NASA Conference Publication 3356

Lfm 97 Fourth NASA Langley Formal Methods Workshop

Compiled by
C. Michael Holloway and Kelly J. Hayhurst

Proceedings of a workshop sponsored by
the National Aeronautics and Space Administration,
Washington, D.C., and held at the Radisson Hotel,
Hampton, Virginia
September 10-12, 1997

September 1997

NASA Conference Publication 3356

Lfm97 Fourth NASA Langley Formal Methods Workshop

Compiled by
C. Michael Holloway and Kelly J. Hayhurst
Langley Research Center • Hampton, Virginia

Proceedings of a workshop sponsored by
the National Aeronautics and Space Administration,
Washington, D.C., and held at the Radisson Hotel,
Hampton, Virginia
September 10-12, 1997

September 1997

This publication is available from the following sources:

NASA Center for AeroSpace Information
800 Elkridge Landing Road
Linthicum Heights, MD 21090-2934
(301) 621-0390

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfield, VA 22161-2171
(703) 487-4650

General Chairman's Message

On behalf of the Langley Formal Methods Team, I welcome you to Lfm97, the Fourth NASA Langley Formal Methods Workshop. The primary purpose of our workshops has always been to bring together leading formal methods researchers and practicing engineers in an environment in which each group can learn from the other. The three previous workshops were limited to invited presentations, but we expanded this year's workshop to include 17 submitted papers. We believe that the program has something to offer to everyone, from those interested in the theoretical aspects of formal methods to those interested in the practical application of formal methods to help solve real problems. I hope that you will agree, and that you will find your time at Lfm97 both interesting and useful.

Many of the slide presentations that will be given at the workshop will be available on the World-Wide Web at <http://atb-www.larc.nasa.gov/Lfm97/>. Information on the NASA Langley formal methods program is also available on the web at <http://atb-www.larc.nasa.gov/fm.html>.

I look forward to meeting you during the workshop. Please let me know if there is anything that I can do to help you while you are here.

[illegible]

C. Michael Holloway, Lfm97 General Chairman
E-mail: c.m.holloway@larc.nasa.gov
Postal Address: Mail Stop 130, NASA Langley Research Center, Hampton VA 23681-0001

Lfm97 Organization

Workshop General Chairman
Michael Holloway, NASA Langley Research Center

Program Committee
Ricky Butler, NASA Langley Research Center (chairman)
Jim Caldwell, NASA Langley Research Center
Victor Carreño, NASA Langley Research Center
Ben DiVito, VíGYAN
David Guaspari, Odyssey Research Associates
Kelly Hayhurst, NASA Langley Research Center
Michael Holloway, NASA Langley Research Center (acting chairman)
Damir Jamsek, Odyssey Research Associates
Pat Lincoln, SRI International,
Paul Miner, NASA Langley Research Center
John Rushby, SRI International

Organizing Committee
Kelly Hayhurst, NASA Langley Research Center
Michael Holloway, NASA Langley Research Center
Lisa Peckham, NASA Langley Research Center
Pamela Verniel, NASA Langley Research Center

Sponsoring Organization
Assessment Technology Branch,
Flight Electronics Technology Division,
Research & Technology Group,
NASA Langley Research Center,
Hampton, Virginia, U.S.A.

Table of Contents

General Chairman's Message iii

Lfm97 Organization v

Why Are Formal Methods Not Used More Widely? 1
John Knight, Colleen DeJong, Matthew Gibble, and Luís Nakano

Plotting The Escape from The Tower: A Formalist's Practicality Primer 13
James Sutton

Proving Properties of Accidents 21
C. W. Johnson

Formalization and Analysis of the Separation Minima for Aircraft in the North Atlantic Region 35
Nancy Day, Jeffrey Joyce, and Gerry Pelletier

Modeling and Validating SAFER in VDM-SL 51
Sten Agerholm and Peter Gorm Larsen

Requirements Analysis of Real-Time Control Systems Using PVS 65
Bruno Dutertre and Victoria Stavridou

Reuse of a Formal Model for Requirements Validation 75
Robyn Lutz

Applying the SCR Requirements Method to a Simple Autopilot 87
Ramesh Bharadwaj and Constance Heitmeyer

A Tabular Language for System Design 103
Steven Johnson

Verifying Communication Related Safety Constraints in RSML Specifications 115
Mats P.E. Heimdahl

Towards High-Assurance High-Performance Program Synthesis 129
Michael Lowry, Steven Roach, and Jeffrey Van Baalen

On the Automatic Discovery of Loop Invariants 137
Andrew Ireland and Jamie Stark

PV: A Model-Checker for Verifying LTL-X Properties 153
Ratan Nalumasu and Ganesh Gopalakrishnan

Automated Deductive Verification of Parallel Systems 163
Hassen Saïdi

Robust Computer System Proofs in PVS 177
Matthew Wilding

Domain Checking Z Specifications 185
Mark Saaltink

Fundamental Hardware Design in PVS 193
James Leathrum, Jr.

Why Are Formal Methods Not Used More Widely?

John C. Knight   Colleen L. DeJong   Matthew S. Gibble   Luís G. Nakano

(knight | cld9h | msg7y | nakano)@virginia.edu
Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Abstract

Despite extensive development over many years and significant demonstrated benefits, formal methods remain poorly accepted by industrial practitioners. Many reasons have been suggested for this situation such as a claim that they extent the development cycle, that they require difficult mathematics, that inadequate tools exist, and that they are incompatible with other software packages. There is little empirical evidence that any of these reasons is valid. The research presented here addresses the question of why formal methods are not used more widely. The approach used was to develop a formal specification for a safety-critical application using several specification notations and assess the results in a comprehensive evaluation framework. The results of the experiment suggests that there remain many impediments to the routine use of formal methods.

1 Introduction

For many years, academics have claimed that the use of formal methods in software development would help industry meet its goals of generating a better software process and increasing software quality. The benefits that have been cited include finding defects earlier, automating checking of certain properties, and decreasing rework. Despite their popularity in academia and these claimed benefits, formal methods are still not widely used by commercial software companies. Industrial authors have expressed frustration in trying to incorporate formal technologies into practical software development for many reasons including: the perception that they add lengthy stages to the process; they require extensive personnel training; or they are incompatible with other software packages. Experts in formal methods have tried to analyse the situation and provide systematic insight into the reasons for this lack of acceptance [4, 7, 9].

The goals of the research presented here are to investigate this disparity between research and industry, and to determine what steps might be taken to increase the benefits realized by industry from formal methods. The initial hypothesis for the relative lack of use of formal methods was that industrial practitioners were reluctant to change their current methods and hence they overlooked the benefits that formal methods could provide. However, upon attempting to apply several formal techniques to a significant application in a case study, several shortcomings that are actually well-known impeded progress dramatically right at the outset. Examples of the difficulties encountered were that a single specification language could describe only a relatively small part of the system, and necessary tools were either not available, not compatible with other development tools, or too slow.

A new hypothesis was formulated in response to these findings. This second hypothesis was that formal methods must overcome a number of relatively mundane but important practical hurdles before their benefits can be realized. These practical hurdles arise from the current state of software-development practice. While the methods used in industry are rarely formally based, they are reasonably well-developed and understood. In order to be incorporated into industrial practice, formal methods must meet this current standard.

After formulating this second hypothesis, we set out to characterize these practical hurdles. A variety of evaluations have appeared in the literature written largely by researchers and with conclusions that tend to praise formal methods. However, further investigation of the evaluations found them lacking. The criteria used for evaluation tended to be vague and ambiguous. They were often derived from the author's experience with a particular project, with little substantiation that the list of criteria was in any sense complete or even applicable to a range of projects. In addition to defects in the criteria themselves, the methods of evaluation were subjective. All of this resulted in little insight into the general characteristics or utility of the formal method.

In this paper we summarize an evaluation framework for formal methods and present results of applying the framework to several formal techniques. The complete framework provides a comprehensive list of evaluation criteria together with the rationale for each. The basis of the evaluation framework is the need for any software technology, including formal methods, to contribute to one overriding goal—the cost-effective development of high-quality software. The results come from the development of formal specifications in several notations for a safety-critical application together with the application of a theorem-proving system to the application.

In the next section we summarize previous work both in the use of formal methods in software development and their evaluation. Then we present a summary of our evaluation framework, and we follow that with a summary of the results of its application in a case study[1]. Finally, we present our conclusions.

2 Previous Work

2.1 Formal Specification

Formal methods have made some inroads into industrial practice. A fairly large number of projects have been undertaken using formal specification in notations such as Z, VDM, PVS, and Statecharts. The most comprehensive report on such work is the well-known study by Craigen, Gerhart, and Ralston [2].

iLogix gives summaries of some of the industrial applications in which the Statemate family of tools has been used [12]. Cardiac Pacemakers, Inc., a unit of Guidant Corp., used Statemate to speed up development of defibrillators and pacemakers. Animations of the Statechart models allowed them to examine interactions between features before building a prototype and to receive feedback on the design from physicians. AOA Apparatebau used Statemate to design a new waste system for the Airbus A330 aircraft. Animation of the system allowed them to easily test single and multiple failures. Boeing used Statecharts in the development and validation of electrical, mechanical, and avionics systems as well as in their integration [14].

The Hursley Park laboratory of IBM UK has used Z in two major projects involving CICS [11]. The first of these was the development of a new release of the system and this release showed quality improvements corresponding to the sections which were formally specified. The second project was the formal specification of the application programming interface.

SCR is a formal method developed at the Naval Research Laboratory during an effort to re-engineer the flight control software for the A-7 aircraft [8]. Since its introduction, the SCR methodology has been expanded and more formally defined. It has been used on several industrial projects, such as a submarine communications system [10] and the certification of the shutdown system for a nuclear generating station [2], but never without the involvement of research or academic experts.

1. A complete report of the research can be found elsewhere [3, 6].

2.2 Formal Verification

Some industrial applications of formal verification have been reported using tools such as HOL, Nqthm, EVES, and PVS. Despite the large number of research projects that have used formal methods, the number of industrial projects that have utilized formal verification is quite small. Of the industrial projects that have taken place, the majority are research projects as opposed to actual practice producing real products.

By far the largest application of formal verification has been in hardware verification. Although hardware verification is not the subject of this paper, we note that successful application to hardware design is a strong indication that similar success with software is possible.

Specification analysis is the area within the software domain where theorem provers are being used. Lutz and Ampo applied mechanical analysis tools, specifically PVS, to the requirements analysis of critical spacecraft software [13]. This project consisted of specifying and analyzing the requirements for portions of the Cassini spacecraft's system-level fault-protection software. This project was more of an experimental study examining the applicability of formal methods and mechanical analysis to industrial software practices.

2.3 Evaluation of Formal Methods

Various authors have proposed evaluation criteria for formal methods and used them in a variety of ways. Rushby introduced some ideas intended to help practitioners select a verification system and also offered a set of evaluation criteria for specification notations [15]. Faulk also proposed a set of evaluation criteria for specification notations [5].

A comprehensive approach to evaluation and some results were presented by Ardis et al [1]. In this work, a set of criteria was established for the evaluation of formal specification notations and the set was then applied to several notations. The evaluation was performed on a sample problem from the field of telecommunications.

3 Evaluation Framework

3.1 Framework Basis

Our objective was to evaluate formal methods in a systematic manner, and an evaluation *framework* enabled us to generate a clear and complete set of evaluation criteria. The alternative was merely to develop a list of seemingly relevant criteria, but such an ad hoc list, though it might appear useful, leaves three important questions unanswered:

- Where did the criteria on the list come from?
- Why are the criteria on the list considered important?
- Is the list complete?

Questions such as these are not answered readily from a list of criteria. An investigation of the development of the criteria could answer these questions, but the framework summarized here provides a defendable list of criteria for the evaluation of specification languages and mechanical analysis tools.

The basis of our evaluation framework is *software development* and the associated *software lifecycle*. In other words, we seek to discover how formal methods contribute to software development and how they fit into the software lifecycle. The criteria used for an evaluation of formal methods should ultimately return to the question, "How will this help build better software?", where the definition of better is not restricted to a certain set of goals. There are two aspects to this question—first, what is needed to build software and, second, how can the use of formal methods augment the current development practices of industry to help build "better" software?

The question of what is needed to build software leads us to examine current practice. Current methods of software development divide the activities into lifecycle phases. Such a division focuses the developers' attention on the tasks that must be completed. But the lifecycle alone is not sufficient to describe the current process of building software since development is guided by program management activities. These activities continue throughout the lifecycle, monitoring and directing it.

An evaluation of formal methods technologies must examine their *compatibility* with current practice and the *actual benefits* they realize over the entire lifecycle. In order to be accepted by industrial practitioners, formal methods have to meet certain objectives:

- They must not detract from the accomplishments achieved by current methods.
- They must augment current methods so as to permit industry to build "better" software.
- They must be consistent with those current methods with which they must be integrated.
- They must be compatible with the tools and techniques that are in current use.

A further difficulty is that each project has different practical requirements. For instance, if the goal of a project is to be the first commercial vendor to develop a certain networked Java application, the reliability is less important than the speed of production. In this context, "better" software would probably imply a faster time to market, whereas in a safety-critical system, "better" would refer to greater reliability of the software.

We present here only a sample of the framework because of space limitations. The complete framework is in two major parts—one for formal specification and the other for mechanical analysis techniques. The framework is structured by the six phases of the software lifecycle—requirements analysis, requirements specification, design, implementation, verification, and maintenance—and for each phase a set of criteria that must be met have been identified and documented along with the rationale for each.

As an example of the way in which the framework operates, consider the oft-cited criterion that a formal specification language must be "readable". In practice, this is completely inadequate as a criterion because it is imprecisely defined and is untestable. In practice, a formal specification is read by engineers with different goals, skills, and needs in each phase of the lifecycle. What is readable to an engineer involved in developing a specification is not necessarily readable to an engineer using the specification for reference during implementation or maintenance. Thus, there are in fact *many* important criteria associated with the notion of a readable specification—the criteria are determined by the lifecycle phase and their relative importance by the product goals.

