of the data values. First, we specify a perfect device that always responds within a certain time. We then refine this specification to model an unreliable device (that may not even accept a request) in conjunction with a timer and a computer that will calculate an approximation to the result when the unreliable device fails to respond in time.

The relationship between the perfect device and its realization can be expressed formally in terms of a *retrieve* function. VDM provides the proof obligations which must be discharged to demonstrate that the realization correctly models the perfect device (essentially by showing that the retrieve function is a homomorphism). These proofs are beyond the scope of the present exercise, but the retrieve function and the proof obligations are outlined for completeness.

The INS and ADC display essentially the same behavior, so the specification here is for an arbitrary device of which the INS and ADC are instances. The way in which this arbitrary device becomes the INS or ADC is through the state on which it operates. The state is declared in the operations on the device and is an explicit paıameter to the pre- and post-conditions of these operations as used by Jones.[1] Therefore, for example, the reliable device specified in the following section is described in terms of an event history *RD_EH*, but when the NAV function uses it to model the INS, it declares an INS event history, *INS_EH*, used when quoting the pre- or post-conditions of operations involving the INS.

A Perfect Device

Our first specification will be of a device that works perfectly. The device will be explicitly initialized by invocation of the *PD_START* operation, and subsequently requests for data will be made by invocation of the *PD_REQ* operation, the latter returning the appropriate data values. We will keep a history of the events in which the perfect device engages.

The events recorded are:

START — The explicit initialization event for the device.

DATA — The event of returning data to the requester.

Note that, unlike the CSP specification, it is not necessary to record the event of requesting data from the device. The CSP specification used this event as a means of delimiting the beginning and end of the data request cycle in order to make appropriate comments with respect to the length of time between the request and the return of the data. Throughout this specification, we do not use an equivalent event to delimit the start of the operation, since we may achieve the same result

[1] See page 142 of Jones [10].

by using the initial and final values of a timestamp variable, which will be used whenever necessary.

After a request, a perfect device always returns with some data within the required time interval *max_delay*.

$$PD\_Event\_Type = \{START, DATA\}$$

$$PD\_Event = Time\_Stamped(PD\_Event\_Type)$$

$PD\_START \;\; (t\colon Time\_Stamp)$
ext wr $PD\_EH \; : \; History(PD\_Event\_Type)$
pre $PD\_EH = [\,]$
post $PD\_EH = [mk\text{-}PD\_Event(START, t)]$

$PD\_REQ \;\; (t, max\_delay\colon Time\_Stamp)$
ext wr $PD\_EH \; : \; History(PD\_Event\_Type)$
pre $(\mathsf{hd}\, PD\_EH).type = START \wedge$
$\quad last(PD\_EH).time \leq t$
post $\exists t' \in Time\_Stamp \cdot$
$\qquad PD\_EH = \overleftarrow{PD\_EH} \frown mk\text{-}PD\_Event(DATA, t') \wedge$
$\qquad t \leq t' \leq (t + max\_delay)$

A Physical Device

Our second specification is of a physical device that is unreliable. It may not accept a request, or may accept the request but not reply with the data within the required time. As with the CSP specification, we do not assume that this device needs any initialization; thus we do not show the *START* event used in the specification of the perfect device.

We considered using common events across histories; for example, much of *APD_REQ* may be described by the previously specified *PD_REQ* operation. However, problems arose with the generic history when instantiated for different devices. Instead, we chose to use disjoint sets of event names. A consequence of this decision is that the post-condition of operations is no longer possible, due to the state variables being of different types. We use the approach of disjoint event names again with *START* and *RSTART*, as well as *PDATA* and *ADATA* later in this specification.

The actual physical device has only one element in its history.

PDATA — The event of recurning physical data to the requester.

Unlike the perfect device, a request for data from the actual physical device (invocation of *APD_REQ*) may fail, in which case the history will be unaltered. We use a disjunction to separate the two possible outcomes of the operation, data or no data. This use of disjunction is common throughout this specification and introduces a non-deterministic element to the specification.

$$APD\_Event\_Type = \{PDATA\}$$

$$APD\_Event = Time\_Stamped(APD\_Event\_Type)$$

$$\begin{array}{l}
APD\_REQ \;\; (t, max\_delay\text{: } Time\_Stamp) \\
\textsf{ext wr } APD\_EH \; : \; History(APD\_Event\_Type) \\
\textsf{pre } last(APD\_EH).time \leq t \\
\textsf{post } APD\_EH = \overleftarrow{APD\_EH} \\
\quad \vee \; \exists t' \in Time\_Stamp \cdot \\
\qquad APD\_EH = \overleftarrow{APD\_EH} \frown mk\text{-}APD\_Event(PDATA, t') \wedge \\
\qquad t \leq t' \leq t + max\_delay
\end{array}$$

The Timer

A timer is started when a request is made on a device, and is used to inform the requester that the device has not responded within the required time interval. If the device responds before the timeout, the requester must cancel the timer.

The events recorded in the timer history are:

HTIMER — The event of canceling the timer before it expires.

TIMEOUT — The event of the timer expiring.

$$Timer\_Event\_Type = \{HTIMER, TIMEOUT\}$$

$$Timer\_Event = Time\_Stamped(Timer\_Event\_Type)$$

$STIMER\ (t, max\_delay\colon Time\_Stamp)$

ext wr $Timer\_EH\ :\ History(Timer\_Event\_Type)$

pre $last(Timer\_EH).time \leq t$

post $Timer\_EH = \overleftarrow{Timer\_EH} \frown$

$\quad mk\text{-}Timer\_Event(TIMEOUT, t + max\_delay) \vee$

$\quad \exists t' \in Time\_Stamp \cdot$

$\qquad Timer\_EH = \overleftarrow{Timer\_EH} \frown mk\text{-}Timer\_Event(HTIMER, t') \wedge$

$\qquad t \leq t' \leq t + max\_delay$

A Reliable Device

A reliable approximation of the perfect device is implemented by the physical device, a timer, and a computer. The reliable device requests data from the physical device and at the same time starts the timer. If the physical device has not responded before a timeout, the computer is used to calculate the approximate values of the data, based on older data.

The event history of the reliable device is similar to that of the perfect device, except that what is just *DATA* in the perfect device is now *ADATA* or *CDATA*, depending on how it was obtained.

The events recorded in the history of the reliable device are:

RSTART — The event initializing the reliable device.

ADATA — The event returning data from the actual physical device. If this event is recorded it implies that the device is working.

CDATA — The event indicating that the returned data has been computed. The implication of this event is that the physical device has, for whatever reason, failed to respond in a timely fashion.

First, we specify the operation *RPD_START*, which initializes the reliable physical device.

$$RPD\_Event\_Type = \{RSTART, ADATA, CDATA\}$$

$$RPD\_Event = Time\_Stamped(RPD\_Event\_Type)$$

$RPD\_START\ \ (t\colon Time\_Stamp)$
ext wr $RPD\_EH\ :\ History(RPD\_Event\_Type)$
pre $RPD\_EH = [\,]$
post $RPD\_EH = [mk\text{-}RPD\_Event(RSTART, t)]$

The *RPD_REQ* operation is the specification of the system's requesting data from a physical device and getting some data returned; the data may be either genuine or have been computed. We use the actual physical device specification and a timer to describe this operation. The effect of the post-condition in the *RPD_REQ* operation is to assert that both the *APD_REQ* and *STIMER* operations have been invoked. The new state of the *RPD_EH* then depends on the last timer event — an *HTIMER* means that the *APD* responded with data and a *TIMEOUT* means that the data had to be computed.

$RPD\_REQ\ \ (t, max\_delay\colon Time\_Stamp)$
ext wr $RPD\_EH\ \ :\ History(RPD\_Event\_Type)$
    wr $APD\_EH\ \ :\ History(APD\_Event\_Type)$
    wr $Timer\_EH\ :\ History(Timer\_Event\_Type)$

pre (hd $RPD\_EH).type = RSTART \wedge$
    $last(RPD\_EH).time \leq t$

post $post\text{-}APD\_REQ(t, max\_delay, \overleftarrow{APD\_EH}, APD\_EH) \wedge$
    $post\text{-}STIMER(t, max\_delay, \overleftarrow{Timer\_EH}, Timer\_EH) \wedge$
    cases $last(Timer\_EH)$ of
      $mk\text{-}Timer\_Event(HTIMER, t') \rightarrow last(APD\_EH) =$
        $mk\text{-}APD\_Event(PDATA, t') \wedge$
        $RPD\_EH = \overleftarrow{RPD\_EH} \frown$
          $mk\text{-}RPD\_Event(ADATA, t')$
      $mk\text{-}Timer\_Event(TIMEOUT, t') \rightarrow RPD\_EH = \overleftarrow{RPD\_EH} \frown$
          $mk\text{-}RPD\_Event(CDATA, t')$
    end

Correspondence between Reliable and Perfect Devices

The reliable device is meant to be an implementation of the perfect device. The retrieve function *retr-PD_EH* maps a reliable device event history (*RPD_EH*) back to its equivalent perfect device event history (*PD_EH*), and the operation of the reliable device models that of the perfect device if *retr-PD_EH* is a homomorphism.

The retrieve function *retr-PD_EH* makes a copy of RPD_EH with *RSTART* replaced by *START* and either *ADATA* or *CDATA* replaced by *DATA*.

```
$one\_rpd : RPD\_Event\_Type \rightarrow PD\_Event\_Type$

$one\_rpd(r)$  $\triangleq$  cases $r$ of
                    $RSTART \rightarrow START$
                      $ADATA \rightarrow DATA$
                      $CDATA \rightarrow DATA$
                    end
```

```
$retr\text{-}PD\_EH : History(RPD\_Event\_Type) \rightarrow History(PD\_Event\_Type)$

$retr\text{-}PD\_EH(rpd\_eh)$  $\triangleq$
    if $rpd\_eh = [\,]$
    then $[\,]$
    else
        let $mk\text{-}RPD\_Event(x, t) \frown rest = rpd\_eh$ in
        $mk\text{-}PD\_Event(one\_rpd(x), t) \frown retr\text{-}PD\_EH(rest)$
```

Jones [10] describes in detail the proof obligation to demonstrate that one specification models another. As an example, we will demonstrate the proof of part of the obligation. Specifically, we want to show part of the obligation arising from a proof that "concrete" operation *R* models "abstract" operation *A* with the retrieve function *retr*.

```
$gen\text{-}prf\text{-}obl()$  $\triangleq$
    $\forall \overleftarrow{r}, r \in R \cdot pre\text{-}A(retr(\overleftarrow{r})) \wedge post\text{-}R(\overleftarrow{r}, r) \;\Rightarrow$
                $post\text{-}A(retr(\overleftarrow{r}), retr(r))$
```

Plugging in *PD_REQ* for *A* and *RPD_REQ* for *R*, we can take advantage of the fact that references to *APD_REQ* are not needed for the proof. In a strictly formal proof these would need to appear and then be removed by the $\wedge$-Elimination proof rule, but here they are omitted altogether for clarity.

$$
\begin{array}{l}
\mathit{pd\text{-}prf\text{-}obl}() \;\; \triangleq \\
\quad \forall \overleftarrow{reh}, reh \in \mathit{History}(\mathit{RPD\_Event\_Type}) \cdot \\
\qquad \forall \overleftarrow{teh}, teh \in \mathit{History}(\mathit{Timer\_Event\_Type}) \cdot \\
\qquad\quad \forall t, \mathit{max\_delay} \in \mathit{Time\_Stamp} \cdot \\
\qquad\qquad (\mathsf{hd}\; \mathit{retr\text{-}PD\_EH}(\overleftarrow{reh})).\mathit{type} = \mathit{START} \wedge \\
\qquad\qquad \mathit{last}(\mathit{retr\text{-}PD\_EH}(\overleftarrow{reh})).\mathit{time} \leq t \wedge \\
\qquad\qquad \mathit{post\text{-}STIMER}(t, \mathit{max\_delay}, \overleftarrow{teh}, teh) \wedge \\
\qquad\qquad \mathsf{cases}\; \mathit{last}(teh)\; \mathsf{of} \\
\qquad\qquad\quad \mathit{mk\text{-}Timer\_Event}(\mathit{HTIMER}, t') \rightarrow reh = \overleftarrow{reh} \frown \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{mk\text{-}RPD\_Event}(\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{ADATA}, t') \\
\qquad\qquad \mathit{mk\text{-}Timer\_Event}(\mathit{TIMEOUT}, t') \rightarrow reh = \overleftarrow{reh} \frown \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{mk\text{-}RPD\_Event}(\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{CDATA}, t') \\
\qquad\qquad \mathsf{end} \\
\qquad\qquad \Rightarrow \\
\qquad\qquad \exists t' \in \mathit{Time\_Stamp} \cdot \\
\qquad\qquad\quad \mathit{retr\text{-}PD\_EH}(reh) = \mathit{retr\text{-}PD\_EH}(\overleftarrow{reh}) \frown \\
\qquad\qquad\qquad\qquad\qquad \mathit{mk\text{-}PD\_Event}(\mathit{DATA}, t') \wedge \\
\qquad\qquad\quad t \leq t' \leq t + \mathit{max\_delay}
\end{array}
$$

7.1.3 The Radar

The behavior of the radar can be thought of as the same as a perfect device, though there may be an indefinite delay between successive data values being returned. This indefinite delay corresponds to the radar device being switched off, and therefore no requests would be made of the device and thus the history would not change. In specification of the radar display, we treat the radar as a perfect device with a specific delay. We could specify the physical radar using an instantiation of the reliable device specification.