A selection of the criteria used for formal specification in the framework is presented in the next subsection. For brevity here, they are not broken down by lifecycle phase. In addition, they were chosen for illustration and are in no sense complete. In general, we use criteria for illustration that have not been noted by others, and we include the rational for each.

3.2 Criteria for Formal Specification

- *Coverage.*
  Real systems are large and complex with many aspects. For a specification notation to serve well, it must either permit description of all of an application

or be designed to operate with the other notations that will be required.

- *Integration with other components.*
  A specification is not developed in isolation, but rather as part of the larger software development process. Any specification technology must integrate with the other components of this process, such as documentation, templates, management information, and executive summaries. Often a system database and version control system are used. A part or all of the specification might be inserted into another document, so the specification must have a common file format. There will almost always be the need for a printed version of the specification. It should be easy to print the entire specification, including comments and non-functional requirements, in a straightforward manner. The formal method technology must be suited to the larger working environment.

- *Group development.*
  Every software project involves more than one person. During the development of the specification, it must be possible for several people to work in parallel and combine their efforts into a comprehensive work product. This means that any specification technique must provide support for the idea of separate development—a notion equivalent to that of *separate compilation* of source programs in languages such as Ada that have syntactic structures and very precise semantics for separately compiled units.

- *Evolution.*
  A specification is not built in one effort and then set in concrete; it is developed and changed over time. The specification technology must support the logical evolution of specification and ease its change. Incompleteness must be tolerated. Functionality such as searching, replacing, cutting, copying, and file insertion must be provided. Modularity and information hiding must be facilitated, so that for example a change in a definition is automatically propagated to every usage of it. Large scale manipulations must also be supported, such as moving entire sections or making them subsections.

- *Usability.*
  The ability to locate relevant information is a vital part of the utility of a specification. The ability to search, for example with regular expressions is valuable, but not sufficient. The specification is intended to serve as a means of communication. Annotating the specification with explanations, rationale, or assumptions is important for both the use of the specification in later phases and for modifications of the specification. This annotation must be easy to create and access, and it must be linked to a part of the specification, so that changes effect the corresponding annotation. The formal method should also provide structuring mechanisms to aid in navigation since the specification document is likely to be large. In a natural language document, the table of contents and index assist in the location of information; many tools allow them to be generated automatically from the text. Another useful capability seen in text editing is the use of hypertext links to a related section or glossary entry. Formal methods must address the usability of the resulting specification documents.

- *Compatibility with design tools.*
  A very strong relationship exists between the specification of a system and its design, therefore the tools should also be closely related. It should not be necessary for the designer to re-enter parts of the specification that are also part of the design. Either the specification tool must also fully support the design phase or it must be compatible with common design tools.

- *Compatible with design methods.*
  Just as the specification technology must be the same as or compatible with popular design tools, it must also be compatible with popular design methods. A difficult transition from a formal specification to say an object-oriented design is an unacceptable lifecycle burden.

- *Communicate desired system characteristics to designers.*
  In order to design the system, the designer must be able to read and understand the specification. Naturally, the specification should describe the normal operating procedure, any error conditions and the response that is appropriate, and non-functional requirements. The specification has to answer every question raised about the system by the designer (who is not likely to be an author of the specification).

- *Facilitate design process.*
  The more easily a design can be developed from the specification, the better. The use of a formal methods could speed up the design process by describing the system clearly and precisely. The designer must take the abstract description in the specification and describe how a real system is going to implement the specification. Information hiding must be maintained and the ability to view the system at varying levels of abstraction must be provided.

- *Implementation performance.*
  Implementation is hindered by any lack of clarity in the specification (and design) and misunderstandings

that cause rework. The more complete, precise, and detailed the specification and design are, the more smoothly implementation will go. An improvement in implementation efficiency is expected, therefore, from the use of formal specification because of its ability to achieve clarity and precision. This efficiency improvement is a critical element in the overall cost effectiveness that is realized by introducing formal specification into the lifecycle.

- *Support for unit testing in implementation phase.*

  A precise, complete, and accurate specification can greatly aid in the formulation of a unit test suite, perhaps through automatic generation. It should also minimize rework, since the requirements are well defined and unambiguously stated in the specification. Again, this expected benefit is a critical element in the overall cost effectiveness argument.

- *Maintenance comprehension.*

  An engineer responsible for maintenance should be able to study the specification and gain either a broad or detailed understanding of the system quickly. The documentation of non-functional requirements and design decisions is vital to a complete understanding of the system. The specification should be easy for the maintainer to navigate, accurate, complete, and easy to reference to find answers to questions. Structure, support for information hiding, and the ability to view the specification at different levels of abstraction are essential.

- *Maintenance changes.*

  When a change is made to an operational software system, the specification, the design, the implementation, and the verification must be changed. This is clearly facilitated if the different work products are carefully linked together so the changes needed in the code, for example, are very similar to those in the specification. In many current developments the specification is changed as an afterthought or not at all. Ideally the specification should be changed first to examine the effects of the change on the system. This requires that the specification be easily changed and that the document remains well-structured. Once changed, formal methods could allow static analysis, animation, or even the establishment of proofs of properties on the new specification before the change is propagated to the code. Clearly both validation and verification of a maintenance change are important and lifecycle support is required.

4 Experimental Evaluation

To evaluate the utility of a formal technique in industrial practice with any degree of statistical rigor, the technique must be tested in a large number and variety of projects. The projects chosen for study should encompass a wide range of application areas with a variety of goals. The population of engineers involved should consist of experienced industrial software practitioners, including clients, managers, designers, developers, technical writers, and maintainers. Finally, projects should be followed from conception through a period of maintenance, and measurements of productivity and product quality made before and after the addition of formal methods to the development process, so that a comparison can be made.

Unfortunately, a study with these characteristics would require many years, the cooperation of thousands of people, and is beyond the scope of this endeavor. The study reported here is quite modest and the results correspondingly modest. What we report: (a) is based on a single application of a particular type; (b) comes from a single development activity; (c) involves specifications that have not yet proceeded to implementation; and (d) is based on specifications that were not developed by experts.

We have applied the evaluation framework to a small but realistic safety-critical application in order to obtain assessments of various formal techniques. The application is the *University of Virginia Reactor* (UVAR), a research reactor that is used for the training of nuclear engineering students, service work in the areas of neutron activation analysis and radioisotope generation, neutron radiography, radiation damage studies, and other research [16].

The experimental evaluation was conducted by first developing three separate specifications for part of a control system for the reactor in three specification languages—Z, PVS, and Statecharts—and establishing proofs of safety properties of the PVS specification using the PVS system. During the creation of these artifacts, various observations and measurements were made by those involved in the development. Once the artifacts were complete, a second set of observations and measurements were made by the developers, computer scientists not involved in the development, and nuclear engineers and reactor operational staff.

4.1 The Case Study Application

The UVAR is a "swimming pool" reactor, i.e., the reactor core is submerged in a very large tank of water. The core is located under approximately 22 feet of water on an 8x8 grid-plate that is suspended from the top of the

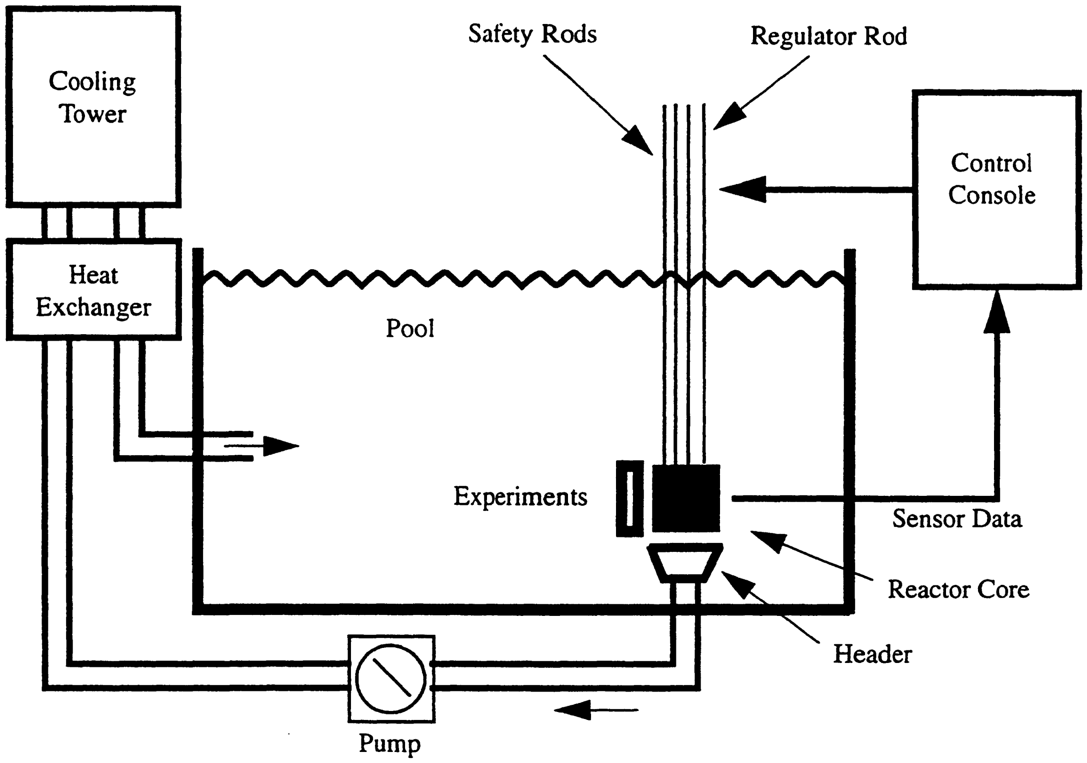


Fig. 1. - The University of Virginia reactor system.

reactor pool. The reactor core is made up of a variable number of fuel elements and in-core experiments, and always includes four control rod elements. Three of these control rods provide gross control and safety. They are coupled magnetically to their drive mechanisms, and they drop into the core by gravity if power fails or a safety shutdown signal (known as a "SCRAM") is generated either by the operator or the reactor protection system. The fourth rod is a regulating rod that is fixed to a drive mechanism and is therefore non-scramable. The regulating rod is moved automatically by the drive mechanism to maintain fine control of the power level to compensate for small changes in reactivity associated with normal operations [16].

The heat capacity of the pool is sufficient for steady-state operation at 200 kW with natural convection cooling. When the reactor is operated above 200 kW, the water in the pool is drawn down through the core by a pump via a header located beneath the grid-plate to a heat exchanger that transfers the heat generated in the water to a secondary system. A cooling tower located on the roof of the facility exhausts the heat and the cooled primary water is returned to the pool. The overall organization of the system is shown in Fig. 1.

The evaluation that we undertook involved the development of formal specifications for the following three components of a proposed new digital control system:

- the alarm system that alerts the operator of conditions needing attention;
- the logic associated with shutting the reactor down in the event of a possible safety problem (the SCRAM logic); and
- the activities undertaken by the operator to start the reactor operating.

For the sake of brevity, we only summarize the results of the study[1] in the following subsections. The first subsection address the specific evaluation criteria outlined earlier and reflect the experience of the developers. The next subsection itemizes specific results obtained from the nuclear engineers. The last subsection documents results obtained from computer scientists

4.2 Specification Assessment By Developers

- *Coverage.*
  Our experience with the UVAR specifications is similar to that of others—many things that have to be

1. Further details can be found elsewhere [3, 6].

specified are not covered by any of the notations we are using. A particularly significant example is the user interface. For systems like the UVAR, the user interface is complex, absolutely critical, and must be formally specified. Even though a model-based specification notation, like Z for example, is not really suitable for such specification, its integration with other notations is essential.

- *Integration with other components.*

  There is a complete lack on compatibility of the tools for the three notations with common text preparation systems. It is remarkably difficult, for example, even to get a printed copy of a specification in any of these notations. Worse is the fact that non-formal elements of a specification cannot be included in a specification and manipulated in a consistent way. A complete specification is more than the formal part. In the UVAR specification, for example, extensive technical background material has to be included.

- *Group development.*

  Statemate offers some support for version control and access control but neither Z nor PVS provide either. The latter is actually preferable because artifacts using the notations can then be handled by existing tools. The Statemate approach to projects and users is completely inconsistent with that which large projects are likely to be using for other purposes.

  Support for separate development (akin, as noted above, to separate compilation of source code in languages like Ada) is completely absent from all three notations.

- *Evolution.*

  The structure of both standard Z and the PVS specification notation offer no support for building specifications with any structure that facilitates evolution. Even the elementary notion of information hiding is absent. Statecharts offer some limited support using the hierarchical chart facility. In the UVAR specification, for example, there is extensive material related to physical devices that might change over time. Similarly, since the digital system is experimental, the concepts it includes are subject to change.

- *Usability.*

  Both the structure and the tool support associated with these three notations provide essentially no support for navigation and searching of a specification. The PVS specification for the UVAR, for example, defines literally dozens of identifiers. Reading, navigating, and changing a specification of even the UVAR's moderate size is difficult and error prone.

- *Compatibility with design tools and design methods.*

  Although specification and design are supposed to be separate activities, there is always a lot of overlap. The SCRAM logic for the UVAR, for example, is a significant part of the specification and a clear implementation structure is implied by the basic functionality required. Despite this, neither Z nor PVS provides a systematic link to any design methods or tools, nor do they explicitly avoid doing so in an effort to support generality. Statemate, on the other hand, embeds the notion of data flow diagrams into the basic specification structure and thereby biases designs towards structured design, an approach that is not universally preferred.

4.3 Specification Assessment By Nuclear Engineers

The following results were obtained by interview. For each of the specification notations, the notation and the associated specification were presented to a nuclear engineer and then the engineer was asked a series of questions derived from the evaluation framework. This process was repeated twice for each notation (making a total of six interviews in all).

The presentation of the notation was informal and brief, intended only to permit the nuclear engineer to understand the subsequent presentation of the specification. The presentation of the specification was intentionally very much like an inspection. As a result, we were able to get very specific information about how understandable the specifications were to domain experts. This is an important issue since, for the most part, human inspection is the primary vehicle for specification validation.