7.1.4 The MPD and HUD

The Multi-Purpose Display

In the following piece of the specification, we model the MPD in a simpler manner than that used for the devices previously specified. As previously explained, the MPD has no explicitly described timing constraints, and we will assume that the system operation to update the device status display on the MPD is invoked at an appropriate time and that the time taken for this system operation is negligible. We have assumed that the MPD does not break and the state variable *MPD_State* refers not to whether or not the MPD is working, but to whether or not the actual device is working.

The state of the MPD may have one of two values.

WORKING — Indicating that the device is functioning normally.

BROKEN — Indicating that the device is not functioning normally.

The operation *MPD_UPDATE* specifies the action of updating the MPD to describe the state of the device.

$Device\_State = \{WORKING, BROKEN\}$

$MPD\_UPDATE\ \ (new\_state\colon Device\_State)$
ext wr $MPD\_State\ :\ Device\_State$
post $MPD\_State = new\_state$

The Heads Up Display

Although the behavior of the HUD is essentially the same as the behavior of the MPD in that the HUD displays the status of the devices, there is a difference; the actions of the HUD are governed by timing constraints. As we stated in Section 7.1.1, we used a history to model the state. In the case of the HUD, it is more convenient to use a state-history rather than the event-histories used for the devices. This state-history is a sequence of (*state,time*) pairs, where the *time* component is the time at which the device entered the state specified by the *state* component. We used the state-history because several of the system's events can cause the same state transition of the system, and we tried to specify the essential behavior of the HUD, so the particular system event that caused the transition is of lesser importance.

Apart from receiving information from the reliable device, a state transition can be caused by a switch operated by the pilot. Thus, there are three operations on the HUD, one to inform it that a device is working, one to inform it that the device is broken, and one to inform it that the pilot has pressed the switch. These operations are described as "informing" rather than doing because following some state changes there is a refractory period (*hud_delay*) during which the incoming information is ignored.

The events recorded in the HUD history correspond to the processes described in the CSP specification.

WO — The HUD is displaying no information about the device and the device is working.

WF — The device has changed from being broken to working and the HUD is flashing an icon indicating that the device is working.

WC — The device is working and more than 2 seconds have elapsed since the HUD started flashing a icon indicating that the device is working. The pilot can now clear the display by pressing the appropriate key.

BR — The HUD is displaying no information about the device and the device is broken.

BF — The device has changed from working to broken and the HUD is flashing an icon indicating that the device is broken.

BC — The device is broken and more than 2 seconds have elapsed since the HUD started flashing a icon indicating that the device is broken. The pilot can now clear the display by pressing the appropriate key.

The operation *HUD_BROKEN* is invoked when the device changes state from working to broken; after the 2-second period has elapsed, the operation *HUD_SWITCH* may be invoked, which clears the display. The operation *HUD_WORKING* is equivalent to *HUD_BROKEN* but is invoked when the device changes state from broken to working.

$$HUD\_State\_Type = \{WO, WF, WC, BR, BF, BC\}$$

$$HUD\_State = History(HUD\_State\_Type)$$

```
$HUD\_BROKEN\ \ (t\colon Time\_Stamp)$
ext wr $HUD\_SH\ :\ History(HUD\_State\_Type)$
post if $last(HUD\_SH).type \in \{WO, WC\}$
    then $HUD\_SH = \overleftarrow{HUD\_SH} \frown mk\text{-}HUD\_State(BF, t) \frown$
                        $mk\text{-}HUD\_State(BC, t + hud\_delay)$
    else $HUD\_SH = \overleftarrow{HUD\_SH}$
```

```
$HUD\_WORKING\ \ (t\colon Time\_Stamp)$
ext wr $HUD\_SH\ :\ History(HUD\_State\_Type)$
post if $last(HUD\_SH).type \in \{BR, BC\}$
    then $HUD\_SH = \overleftarrow{HUD\_SH} \frown mk\text{-}HUD\_State(WF, t) \frown$
                        $mk\text{-}HUD\_State(WC, t + hud\_delay)$
    else $HUD\_SH = \overleftarrow{HUD\_SH}$
```

```
$HUD\_SWITCH\ \ (t\colon Time\_Stamp)$
ext wr $HUD\_SH\ :\ History(HUD\_State\_Type)$
post cases $last(HUD\_SH).type$ of
    $WC \rightarrow HUD\_SH = \overleftarrow{HUD\_SH} \frown mk\text{-}HUD\_State(WO, t)$
    $BC \rightarrow HUD\_SH = \overleftarrow{HUD\_SH} \frown mk\text{-}HUD\_State(BR, t)$
    otherwise $HUD\_SH = \overleftarrow{HUD\_SH}$
    end
```

Updating the MPD and HUD

The *UPDATE_DISPLAYS* operation offers a single operation to update both the MPD and HUD. It is invoked by the navigation function immediately after a request to the INS or ADC has returned some data and uses the fact that the last item in the device's history is *ADATA* or *CDATA*, according to whether the device responded or not.

```
$UPDATE\_DISPLAYS\ \ (t\colon Time\_Stamp)$
ext rd $RPD\_EH$      : $History(RPD\_Event\_Type)$
    wr $MPD\_State$ : $Device\_State$
    wr $HUD\_SH$      : $History(HUD\_State\_Type)$
```

```
post cases $last(RPD\_EH).type$ of
    $ADATA \rightarrow post\text{-}MPD\_UPDATE(WORKING, \overleftarrow{MPD\_State},$
                                $MPD\_State) \wedge$
                $post\text{-}HUD\_WORKING(t, \overleftarrow{HUD\_SH}, HUD\_SH)$
    $CDATA \rightarrow post\text{-}MPD\_UPDATE(BROKEN, \overleftarrow{MPD\_State},$
                                $MPD\_State) \wedge$
                $post\text{-}HUD\_BROKEN(t, \overleftarrow{HUD\_SH}, HUD\_SH)$
end
```

The Radar Display

The operation *RADAR_ON* specifies the events that must occur when the pilot switches the radar display on, and the operation *RADAR_OFF* specifies the events that occur when the radar display is switched off. The *RADAR_DISP* operation specifies the events involved in updating the radar display.

There is a separate event history indicating whether at time t the radar display is on or off; this history is updated by the operations *RADAR_ON* and *RADAR_OFF*. The actual cycle of requesting data from the radar, doing a computation if necessary, and displaying the data, is specified by the *RADAR_DISP* operation. This is invoked by *RADAR_ON* and continues cycling until *RADAR_OFF* updates the on/off history to *off*. At this point, *RADAR_DISP* generates an *RDCLEAR* event and stops.

The radar must supply data for display every 200 ms. and if this data is not forthcoming an approximation must be computed. If it takes c ms. to compute the approximation, the computation must start no later than $200 - c$ ms. after the previous display to ensure the data is available in time for the next display. For this reason, the radar itself is treated as a perfect device with maximum delay $200 - c$ ms.

The events recorded in the radar history display are:

RDDATA — The event of the radar returning data in a timely fashion.

RDCOMP — The event of the radar failing to return data in a timely fashion and some computation to estimate data.

RDDISP — The event of updating the radar display.

RDCLEAR — The event of clearing the radar display; this event is recorded when the radar is switched off.

$RD\_Event\_Type = \{RDDATA, RDCOMP, RDDISP, RDCLEAR\}$

$RD\_EVENT = Time\_Stamped(RD\_Event\_Type)$

$RD\_Status\_Type = \{RDON, RDOFF\}$

$RD\_Status = Time\_Stamped(RD\_Status\_Type)$

$RADAR\_ON \;\; (t\colon Time\_Stamp)$
$\mathsf{ext\ wr}\ RD\_EH \;:\; History(RD\_Event\_Type)$
$\quad \mathsf{wr}\ RD\_ST \;:\; History(RD\_Status\_Type)$
$\mathsf{post}\ RD\_ST = \overleftarrow{RD\_ST} \frown mk\text{-}RD\_Status(RDON, t) \wedge$
$\quad post\text{-}RADAR\_DISP(t, \overleftarrow{RD\_EH}, \overleftarrow{RD\_ST}, RD\_EH, RD\_ST)$

$RADAR\_OFF \;\; (t\colon Time\_Stamp)$
$\mathsf{ext\ wr}\ RD\_ST \;:\; History(RD\_Status\_Type)$
$\mathsf{post}\ RD\_ST = \overleftarrow{RD\_ST} \frown mk\text{-}RD\_Status(RDOFF, t)$

$RADAR\_DISP \;\; (t\colon Time\_Stamp)$
$\mathsf{ext\ wr}\ Radar\_EH \;:\; History(PD\_Event\_Type)$
$\quad \mathsf{wr}\ RD\_EH \;:\; History(RD\_Event\_Type)$
$\quad \mathsf{wr}\ RD\_ST \;:\; History(RD\_Status\_Type)$
$\mathsf{pre}\ RD\_EH \neq [\,] \wedge last(RD\_ST).type \neq RDOFF$

$$
\begin{array}{l}
\textbf{post } \mathit{post\text{-}PD\_REQ}(t, 200 - c, \overleftarrow{\mathit{Radar\_EH}}, \mathit{Radar\_EH}) \land \\
\quad \exists \mathit{tail} \in \mathit{History}(\mathit{RD\_Event\_Type}) \cdot \\
\qquad \mathit{RD\_EH} = \overleftarrow{\mathit{RD\_EH}} \frown \\
\qquad\qquad\qquad \mathit{mk\_RD\_Event}(\mathit{RDDISP}, t + 200) \frown \mathit{tail} \\
\qquad \land \\
\qquad (\mathit{last}(\mathit{RD\_ST}).\mathit{type} = \mathit{RDOFF} \land \\
\qquad\quad \mathit{tail} = \mathit{mk\text{-}RD\_Event}(\mathit{RDCLEAR}, t + 200) \\
\qquad \lor \\
\qquad\quad \mathit{last}(\mathit{RD\_ST}).\mathit{type} = \mathit{RDON} \land \mathit{tail} = [\,] \land \\
\qquad\quad \exists \mathit{RD\_EH}' \in \mathit{History}(\mathit{RD\_Event\_Type}) \cdot \\
\qquad\qquad \exists \mathit{RD\_ST}' \in \mathit{History}(\mathit{RD\_Status\_Type}) \cdot \\
\qquad\qquad\quad \mathit{post\text{-}RADAR\_DISP}(t + 200, \mathit{RD\_EH}, \mathit{RD\_ST}, \\
\qquad\qquad\qquad\qquad \mathit{RD\_EH}', \mathit{RD\_ST}'))
\end{array}
$$

7.1.5 The Waypoint Manager

A waypoint is a *Position* (we do not define *Position* any further in this specification) along with an indication of whether that position is to be approached or avoided.

$$
\begin{array}{lll}
\mathit{Waypoint} :: & \mathit{pos} : & \mathit{Position} \\
 & \mathit{ind} : & \{\mathit{APPROACH}, \mathit{AVOID}\}
\end{array}
$$

The waypoint manager maintains an ordered list of up to 20 waypoints.

$$\mathit{Waypoint\_List} :: \textsf{seq of } \mathit{Waypoint}$$

where

$$\mathit{inv\text{-}Waypoint\_List}(w) \quad \triangleq \quad \textsf{card dom } w \leq 20$$

There are several possible operations that could be performed on the waypoint list, for example, *create* and *update*. The only operation specified here, *W_REQ*, is an ›peration that removes the first element of the list of waypoints indicating that the pilot has just passed the current waypoint, so that the next waypoint is now at the head of the list of waypoints, corresponding to the CSP *wreq* event. A function *w_curr* returning the current waypoint is also provided.

$$\mathit{w\_curr} : \mathit{Waypoint\_List} \rightarrow \mathit{Waypoint}$$

$$\mathit{w\_curr}(w) \quad \triangleq \quad \textsf{hd } w$$

W_REQ
ext wr $w\ :\ Waypoint\_List$
pre $w \neq [\,]$
post $w = \mathsf{tl}\ \overleftarrow{w}$

7.1.6 The Autopilot

We have minimal information from the requirements with respect to the autopilot; thus we have specified a device that simply emits appropriate course control commands on invocation.