The results were quite unexpected and the detailed discussion resulting from the interviews was more enlightening than the specific answers to the framework questions. The majority of the following are observations that resulted from these discussions. The first three points are general and the remainder are language specific.

- *The role of the specification has to be understood.*

  Communicating with people from a different field of expertise is always difficult. In this experiment, a particularly troublesome issue was the role of the specification in software development—the nuclear engineers were not familiar with this role. One of the participants considered one of the specifications to be source code and wanted to see the execution to check correctness. Another considered it a summary that should be easy to read and not contain many details. The lesson learned was that it is vital that application engineers understand the role of a specification before trying to read or manipulate one.

- *Direct and indirect influence on the system are difficult to distinguish.*

  A common difficulty for the nuclear engineers in understanding the specifications was with the difference between direct and indirect influence on the state of the system. The nuclear control system is reactive—it is constantly making alterations in response to input received from sensors. A change in the height of a rod causes changes in the sensor values. The height of the rod can be altered directly by the system, but the sensor values change indirectly as a result of the movement of the rod.

  The formal specification notations designate parts of the system that can be influenced directly differently than those that cannot, for example Z uses primed identifiers to indicate items that are changed directly by operations. These designations were a constant source of questions because, along with the changes in the system from direct influence, there are expected indirect changes in the state of the system. By no means is this an argument to abolish the separate designations for items that can be directly influenced, rather to point out a difficulty in understanding these notations that is forgotten once the notation is familiar.

- *The use of symbolic constants is problematic.*

  An interesting anecdote involves the use of constants. It is customary, in fact preached, in computer science that constants should be defined in one place and given symbols so that no "magic" numbers are used throughout the rest of the system. The reasons are first that the numbers are unexplained, and second that every location of use has to be found if the constant is changed. To most of the nuclear engineers, this organization was clear and desirable since they did not have the constants memorized and the values would have to be checked against other documentation in an effort separate from the general perusal of the specification. However, one participant was confused by the use of constant identifiers rather than numbers because the specific values have important meanings in the context of the application.

- *There is no road back to natural language specification.*

  Once the nuclear engineers had experience with one or more of the formal specification notations, they said they would never trust a natural language specification again. They were impressed by the level of understanding of the system that was required to write the specifications and felt that with natural language they could never be sure that the words were not just copied down with little understanding of the system. While they would have liked some natural language to accompany the formal specifications, they wanted to retain the formalisms.

4.3.1 Z

- *Effective for communication.*

  The Z specification was described as meaningful and useful for communication by the nuclear engineers. One participant felt comfortable with the notation after a short period of time, no longer needed full translations of the schemas, and began to find errors in the specification. This participant felt that, after a few iterations of discussion and correction of the specification, he would feel that there was a mutual understanding of the system.

- *Mathematical notation is not familiar.*

  A surprising discovery was that the mathematical notation used in Z was not familiar to the nuclear engineers. One participant expressed the desire for a glossary of symbols, including for all, there exists, and implies. Another asked why words, which are universally understood, were not used in place of the symbols.

- *Validation by inspection was effective.*

  In this case, the presenter of the Z specification was not the author, but another computer scientist familiar with the project, and the process of explaining the specification to the nuclear engineers uncovered errors. This helps to substantiate the generally accepted view of the community that inspection is valuable and cost effective.

4.3.2 PVS

- *Looks like source code.*

  The first impressions of the PVS specification were that it looked like source code, it was too long, and there was too much text. One participant said he did not even want to try to read it. Another criticism from another participant was that there were too many variables leading to confusion.

- *Validation by inspection was effective.*

  Although one of the participants was not comfortable reading the PVS notation, a detailed explanation of the specification facilitated useful discussions that identified errors in the specification and in the specifiers' understanding of the system. One way that this occurred was that the nuclear engineer asked questions to check the model. He identified a misunderstanding of the power levels of the reactor that necessitated the redesign of a section of the specification. If this error had not been found until the system had been implemented, it would have been impossible to increase the power level of the reactor above about half of the value at which it is licensed to operate. The use of meaningful variable names was key to the understanding of the specification.

In addition to errors found by the nuclear engineers, presenting the specification caused the specifier to discover an error in his own specification.

4.3.3 Statecharts

- *Effective for communication.*
  After less than an hour of introduction to the Statecharts notation and specification, one participant was no longer intimidated by the notation and was able to understand the specification without assistance. The graphical notation was appealing, as well as the obvious flow of the system following the arrows. The cliche "a picture is worth a thousand words" was used repeatedly. The structure of the specification was much more evident in Statecharts than the other two notations because of its hierarchical nature.
- *Difficult to search and navigate.*
  In a very detailed examination of the specification, participants complained of the difficulties of knowing the state of the whole system at once and of identifying the results of actions since the actions could affect any page of the specification. Whenever the details of a state were included in the diagram of that state rather than being saved in another file, the lack of abstraction seemed to be confusing.
- *Easy to learn.*
  Within two hours of discussion of the specification, the participants displayed the desire to learn the syntax of the notation in order to understand the subtleties of the specification. A large number of errors were identified during the discussion of the specification and the need for additional robustness was evident. The participants found the specification easy to understand with the explanation from the specifier and felt that they could then continue to study it alone. They also felt comfortable enough with the notation that, if there were changes to be made to the system, they felt they could write Statecharts of the proposed changes.
- *Specification is superior to existing documentation.*
  The participants from the nuclear reactor staff felt that the specifiers understood the system better than most of the operators. They felt that they could eventually come to an agreement that the Statechart specification correctly described the system and did not feel that they would have the same confidence with an English document. They said that this specification had the potential to be used in the training of their operators and perhaps even to replace their SAR which describes the control of the nuclear reactor. These are significant comments.

4.4 Specification Assessment By Computer Scientists

The participants in this portion of the study were seven computer science students. There was one undergraduate, four students working toward or finished with a master's degree, and two Ph.D. candidates. Two participants had a year or less work experience developing software, three had one to five years experience, and two had more than five years of work experience. All had knowledge of the C programming language. Regarding their experience with formal specification methods, four had no experience prior to this study, two had a segment of a course, and one had an entire course. All had some, but not extensive, knowledge of basic science and engineering and little to no knowledge of nuclear reactors.

4.4.1 Z

- *Fairly easy to understand, navigate, and search.*
  The Z specification was generally well-structured and this aided the participants in understanding and searching the specification. However, one participant expressed difficulty locating the definitions of types since they are not defined near their use and another suggested that the specification would be easier to search, navigate, and use for reference if there were a table of contents. The participants felt strongly that Z would aid communication about the system, however they considered it only average for use in the maintenance phase as an introduction to the system and as a reference document about the system. Familiarity with logic symbols, the smallness and simplicity of the notation, and the natural language descriptions aided the participants in understanding the specification.
- *Reasonably easy to learn.*
  None of the participants felt very confident in their ability to use Z after this short introduction. A few of the participants felt that Z was harder to learn than a programming language, but most felt that it was as easy or easier to learn. Difficulties in learning Z were attributed to the mathematical notation, the unusual delimiters of inputs and outputs, and the unfamiliarity of the notation in general. No one thought that Z was too large a notation and almost everyone thought the complexity of the notation was appropriate for speci-

fication.

- *Implementable.*

  After a thorough inspection of the description of the SCRAM logic in the specification, everyone saw ways that it could be implemented. No one was sure that the description was complete, however. Some participants found errors in the specification. Upon quick perusal of the rest of the specification, almost everyone felt that all the features of the notation were implementable. It was practically unanimous that Z provided the appropriate level of detail about the system for a specification.

4.4.2 PVS

- *Difficult to understand, navigate, and search.*

  Although PVS is structured a lot like source code in C (of which all participants claimed a lot or extensive knowledge), it received low ratings in the areas of structure, understandability, and searching. One participant cited the formatting as hindering understanding. It was deemed average to bad for use during the maintenance phase as an introduction to the system or as a reference document. The answers were widely varied as to whether PVS would aid communication between people involved in the software development process.

- *Ease of learning mixed.*

  None of the participants felt confident using PVS after this short introduction. Most felt that PVS was as easy or easier to learn than a programming language, but a few felt that it was harder to learn. No one thought that the PVS notation had too few features and most people thought that it had the appropriate amount of complexity, while a few felt that it was too complex. Difficulties in learning the notation were attributed to the size and complexity of the notation and the difficulty in understanding the keywords and constructs. However, some participants felt that the keywords and constructs were easy to learn and PVS was similar to other notations with which they were familiar.

- *Implementable.*

  After examining the scram logic in the PVS specification, everyone saw ways that it could be implemented, but a few saw some problems. No one was certain whether the description of the scrams was complete. After a quick inspection of the rest of the specification, the participants felt that everything was implementable. There was a wide range of responses when asked whether PVS provided the appropriate level of detail for a specification.

4.4.3 Statecharts

- *Easy to understand.*

  Statecharts was described as well-structured and this aided the participants in understanding the specification. Difficulties in understanding the specification were attributed to the global nature of events and the division of the specification over many pages. The responses indicated strongly that Statecharts would aid communication between people in the development of a software product.

- *Difficult to navigate and search.*

  The structure of Statecharts aided in searching, but one participant noted that the specification would be easier to navigate, search, and use as reference, if it had a table of contents. It was deemed average for use in the maintenance phase as an introduction to the system and as a reference document.

- *Fairly easy to learn.*

  The participants did not feel confident in their ability to specify a system using Statecharts at this point. Difficulties in learning Statecharts were attributed to the notation being unlike any notation they had seen before and the constructs being difficult to understand. However some people felt that Statecharts was easy to learn because the notation was familiar, graphical, small and simple, and the constructs were easily understood. Most of the participants thought that Statecharts was as easy or easier to learn than a programming language.

- *Implementable.*

  After studying the scram logic described in the Statecharts specification, everyone saw ways to implement it, however no one was certain the description was complete. After a quick survey of the specification, almost every participant thought that all the features of the notation were implementable. It was almost unanimous that Statecharts provided the appropriate level of detail about the system. Most of the participants thought that Statecharts notation contained the appropriate level of complexity.

4.5 Mechanical Analysis With PVS

The PVS specification was subjected to limited analysis with the PVS theorem-proving system. The purpose was to evaluate the difficulties involved in dealing with this modest sized specification and to learn what the practical issues might be that are limiting the wide-spread application of mechanical theorem proving. This part of the study was performed by the authors.

The conclusions from this part of the study fall into two basic categories. The first concerns the "method" part of formal methods. Devising the requisite theorems and developing a proof strategy for them proved to be a significant challenge and there is no real "method" that can assist the specifier.

The second category of conclusions is in the area of tool performance. Although the PVS system is very powerful, this power is difficult to use. Some of the difficulties with the tool are the following:

- *Syntax and type checking are laborious because the system reports errors individually.*
- *The specification interface is very awkward to use since, for example, it does not permit many display items to be customized, does not provide status information conveniently, and lacks expressivity.*
- *Navigation through a specification using the toolset is extremely labored.*
- *The variation in delays that occur with different user actions makes interactive use very difficult.*
- *The theorem prover interface is awkward to use since, for example, information is not displayed conveniently during proof attempts, proofs are re-displayed after invalid commands, and certain commands generate an overabundance of output.*

These and many other observations lead us to conclude that the practical adoption of mechanical theorem proving by industrial practitioners is being severely limited by one major problem—the difficulty of determining what should be proved to gain confidence in a specification, and one relatively minor problem (or at least a problem that should be minor)—the relatively poor usability of the toolset.

5 Conclusions

Our assessment of the formal technologies that we used is that there are many practical barriers to their routine use in industrial software development projects. In most cases, this will not be "news" either to the developers of the techniques or the community at large. In fact, some developers have been quite open in their discussion of the pragmatic weaknesses of their technologies. Thus, we offer little specific new information. However, the accumulation of all the different criteria in our framework together with their systematic development provides a clear picture of what is needed to achieve success in industrial applications. It is important to keep in mind that the criteria are not sufficient, merely necessary.

Several of our results are surprising but two are repeated here because of their significance. Both of these comments arose during the interviews with the nuclear engineers:

> *They felt that they could eventually come to an agreement that the Statechart specification correctly described the system and did not feel that they would have the same confidence with an English document.*

> *Once the nuclear engineers had experience with one or more of the formal specification notations, they said they would never trust a natural language specification again.*

These are *very* positive comments although when reading them it must be kept in mind that the nuclear engineers involved had been exposed to this technology for only a short time. However, these remarks provide strong motivation for continued work in the area of formal methods.

Perhaps the most important conclusion to be drawn from this work is that the framework provides a detailed research agenda for workers in this field. The potential is tremendous but unless the criteria in the framework are met by specific formal methods, their chance of widespread acceptance is remote at best.

6 Acknowledgments

It is a pleasure to acknowledge those who patiently described the requirements of the UVAR control system including Tom Doyle, Bo Hosticka, Don Krause, Bob Mulder, and Reed Johnson. The participants in this study also deserve recognition: Roger Rydin, Reed Johnson, William Dixon, Jim Kelly, Bob Mulder, Tom Doyle, Emily West, Dale Newfield, Russ Haddleton, Craig Chaney, Cassie Trontoski, Meng Yin, and Chenxi Wang. This work was supported in part by the National Science Foundation under grant number CCR-9213427, in part by NASA under grant number NAG1-1123-FDP, and in part by the U.S. Nuclear Regulatory Commission under grant number NRC-04-94-093. This work was performed under the auspices of the U.S. Nuclear Regulatory Commission. The views expressed are those of the authors and do not necessarily reflect the position or policy of the U.S. Nuclear Regulatory Commission.

References

[1] Mark A. Ardis, John A. Chaves, Lalita J. Jagadeesan, Peter Mataga, Carlos Puchol, Mark G. Staskauskas, and James Von Olnhausen. A Framework for Evaluating Specification Methods for Reactive Systems: Experience Report. *IEEE Transactions on Software Engineering*, 22(6):378–

389, June 1996.

[2] Dan Craigen, Susan Gerhart, Ted Ralston. *An International Survey of Industrial Applications of Formal Methods.* U.S. Department of Commerce, March 1993.