The autopilot is treated as a device with a single operation, corresponding to the CSP *course* event from the navigation unit. As a result of invoking this operation, the autopilot either does nothing or emits commands to steer the plane. The autopilot operation, *AP_COURSE*, is invoked every time the navigation function has performed its computation. The history of the autopilot is just the sequence of *COMMANDS* events issued.

COMMANDS — The event corresponding to issuing commands.

$$AP\_Event\_Type = \{COMMANDS\}$$

$$AP\_Event = Time\_Stamped(AP\_Event\_Type)$$

$AP\_COURSE\ \ (t{:}\ Time\_Stamp)$
ext wr $AP\_EH\ :\ History(AP\_Event\_Type)$
pre $last(AP\_EH).time \leq t$
post $AP\_EH = \overleftarrow{AP\_EH} \frown mk\text{-}AP\_Event(COMMANDS, t) \vee$
$\quad AP\_EH = \overleftarrow{AP\_EH}$

7.1.7 The Navigation Function

The navigation operation controls the operations described in this specification. The state of the navigation operation consists of the states of the previously described operations, and we will not list the events occurring in the history here. The interpretation of these events may be found in the description of the appropriate piece of specification presented previously in this chapter.

The operation of the navigation function is to request data from the ADC and the INS and to perform some computation *ncomp* on the data, which returns the course information to be transmitted to the pilot (via displays) and the autopilot.

This cycle of requests, data, and computations must be repeated every 59 ms. The cyclic nature of the operation is specified by quoting the post-condition recursively. In this case there is no termination condition for the recursion, since there are no *start* or *stop* operations which, if present, would allow termination to be handled in a similar manner to that of the radar display.

The ADC and INS are treated here as instances of the reliable device described earlier, with maximum delays of *adc_delay* and *ins_delay* respectively. The function *ncomp* assumes there is some actual data content in the *ADATA* and *CDATA* events of the perfect device.

The course information is of type *course*, which is not further defined. The parameterless function *atwaypt*, corresponding to the event of the same name in the CSP version, indicates whether the current waypoint has been reached, and hence whether the next waypoint is to be requested.

To simplify some of the expressions, we introduce a single type, *RPD_State*, embracing all facets of the reliable device. The actual ADC and INS are then represented as objects of this type in the navigation function.

$$\begin{array}{lrcl}
RPD\_State :: & rpd\_eh & : & History(RPD\_Event\_Type) \\
 & apd\_eh & : & History(APD\_Event\_Type) \\
 & timer & : & History(Timer\_Event\_Type) \\
 & hud\_sh & : & History(HUD\_State\_Type) \\
 & mpd\_state & : & Device\_State
\end{array}$$

$NAV\ (t: Time\_Stamp)\ c: Course$

$$\begin{array}{lll}
\textsf{ext wr}\ ADC & : & RPD\_State \\
\quad \textsf{wr}\ INS & : & RPD\_State \\
\quad \textsf{wr}\ WL & : & Waypoint\_List
\end{array}$$

$$\begin{array}{l}
\textsf{pre}\ pre\text{-}RPD\_REQ(t, ADC.rpd\_eh, ADC.apd\_eh, ADC.timer) \land \\
\quad pre\text{-}RPD\_REQ(t, INS.rpd\_eh, INS.apd\_eh, INS.timer) \land \\
\quad pre\text{-}W\_REQ(t, WL)
\end{array}$$

$$
\begin{array}{l}
\textbf{post}\ \exists t', t'' \in \mathit{Time\_Stamp}, w \in \mathit{Waypoint} \cdot \\
\qquad \mathit{post\text{-}RPD\_REQ}(t', \mathit{adc\_delay}, \overleftarrow{\mathit{ADC.rpd\_eh}}, \overleftarrow{\mathit{ADC.apd\_eh}}, \\
\qquad\qquad\qquad\qquad \overleftarrow{\mathit{ADC.timer}}, \mathit{ADC.rpd\_eh}, \\
\qquad\qquad\qquad\qquad \mathit{ADC.apd\_eh}, \mathit{ADC.timer}) \wedge \\
\qquad \mathit{post\text{-}RPD\_REQ}(t'', \mathit{ins\_delay}, \overleftarrow{\mathit{INS.rpd\_eh}}, \overleftarrow{\mathit{INS.apd\_eh}}, \\
\qquad\qquad\qquad\qquad \overleftarrow{\mathit{INS.timer}}, \mathit{INS.rpd\_eh}, \\
\qquad\qquad\qquad\qquad \mathit{INS.apd\_eh}, \mathit{INS.timer}) \wedge \\
\qquad t \leq t' \leq t + 59 \wedge \\
\qquad t \leq t'' \leq t + 59 \wedge \\
\qquad \mathit{post\text{-}UPDATE\_DISPLAYS}(t, \overleftarrow{\mathit{ADC.rpd\_eh}}, \\
\qquad\qquad\qquad \overleftarrow{\mathit{ADC.mpd\_state}}, \overleftarrow{\mathit{ADC.hud\_sh}}, \\
\qquad\qquad\qquad \mathit{ADC.rpd\_eh}, \mathit{ADC.mpd\_state}, \mathit{ADC.hud\_sh}) \wedge \\
\qquad \mathit{post\text{-}UPDATE\_DISPLAYS}(t, \overleftarrow{\mathit{INS.rpd\_eh}}, \\
\qquad\qquad\qquad \overleftarrow{\mathit{INS.mpd\_state}}, \overleftarrow{\mathit{INS.hud\_sh}}, \\
\qquad\qquad\qquad \mathit{INS.rpd\_eh}, \mathit{INS.mpd\_state}, \mathit{INS.hud\_sh}) \wedge \\
\qquad (\mathit{atwaypt} \wedge \mathit{post\text{-}W\_REQ}(\overleftarrow{\mathit{WL}}, \mathit{WL}) \vee \\
\qquad \neg \mathit{atwaypt} \wedge \mathit{WL} = \overleftarrow{\mathit{WL}}) \wedge \\
\qquad c = \mathit{ncomp}(\mathit{last}(\mathit{ADC\_EH}), \mathit{last}(\mathit{INS\_EH}), \mathit{w\_curr}(\mathit{WL})) \wedge \\
\qquad \exists \mathit{ADC}', \mathit{INS}' \in \mathit{RPD\_State} \cdot \\
\qquad\quad \exists \mathit{WL}' \in \mathit{Waypoint\_List}, c' \in \mathit{Course} \cdot \\
\qquad\qquad \mathit{post\text{-}NAV}(t + 59, \mathit{ADC}, \mathit{INS}, \mathit{WL}, \\
\qquad\qquad\qquad\qquad \mathit{ADC}', \mathit{INS}', \mathit{WL}', c')
\end{array}
$$

7.2 Comments on the Specification

The avionics example is not typical of the class of problems to which VDM is normally applied in that it exemplifies concurrency and requires explicit modeling of time.

Event histories were introduced to enable us to write specifications with statements about time. Although these are similar to CSP traces, we tried to avoid simply reproducing the CSP specification in VDM. Nevertheless, taking the CSP as the starting point is bound to have had an effect on the end product.

Similarly, the decomposition of the problem has inevitably been influenced by the CSP decomposition in order to maintain a visible correspondence between the specifications. Other decompositions are possible, and indeed might be considered more natural to the VDM style.

Polled devices, that is devices which receive a request and then return some data, are treated as objects on which a REQUEST operation may be performed, and this seems a fairly natural way to handle them. A consequence of this is that an operation now takes a finite time to execute, during which other events may have occurred. This in turn gives rise to a need for some form of protection of that part of the state being updated. In the past, constructs such as rely/guarantee conditions have been proposed. In this example, we have avoided rely/guarantee conditions by having predicates over the history of the state rather than just the current state. A concurrent state change affected by an operation executing in parallel is detected by inspecting the last item in that state's history and comparing it with the current time (that is, the time parameter to the operation).

The handling of cyclic processes like the navigation function and radar display is not entirely satisfactory. The method used to specify cyclic repetition is to quote the post-condition of the operation recursively. In the case of the radar display, there are ON and OFF operations with their own history. If an OFF operation has occurred during one of the cycles, this causes termination of the function. In the navigation function there are no such ON/OFF operations and so there is no guard on the recursion to ensure termination. It has been assumed that, at some stage, some kind of ON/OFF facility would have to be introduced into the requirements to allow orderly start-up and shut-down.

One area in which VDM could have demonstrated its superiority is in the representation and manipulation of data. In several cases a process returns either some actual data or some computed data depending on whether a device has responded in time or not. Following the CSP specification (with no further information available) the data is simply treated as an event. VDM has the capability of defining both the structure of the data and the computation performed to produce the data. The waypoint manager is the component that comes closest to illustrating this, the most conventional use of VDM.

7.3 Classification of VDM

This section presents the classification of VDM according to our criteria. Note that although most of the classification is based on our example specification, part of the classification is based on available VDM literature.

7.3.1 Representation

Style — VDM is model-oriented and specifies objects by their functions. It allows both implicit and explicit specification, and explicit specifications may be declarative or procedural.

Concurrency — VDM as generally used has no notion of concurrency. Thus the aspects of concurrency that occur in reactive systems have to be modeled explicitly. The variation of VDM used to specify our example uses event and state histories to model the concurrent behavior of the system. Other approaches that have been used are to introduce *rely* and *guarantee* conditions for VDM operations [9] and then to model communication between concurrently executing processes by means of shared variables.

Communication — Communication may only be introduced into VDM by explicit modeling. Shared data is probably the easiest approach to such modeling, but no approach is elegant.

Non-Determinism — An operation that has non-deterministic behavior may be represented in VDM by means of disjunctions in the post-condition of the operation. If we assume that an operation will either terminate resulting in post-condition *P1* or will terminate resulting in *P2* then $P1 \vee P2$ describes the post-condition of the operation without stating what determines which state will result.

Fairness — There is no concept of fairness.

Modularity — In the draft version of the standard [1], a form of parameterized modules was described. Subsequently, doubt has been cast on the soundness of these parameterizations and the constructs have been removed from the main body of the standard. It is true that the definition of functions provides some modularity. However, for specification of large systems a stronger notion of modularity is needed, perhaps something closer to abstract data types is appropriate.

Time — Time must be modeled explicitly. In the avionics example, sporadic events are easier to handle (by treating them as operations) than periodic events.

Data — Although the capability is not used within our example, VDM has a very rich notation for modeling data structures.