[3] Colleen L. DeJong, Matthew S. Gibble, John C. Knight, and Luís G. Nakano. Formal Specification: A Systematic Evaluation. Technical Report CS-97-09, Department of Computer Science, University of Virginia, Charlottesville, VA, June 1997.

[4] David Dill and John Rushby. Acceptance of Formal Methods: Lessons from Hardware Design. *IEEE Computer*, 29(4):23-24, April 1996.

[5] Stuart Faulk. Software Requirements: A Tutorial. *Technical Report NRL/MR/5546—95-7775*, Naval Research Laboratories, November 14, 1995.

[6] Matthew S. Gibble and John C. Knight. Experience Report Using PVS for a Nuclear Reactor Control System. Technical Report CS-97-13, Department of Computer Science, University of Virginia, Charlottesville, VA, June 1997.

[7] Anthony Hall. What is the Formal Methods Debate About? *IEEE Computer*, 29(4):22-23, April 1996.

[8] Kathryn L. Heninger. Specifying Software Requirements for Complex Systems: New Techniques and Their Application. *IEEE Transactions on Software Engineering*, 6(1):2–13, January, 1980.

[9] C. Michael Holloway and Ricky W. Butler. Impediments to Industrial Use of Formal Methods. *IEEE Computer*, 29(4):25-26, April 1996.

[10] Constance Heitmeyer and John McLean. Abstract Requirements Specification: A New Approach and Its Application. *IEEE Transactions on Software Engineering*, 9(5), Sept. 1983.

[11] I. Houstan and S. King. CICS Project Report: Experiences and Results from The Use Of Z In IBM. *VDM '91. Formal Software Development Methods, Vol. 1: Conference Contribution.* Lecture Notes in Computer Science, Volume 552, Springer Verlag, 588–596.

[12] Some industrial uses of iLogix tools can be found on-line at: <*http://www.ilogix.com/company/success.htm*>, 1997.

[13] Robyn Lutz and Yoko Ampo. Experience Report: Using Formal Methods For Requirements Analysis Of Critical Spacecraft Software. In *Proceedings of the 19th Annual Software Engineering Workshop*, pp. 231–248, Greenbelt, MD, December 1994. NASA Goddard Space Flight Center.

[14] C. R. Nobe and W. E. Warner. Lessons Learned from a Trial Application of Requirements Modeling using Statecharts. In *Proceedings the Second International Conference on Requirements Engineering*, pp. 86–93, April 15–18, 1996.

[15] John Rushby. Formal Methods and the Certification of Critical Systems. *Technical Report CSL-93-7, SRI International*, December 1993.

[16] University of Virginia Reactor, The University of Virginia Nuclear Reactor Tour Information Booklet can be found on-line at: <*http://minerva.acc.Virginia.EDU/~reactor/*>, 1997.

Plotting The Escape from The Tower: A Formalist's Practicality Primer

James M. Sutton

james.m.sutton-iii@boeing.com
Boeing North American
1800 Satellite Blvd., Mail Stop DL23
Duluth, Georgia 30155

Abstract

Formality will eventually become the norm in software development. It will happen for the same reasons that formality has become the norm in every other engineering discipline: Quality, confidence, objectivity, and even cost only make their greatest strides when mathematics becomes the basis for a discipline.

The theory of software formality has matured greatly in the last ten years. Enough is now understood to make formality useful not just to academia but also to industry. The main impediment to widespread adoption is financial: as formal methods are typically applied, they cost their users more than they pay them back. Nobody gets career credit for "doing the right thing" to the detriment of their company.

Thus, formality will only be adopted when it pays its own way. This is already happening on a few projects. Achieving payback requires treating the software lifecycle as an integrated whole of which formality is just one aspect.

The formal methods available at present share similar strengths and weaknesses. An effective formal process takes advantage of those strengths and compensates for those weaknesses. Compensation comes through integration...using other methods and approaches to "fill in" where formal methods are weak, while allowing their strengths to continue to shine.

In such a context formal methods have proven, on real industrial projects, they can benefit everyone and become the best *business* option. Adoption then follows without a need for further cajoling or coercion.

This paper will explore the use of formality in a practical and self-justifying way, in the realistic industrial setting. The principles given will be illustrated from Boeing North American's development of the Brimstone missile system, and other programs.

1 Philosophical Foundations

Practicality always boils down to economics. If a method is so unpleasant or difficult to use that workers resist it or simply never become proficient, productivity suffers and money is lost. If a method requires such expensive tooling that resource costs can never be recouped before technology evolution has obsoleted the tool, and if productivity gains from that tool's use are less than the capital loss, again there is no net benefit.

Therefore, the "gatekeeper" for adoption of formality is that the benefits from its use must exceed its costs. The challenge is to find ways of using formality which maximize benefits and minimize costs until breakeven is achieved. As one would expect, the benefits of formality come from its strengths, while the costs come from its weaknesses.

The strengths of formality are already benefiting users regularly. They are well known and understood: assurance of internal correctness, consistency, completeness, traceability and so forth. Much work is ongoing to further improve these strengths.

The weaknesses of formality are also well known. They include labor intensiveness, poor communicability to others than formalists. lack of scaleability to handle large problems, and poor efficiency in the face of system change.

These weaknesses typically receive somewhat less attention than do the strengths, perhaps because we assume that little can be done about them for now (placing our hopes on future breakthroughs in formal technology to somehow reduce the problems). However, the assumption of present helplessness is incorrect.

Since significant benefits are already being obtained from the strengths, most of the challenges in making formality practical are currently found in ameliorating its weaknesses and thus reducing its costs. Not only are the weaknesses reducible, they have on occasion been effectively overcome to make formality a net contributor to project financial success.

In industrial process engineering, one combines differing methods, tools or procedures so the strengths of some will always be offsetting the weaknesses of others. The goal is to yield products of higher quality and better profitability than could have been obtained through the use of any of the methods, tools or procedures in isolation.

When formal methods are incorporated into the industrial software development process, the formal methods chosen must either be those whose weaknesses are most easily ameliorated, or those which possess weaknesses to the least degree.

Then synergies must be found with non-formal (at least, for now) methods or approaches to offset the formal-method weaknesses. The formality must be "framed" in such a way that, despite its low scaleability, large programs can still be created. Since change is a primary characteristic of most product developments, formal methods must be couched such that system change propagates as little as possible and thus has minimal effect on formal product elements. And so on.

This is the approach being taken on significant portions of Boeing North American's development on the Brimstone missile system, as well as the approach taken on the mission software of the Lockheed Martin C-130J airlifter (in which the author also participated [1]). Resonance with aspects of this viewpoint have also been found on certain other projects, primarily the NRL's (Naval Research Laboratory) A-7 Avionics Upgrade Program [2], Allied Signal's TCAS system [3], and Rockwell Avionics and Communications Software Requirements Engineering Project [4].

Discussing these principles in a paper for a formal methods conference can give the impression that the rationale for creating such an approach is to bring formal methods into the software development mainstream. Nothing could be further from the truth. The rationale is to make industry more productive and competitive. Formal methods are of interest to industry only inasmuch as they contribute to that goal. Ultimately, the ability of formal methods to now do so is one of the best things that has ever happened to the discipline of formality.

2 Process Guidelines

There are two overarching guidelines for creation of a software lifecycle which supports the goal of industry, i.e., that increases productivity and competitiveness. Those guidelines are:

- correctness by construction
- verification-driven development

2.1 Correctness by Construction

Software errors are one of the biggest cost drivers in industry. Software errors have many kinds of costs. Verification, with its associated activity of correction, is almost always the most costly and time-consuming activity in software development.

The more errors there are present, the greater the cost in finding, removing, and confirming removal of them. Errors also cost in lost customer confidence and good will, creation of an adversarial relationship with regulators and government, and in the fallout from failures in fielded systems. And since it is in general impossible to detect all errors in a program, the more errors injected as the code is produced, the more that will remain at delivery.

Industry has historically assumed that error creation was unavoidable. This assumption is no longer valid. Errors can be largely avoided through a "correctness by construction" development process.

"Correctness by construction" means that the process must create software that, to the maximum extent practical, is inherently caused and constrained to be correct by the development processes used. Without formality, this would be impossible. With formality alone, it is unaffordable. Only by an integrated lifecyle of formal and other (currently non-formal) methods, tools and procedures can correctness by construction be implemented in the industrial setting.

Experience has shown that such a lifecycle need be no more expensive than the typical lifecycle; indeed, it has provided productivity above industry norms on industrial programs like the Lockheed Martin C-130J [5].

The selection and synthesis of complementary lifecycle methods will be examined shortly.

2.2 Verification-Driven Development

In the ideal world, a software development process which reliably prevents any errors from being injected into its developed systems need not spend anything on verification. Since no real-world process will attain such perfection, some errors will continue to be found in newly-developed software.Therefore, verification will continue to be required even in a "correctness by construction" lifecycle.

Unfortunately, the nature of verification is not changed by the correctness approach. Verification continues to the most expensive task in the software lifecycle. Correctness by construction simply lessens the number of correction/reverification cycles required (which is nevertheless a great productivity booster).

Verification therefore becomes the other great opportunity to reduce costs. Formality is a great enabler for such savings, by playing a role in the efficient static analysis and testing of software. The lifecycle is then optimized to enable efficient "endgame" verification, while not ignoring other concerns such as execution efficiency.

Verification is so seldom addressed as part of lifecycle process planning, that a little attention early on can have dramatic effects late in the project. While the remainder of this paper will give process-wide steps which implement this guideline, smaller things can also make a great difference. For instance, restricting the coding standards to make the produced code compatible with the best formal and test tools is very important.

3 Principles of Practicality

We will now explore some principles for integrating formality into the industrial software lifecycle, and thus economically implementing correctness by construction and verification-driven development.

These principles are presented without any claims either that they are exhaustive or that they "the only way" to use formality practically. However, these principles have been proven to enable the rather extensive use of formality in real industrial programs while yielding some very commercially significant benefits: greater software productivity than traditional, non-formal development (by approximately a factor of two), and much higher resulting software quality (by a factor of ten fewer anomalies, which includes both errors and inconsistencies).

These principles will be illustrated by naming methods which support them, as well as examples from industry.

The principles we will discuss are:

- Factorization of product and process
- Change-driven design
- Closed-loop formality

3.1 Factorization

The main principle for dealing with formality's scale-up limitations is factorization. Factorization is defined here as the decomposition of both product and process into small, relatively self-contained elements that are of an efficient scale for both individual human effort, and for use with formal methods. Factorization applies to both the product and process.

The primary method used for product factorization is domain engineering. Domain analysis allows factoring a large problem space into relatively small, manageable and naturally interrelated collections of requirements.

The SPC's (Software Productivity Consortium) CoRE (Consortium Requirements Engineering) method combines formal requirements specification with mechanisms for factoring the problem space as a result of domain analysis [6]. Few formal requirements methods support factoring so directly.

CoRE has other advantages in a practical industrial setting. Requirements are recorded essentially in an algebraic format whose use is easily learned, and whose representation is easily understood even by non-users. CoRE shares this characteristic with the NRL (Naval Research Laboratory) SCR (Software Cost Reduction) method [7], and the T-VEC method originated at Allied Signal.

Figure 1 shows an example CoRE requirement, captured in the syntax of a Cadre Teamwork control specification table.

The "abstracted output" is a discontinuous function of the "abstracted inputs" (actually, it is literally a relation, though the distinction is not critical to the purposes of this paper). Therefore the output must be defined across all combinations of subranges of the inputs. Each row in the table (besides the top label row) defines the function across one combination of subranges. All rows together fully define the output. The breaks between rows represent the points of discontinuity, often called "boundaries" in the testing arena.

Some other common formal requirements methods, such as Z and VDM, lack support for factoring and general communicability. This makes their use in the industrial setting more problematic. Other amelioration strategies would need to be found to overcome their weaknesses in these areas.

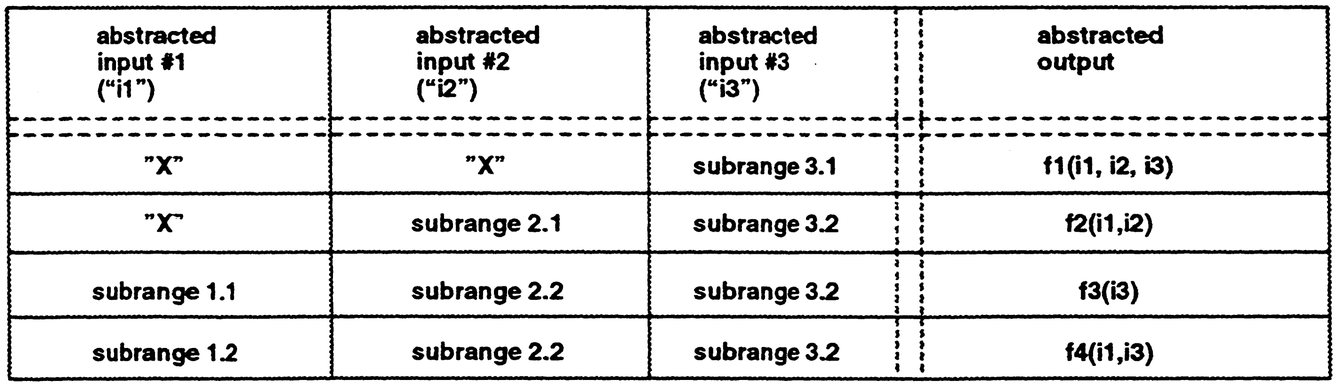
An architecture should be factored according to the nature of the solution domain, and not, in general, the problem domain (as frequently happens in object orientation). Problem-domain factorization leads to systems inefficient to develop and to execute.

Process factorization divides the analysis through verification of a system into a sequence of steps. Each step produces its own well-defined product according to very strict rules of production, with the assistance and rule enforcement of software tools tailored for the purpose. Because the steps are small, confirmatory Verification and Validation (V&V) can be performed for each step in an economical manner.

In an overall software lifecycle, these rules of production can not as yet always be completely formal. This is due primarily to the current immaturity of early-lifecycle formal methods. In all cases, however, the production rules should be specified as rigorously as is practical, primarily to preserve the quality of products as they pass from phase to phase.

Additional factoring of the process can be obtained through means such as a spiral lifecycle process (as in the SPC's ESP or Evolutionary Spiral Process [8]).

A variation on this theme, that fits within a more typical waterfall of "V" lifecycle, is narrow-slice development. In narrow-slice development, during each lifecycle phase an example of the products of the next phase is developed using the planned process (not via an ad-hoc "prototype" approach). The narrow slice takes a "trial run" at the development process, and works out its problems "before the herd arrives" to do the main work of that phase. This approach detects many blind alleys or simple inefficiencies that may hidden in the process (especially in an unfamiliar formal process) before much effort has been expended, and thus improves the overall efficiency and mitigates the risks of adopting formality.



abstracted input #1 ("i1")	**abstracted input #2 ("i2")**	**abstracted input #3 ("i3")**	**abstracted output**
"X"	**"X"**	**subrange 3.1**	**f1(i1, i2, i3)**
"X"	**subrange 2.1**	**subrange 3.2**	**f2(i1,i2)**
subrange 1.1	**subrange 2.2**	**subrange 3.2**	**f3(i3)**
subrange 1.2	**subrange 2.2**	**subrange 3.2**	**f4(i1,i3)**

figure 1: Example CoRE Requirement

3.2 Change-Driven Design

Formal methods are exceptionally sensitive to changes. A mathematical approach will often take longer to perform than a traditional heuristic one, at least in the initial definition activity and in activities like theorem proving. These costs can be more than recouped during the other lifecycle activities as long as the costly activities need not be repeated too often.

However, in real industrial projects change is the rule. It is typically frequent, and often extreme. The only way to retain the advantages of formality without being overwhelmed by its weaknesses in this area is to strictly limit the propagation of change through the system...so that when change occurs, it affects only the absolute minimum portion of the system inherently necessary to implement the change.

The propagation of change can be strictly limited by constructing "change scenarios"...identifying across the expected lifecycle of the product and its variants every type of change that could plausibly occur. This should include changes likely to occur as part of the initial development cycle.

Then different architectural organizations can be postulated to attempt to encapsulate those changes as severely as possible. The goal is to achieve an architecture which will only need to be changed in one place (ideally, one subprogram or data structure) for each individual "change stimulus."

This approach is consistent with the philosophy behind certain domain design approaches (e.g., SPC's Synthesis approach [9]).

For instance, a class structure could be constructed to encapsulate changes likely to occur in a system that must communicate significantly with other devices, systems or its environment, e.g., via data bus, and must transform such information to perform its tasks (a fairly generic type of processing).

At the highest level of abstraction, most of what such a system does could be covered by three classes (this is the approach planned for portions of Boeing North American's development on the Brimstone program, and which also was used on the Lockheed Martin C-130J mission software).

One class could handle the translation of bus-encoded data into more abstracted information suitable for use by the system being developed, and vice versa, from abstracted system information into low-level bus data.

Another class could provide read-only access to the abstracted information which would represent the state of the outside systems or environment.

The third class would perform the transformations of abstracted information about current state of the external environment, into abstracted information about the desired effect upon the external environment. Note the similarity of this class's charter and the components of a CoRE table (i.e., abstracted inputs, abstracted outputs, and discontinuous functions relating them). This similarity is exploited in a way which will be described shortly.

The three classes work together to form a complete processing engine. There would be a group of three instances, one of each class, for each external system to which the system under design was interfaced.

These three classes could be called "device interface," "device current state," and "device control," respectively. This is illustrated in figure 2, in a variant of Buhr notation. Note that dashed outlines of classes (outer box) or methods (inner boxes) indicate there can be multiple instances thereof.

There are several plausible change scenarios for this type of system. Most of them distill to two basic patterns: The external systems or environmental interfaces could change, or the purposes of the system could change. In the first case, change will usually be limited to the instance of the device interface class for the system which changed. In the second case, change will usually be limited to a single procedure in the control class.

Another design criteria, with benefits not just as change occurs but also for traceability and testing, is to localize the implementation of each formal (e.g., CoRE) requirement to a single method in an object of the control class. This takes advantage, noted earlier, of the similarity between the nature of CoRE requirements and the charter of the control class.

This approach is in contrast to the typical design decomposition process, which "smears out" individual requirements across many design artifacts. Experience has shown that this approach is valid even on large programs (>100 KSLOC, e.g., like the C-130J): appropriate domain-oriented factorization is the key to making this possible.

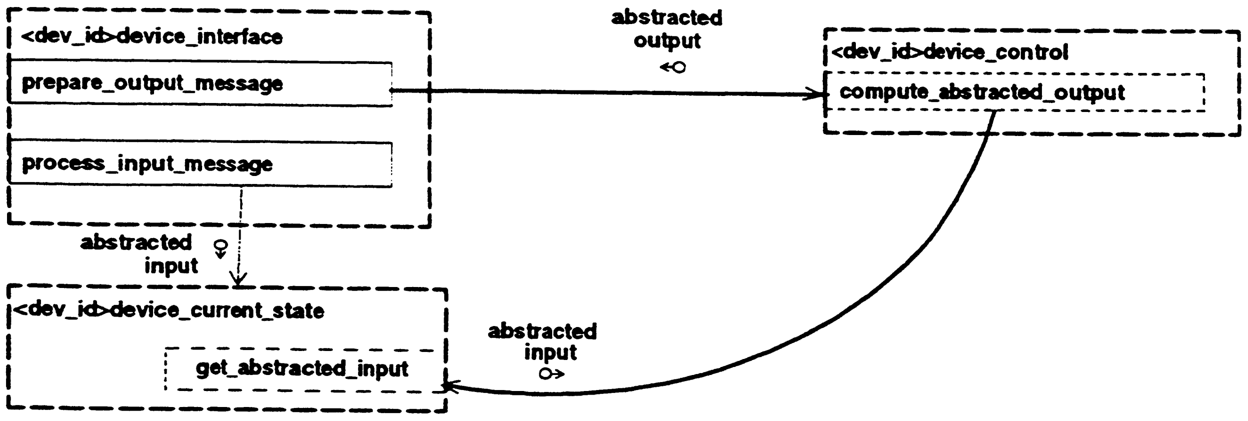


figure 2: Change-Tolerant Class Structure

In this type of design structuring, changes to a single requirement affect, in general, just one subprogram. Tests of a single requirement become unit-level tests(using the rows of the CoRE table as the specifications for individual test cases!). Traceability of requirements to design to code to test cases becomes trivial. All these factors dramatically decrease costs.

By creating a class structure which reflects the inherent repeatabilities in the solution domain, one can craft a "syntax" for design. Each type of class becomes a "part of design," just as nouns, verbs and adjectives are "parts of speech." As the English language includes strict syntactic rules for how its parts may interact with one other to express meaning, so also a strict set of syntactic rules can be constructed between the classes. In general, this approach is called an ADL or "Architecture Design Language" [10].

If the choice of the "parts of design" for such a syntax is directed by a domain design methodology, the result will be both factored and change-driven. Such an ADL can be called a DSDL or "Domain-Specific Design Language."

This is a "semi-formal" method in the sense that syntax is often considered a mathematical construction, and can be subjected to mathematical verification arguments. Further, tools (homegrown or commercial, e.g., Rational Apex subsystems and views) can enforce the restrictions on interactions between the classes, and thus prevent the introduction of many types of errors.

Finally, identification of change-driven classes allows one to create "implementation templates." These templates provide the final implementors (e.g., detailed designers/coders) with the required structure of their portion of the system. The pre-defined allocation of formal requirements to detailed-design elements makes implementation much simpler, verification much quicker (against the pre-defined allocation), and control of unwanted interactions between code elements easier (the "universe" of possibilities has been strictly limited by the templates).

3.3 Closed-Loop Formality

Formal requirements provide benefits even when used in isolation. However, the added costs of using formality make it imperative to obtain every possible benefit of formal requirements. The remaining benefits come only through using formality in an integrated way throughout the lifecycle.

Formality throughout the lifecycle must apply the strong mathematical foundation provided by the formal requirements to facilitate every lifecycle activity: design, code, static analysis, and testing. By eliminating much of the "guess work" typically in these activities, lifecycle formality increases their efficiency.

The relationships between formal requirements and formality in the rest of the lifecycle are illustrated in figure 3.

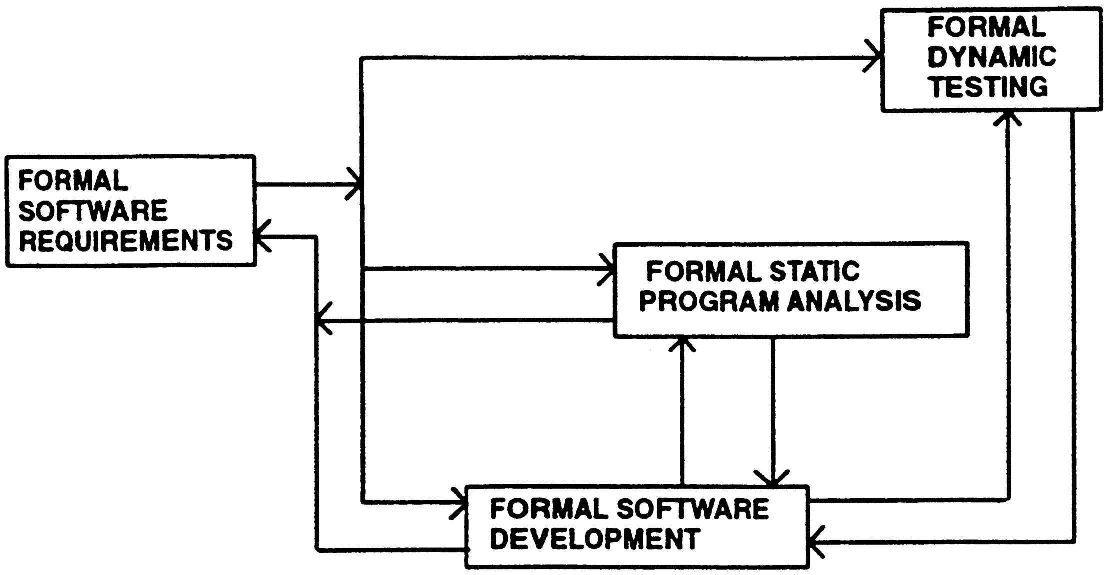

FORMAL DYNAMIC TESTING
FORMAL SOFTWARE REQUIREMENTS
FORMAL STATIC PROGRAM ANALYSIS
FORMAL SOFTWARE DEVELOPMENT

figure 3: Formal Requirements and Lifecycle Formality

Formal development takes advantage of formal requirements, the DSDL syntax and associated templates, and the previously-mentioned architectural strategy of implementing each formal method in a specific procedure in the software, to simplify and speed implementation. If the software is then coded in a formal language like the SPARK Ada subset [11], the code will largely be correct as constructed.

Formal verification begins with static analysis. With formal requirements and formal code, static analysis of correctness is more efficient than traditional unit testing. Thus, static analysis is performed first, any errors found are corrected, and the code is then submitted to test This also "closes the loop" of code back to requirements.

Formal testing remains necessary because of incomplete formalization of the software product, the need to verify target-compiler correctness, and hardware issues (e.g., was the original software specification based on a correct understanding of the hardware environment).

Formal testing derives the test cases from the formal requirements. This provides very high statement and path coverage compared to typical non-formal requirements-based testing. Since requirements-based testing is often the most efficient testing approach, high confidence is provided at relatively low testing cost.

If the CoRE (or SCR) method is used to specify requirements, black-box testing is as simple as setting up the abstracted input values in a given table row, and comparing the result to the abstracted output for that row. If the architecture has applied the heuristic of "one requirement to one procedure," white-box testing of the most semantically-significant modules in the software system will also be directly driven from the formal requirements. This too has proven to be highly efficient.

Conclusion

Formal methods are the future of software development. The sheer number of failed software systems is proof of the need for more robust means of production. Failed systems are intolerable in business; mathematics provides the needed robustness. Business and mathematics are a marriage made in heaven; they will meet again in software as they have so often before in other disciplines.

Business will not, however, embrace mathematics to its own loss. Mathematics has always provided net benefits in other endeavors, and must do so now in software. This requires that the software theorist at least consider the business perspective; i.e., asking "*why* are companies developing software?" The answer almost invariably reduces to "for the shareholders."

Re-examining formality's role in such a purposeful software lifecycle leads to principles of practicality. As those principles are identified and refined, formality will take an ever-increasing role in for-profit software development. And this, in the end, will benefit everyone; the industrialist, by improving the bottom line, the consumers and public, by providing them with more reliable and affordable systems, and the theorists, by providing more compelling reasons, a sharper focus, and a ready outlet for their creativity and research.

Biographical

James Sutton is the lead software methodologist and software safety criticality engineer for Boeing North American's Brimstone missile system development. He previously served as lead methodologist and architect on the mission software for Lockheed Martin's C-130J program, as well as safety critical methods liaison with US and international regulatory agencies, reuse IRAD principal investigator, and reuse lead engineer for the F-22. He has authored a college textbook entitled "Power Programming" for Prentice Hall, and has presented and/ or published for numerous conferences including NAECON, Tri-Ada, ERA Avionics Conference (U.K.), Ada-Europe, IEEE DASC, and Compass (safety critical software).

References

[1] James M. Sutton. Lean Software for the Lean Aircraft. *Proceedings of the IEEE DASC 96 Conference*, Atlanta, Georgia, 1996.

[2] Thomas A. Alspaugh, Stuart R. Faulk, Kathryn Heninger Britton, R. Allan Parker, David L. Parnas, John E. Shore. Software Requirements for the A-7E Aircraft. NRL/FR/5530-92-9194, Washington DC: Naval Research Laboratory, 1992.

[3] Mark R. Blackburn, Robert D. Busser. T-VEC: A Tool for Developing Critical Systems. *Proceedings of the IEEE Compass 96 Conference*, 1996.