User Presentation — Not addressed in the avionics example, but we believe that VDM should be good for representing the user interface.

7.3.2 Derivation

Transformation, Elaboration, & Equivalence — VDM is a general purpose formal language so that both transformations and elaborations can be performed within the same language and are hence subject to the same proof rules and proof obligations.

Composition — Composition of operations by operation quotation is made awkward because post-conditions tend to have large signatures. There is little else to allow composition of specifications.

7.3.3 Examination

Consistency — The construction of the state model is useful in removing inconsistencies from a natural language specification. Under- and over- specification are addressed by Jones [10] in terms of the ability to provide a *retrieve* function and in the many-oneness of the function.

Safety & Liveness — These are not built-in concepts, and proofs of such properties depend on how concurrency and communication are being modeled.

Determinacy — It is possible in VDM to specify the requirement for a determinate solution to a problem with a non-deterministic implementation. For example, it would be possible to specify a function that given a set returned the maximal element in the set. By the definition of sets in VDM, the implementation will be non-deterministic, but the result will be determinate.

Correctness — Validation is a problem in all specification methods. It is possible in principle to animate some VDM specifications or to paraphrase the VDM back into natural language, but there is nothing in the method which specifically addresses validation. Verification, on the other hand, is VDM's *raison d'être* and it defines its own set of proof rules and proof obligations.

Chapter 8

Temporal Logic

This chapter specifies the avionics problem using temporal logic and model checking. The system is specified in temporal logic, describing its operation by the set of logical conditions which are disallowed (the safety conditions) and the set of logical conditions which are allowed (the liveness conditions). Pnueli first introduced the application of temporal logic to concurrent software specification, and his later paper [15] is a good survey of the field. The specification in this section, however, follows the CTL model of Clarke, and this is described in [4]. The model checking part of the specification consists of building a finite state machine, against which the temporal logic specifications can be checked for consistency. Although we only show the temporal formulas in this example, we consider both the formulas and the finite state machine model to be a part of the specification of the system. The steps in the development of a specification are outlined below.

1. Organize the system into a collection of processes and communications between the processes. The organization of the system into smaller components (the processes) makes it easier to understand the operation of the system.

2. Specify the operation of each process as temporal logic formulas, adhering to some naming convention for information exchanged between processes.

3. Build a finite state machine representing a model of the process' operation for each process separately. Check the temporal formulas for each process against the model, to verify the consistency of the formulas.

4. Coalesce the process models into a system representation, checking against the formulas involving the inter-process communications conditions, and then verify the system operations.

The temporal logic used throughout is Clarke's CTL, though the syntax used differs from this, since we prefer to use icons rather than alphabetic characters, believing that this is more easily readable to the uninitiated and is more in keeping with other temporal logic specifications.

Temporal Operators

The expected logical operators used throughout are **and** ($\wedge$), **or** ($\vee$), **not** ($\neg$), **equivalence** ($\equiv$), and **implies** ($\rightarrow$).

The temporal operators to be used are listed below, and the bold word in the description of the operator is the word to be visualized when reading the formula.

$\Box a$ — This means that **henceforth** a is always true.

$\Diamond a$ — This means that in **all** cases **eventually** a will be true.

$\underline{\Diamond} a$ — This means that in **some** cases **eventually** a will be true.

$\circ a$ — This means that in all cases at the **next** state (instant in time) a will be true.

$\underline{\circ} a$ — This means that in some cases at the **next** state (instant in time) a will be true.

$\bullet a$ — This means that at the **previous** state (instant in time) a was true.

$a \Rightarrow b$ — This is read as **strictly implies**, and it means that henceforth, if a is true, then b is true. $(a \Rightarrow b) =_{\mathrm{df}} \Box(a \rightarrow b)$.

$a \,\mathcal{U}\, b$ — This means that in all cases, a is true **until** the state (instant in time) when b occurs. The strong **until** implies that b will eventually occur.

$a = b$ —This means that a is **strictly equivalent** to b. $(a = b) =_{\mathrm{df}} \Box(a \equiv b)$.

Conventions

Temporal logic is useful for specifying the system as a set of formulas which must be satisfied. The problem we are dealing with is one in which the same conditions are repeated over many cycles of operations. Temporal logic does not distinguish between events occurring at different cycles, but restricts itself to describing what logical conditions must always be true, and in general this leads to describing the logical conditions which may repeat during a cycle.

The occurrence of an event is mimicked in temporal logic by stating that a logical condition changes state.

8.1 Specification

Figure 8.1 shows an overview of the processes specified (the MPD is not shown to avoid clutter) and the interfacing conditions between processes. The processes specified in the figure relate directly to those represented as processes in the model-building language SML. Temporal logic per se has, of course, no concept of a process, but it is a useful way of organizing the problem. Each of the processes in the diagram is specified using temporal logic in this section.

8.1.1 Handling Time

Time is not represented explicitly in temporal logic. It is extremely convenient in temporal logic to discuss the changes in logical conditions which occur, and to relate these to the eventuality of other conditions at some later time. The problem we are specifying, however, does depend on real time intervals, and we get around this by using the following tricks.

1. In cases where one event must occur before a timer completes, we can state that the timer will always eventually recur. This does not, of course, explicitly state that the timer occurs at exact intervals. But in the FSM modeling we can introduce real time intervals, and if the formulas are true against this model, then we know that they are true over the real time intervals.

2. In other cases, we make use of the power of the **until** operator, which states that the condition following the operator will always occur.

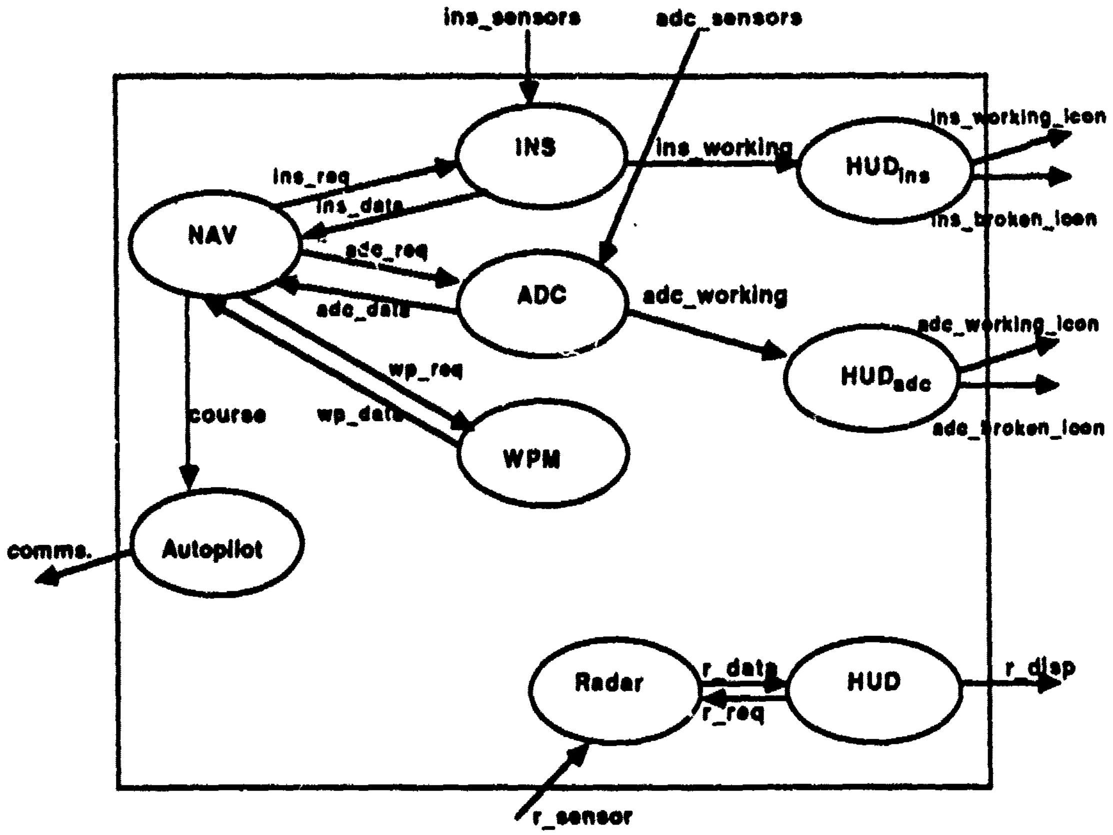


Figure 8.1: Processes in Temporal Logic Specification

8.1.2 The ADC and the INS

This subsection specifies the operation of a device, and this device describes the behavior of the inertial navigation system (INS) and the air data computer (ADC). We first specify the generic device, and then distinguish the logical variables which are 'local' to each device, and those which are 'shared' with other subsystems. Obviously, we need to supply distinguishing names for those shared variables, and this is done at the end of this subsection.

The generic device is specified first as a perfect device, which includes all of the behavior to supply a response to a request for data. In a real system, this would encompass the response of the device, some timers, and a computation if the actual device fails to respond. We then proceed to introduce the actual device and the computation.

Perfect Device

The perfect device does not exist. It is valuable, however, to model a perfect device; we may then compare the specification of the combination of a device and MCC software to the model. When a request is received by this device, it will always respond with data within a predefined maximum time interval. Of course, the device must be used by a disciplined user, who will not issue a further request within the maximum response time.

No request will be made of the perfect device until a start has been issued.

$$\Box\neg requ\,\mathcal{U}\ start$$

Once the condition start is set, it will always remain set.

$$start \Rightarrow \Box start$$

When the request is received by a perfect device, the data will eventually be supplied by the device before the maxtimer has expired. This implies that the *maxtimer* is *false* until the *data* becomes set.

$$\bullet\neg requ \wedge requ \Rightarrow \circ\neg maxtimer\ \mathcal{U}\ data$$

When the data signal is set, it remains set until the next request.

$$data \Rightarrow data\,\mathcal{U}\ requ$$

The next request can only be raised after completion of the maximum time interval.

$$requ \Rightarrow \circ\neg requ\ \mathcal{U}\ maxtimer$$

There is no mechanism required for the **maxtimer** quantity. In the previous formula, however, it is explicitly required that maxtimer must occur, and the sequence of states leading to its occurrence is specified. We could have introduced a **startimer**, which led eventually to a maxtimer, but this would have no effect on the specification completeness, though it may have made the intention more obvious. In the model building of a process in SML, there is the ability to measure time explicitly as a number of cycles of operation. This allows us to determine that the formulas are true with an explicit timing considerations.

Actual Physical Device

An actual physical device can, of course, fail to return a signal, since it will often break or lose its communication channel. This is represented by stating that for some paths eventually the data will be returned.

When a request is generated, the actual physical device will sometimes respond with a physical signal.

$$\bullet\neg apd\_requ \wedge apd\_requ \Rightarrow \underline{\Diamond} apd\_data$$

Reliable Device

A reliable device mimics a perfect device by requesting the data from the actual physical device and computing an estimate if the device does not respond within a certain time period. Before describing the safety and liveness conditions, a strict equivalence is specified. The data response of the perfect device is strictly equivalent to either the data signal or the estimated data.

$$data = apd\_data \vee est\_data$$

Safety Condition

One condition which should never occur is to have both the device data and the estimated data present at the same instant.

$$\Box\neg(apd\_data \wedge est\_data)$$

Liveness Conditions

The request for a data value generates a request to the actual physical device, and eventually either the data arrives or a timeout occurs.

$$\bullet\neg \mathit{requ} \wedge \mathit{requ} \Rightarrow \circ \mathit{apd\_requ} \wedge \Diamond(\mathit{timeout} \vee \mathit{apd\_data})$$

If the actual physical device responds, the timeout signal is lowered and remains in this state until a new request to the device is issued. In addition, the data signal remains raised until a new request is received.

$$\mathit{apd\_data} \Rightarrow \circ((\neg \mathit{timeout} \wedge \mathit{apd\_data})\ \mathcal{U}\ \mathit{apd\_requ}) \wedge (\neg \mathit{est\_data}\ \mathcal{U}\ \mathit{apd\_requ})$$

If the timeout occurs, then the actual data will not be accepted, and an estimate will be made and maintained until the next request.

$$\neg \mathit{apd\_data} \wedge \mathit{timeout} \Rightarrow (\neg \mathit{apd\_data}\ \mathcal{U}\ \mathit{apd\_requ}) \wedge (\mathit{timeout}\ \mathcal{U}\ \mathit{est\_data})$$

The estimated data is ready before the next request is issued.

$$\mathit{est\_data} \Rightarrow (\mathit{est\_data}\ \mathcal{U}\ \mathit{requ}) \wedge \circ(\neg \mathit{timeout}\ \mathcal{U}\ \mathit{apd\_requ})$$

There is a need to maintain a persistent variable describing whether the device is working or broken. If it is working, it will remain *working* until the *est_data* is computed. If it has failed, it will remain $\neg$*working* until *apd_data* is available.

$$\mathit{apd\_data} \Rightarrow \circ(\mathit{dev\_working}\ \mathcal{U}\ \mathit{est\_data})$$

$$\mathit{est\_data} \Rightarrow \circ(\neg \mathit{dev\_working}\ \mathcal{U}\ \mathit{apd\_data})$$

The strong **until** used in the previous formulas implies that the device will fail and be restored to operation time after time. Although these conditions seem rather strange, they are a consequence of the lack of a stopping condition as are the fairness conditions listed in the next section. The awkwardness caused by the lack of a stopping condition recurs throughout the specification.