[4] Steven P. Miller, Carl F. Hoech. Specifying the Mode Logic of a Flight Guidance System in CoRE. Unpublished working paper of Rockwell Avionics and Communications, 1997.

[5] B. Carre', J. Sutton. Achieving High Integrity At Low Cost: A Constructive Approach. *Proceedings of the ERA Conference*, London, 1995.

[6] S. Faulk, L. Finneran, J. Kirby, Jr., J. Sutton. Experience Applying the CoRE Method to the Lockheed C-130J Software Requirements. *Proceedings of the Ninth Annual Conference on Computer Assurance,* 1994.

[7] C. Heitmeyer, A. Bull, C. Gasarch, B. Labaw. SCR*: A Toolset For Specifying And Analyzing Requirements. *Proceedings of Tenth Annual Conference on Computer Assurance*, 1995.

[8] Process Engineering with the Evolutionary Spiral Process Model. SPC-93098-CMC version 01.00.06; Software Productivity Consortium; Herndon, VA, US January, 1994.

[9] Reuse-Driven Software Processes Guidebook. SPC-92019-CMC version 02.00.03; Software Productivity Consortium; Herndon, VA, US November 1993.

[10] M. Graham, E. Mettala. The Domain-Specific Software Architecture Program. *Proceedings of the 1992 DARPA Software Technology Conference*, 1992.

[11] B. Carre', J. Garnsworthy. SPARK - An Annotated Ada Subset for Safety-Critical Programming. *Proceedings of Tri-Ada Conference*, Baltimore, December 1990.

Proving Properties of Accidents

C.W. Johnson,

Glasgow Accident Analysis Group,
Department of Computing Science,
University of Glasgow,
Glasgow, United Kingdom, G12 8QQ.
E-mail:johnson@dcs.glasgow.ac.uk
WWW: http://www.dcs.gla.ac.uk/~johnson

Abstract

Accident reports are produced by regulatory and commercial authorities, such as the UK Air Accident Investigation Branch [1] and the US National Transportation Safety Board [17], in response to most major accidents. They, typically, contain accounts of the human and system failures that lead to major accidents. These descriptions are then used to identify the primary and secondary causes of the failure. Finally, recommendations are made so that the operators and regulators of safety-critical systems can avoid future accidents. Unfortunately, it is often difficult for readers to trace the way in which particular conclusions are drawn from many hundreds of pages of evidence. Natural language arguments often contain implicit assumptions and ambiguous remarks that prevent readers from understanding the reasons why a particular conclusion was drawn from a particular accident. This paper argues that mathematical proof techniques can be used to support the findings of accident investigations. These techniques enable analysts to formally demonstrate that a particular conclusion is justified given the evidence in a report. Conclusion, Analysis and Evidence diagrams can then be used to communicate the results of a formal analysis. The intention is not to replace the natural argumentation structures that are currently used in accident reports. Rather, our aim is to increase confidence that particular conclusions are well supported by the evidence that is presented within a report.

1 Introduction

Accident reports are intended to ensure that major failures do not recur. They are produced by a wide range of national [4, 6] and international bodies [23]. Typically, these documents begin by providing a brief synopsis of the events leading to an accident. The synopsis is then followed by a number of expert analyses. These identify and prioritise the failures leading to the accident. Finally, conclusions are drafted from the experts' findings. These form the basis of any actions that companies or regulatory authorities might take to prevent future failures.

1.1 Conventional Reporting Techniques

Unfortunately, it is not always easy for readers to understand the justifications that support particular conclusions [8]. Accident reports are often many hundreds of pages in length. The evidence that supports a particular line of analysis may be lost amongst the paragraphs of contextual detail. A further problem is that natural language can be ambiguous. For example, accident reports often refer to situations of 'high workload' and 'operator error' without explaining the precise meaning of these terms [20]. Many accident reports are also inconsistent in the sense that the same term is used to refer to several different objects or people [14]. Later sections of this paper will argue that these problems create considerable confusion for the reader and may even lead them to doubt the accuracy of the final report.

1.2 Formal Methods and Accident Analysis

Formal proof techniques can be used to avoid the ambiguity and inconsistency of natural language [2]. A number of authors have also used these techniques to support the design of dynamic, interactive systems. For example, Dix [5] has used an algebraic notation to reason about high level properties of multi-user systems. Paterno, Sciacchitano and Lowgren [19] have used the LOTOS notation to examine interaction with complex multimedia applications. Palanque and Bastide [18] have applied Petri Nets to examine safety and liveness properties of distributed systems. None of this work has been applied to reason about accident reports. In particular, there has been no attempt to use mathematical techniques to prove that conclusions are well-founded with respect to the analysis that is presented in an accident report.

2 The Case Study

This paper focuses upon an accident report that was produced by the United States' Coast Guard in response to a collision between the passenger vessel Noordam and the bulk carrier Mount Ymitos [22]. We are interested in this case study because it typifies the many different operator errors and organisational failures that exacerbate accidents with complex, interactive systems. The remainder of this section brief outlines the course of the accident. The Noordam collided with the Mount Ymitos at 20.42 (Central Standard Time) on November 6th, 1993. The accident occurred two miles south of the Southwest Pass Entrance Light Buoy in the Gulf of Mexico. The exact location was recorded as 28 degrees, 50.0 minutes North and 89 degrees, 25.7 minutes West. Both ships were damaged in the collision but there was no loss of life.

The Mount Ymitos was outbound from the Mississippi River en route to St Petersburg, Russia. It had cleared the Southwest Pass out of the River when the Third Officer noticed an inbound passenger vessel using their binoculars. At this stage, he estimated that the vessel was approximately six miles from the Ymitos. He did not immediately report his observation as the Captain was busy with the Pilot who was preparing to leave the Mount Ymitos. The watch-standers re-established visual contact when the Noordam had closed to two miles from the Mount Ymitos. The Captain reduced their speed to dead slow and expected the Noordam to alter its heading. At this point the ARPA (Automatic Radar Plotting Aid) showed that the closest point of approach was under six hundred feet. The Captain made several attempts to alert the Noordam. At 20:40:08 the Coast Guard logged a Channel 16 VHF call: 'Passenger Vessel, Passenger Vessel, Go to South Pass'. At 20:40:50 they logged 'Passenger Vessel Going to South Pass, I Turn Hard Starboard'. The third officer then attempted to communicate the warning using an Aldis lamp. No response was received.

The Noordam was en route to New Orleans from Cozumel, Mexico. At approximately 20:00:00, Second Officer Smit called the Pilot Station and learned that two other vessels were also in-bound towards the Mississippi and could be overtaken. The Pilot did not alert the Noordam to the presence of any outward bound vessels. Quartermaster Salyo was the designated lookout He left the bridge on two separate occasions during the approach. Shortly after 20:00:00 he left, with the permission of Second Officer Smit, to make sandwiches and coffee for the bridge crew. At 20:10:00 he unlashed the anchors in preparation for entering port. He returned at 20:20:00 but did not detect the Mount Ymitos until immediately prior to the collision. A scheduled watch change took place at 20:30:00. Second Officer Smit performed navigation checks using the radar, together with Chief Officer Broekhoven, before handing over to Third Officer Veldhoen. Veldhoen, in turn, handed over to the Chief Officer at 20:36:00 when an 'end of sea voyage' was declared. This is a point of convenience determined by the watch officer and represents the point at which the Chief Officer assumes control for the manoeuvring watch prior to arrival in port. In order to complete this hand-over Veldhoen had to fix the vessel's position, complete the log and notify the engine room. As the Noordam changed course to enter the final leg of the approach, Fourth Officer Kuiper, who was on the bridge but who was not on duty, saw the lights of the Mount Ymitos and immediately issued a curse. The manoeuvre was halted while the crew determined the course and position of the vessel that they had seen, Approximately one minute before the collision, Chief Officer Broekhoven ordered left full rudder to pull away from the danger.

The Coast Guard's report argues that the principle reason for the collision was the failure by the Noordam's crew to keep an adequate watch. Unfortunately, the report does not provide a detailed explanation of why this failure occurred. The reader is left to infer the causal relations that link the observations about the accident and the conclusions that are listed at the end of the document. The following pages, therefore, show how formal techniques can be used to explicitly link the findings of an investigation to the account of an accident.

3 Formalisation of the Accident

In order to reason about the findings of an accident report, it is first necessary to model the events leading to the failure. The first stage in this process is to identify the critical operators, tasks, roles, communications, systems and locations that helped to shape the course of the accident.

3.1 Critical Components

A limitation with natural language approaches to accident reporting is that it can be difficult to identify critical information from a mass of background detail. For example, the Coast Guard's report into the Noordam collision includes the following account:

> "Fourth Officer Daniel Kuiper, who was not on duty, was the first to notice the lights of a vessel off the starboard side of the Noordam. This was between one and two minutes before the time of collision. He saw a red light that he estimated was approximately 2 points off the NOORDAM's starboard bow - a point being 11.25 degrees of arc. First Officer Kuiper uttered a curse word that attracted the attention of others on the bridge. Third Officer Veldhoen, upon looking to starboard, also saw lights." [Paragraph 42]

Additional information, such as the conversion between points and degree of arc, is included to help the reader form a picture of the accident. Unfortunately, such details may actually obscure the underlying causes of operator 'error' and system 'failure'. Our previous work on accident analysis has, however, identified a number of categories that can be used to identify critical components in an accident:

• **operators.** It is necessary to represent the people involved in an accident so that readers can follow the way in which operator intervention affects the course of system failures;
• **roles.** It is important to distinguish particular individuals from the roles that they perform during an accident. For example, a number of individuals performed the tasks associated with the role of watch-stander during the Noordam collision;
• **tasks.** It is necessary to identify the tasks that operators were or should have been performing during accidents if readers are to understand the ways in which human intervention safeguarded the system or exacerbated any key failures;
• **speech acts.** It is vital to represent communication between the operators that are involved in an accident. Misunderstandings have a profound impact upon the safety of many applications;
• **information and control systems.** This type of information is included because the quality of information that is available to system operators is often determined by the channel that is used to support their observations. For instance, ARPA radar provides more detailed and arguably less reliable information than direct visual contact;
• **physical locations.** It is necessary to represent the place in which an accident occurs because the location of a failure can have a profound impact upon an operator's ability to respond to an accident [11].

Paragraph [42], cited above, can be used to identify physical locations, such as the Noordam and the Mount Ymitos. It is also possible to identify operators such as Veldhoen and Kuiper who perform the roles of First and Fourth Officers respectively. We can identify observation channels; in this case the visual observation of the Ymitos' lights as well as critical speech acts such as Kuiper's curse. Table 1 shows the results that can be obtained by extending this analysis throughout the Coast Guard's report. In formal terms, the elements of this table define the types that model the Noordam accident. The process of building such a table helps to strip out irrelevant detail that obscures critical properties of major accidents.

Physical Locations	Roles
captain veniamis	*lookout*
pacific trident	*chief officer*
mount ymitos	*first officer*
noordam	*third officer*
	fourth officer
	watch officer

Operators	Speech Acts
pilot station	*inbound pacific trident*
engine room.	*inbound capt veniamis*
smit.	*outbound mount ymitos*
salyo	*curse*
broekhoven	*take bearing on lights*
veldhoen	*lights moving right*
kuiper	*officer change*
	end of voyage
	left full rudder
Tasks	**Information Systems**
navigation radar check	*arpa radar*
collision radar check	*visual*
correlate radar targets	*binoculars*
declare end ofvoyage	
fix vessel position	
complete log	
notify engine room	

Table 1: Critical Entity Table for the Noordam Accident

3.2 Axioms for the Accident System

The identification of operators, roles, tasks, speech acts, information systems and locations is of little benefit if analysts cannot represent and reason about the relationships that exist between these components. The following section uses a simple form of temporal logic to demonstrate how this might be done for the Noordam case study.

3.2.1 Operators and Roles

The previous sections argued that it is important to identify the critical roles that operators play in an accident. This affects the range of tasks that operators are expected to perform. For example, Broekhoven was the Noordam's Chief Officer during the incident, Smit was the First Officer:, Veldhoen was the Third Officer and Salyo was the lookout

role(chief_officer, broekhoven). (1)
role(first_officer, smit). (2)
role(third_officer, veldhoen). (3)
role(lookout, salyo). (4)

Such clauses gather together information that is, typically, scattered throughout conventional, natural language documents. The roles performed by key individuals in the Coast Guard's report are listed in paragraphs [13, 16, 25, 30, 37, 42]. Such a formalisation is also important if an individual's role changes during the course of an accident. For example, the officer in charge of the watch on the Noordam changed at 20:30 hrs:

at(role(watch_officer, smit), 2029). (5)
at(role(watch_officer, veldhoen), 2030). (6)

The previous clauses exploit a simple form of temporal logic in which the binary *at* operator takes a proposition and a term denoting a time such that *at(p, t)* is true if and only if *p* is true at *t*. A number of technical problems surround the general application of this simple extension to propositional logic. In particular, the philosophical issue of reification forces analysts to clearly state the relationship between particular terms and objects over time. This theoretical problem is less of an issue for our purposes because we are always referring to definite entities at specific times during an accident. We, therefore, retain this simple temporal framework rather than the more elaborate temporal languages in our previous work [7, 10, 21].

3.2.2 Operators and Communications

Communications problems exacerbate many major accidents. It is, therefore, important to represent and reason about this source of 'error'. During the accident, Smit requested and received specific information about the Captain Veniamis and the Pacific Trident that were inbound towards the Mississippi:

∃ t: at(message(pilot_station, smit, inbound_capt_veniamis), t). (7)
∃ t: at(message(pilot_station, smit, inbound_pacific_trident), t). (8)

The existential ∃ quantifier (read as 'there exists') is used because the accident report does not represent the precise times associated with each of these individual communications. The following clause shows how the same approach can be adopted to represent a lack of communication. Smit did not receive information about outbound traffic from the Pilot Station:

∀ t: not at(message(pilot_station, smit, outbound_mount_ymitos), t). (9)

The universal ∀ quantifier (read as 'for all') is used because it was never the case that Smit received information from the Pilot Station about the Mount Ymitos. Similar clauses can be used to represent more complex verbal exchanges. For example, Kuiper first observed the Ymitos' lights and issued a curse which was heard by Veldhoen and Broekhoven. Broekhoven then requested that Veldhoen take a bearing on the lights. Veldhoen responded that the lights were moving right. The following clauses represent these individual speech acts:

∃ t, t' : at(message(kuiper, [veldhoen, broekhoven], curse), 2040)
∧ at(message(broekhoven, veldhoen, take_bearing_on_lights), t)
∧ at(message(veldhoen, broekhoven, lights_moving_right), t')
∧ after(2040, t) ∧ after(t, t').
(10)

It is important to note that the preceding clauses do not represent the precise verbal components of each speech act. This information could be introduced if it were available, for instance through studying cockpit voice recordings. In the case of the Noordam there was no such record. Place holders, such as *curse*, are used to capture the recollected sense of the communication without specifying its exact form.