Fairness Conditions

The manner in which the *dev_working* condition is specified requires the repeated recurrence of both *est_data* and *apd_data*. This, of course, will only happen if the device can be restored after failure and can fail after each restoration.

$$\Box\Diamond \mathit{est\_data}$$

$$\Box\Diamond \mathit{apd\_data}$$

INS subsystem

The INS subsystem communicates with the rest of the world through the variables defined below. These are equivalent to the variable *requ, data*, and *dev_working* in an instantiation of the generic device specification. The INS subsystem is viewed as a *perfect device* by the NAV process, and as a *reliable device* by the HUD and MPD processes.

ins_requ — True when data is requested of the INS.

ins_data — True when data is available from the INS.

ins_working — True when the INS sensor is working.

ADC subsystem

The ADC subsystem communicates with the rest of the world with the variables defined below. These are equivalent to the variable *requ, data*, and *dev_working* in an instantiation of the generic device specification. The ADC subsystem is viewed as a *perfect device* by the NAV process, and as a *reliable device* by the HUD and MPD processes.

adc_requ — True when data is requested of the ADC.

adc_data — True when data is available from the ADC.

adc_working — True when the ADC sensor is working.

8.1.3 The Radar

The radar subsystem communicates with the rest of the world through the variables defined below. These are equivalent to the variable *requ* and *data* in an instantiation of the generic device specification. The radar is viewed as a perfect device by both the HUD and the MPD.

r_requ — True when data is requested of the radar.

r_data — True when data is available from the radar.

8.1.4 The MPD and the HUD

This section describes the display of data on the HUD from all devices. Information from the INS, ADC, and radar is displayed on the HUD. The ADC and the INS behave in the same manner, though presumably in different windows on the HUD. The radar has a somewhat different behavior.

INS and ADC Displays on the HUD

This section describes the display of information on the HUD from the INS and ADC processes.

Definitions

switch — Indicates whether a switch was touched.

dev_working — True whenever the device is working.

disp_broken — True when the "device broken" icon is flashing on the HUD.

disp_working — True when the "device working" icon is flashing on the HUD.

timeout — True when 2 seconds or more have elapsed since the last device icon started to flash on the display.

Preliminaries

remove — This is the condition for removing the icon currently flashing on the screen.

$$remove = switch \wedge timeout$$

Safety Conditions

The "device working" and "device broken" icons can never be displayed simultaneously.

$$\Box\neg(disp\_working \wedge disp\_broken)$$

Liveness Conditions

When neither icon is flashing and the device fails, the *disp_broken* icon is displayed. This seems rather simple and straightforward, but it turned out to be somewhat more complicated when the model was built and verification attempted. In fact, the formula chosen (given below) differs somewhat from the simple statement above. The reason for this restatement is to cover the condition where simultaneously the device fails and the icon denoting device failure is removed (implying that the device recovered temporarily and then failed again).

$$\begin{array}{l}(\bullet(\mathit{dev\_working} \wedge \neg \mathit{disp\_broken}) \quad \wedge \\ (\neg \mathit{dev\_working} \wedge \neg \mathit{disp\_working})) \quad \Rightarrow \quad (\mathit{disp\_broken}\ \mathcal{U}\ \mathit{remove})\end{array}$$

When the "device broken" icon is flashing and the remove condition arises, then the display must remove the "device broken" icon. The "device broken" icon cannot be redisplayed until after a "display working" signal occurs.

$$\mathit{remove} \wedge \mathit{disp\_broken} \Rightarrow \circ(\neg \mathit{disp\_broken}\ \mathcal{U}\ \mathit{disp\_working})$$

When the "device broken" icon disappears and the device is working, then the "device working" icon must start flashing.

$$\bullet \mathit{disp\_broken} \wedge \neg \mathit{disp\_broken} \wedge \mathit{dev\_working} \Rightarrow \mathit{disp\_working}\ \mathcal{U}\ \mathit{remove}$$

When neither icon is flashing and the device starts working, the "device working" is displayed.

$$\begin{array}{l}(\bullet(\neg \mathit{dev\_working} \wedge \neg \mathit{disp\_working}) \quad \wedge \\ \quad (\mathit{dev\_working} \wedge \neg \mathit{disp\_broken})) \quad \Rightarrow \quad (\mathit{disp\_broken}\ \mathcal{U}\ \mathit{remove})\end{array}$$

When the "device working" icon is flashing and the remove condition arises, then the display must remove the "device working" icon. The "display working" signal cannot be redisplayed until the "display broken" icon has been displayed.

$$\mathit{remove} \wedge \mathit{disp\_working} \Rightarrow \circ(\neg \mathit{disp\_working}\ \mathcal{U}\ \mathit{disp\_broken})$$

When the "device working" icon disappears and the device is broken, then the "device broken" icon must start flashing.

$$\bullet \mathit{disp\_working} \wedge \neg \mathit{disp\_working} \wedge \neg \mathit{dev\_working} \Rightarrow \mathit{disp\_broken}\ \mathcal{U}\ \mathit{remove}$$

Fairness Conditions

The strong **until** used in two of the preceding formulas is incorrect for all considerations, since the device may never again fail (or start to work). This can be circumvented by adding fairness conditions such that the device fails and recovers infinitely often.

$$\Box\Diamond\neg dev\_working$$

$$\Box\Diamond dev\_working$$

Display of Radar Data on the HUD

The display of radar data on the HUD is under the pilot's control. He can select the radar to be on or off. When the radar is switched off, the display is cleared. When the radar is switched on, the data from the radar is updated on the HUD every 200 ms.

Definitions

r_on — True when the radar display is switched on.

r_maxtimer — True when the maximum time interval between displays has expired.

r_requ — True when a request to the radar for data is issued.

r_data — True when the radar data is available.

r_disp — True when the radar display has been updated.

r_clear — True when the radar display is to be cleared.

Liveness Conditions

When the radar display is switched off, a clear signal is sent and the timer is disabled.

$$\bullet r\_on \wedge \neg r\_on \Rightarrow \circ(r\_clear \wedge (\neg r\_disp\, \mathcal{U}\, r\_on))$$

A maxtimer signal is generated autonomously on a periodic basis.

$$\Box\Diamond r\_maxtimer$$

When the *r_maxtimer* is set and the radar is on, eventually a request to the radar subsystem is generated.

$$r\_maxtimer \wedge r\_on \Rightarrow \circ(\neg r\_maxtimer\ \mathcal{U}\ (r\_disp \vee r\_clear)) \wedge \Diamond r\_requ$$

When a request is made, eventually the data will be available; it will then be displayed.

$$r\_requ \Rightarrow \Diamond(r\_data\ \mathcal{U}\ (r\_disp \vee r\_clear))$$

Notes

1. It may seem to the reader on initial inspection that the raising of the *r_on* signal should trigger the starting of the timer, which should then occur periodically until the $\neg r\_on$ condition arises. Indeed, this was the approach we took initially; however, the problem with this approach was that the condition could change from on to off to on to off within a display cycle, and the model to handle those situations, while still satisfying the temporal formulas (which obviously differed from those above), was becoming quite complicated and still did not work. We compromised, changed the original formulas until we derived those presented above, and were able to verify them against the new (simple) model quite easily. The only compromise to system operation is that the display of data to the pilot after he switches the radar on will be delayed by (at most) the interval of the radar update (200 ms. in this case).

2. The maxtimer signal can be modeled exactly, if awkwardly, by using the next ($\circ$) construct repeatedly. For example, if an individual cycle in the model took 50 ms., then a delay of 4 cycles between maxtimers could be expressed as:

   $$maxtimer \Rightarrow \circ(\neg maxtimer \wedge \circ(\neg maxtimer \wedge \circ(\neg maxtimer \wedge \circ maxtimer)))$$

   We were not especially comfortable with this particular expression of the problem and did not use it.

Multi-Purpose Display (MPD)

The multi-purpose display is another display available to the pilot. Once again, the behavior of the INS and ADC components is exactly the same on this display, though they differ somewhat from the displays on the HUD. There is also a radar display on the MPD, but this is exactly the same as that of the HUD.

INS and ADC displays on the MPD

The MPD display for both these devices shows the device as either working or broken. There are no particular timing requirements. In this section, the generic variables are denoted by *xxx*, which is replaced by *adc* or *ins*.

xxx_icon_working — The icon showing that device *xxx* is working is currently being displayed.

xxx_icon_broken — The icon showing that device *xxx* is broken is currently being displayed.

Safety Condition

The device cannot display working and broken simultaneously.

$$\Box\neg(xxx\_icon\_working \wedge xxx\_icon\_broken)$$

Liveness Conditions

When the icon designating the device is working is displayed, then the device must have been working in the previous state.

$$xxx\_icon\_working \Rightarrow \bullet xxx\_working$$

When the icon designating the device is broken is displayed, then the device must have been broken in the previous state.

$$xxx\_icon\_broken \Rightarrow \bullet\neg xxx\_working$$

8.1.5 The Waypoint Manager

This is an initial description of the waypoint manager. The waypoint manager nandles a sequence of waypoints (up to 20). It receives a request for a waypoint, and responds by giving the next waypoint in the sequence. We assume that the new waypoint given by the waypoint manager allows the navigation subsystem to determine when to ask for a new waypoint. Hence, the waypoint manager merely accepts a request for a new waypoint and returns a new waypoint from a sequence of waypoints.

Definitions

In this section, we are dealing with a sequence K of waypoints, and each waypoint is denoted by an index $(1 \leq k \leq 20)$. It is convenient to introduce a second sequence, K^-, that is the same as sequence K, except that it is missing the last element.

$$\forall i.1 \leq i \leq 19 \Rightarrow K_i = K_i^-$$

beyond$_k$ — When this is true, the aircraft is beyond waypoint$_k$.

w_requ — The next waypoint in the sequence is requested.

w_data — The next waypoint is available.

Safety Conditions

When the aircraft has not yet reached waypoint$_{k1}$, it obviously has not reached any of the waypoints beyond waypoint$_{k1}$ in the sequence.

$$\forall k, k1 \in K, k \geq k1 \neg beyond_{k1} \Rightarrow \neg beyond_k$$

When the aircraft is beyond sector$_{k1}$, it must also be beyond the previous waypoints.

$$\forall k, k1 \in K, k \leq k1, beyond_{k1} \Rightarrow beyond_k$$

Liveness Conditions

When a request for the next waypoint is issued, then the *w_data* signal will be false during the computation of the new waypoint.

$$\mathit{w\_requ} \wedge (\exists k \in K^{-}, \mathit{beyond}_k \wedge \neg \mathit{beyond}_{k+1}) \Rightarrow \neg \mathit{w\_data}\ \mathcal{U}\ \mathit{beyond}_{k+1}$$

When the new waypoint is available, the appropriate *beyond* signal is raised at the same time as the data available signal.

$$\forall k \in K, \bullet\neg \mathit{beyond}_k \wedge \mathit{beyond}_k \Rightarrow \mathit{w\_data}$$

If there is a request for a waypoint after the maximum number have been given, the request is ignored by leaving the value *w_data* set.

$$\mathit{w\_requ} \wedge \mathit{beyond}_{20} \Rightarrow \mathit{w\_data}$$

8.1.6 The Autopilot

The autopilot accepts commands from the navigational unit and turns them into control actions.