3.2.3 Operators and Locations

It is important to consider the physical location of system operators during major accidents. For example, the lookout left his position on the bridge at critical moments during the lead-up to the Noordam collision. Clause (11) states that *salyo* was in the galley at 20:00hrs. Similarly, clauses (12) and (13) describe Salyo's subsequent movements from the galley back to the bridge at 20:10 hrs and from the bridge down to the decks at 20:15 hrs. They do not specify when Salyo moved from each of these locations because the report does not provide accurate journey times:

at(position(salyo, galley(noordam)), 2000). (11)
at(position(salyo, bridge(noordam)), 2010). (12)
at(position(salyo, decks(noordam)), 2015). (13)

The previous clauses do not specify the relative position of the galley, bridge or decks. Such information can be introduced by formalising a three-dimensional co-ordinate scheme [11]. This was not done because clauses (11,12,13) reflect the level of detail in the Coast Guard's report. This illustrates an important benefit of the formalisation. Logic provides an explicit representation of the level of abstraction that is considered appropriate for the readers of the report. They do not need to know the relative positions of the galley, bridge and decks in order to understand the events leading to the collision. Such decisions are extremely important. Too much detail and readers will be swamped amongst a mass of contextual information. Too little detail and it will be difficult for them to reconstruct the flow of events leading to disaster. Clauses, such as (11,12,13), can be used to represent and reason about appropriate levels of abstraction. This helps to avoid the ad hoc decisions that frequently seem to be made about the amount of location information that is included in accident reports [14].

3.2.4 Operators and Tasks

The Coast Guard's report contains the following paragraph:

"Between 2030 and 2036, Broekhoven and Veldhoen checked the radars occasionally, using the six mile scale. Broekhoven was planning the turn from 325 degrees to 000 degrees to coincide with bringing the Racon 'T' platform abeam, at 1.5 miles to port. Both Veldhoen and Broekhoven used the 10-centimetre and centimetre radars to check the distance of the domino platforms, and particularly the bearing and range of the Racon 'T'. They were not using the radars for collision avoidance and observation of moving targets, and did not attempt to correlate every fixed target contact in the radar with fixed platforms observed visually to see if any were any underway contacts rather than fixed platforms." [Paragraph 39]

From this it is possible to extract two critical observations about the operation of the Noordam. Firstly, that between 20:30, and 20:36 both Broekhoven and Veldhoen were performing navigation radar checks. Secondly, that during this interval they did not correlate radar targets with visual observations. The following clauses introduce a *during* operator such that *during(p, t)* is true if and only if the situation denoted by *p* occurs at sometime during the interval denoted by *t*. Formally, this can be given as follows:

$$\forall t : during(p, t) \Leftrightarrow \exists t' : at(p, t') \wedge before(t', end(t)) \wedge before(begin(t), t'). \qquad (14)$$

This assumes that *before(t, t')* is true if *t'* occurs at some time after *t* or at the same instant as *t*. The following clauses also introduce the operator, *in*, such that *in(t, t', t")* is true if *t* is wholly contained within *t'* and *t"*. This can be formalised in a similar manner to *during*. In contrast, the following clauses formalise the observations made in paragraph [39] of the accident report:

$$\exists t : during(perform(broekhoven, navigation\_radar\_check), t) \wedge not\ during(perform(broekhoven, correlate\_radar\_targets), t) \wedge in(t, 2030, 2036). \qquad (15)$$

$$\exists t : during(perform(veldhoen, navigation\_radar\_check), t) \wedge not\ during(perform(veldhoen, correlate\_radar\_targets), t) \wedge in(t, 2030, 2036). \qquad (16)$$

An important benefit of the formalisation process is that clauses, such as (15) and (16), can be translated back into natural language sentences; between 20:30 hrs and 20:36 hrs Broekhoven and Veldhoen performed navigation radar checks but did not correlate radar targets. The formalisation process helps analysts to focus upon critical aspects of an accident, such as operator tasks. This benefit might be obtained using a conventional task analysis technique such as TAKD [15]. Later sections will, however, argue that formal reasoning techniques can be used to prove properties of accident reports. This provides the additional degree of assurance that is demanded by bodies such as NASA and the UK Ministry of Defence [2].

The previous example describes a relatively simple set of observations about operator tasks. Typically, the co-ordination of group activities is more complex. For example, Veldhoen declared an 'end of sea voyage' between 20:34 and 20:38. This procedure handed over control of the watch to the First Officer Broekhoven. He was responsible for navigating the Noordam into port. This change was not, however, announced to the lookout:

$$\exists t : during(perform(veldhoen, declare\_end\_of\_voyage), t) \wedge not\ during(message(broekhoven, salyo, officer\_change), t) \wedge during(role(watch\_officer, broekhoven), t) \wedge \wedge in(t, 2034, 2038). \qquad (17)$$

The failure to inform the lookout was important because the task of declaring the 'end of sea voyage' involves the watch officer in a number of sub-tasks that reduce the amount of time that they have available for navigation and collision avoidance:

$$\forall t: during(perform(veldhoen, declare\_end\_of\_voyage), t) \Leftrightarrow during(perform(veldhoen, fix\_vessel\_position), t) \wedge during(perform(veldhoen, complete\_log), t) \wedge during(message(veldhoen, engine\_room, end\_of\_voyage), t). \qquad (18)$$

Such clauses illustrate how the products of hierarchical task analysis might be introduced into formal models of major accidents. The higher order task of declaring the 'end of sea voyage' is comprised of three sub-tasks: fixing the vessel's position; completing the log and notifying the engine room.

3.2.5 Operators and Observations

The entities that were identified in Table 1 are generic in the sense that operators, roles, tasks, speech acts, information systems and physical locations are central to all of the accidents reports that we have examined [8, 9, 14]. This does not mean that the list is exhaustive. A related point is that the significance of individual entities will vary from accident to accident. For example, automated control systems did not have a significant impact upon the course of the collision between the Ymitos and the Noordam. In contrast, information systems played a critical role in the observations that operators made during the accident. Veldhoen made visual observations of the ship but did not use an azimuth circle to verify his observation:

∃t : at(observe(veldhoen,
mount_ymitos, visual), t). (19)
∀t : not at(observe(veldhoen,
mount_ymitos, azimuth), t). (20)

As before, the existential ∃ quantifier is used in clause (19) because the accident report does not identify the particular interval when Veldhoen made his observation. All we know is that there exists a time at which Veldhoen made a visual observation of the Ymitos. The universal∀ quantifier is used in clause (20) because Veldhoen did not use an Azimuth circle at any time in the accident. This shows how an analysts concerns can direct the formalisation process. Clause (20) represents something that the officer did not do. If it had not been formalised then readers would not have been aware of this omission. In fact, Veldhoen's failure to verify his visual observations reinforced Broekhoven's judgement that the ships would pass starboard to starboard. He had seen a green (starboard) light shortly after the initial observation made by Kuiper:

∃t : at(observe(broekhoven,
green_light(mount_ymitos), binoculars), t). (21)

It was only when Broekhoven saw a red light that he realised the imminent possibility of a collision with the Mount Ymitos and took evasive action:

at(observe(broekhoven,
red_light(mount_ymitos), visual), 2041)
∧ at(message(broekhoven,
engine_room, left_full_rudder), 2041).
(22)

This section has used temporal logic to formalise the events leading to an accident. This formalisation process helps to strip out the contextual detail that hides critical observations in the many hundreds of pages that form conventional reports. We have not, however, shown that this approach can be used to reason about the conclusions that are drawn from an accident report.

4 Reasoning About Accident Reports

This section argues that formal methods can be used to establish the relationship between the evidence presented in an accident report and the conclusions which boards of enquiry use to draft future legislation. Unless this can be done, it will be difficult for commercial organisations to understand the reasons why particular sanctions may be imposed in the aftermath of major accidents [8]. For example, the Coast Guard enquiry made the following observation about the collision between the Noordam and the Ymitos:

'The proximate cause of the casualty was the failure of Chief Officer Broekhoven, the person in charge of the watch on the NOORDAM at the time of the casualty, to maintain a vigilant watch in that he did not detect the presence of the MOUNT YMITOS visually or on radar until the MOUNT YMITOS was less than 1 mile away, less than 2 minutes before the collision.' [Conclusion 1].

Such findings create a number of problems for organisations that must prevent the recurrence of future accidents. In particular, it does not explain the reasons why Broekhoven failed to spot the Mount Ymitos. Readers are left to piece together or infer these justifications from the evidence presented in the many previous pages of analysis. This can have extremely serious consequences. Two readers might easily infer two different reasons why Broekhoven failed to keep an efficient watch. Each might, therefore, adopt quite different strategies for avoiding future failures [20].

Formal proof techniques can be used to demonstrate that a conclusion is valid given the evidence that is presented in an accident report. For instance, the following clause is derived from Conclusion 1 in the Coast Guard report:

∀ t: not during(vigilant(broekhoven), t)
⇔ not(during(observe(broekhoven,
mount_ymitos, visual), t)
∨ during(observe(broekhoven,
mount_ymitos, arpa_radar),t))
∧ before(t, 2040). (23)

We can re-write this clause as follows:

⇔ not during(observe(broekhoven,
mount_ymitos, visual), t)
∧ not during(observe(broekhoven,
mount_ymitos, arpa_radar),t)
∧ before(t, 2040).
[DeMorgan's Law (23)] (24)

⇔ (not during(observe(broekhoven,
mount_ymitos, visual), t) ∧ before(t, 2040))
∧ (not during(observe(broekhoven,
mount_ymitos, arpa_radar), t) ∧ before(t, 2040)).
[∧ Identity (24)] (25)

In order to justify Conclusion 1 we must consider two different cases. The first concerns the reasons why Broekhoven failed to make visual contact with the Mount Ymitos. The second addresses the failure to detect the Ymitos using the ARPA radar. In order to establish the connection between the conclusion and the evidence presented in the body of the report it is necessary for analysts to explicitly state the reasons supporting particular findings. For example, one of the reasons why Broekhoven failed to observe the Mount Ymitos was that he used the radar for navigation and not for collision avoidance:

$\forall t$: *not during(observe(broekhoven, mount_ymitos, arpa_radar), t)*
$\Leftarrow$ *during(perform(broekhoven, navigation_radar_check), t)*
$\wedge$ *not during(perform(broekhoven, correlate_radar_targets), t).* (26)

We can now prove that the second part of our formalisation of Conclusion 1 is satisfied by the evidence in the accident report. This can be done by applying the following inference rule to (15) and (26).

$\forall t: P(t) \Rightarrow Q(t),\ \exists t': P(t') \vdash \exists t': Q(t')$ (27)

Informally, this argument can be expressed as follows. From clause (26), we conclude that Broekhoven failed to observe the Mount Ymitos using the ARPA radar during any interval in which he was performing a navigation radar check and did not correlating radar targets. From clause (15) we know that know that Broekhoven was performing a navigation radar check and was not correlating radar targets between 20:30 and 20:36. Clause (27) tells us that if, we have clause (26) and clause (15) we can infer that Broekhoven failed to observe the Mount Ymitos using the ARPA radar during the interval between 20:30 and 20:36.

The previous proof illustrates a weakness in the accident report. Our formalisation of Conclusion 1 stated that Broekhoven did not observe the Mount Ymitos using the radar until 20:42. Our model has been used to prove that Broekhoven was pre-occupied with navigation checks between 20:30 and 20:36. This leaves at least six minutes unaccounted for. During that time, Broekhoven began turning the Noordam to the North. The accident report makes no reference to the use of the ARPA during this interval. The reader has to assume that the system was not used during this or subsequent operations prior to the collision at 20:42. Such findings are significant because they have important consequences for the recommendations that might be drawn from the report. For example, it is normal practice for officers to correlate radar targets when approaching an unfamiliar port. In the interval from 20:30 to 20:36 we can clearly see that navigation problems explain **why** Broekhoven did not perform these checks. We cannot, however, explain the omission during the final six minutes before the collision.

The second part of Conclusion 1 states that Broekhoven did not make any visual observation of the Mount Ymitos until 20:42. The justification for this finding can be found in a subsequent conclusion, rather than in the body of the accident report:

'The number of personnel (both watch-standing and non-watch-standing) on the bridge of the NOORDAM between 2020 and the time of the collision may have raised the complacency level and lowered the attentiveness of the bridge watch-standers with regards to maintaining a dedicated visual and radar watch.' [Conclusion 5].

The evidence for this conclusion can be found in paragraph [41] which states that:

'There were seven other persons on the bridge of the NOORDAM at this time (20:37hrs) in addition to the chief officer, who was in control of the vessel - three other licensed officers (one on duty, and two off duty), one cadet, two quartermasters and the chief officer's wife' [Paragraph 41].