Definitions

course — Data from the navigational unit is available.

comms — Commands are to be sent to the autopilot.

Liveness Conditions

The autopilot performs these actions by awaiting the navigational unit's signal, changing state, then initiating the commands before the course signal is changed again.

$$\bullet\neg \mathit{course} \wedge \mathit{course} \Rightarrow \mathit{course}\ \mathcal{U}\ \mathit{comms}$$

8.1.7 The Navigation Function

The navigation subsystem computes the course to be followed, after collecting data from the INS, ADC, and waypoint manager subsystems. Hence, the navigation process has to issue commands to each of the above perfect devices, receive all results, and compute a course.

Definitions

n_maxtimer — Maximum time interval between invocations of navigation.

ins_req — Request for data from the INS.

adc_req — Request for data from the ADC.

ins_data — Data is available from the INS in response to the request.

adc_data — Data is available from the ADC in response to the request.

course — The course heading has been computed.

w_requ — Request for a new waypoint.

w_data — A new waypoint has just been delivered.

Liveness Conditions

The n_maxtimer signal is always repeated.

$$\Box\Diamond n\_maxtimer$$

When the *n_maxtimer* occurs, it is reset, and the *course* must be computed before it is again set. In addition, the requests to the INS and ADC are eventually issued. The use of **eventually** when issuing commands is deliberate, since it allows the designer the maximum flexibility in communicating with the two devices.

$$n\_maxtimer \Rightarrow \circ(\neg n\_maxtimer\ \mathcal{U}\ course) \wedge \Diamond ins\_requ \wedge \Diamond adc\_requ$$

When the *ins_requ* signal is set, the *ins_data* signal must be reset at the next state, and eventually the *ins_data* signal will set. This corresponds to communicating with the perfect device described previously.

$$ins\_requ \Rightarrow \circ\neg ins\_data \wedge \Diamond ins\_data$$

When the *adc_requ* signal is set, the *adc_data* signal is reset, and eventually the *adc_data* signal will be set. Once again, this is a communication with a perfect device.

$$adc\_requ \Rightarrow \circ\neg adc\_data \wedge \Diamond adc\_data$$

When both requests have been made, the course signal must be reset and remain reset until the data signals arrive and the waypoint data is available.

$$adc\_requ \wedge ins\_requ \Rightarrow \circ(\neg course\,\mathcal{U}\;(ins\_data \wedge adc\_data \wedge \circ w\_data))$$

When the *ins_data* and *adc_data* are both determined, it is now also known if the aircraft is at a waypoint. If it is, a new waypoint is requested from the waypoint manager. This can be expressed by stating that some of the time, a waypoint manager request will be issued in the next time interval. When the waypoint request is issued, the waypoint data will be available eventually.

$$ins\_data \wedge adc\_data \Rightarrow \underline{\circ} w\_requ$$

$$w\_requ \Rightarrow \neg w\_data \wedge \Diamond w\_data$$

When the *adc_data* and *ins_data* signals are available, and the waypoint data is also available, then eventually the course signal will be computed

$$ins\_data \wedge adc\_data \wedge \circ w\_data \Rightarrow \Diamond(course\,\mathcal{U}\;n\_maxtimer)$$

Note that we need the **next** symbol in the above formula because we use the **next** symbol when setting *w_req* in the previous formulas.

8.2 Comments on Specification

General

The previous section specified the problem in temporal logic, with an overview graphical model showing the breakup into large-grained communicating processes. The model-checking part of the method consists of two parts:

- Building a Finite State Machine (FSM) model of the system using the tool State Machine Language (SML); and
- Checking the consistency of the temporal logic formulas against the constructed model using the Model Checking Tool (MCB).

Both of these tools were developed by Ed Clarke and others at CMU.

The preceding description is deceptively simple, and the FSMs have to be built in a modular fashion. This was done by building a model for each process, and checking the formulas against the standalone models. In some cases, of course, stubs were needed in these models of individual processes to account for external conditions. Once all of the models for each individual process were checked, larger grained subsystems were formed by composing (and changing) the individual process models. Each of these subsystems could then be checked in turn against the applicable temporal formulas. It was not possible to compose all of the models into a single system model, since this exceeded the capacity of the modeling tools. However, one model included the INS, ADC, waypoint manager, navigation, and autopilot. Another included the HUD display for the INS, and yet another included the radar display.

Lack of a "stop" condition

One of the obvious flaws in the requirements is the lack of a "stop" condition complementing the "start" condition for devices. This oversight caused some problems in two ways. First, the strong until ($\mathcal{U}$) as used in a number of the formulas states that the device **must** eventually fail and then eventually work again, which is not a particularly comfortable representation of reality. A "stop" condition would have made these conditions more reasonable. For example, we could state that the device was working until it failed or a "stop" condition arose. The fairness conditions would also have made more sense if a "stop" condition were present. We considered using a weaker form of **until** but resolved that if changes were to be made, the introduction of a "stop" condition would make more sense.

Understatement

One of the convenient properties of temporal logic is that one can in any formula understate the conditions on the expected **until** operator. For example, in the formulas for the display of the INS and ADC data on the HUD, it was straightforward to declare that when the icon representing the device is broken is being displayed and the remove condition arises, the "device broken" icon will be reset until the "display working" icon has been displayed. In actual fact, of course, the "device broken" icon cannot be redisplayed until the "display working" icon has disappeared. However, there is no need to state this more complicated condition, since it is implied by other conditions. For example, the safety condition does not allow the "device broken" icon and the "device working" icon to be displayed simultaneously.

Independence

In deriving a set of temporal formulas for a particular process, it is best to state the formulas as independently from the other formulas as possible, such that changes to the other processes will have minimal affect on the process being specified.

Generality

In deriving the specification for any process, it is best to state the formulas in the most generally applicable fashion. This allows the engineer doing the refinement the greatest degree of freedom in his optimization choices. For example, in the navigation subsystem, the second formula states that after the maxtimer has been set, eventually a request for data will be issued to the INS and ADC. It leaves the designer at the next stage the freedom of deciding the order of the two requests, and whether to do them in parallel or sequentially. If, for example, at a later stage in the development the two must be requested sequentially (for example, because they shared some hardware channel), then this can be done within the context of this formula, and the restriction can be added as a further formula at the appropriate time. This allows for postponement of design decisions until as late as possible in the development. The advantage of this is obviously that when maintenance changes occur, the highest level specifications should remain relatively untouched (since they are very general), and the changes can be introduced at the appropriate lower level.

8.3 Classification of Temporal Logic

8.3.1 Representation

Style — Temporal logic specifies the behavior of the system by describing how the logical conditions associated with the system change over time. In order to prove consistency between the formulas, a model of the system must be built in the form of a finite state machine. The temporal logic manner of description is declarative, while the model-building language is procedural.

Concurrency — There is no direct representation of processes or events in temporal logic, but the FSM models represent concurrent processes and the events are easily mimicked as changes in logical conditions. Each formula has to hold independently of the other formulas at all times, and many conditions can change in each instant in time. The formulas represent conditions which hold across a number of processes.

Communication — Temporal logic communicates by shared data, and other types of communication have to be modeled.

Non-Determinism — Temporal logic is non-deterministic, and the "for some paths" operators are especially useful in this regard, though non-determinism can also be achieved without using these operators.

Fairness — Temporal logic can explicitly express the fairness conditions required to constrain the non-deterministic behavior. Although fairness plays only a small part in the avionics example specification, there are many applications in which it has a significant role.

Modularity — The modularity of the temporal logic is at the level of individual formulas, while that of the modeling language is at the level of processes. The specifier is forced by the dual nature of the technique to break the system into processes to be modeled by FSMs, to apply the temporal logic formulas to these models for verification, and to compose the models into larger units which satisfy the combined formulas. Formulas obviously apply to different processes.

Time — The purpose of temporal logic is to describe sequences of changing conditions over time, and it is a perfect instrument for this purpose. However, it does not handle explicit timing requirements comfortably. The FSM model does, however, explicitly represent time as a sequence of clock ticks. Hence explicit wall clock requirements can be loosely specified in the logic and verified against the FSM.

Data — Temporal logic is restricted to representing logical conditions.

User Presentation — Temporal logic can describe only those aspects which can be described as logical conditions.

8.3.2 Derivation

Transformation — This does not seem to be particularly applicable to temporal logic. Transforming the FSM into an application would seem like a reasonably easy step to achieve, though it was not done in this study.

Elaboration — There are numerous places in the specification where a specification was created at one level and then elaborated to include more functionality, for example, by defining a perfect device, then an actual device, and finally a reliable device. This was done without changing in any way the perfect device definition, and the reliable device still satisfies the properties of the perfect device.

Composition — The specification is produced incrementally, with the temporal logic specifying the behavior of each part, an FSM model for the part being built, and the formulas verified against the model. The composition of the FSM models usually included making small changes to one or other of the models. The major problem, however, is one of scale; a single FSM representing the whole system could not be generated, since it exceeded the capability of the toolset. (One partial FSM was about 2200 nodes.) This problem may be alleviated with a new toolset, compositional state machine language (CSML) completing development, but we have not yet used the new method. In addition, the temporal logic is propositional rather than predicate, and hence lacks the ability to deal with enumerated quantifiers– this was not a problem in the chosen example, but is a problem in many systems.

8.3.3 Examinations

Equivalence — This is demonstrated by the verification that formulas are consistent against an FSM. Two sets of formulas can be true against a single FSM, yet still allow different behaviors. Similarly one set of formulas can be true against two different FSM models. There is no strong evidence that such equivalencing is particularly meaningful. On the other hand, we were able to demonstrate that the behavior of the comibination of the actual device and the reliable device was equivalent to the behavior of the perfect device.

Consistency — The consistency of the temporal formulas may be verified by checking them against a finite state machine. This may be done by considering the system to consist of a number of processes, specifying the behavior of each process and its interaction with other processes in temporal logic, building FSMs to model each process, and verifying the truth of each formula against the corresponding FSM. The FSMs may be coalesced into larger models and the verification continued. This approach allows the verification of consistency as specification proceeds. Unfortunately for this approach, problems of scale arise and it is often impossible to check a complete specification for consistency.

Over-Specification — In principle, once the engineer has a consistent specification, he can then determine if any parts of the specification can be derived from other parts. But we did not attempt to do this and believe that the difficulty of doing this outweighs any foreseeable advantages.

Implementation Bias — With a little self-discipline, the specifier can easily avoid adding too much detail to the specification, and it is straightforward to express the problem in a very abstract manner, so that decisions can be delayed until the appropriate point where optimization is based on obvious choices.

Ambiguity — Temporal logic allows the specifier to mimic any desirable ambiguity in the requirements, thus allowing a decision to be made at a later stage in the life cycle, when the grounds for an optimal decision may be clearer.

Under-Specification — We consistently under-specified parts of the system initially, and proceeded to improve the specification as we went along. There is no explicit way in which under-specification reveals itself; one has to rely on the good judgement of the specifier. It is reasonable to expect that guidelines to avoid under-specification could be generated.

Safety — Temporal logic encourages the separation of safety conditions from liveness conditions. Since the safety conditions express *conditions which must not occur*, this is very useful. It forces the specifier to organize his thoughts in terms of the conditions which should not arise in the system. The safety conditions were not very useful in deriving an FSM model of system operation, since they merely state the conditions which must not happen. However, they insure that the model built does not accidently have an undesirable side effect.

Liveness — Temporal logic allows one to express the liveness conditions describing the desired operations of the system, and its responses to changes in input conditions. The liveness conditions were useful in building the FSM model.

Validation — The duality of the specification, in terms of developing both a temporal logic specification and an FSM model helps in the validation process. One can demonstrate to the user both a declarative and a procedural description of the system. It is straightforward to have the customer express further scenarios, write these as temporal logic formulas, and verify them against the model for consistency.

Correctness — The correctness of the formulas against the model is a given in this process of creating both a specification and a model. Since no development of lower level objects (design, code) was attempted, it is not clear whether this is easy or difficult. The fact that an FSM model of the system was built and verified would lead one to expect that further development would be started from the FSM.

Chapter 9

Comparative Evaluation

This chapter summarizes our comments on the problem defined in Section 5 and presents a tabular summary of the classifications.

9.1 Comments on the Problem

The features of the avionics system that we believe are ambiguous or at least need some dialogue between the specifiers and the developers of the requirements are organized into three groups: unexpected behaviors, design decisions, and changes in requirements.

9.1.1 Unexpected Behaviors

As stated in the requirements, the HUD and MPD get updated when a device changes status from working to broken or vice versa. However, there is an additional requirement that the HUD will blink for at least 2 seconds when such an event occurs. We have assumed from the text that the display will not be changed for a minimum of those 2 seconds. This can lead to an anomalous and perhaps unexpected behavior. We made this discovery while creating the specification and, though we cannot say that we would not have discovered this anomaly when designing the system, we can say that we discovered the problem before any design occurred (and, in a real development, before any effort would have been wasted).

Consider the case, for example, when the INS fails to provide data on time (for whatever reason). Both the HUD and MPD in this case display that the INS is broken, the HUD flashing the information. Now, assume that before the 2 seconds have passed, the INS starts responding normally. At this point, the MPD starts displaying that the INS is working, but the HUD still flashes the message that the INS is broken. Thus, the pilot is presented with inconsistent information that may not be a desirable consequence of the requirements. Such anomalies are recognized during the process of validating the specification, a process we only carried out loosely. During the validation process, the validators will create scenarios and use the specification as a model to predict the behavior of the system. Once the scenario has been created, the system behavior can be accurately predicted. This is different from the case in which the requirements document is being validated, since ambiguities in interpretation of the document may lead to inaccurate predictions of behavior.

It may be that the requirements should be altered in order to eliminate this anomaly.

An interesting and unexpected (though not harmful) behavior was the discovery that the INS and ADC devices could be treated as having identical behaviors. Using the specification techniques to create an abstract specification of the system makes this similarity in behavior obvious. Such a clear statement of common behavior has benefits for implementation where communication between the MCC and, say, the INS may reuse the code implementing communication between the MCC and the ADC.

9.1.2 Design Decisions

For the purpose of developing our specifications, we have assumed that all of the devices work in some polled manner. This may be an incorrect decision on our part. Equally well, the devices could operate by placing somewhere into memory the required information and potentially even reporting on their own status. This would fundamentally change the specification. Were we developing a real system, we would expect to query the system architects to discover how the devices actually work before progressing very far in writing the specification.

Similarly we have assumed that we may have as many timers, of varying lengths as we need. Again, were we developing a real system specification, we would have to query the system architects concerning the nature of clock interrupts and the granularity of the timers that could be derived. It might be the case that the only interrupt available would be a simple cyclically generated pulse. This would fundamentally alter the structure of our specification.

The second of these decisions may be thought of as a specification convenience, in that it is possible to model all of these independent clocks using a single clock process generating interrupts to appropriate processes at the desired intervals. However, the issue of the manner in which we have assumed that the devices communicate information is a more fundamental issue. Our specification assumes that the devices have to be requested to provide information; an alternative form of device is one that continually updates an information store and that the MCC reads this store when it needs the information. Such a device behavior would alter (perhaps simplify) the structure of the specification and we would not expect it to be modeled by the behavior described in our specifications.

We had to resolve an ambiguity in the requirements document. The text

> the pilot and autopilot steering commands

in the system statement is ambiguous. It is not clear whether these are commands given to, or accepted from, the pilot and autopilot by the rest of the system. We decided that these were commands sent to the pilot and autopilot.

9.1.3 Choices in the Requirements

As was shown in the CSP specification, there were a number of alternative specifications for the ıadar displays. On the assumption that the requirements meant that the display had to be updated precisely every 200 ms., the most complicated specification was the one that most closely satisfied our interpretation of the requirements. However, this specification required the use of two timers that would make the resultant developed system more complicated, place a higher burden on interrupts, and would therefore be harder to test (or verify) for correctness. We could offer the system architects the alternative specifications pointing out the consequences of each of the alternatives, and an appropriate change in the requirements could simplify the specification and resultant system, potentially leading to a cheaper system.

9.2 Comparison of Classifications

We have, in the previous chapters, presented our specifications of the sample problem and classified the techniques according to our criteria. This section collates those classifications and presents them in a tabular form. We intend this section to provide a rapid way of searching for the technique most suited to a problem.

We repeat that the data for these tabulations arises from two major sources: first, from the sample problem specifications and, second, (to evaluate the techniques against criteria not addressed by our example system) from available literature describing the techniques. We would also like to state that the data does represent a value judgment on our part. We have attempted to maintain a consensus opinion for each of these data values in order to minimize the personal bias of each of the authors. Because each author wrote one of the specifications, each of us would have probably evaluated the specification techniques we used higher than deserved. Further, a consensus made a genuine comparison between the techniques in each of the classification criteria easier.

Before we describe the tabulations, we stress that it is easy to misinterpret the results presented, and so we will attempt to convey what information may be inferred from the tables, as well as the information that could be incorrectly inferred from the tables.

We use a numeric scheme to compare the techniques on specific criteria, not to score them. We use higher values to indicate a better performance of the technique in the particular classification. It would not be correct to sum the values for a technique and compare that sum against the equivalent value for some other technique and then to state "technique A is better than technique B." Differences in data values imply differences in the techniques and are not a precise measure of quality. It is fair to say that a higher value in a specific category does imply that, in our opinion, the technique performs better in that category than a technique with a lower value.

9.2.1 Interpretation of Table Data

As stated, we use numeric values to compare the techniques in each category. The following are the interpretations to be assigned to the numeric values.

0. The specification technique has no capability in the particular classification category.

1. The specification technique can be used to address the concerns of the category, but the manner of doing so obscures the expression of the system requirements.
2. The specification technique addresses the concept in a natural manner, but comments in the form of annotations to the specification may be required or a non-standard interpretation of the meaning of the constructs of the specification language might have to be used.
3. The specification technique addresses the concept in a natural manner, and there is no need for changes to the meaning of the constructs.

After each table, we discuss the relative importance of the fields and compare the specification techniques.

9.2.2 Representations

Table 9.1 is a collection of the criteria in the representation category for the three techniques evaluated in this report. Concepts such as "style" and "communication" are not evaluated numerically because it makes no sense to state, for example, that synchronous message passing is better than use of shared data.

Category	CSP	VDM	Temporal Logic
Style	Behavioral	Functional	Behavioral
Concurrency	3	1	3
Communication	Synchronous message passing	None	Shared Data
Non-determinism	3	3	3
Fairness	0	0	3
Modularity	3	1	1
Time	2	1	2
Data	0	3	0
User Presentation	0	2	0

Table 9.1: Representations

Concurrency — Both CSP and temporal logic were designed for the specification of concurrent systems, whereas with VDM we had to explicitly model the concurrent aspects of our sample system, thus obscuring the description of the function of the system. This explicit modeling led to specifications that we consider to be harder to read than the specifications in CSP and temporal logic.

Non-Determinism — The three techniques evaluated are well able to specify non-deterministic systems. In CSP, the user must explicitly introduce non-determinism through the use of the appropriate operator. This means that if a system is to be deterministic, the specifier may use a deterministic subset of the notation.

Fairness — CSP treats fairness as an implementation issue, and thus the concept cannot be expressed using the CSP notation. VDM does not explicitly address issues of fairness. However, since VDM has to model concurrency, any notion of fairness required can be added to the language. Temporal logic allows the expression of fairness constraints, both in the logic and in the model checking language SML. It should be noted that we did not find the inability to represent fairness to be a problem in the CSP and VDM specifications, indicating that fairness is not an issue in the creation of a specification, though it may become more important when examining the specification.

Modularity — Only CSP had a true notion of modularity with the division of the specification into a number of processes. There are some experimental ways to represent modularity in temporal logic and VDM; however, we have not considered such variants in this report due to lack of resources and an attempt to evaluate stable specification techniques. Modularity, however, was an important factor in the specification, and the ability to break the specification into a number of units was helpful.

Time — None of the techniques explicitly discusses time. We found that modeling timing concepts was straightforward in both CSP and temporal logic. Again, in VDM time had to be explicitly modeled; however, this was less natural than in CSP or temporal logic. We would add that although the expression of time used in the specifications might not be considered as fully formal, we do consider that the specification accurately and unambiguously expressed our requirements.

Data — Of the techniques investigated, only VDM has the capability of expressing complex data constructs. We note, though, that the problem itself does not contain many requirements for the specification of data constructs; the waypoint manager would have been the component to benefit most by such a capability.

User Presentation — Only VDM has the capability of specifying "screen real-estate," that is, the position of icons or characters on a screen. Our specification does not do this. We believe that we have specified the important information, the type of data to be displayed and the circumstances under which such data should be displayed. We consider that it is inappropriate to make explicit decisions about specific screen displays at this early stage in the development of a system.

9.2.3 Derivations

Table 9.2 displays the collected classification for the ways in which a specification may be manipulated.

Category	CSP	VDM	Temporal Logic
Transformation	3	3	3
Elaboration	2	3	1
Composition	3	1	1

Table 9.2: Derivations

Transformation — All the techniques are well able to transform specifications and provide appropriate transformation rules.

Elaboration — VDM was designed as a language for expressing both specification and design issues, so it is able to elaborate specifications easily. With some restrictions, CSP is also able to elaborate specifications, as we have shown in our specifications. However, using CSP for refinement of events requires the specifier to introduce into a design the same event names as were used in the specification, or at least some sequence of events in the design that represents the event described in the specification.

Composition — The use of the CSP parallel combinator proved to be important in the specification. Given the modular construction of the specification, a form of composition was important. It should be stated that we found this a natural way to develop the specification of the system. Since neither VDM nor temporal logic expresses concepts of modularity, each has only limited capability for composing pieces of specification to form new specifications.

9.2.4 Examinations

Table 9.3 lists the classifications for the types of examination that may be performed on the specifications.

Category	CSP	VDM	Temporal Logic
Equivalence	3	3	3
Consistency	3	3	3
Over Specification	1	2	2
Safety and Liveness	3	1	3
Determinacy	1	2	2
Validation	2	1	1
Verification	2	3	2

Table 9.3: Examinations

Equivalence — Since all of the techniques use transformation as a derivation process, they all provide mechanisms for determining that two specifications are equivalent.

Consistency — In all three methods, the consistency of the specification can be checked.

In principle, both CSP and temporal logic have sufficient rules to determine if any portion of the specification can be derived from some other portion of the specification. However, this is difficult to do and not easily automated. For VDM this may be performed in a localized fashion. For example, it is possible to show that part of a pre-condition is implied by other parts of the pre-condition. However, in the more global case of considering the entire specification, it is not clear how examinations for over-specification may be performed.

In terms of implementation bias, a CSP model-based specification introduces a great deal of implementation bias since it may be too explicit; trace-based specifications should introduce less bias. Both temporal logic and VDM, by use of more abstract specification styles, introduce less bias than the CSP model-based specifications.

Safety and Liveness — Temporal logic encourages the specification to contain standalone safety conditions describing what the system is not allowed to do. For example, the ***device working*** and ***device broken*** icons cannot be displayed simultaneously. This is stated quite independently of the liveness conditions specifying how these two icons are to be displayed as the device changes state. Since the liveness conditions in most significant snecifications are quite complicated, the safety conditions allow the explicit and straightforward specification of conditions that must not arise. This can also be done using traces in VDM and CSP; however, these methods do not especially encourage the specifier to write separate safety conditions.

All three methods allow the liveness conditions to be expressed, though the traces have been grafted on to VDM, making it more cumbersome.

Determinacy — The manner in which we have used CSP in our specification does not describe data values or the notion of passing data between processes. Thus, we cannot examine a process to see if it is determinate. For VDM and temporal logic we can perform some tests of the systems being determinate, though these examinations have not been performed in our example.

Validation — CSP validates that the specification meets the intentions of the requirements. This is true because CSP was developed to deal with processes and events, does so in an obvious way, and the composition of different modules is relatively painless.

Verification — One of the strengths of VDM that is not demonstrated in this report is the ability to refine a specification through design and implementation levels and to verify that the lower levels satisfy the upper level specifications.

9.3 Conclusions Based on Evaluations

Looking at the evaluations, we can conclude that the three techniques seem fairly equal. They each have strengths (or weaknesses) in different criteria. We note that some of the judgements such as user presentation are based on available literature rather than on our example specifications and may not be born out in practice.

Although the techniques appear relatively equal in terms of applicability to the specification of reactive systems, we do not believe this to be the case. This appearance of equality arises from giving each of the criteria equal weight, which may not be appropriate for all classes of system.

For example, we consider that the specification of the user's view of data being presented may not be appropriate at this level of specification when describing reactive systems. Indeed, to attempt to do so at this stage may be a case of making decisions too early in the life of the project and certainly a case of making the decision before other considerations that affect that decision are fully understood. We restate that ***for reactive systems, user presentation may not be an important criterion at the specification stage***; obviously, if the system being specified were a user interface system, the user presentation would be very important.

Similarly, in the avionics example, there was only one relatively uncomplicated data structure (the waypoint manager). We consider this to be typical of reactive systems; that description of data structures may not be as important a factor as the description of interactions between processes.

In the derivations category, we found that perhaps the most important of the criteria, in terms of constructing a specification, was the ability to compose pieces of specification to form a specification of larger systems.

Based solely on the work performed in developing the example specifications, we considered CSP to be the easiest technique to use, with temporal logic a close second. We felt that the extended VDM was not appropriate for this type of specification, though we believe that this may be a biased opinion and that a specification developed independently of the CSP specification may have provided better results for VDM.

Chapter 10

Conclusions

In the preceding chapter, we presented our evaluations of the three techniques investigated in this report. This chapter presents our conclusions with respect to the applicability of formal specification techniques based on our examples and our discussions with method developers and users. We also discuss possible extensions to this work, both in terms of widening the survey effort to cover more techniques and in terms of deepening the investigation into those techniques covered so far.

10.1 Conclusions

Our experience with the sample problem gives us confidence that use of an appropriate formal specification technique clarifies issues at the specification phase of a system's development and that formal specification techniques are applicable to the domain of avionics systems. Indeed, based on our work and the result of the survey of practitioners, we believe that formal specification techniques are ready for application to real systems.

Formal specifications provide a good basis for communication between team members. When we were comparing our specifications it was easy to understand exactly what other authors had written due to the precise nature of the descriptions. Thus, we were able to quickly see differences between the systems specified using the different techniques and to remove these differences.

There is a growing interest in the use of formal specification, especially in Europe. To date, formalists in the U.S.A. have concentrated on verifying pieces of systems; there is a growing interest in the use of formalism to specify entire systems.

That the classifications of the three techniques presented in this report showed differences between the techniques gave us some confidence in our classification criteria as being a useful guide to distinguishing between specification techniques.

There are differences between specification techniques. The choice of formalism will affect the structure of the specification, which will in turn probably bias the program developed from the specification. Thus, the initial choice of specification technique is a very important step in the development of a system.

In Chapter 9 we presented a summary of the classifications of the techniques we investigated. Based on this summary we would choose CSP for the specification of real-time, software dependent systems. However, we do not consider that CSP is so much better than temporal logic that developers already versed in temporal logic should switch to CSP. A second issue is that we have not considered the process of constructing a design from the specification. It is unclear how to construct a system design from a CSP specification, whereas the use of state machine models and temporal logic provide a more familiar method for the construction of a design. So although we prefer CSP for the construction and manipulation of a specification, considering the entire development life cycle, temporal logic may prove to be a more worthwhile approach.

10.2 Future Work

In this section we attempt to capture some of the ideas for future effort that occurred to us while performing this survey. These ideas seem to form natural extensions to the survey.

The most obvious extension is to construct more specifications of the same example using yet more specification techniques, specifically, techniques such as Petri nets and LOTOS, both of which are being used in the field of telecommunication protocol specification by various companies and researchers. Although LOTOS is similar in some ways to CSP, the addition of an algebraic style of specifying abstract data would mean that a LOTOS specification should be able to describe more of the system than the CSP specification. LOTOS is based in part on Milner's Calculus of Communicating Systems (CCS) [13], and an investigation into the differences between a LOTOS specification and a CCS specification would prove interesting. Another promising technique that should be applied to our problem is the UNITY approach [2].

While answering our survey questions, a number of specification technique developers suggested that they would like to anply their methods to our problem. So a related possibility is to pass out this report to anyone who wishes to use My Favorite Method (MFM). We would then act as a "clearinghouse" and at intervals produce updates to this report with the additional specifications and classifications of the techniques. We will attempt to keep the report object've by examination of the additional specifications and stating our views with respect to the evaluation of the technique.

In our classification criteria, we list a number of miscellaneous criteria which, though important, we did not intend to use in classifying the techniques. Some of these criteria, such as whether or not interesting subsets of the technique exist, could be investigated. If such subsets existed and the system could be specified using such a subset, then the effort involved in developing and subsequently reasoning about the specification might be reduced. Similarly, investigations into the differences in the reasoning systems of the specification techniques would prove valuable, since these differences may affect the information that can be determined about the behavior of the system from the specification.

Our classification shows that there are differences between the specification techniques investigated. To determine whether or not these differences are significant, we should take a system and classify the system according to our criteria. This should then provide us with some sort of suitability index of the specification techniques. We could then use the specification techniques on the problem to determine whether the different techniques match our expectations. If they did, we would have greater confidence that the selection of technique could be made on the basis of classifying the problem according to our criteria.

The problem we chose was sufficiently small that it could be specified in a number of different ways. It would be interesting to take a larger, more realistic problem and, using just one of the specification techniques, perform a more in-depth study of the use of the technique. This would determine both the strengths and weaknesses of the chosen technique, but would also provide an example of a formal specification of a real system.

The process of deriving a specification needs to be investigated. In our example, we first wrote a specification in CSP and the other specifications were derived, to some extent, from that CSP specification. However, the process of deriving the CSP specification was not studied and was, in fact, carried out fairly informally. An investigation into the process of deriving a specification and validating that the specification matches the user's requirements should suggest requirements on the language of the technique as well as the manipulations that may be performed.

For example, it would seem that a technique that lends itself to animation may be easier to validate than a technique that cannot be animated. However, this matter needs some experimentation.

Finally, we mentioned at the start of this report that we consider formal specification to be the basis of a formal development. A study should be made of how each of the techniques fits into the entire life cycle. For example, having derived the formal specification, how may it be developed into an executable program? How does the specification assist other parts of the entire development team, such as the quality assurance and maintenance groups? These questions need to be answered so that specification becomes not an end in itself, but a means of developing higher quality systems at a reduced total development cost.

Bibliography

[1] British Standards Institute BSI IST/5/50. *VDM Specification Language* (Draft), April 1989.

[2] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[3] E.M. Clarke, M.C. Browne, E.A. Emerson, and A.P. Sistla. Using Temporal Logic for Automatic Verification of Finite State Systems. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 3–26. Springer-Verlag, 1985.

[4] E.M. Clarke and O. Grümberg. Research on Automatic Verification of Finite State Concurrent Systems. *Annual Review of Computer Science*, 1987.

[5] Robert Firth, Bill Wood, Rich Pethia, Lauren Roberts, Vicky Mosley, and Tom Dolce. *A Classification Scheme for Software Development Methods.* Technical Report CMU/SEI-87-TR-41; DTIC: ADA200606, Software Engineering Institute, Carnegie Mellon University, November 1987.

[6] N. Francez. *Fairness.* Springer-Verlag, 1986.

[7] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[8] International Organization for Standardization ISO 8807. *LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, 1989.

[9] C.B. Jones. *Development Methods for Computer Programs Including a Notion of Interference.* PhD thesis. Oxford University, June 1981.

[10] C.B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, 1986.

[11] J.A.N. Lee and Karl A. Nyberg. Strategies for Introducing Formal Methods into the Ada Life Cycle. Technical Report SPC-TR-88-002, Software Productivity Consortium, January 1988.

[12] C. Douglass Locke, David R. Vogel, and Lee Lucas. Generic Avionics Software Specification. IBM Draft for the SEI, November 1988.

[13] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[14] Jan Pedersen and Mark Klein. *Using the Vienna Development Method (VDM) to Formalize a Communication Protocol.* Technical Report CMU/SEI-88-TR-26; DTIC: ADA204757, Software Engineering Institute, Carnegie Mellon University, November 1988.

[15] A. Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. In *Current Trends in Concurrency*, pages 510–584. Springer-Verlag, 1985.

[16] W. Reisig. *Petri Nets, An Introduction.* Springer-Verlag, 1985.

[17] Donald Sannella. A Survey of Formal Software Development Methods. Technical Report ECS-LFCS-88-56, Edinburgh University, July 1988.

[18] Władysław M. Turski and Thomas S.E. Maibaum. *The Specification of Computer Programs.* Addison-Wesley, 1987.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

1b. RESTRICTIVE MARKINGS
NONE

2a. SECURITY CLASSIFICATION AUTHORITY
N/A

2b. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

3. DISTRIBUTION/AVAILABILITY OF REPORT
APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
CMU/SEI-90-TR-5

5. MONITORING ORGANIZATION REPORT NUMBER(S)
ESD-TR-90-206

6a. NAME OF PERFORMING ORGANIZATION
SOFTWARE ENGINEERING INST.

6b. OFFICE SYMBOL (If applicable)
SEI

7a. NAME OF MONITORING ORGANIZATION
SEI JOINT PROGRAM OFFICE

6c. ADDRESS (City, State and ZIP Code)
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PA 15213

7b. ADDRESS (City, State and ZIP Code)
ESD/XRS1
HANSCOM AIR FORCE BASE
HANSCOM, MA 01731

8a. NAME OF FUNDING/SPONSORING ORGANIZATION
SEI JOINT PROGRAM OFFICE

8b. OFFICE SYMBOL (If applicable)
ESD/XRS1

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
F1962890C0003

8c. ADDRESS (City, State and ZIP Code)
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PA 15213

10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
63752F	N/A	N/A	N/A

11. TITLE (Include Security Classification)
SURVEY OF FORMAL SPECIFICATION TECHNIQUES FOR REACTIVE SYSTEMS

12. PERSONAL AUTHOR(S)
Patrick R. H. Place, William G. Wood, Mike Tudball

13a. TYPE OF REPORT
FINAL

13b. TIME COVERED
FROM ________ TO ________

14. DATE OF REPORT (Yr., Mo., Day)
May 1990

15. PAGE COUNT
128

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
avionics
concurrency
formal methods specification
formal methods survey
reactive systems

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Formal methods are being considered for the description of many systems including systems with real-time constraints and multiple concurrently executing processes. This report develops a set of evaluation criteria and evaluates Communicating Sequential Processes (CSP), the Vienna Development Method (VDM), and temporal logic. The evaluation is based on specifications, written with each of the techniques, of an example avionics system.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. □ DTIC USERS □

21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED, UNLIMITED DISTRIBUTION

22a. NAME OF RESPONSIBLE INDIVIDUAL
KARL H. SHINGLER

22b. TELEPHONE NUMBER (Include Area Code)
412 268-7630

22c. OFFICE SYMBOL
SEI JPO

EDITION OF 1 JAN 73 IS OBSOLETE.

SECURITY CLASSIFICATION OF THIS PAGE