This led to considerable confusion during our analysis of the report. We initially identified eight, and not seven, other individuals on the bridge in the final minutes before the collision. This confusion arose because Salyo was identified both by his name and by his role as Quartermaster In order to form this association, the reader must remember the allocation of responsibilities that was introduced in paragraph [25] when reading paragraph [41]:

$\forall t$: *not during(observe(broekhoven, mount_ymitos, visual), t)*
$\Leftarrow$ *during(position(kuiper, bridge(noordam)), t)*
$\wedge$*during(position(veldhoen, bridge(noordam)), t)*
$\wedge$ *during(position(helmsman, bridge(noordam)), t)*
$\wedge$ *during(position(chief_officers_wife, bridge(noordam)), t)*
$\wedge$ *during(position(quartermaster_1, bridge(noordam)), t)*
$\wedge$ *during(position(quartermaster_2, bridge(noordam)), t)*
$\wedge$ *during(position(cadet, bridge(noordam)), t)*
$\wedge$ *during(position(broekhoven, bridge(noordam)), t).* (28)

Paragraph [41] suggests that there were nine people on the bridge at 20:37:

at(position(kuiper, bridge(noordam)), 2037). (29)
at(position(veldhoen, bridge(noordam)), 2037). (30)
at(position(helmsman, bridge(noordam)), 2037). (31)
at(position(chief_officers_wife, bridge(noordam)), 2037). (32)
at(position(quartermaster_1, bridge(noordam)), 2037). (33)
at(position(quartermaster_2, bridge(noordam)), 2037). (34)
at(position(cadet, bridge(noordam)), 2037). (35)
at(position(broekhoven, bridge(noordam)), 2037). (36)

We can apply our definition of *during*, given in clause (14), to re-write each of the clauses from (29) to (36) in the following form:

∃ t: during(position(kuiper, bridge(noordam)),t)
∧ before(2037, end(t)) ∧ before(begin(t), 2037).
[Application of (14) to (29)] (37)

By repeating the application of (14) in the manner described above, we obtain the following:

∃t : during(position(kuiper,
bridge(noordam)), t)
∧ during(position(veldhoen,
bridge(noordam)), t)
∧ during(position(helmsman,
bridge(noordam)), t)
∧ during(position(chief_officers_wife,
bridge(noordam)), t)
∧ during(position(quartermaster_1,
bridge(noordam)), t)
∧ during(position(quartermaster_2,
bridge(noordam)), t)
∧ during(position(cadet,
bridge(noordam)), t)
∧ during(position(broekhoven,
bridge(noordam)), t)
∧ before(2037, end(t))
∧ before(begin(t), 2037).
[Introduction of ∧
from application of (14) to (29..36)] (38)

Finally, by applying inference rule (27) we get the following clause which corresponds to the second condition in Conclusion 1. In other words, the derivation of the following clause formally proves that our conclusions are consistent with the information contained in the body of the report:

∃t : not during(observe(broekhoven,
mount_ymitos, visual), t)
∧ before(2037, end(t))
∧ before(begin(t), 2037).
[Application of (27) to (28) using (38)] (39)

This proof helps to identify a further problem with the Coast Guard report We have previously cited Conclusion 5 which states that the number of personnel on the bridge between 20:20hrs and the time of the collision may have lowered the attentiveness of Broekhoven with regards to maintaining a visual and radar watch. Our formal analysis reveals that the evidence for this assertion only applies to the interval from 20:37hrs until the time of the collision. This poses a number of problems. We do not know why Conclusion 5 mentions 20:20hrs rather than 20:37hrs as stated in the body of the report. It can only be speculated that a number of people arrived on the bridge at this time earlier time. Alternatively, if additional crew members gradually were arriving from some time before 20:20hrs then we do not know why this was chosen as the critical moment at which collision avoidance tasks were impaired.

5 Communicating the Results of Formalisation

Unfortunately, mathematical analysis provides non-formalists with an extremely poor idea of the argumentation processes that support particular conclusions. It is difficult for people without some mathematical background to understand the various proof rules that are applied during the formal derivation of particular conclusions. This section, therefore, describes how literate specification techniques can be extended from the field of software engineering to support the formal analysis of accident reports.

5.1 Literate Specification

Communicating the results of mathematical analysis is a general problem for the application of formal methods. It affects the techniques described in this paper. It also affects the development of safety -critical systems. For example, designers might use the following clause to specify that a control system automatically removes a warning at some time after a failure has occurred. This is an important requirement if users are not to be over-whelmed by obsolete error messages. Unfortunately, it is not easy for non-formalists to understand the natural language requirement from its formal statement. A related point is that the formal expression of the requirement provides no clues as to the motivation or justification behind the requirement. In other words, it describes **what** the system should do, it does not describe **why** it should do it:

∀ t , ∃ t': at(automatically_remove_warning(
blow_back_error), t')
⇐ at(state(blow_back, failed), t)
∧ at(display(blow_back_error_icon), t)
∧ at(sys_cancel(blow_back_error_icon), t')
∧ before(t, t'). (40)

In previous papers, we have addressed these problems by developing literate specification techniques [12, 13]. This approach uses the semi-formal argumentation of design rationale to support the use of formal methods during the systems development. Figure 1 illustrates this approach. Rank Xerox's Questions, Options and Criteria (QOC) notation is used to document the reasons why the previous clause might be adopted within the design of a particular system. QOC diagrams are built by identifying the key questions that must be addressed during the development of an interactive system [3].

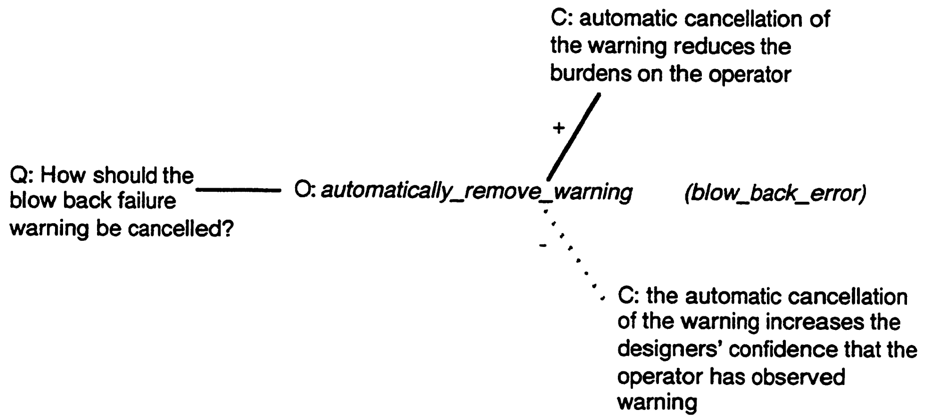

C: automatic cancellation of the warning reduces the burdens on the operator
+
Q: How should the blow back failure warning be cancelled?
O: automatically_remove_warning
(blow_back_error)
-
C: the automatic cancellation of the warning increases the designers' confidence that the operator has observed warning

Figure 1: Literate Specification for the Warning Cancellation.

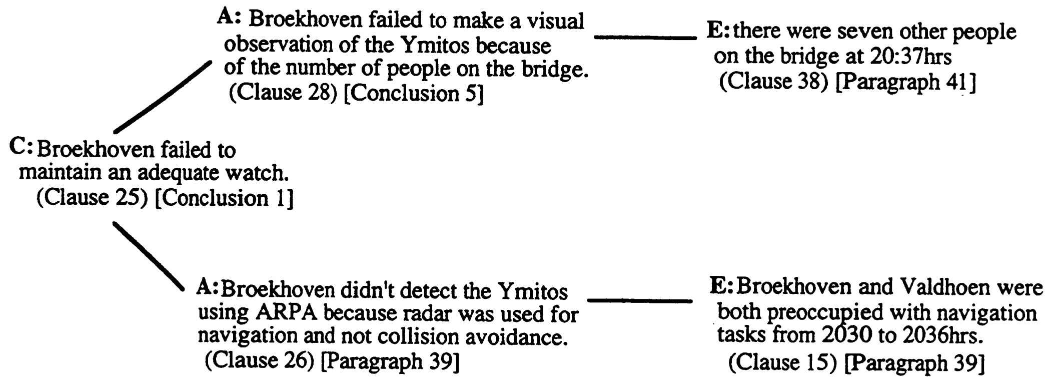

A: Broekhoven failed to make a visual observation of the Ymitos because of the number of people on the bridge. (Clause 28) [Conclusion 5]
E: there were seven other people on the bridge at 20:37hrs (Clause 38) [Paragraph 41]
C: Broekhoven failed to maintain an adequate watch. (Clause 25) [Conclusion 1]
A: Broekhoven didn't detect the Ymitos using ARPA because radar was used for navigation and not collision avoidance. (Clause 26) [Paragraph 39]
E: Broekhoven and Valdhoen were both preoccupied with navigation tasks from 2030 to 2036hrs. (Clause 15) [Paragraph 39]

Figure 2: Conclusion, Analysis, Evidence (CAE) Diagram for the Noordam Collision

The options that answer a particular question are then linked to it using the lines shown in Figure 1. Finally, options are linked to the criteria that support them, using solid lines, or weaken them, using broken lines. In Figure 1, the question of how to cancel blow-back warnings is answered by the design option that is specified by clause (40). This is justified by the criteria that the automatic cancellation of warnings reduces burdens on system operators. This does not help the operator to observe the warning.

The diagram shown in Figure 1 is relatively simple in that it only shows a single option for the design question. In practice, these diagrams tend to show a number of alternative clauses each of which represents a different design option for the problem being considered. The interested reader is directed to Johnson [13] for more detail on the application of this approach.

This blend of formal and semi-formal notations can also support the formal analysis of accident reports. Natural language annotations of the Questions and Criteria provide non-formalists with an entry-point into the clauses that represent particular Options. In literate specification, these annotations provide the justifications for and against formal design requirements. In accident analysis, they link source material to the clauses that describe the relationship between evidence and conclusions.

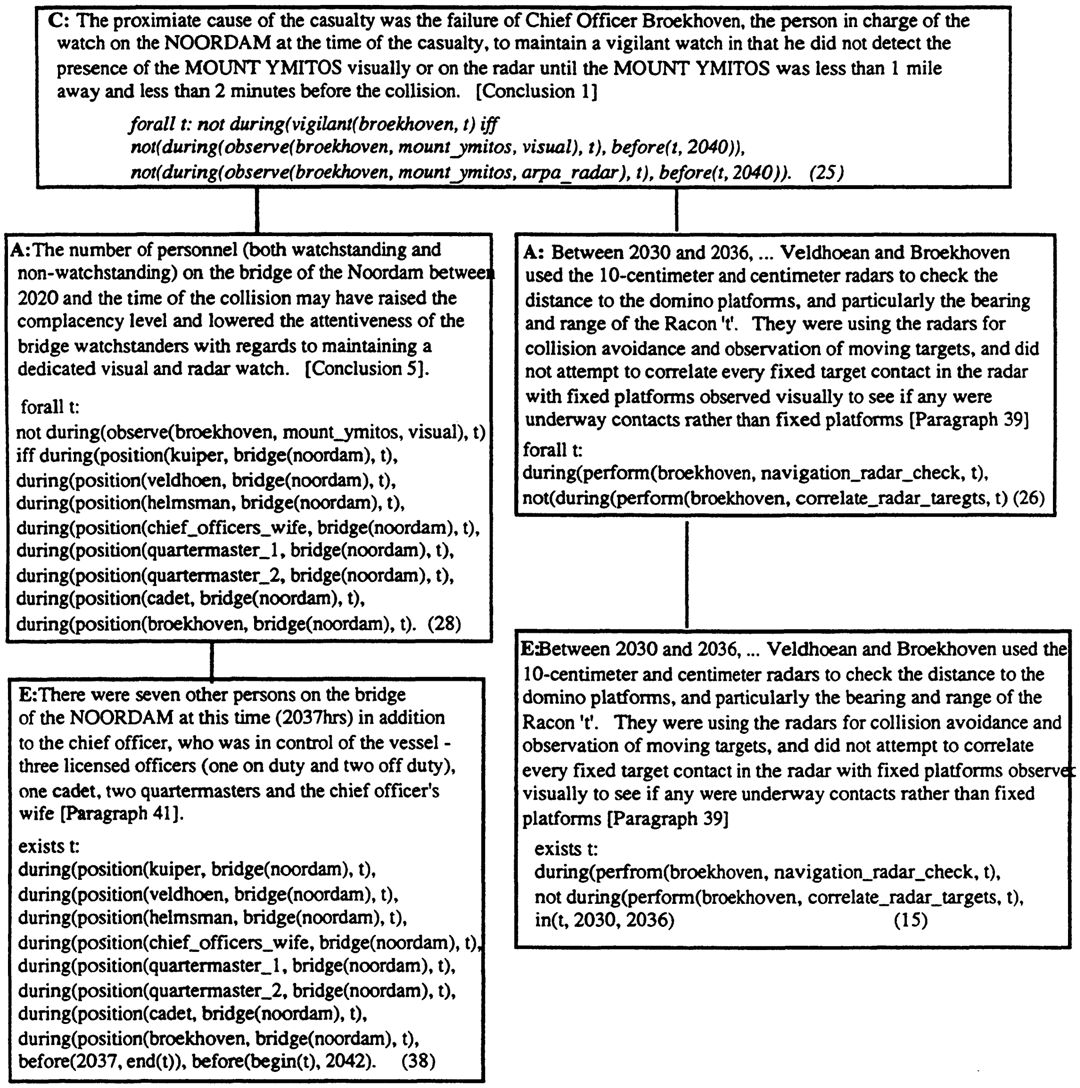
5.2 Conclusion, Analysis and Evidence (CAE) Diagrams

The Questions, Options and Criteria notation can be translated into a form that directly supports the formal analysis of accident reports. Instead of using questions to represent critical design issues, diagrams can represent the conclusions that are presented in a report. The options of a QOC diagram correspond to alternative interpretations of the events leading to a conclusion. Criteria can be compared to the evidence that supports or weakens the interpretation of an accident. Figure 2 presents a Conclusion, Analysis and Evidence (CAE) diagram for the Noordam collision. Broekhoven failed to maintain a vigilant watch. This is supported by [Conclusion 1] in the report and is formalised in clause (25). The conclusion relies upon an analysis which suggests that the number of people on the bridge prevented Broekhoven from visually detecting the Mount Ymitos. This is supported by the analysis in [Conclusion 5] of the report and is formalised in clause (28). The analysis rests upon evidence presented in [Paragraph 41] of the report. The conclusion also depends upon an analysis of the way in which Broekhoven used the radar. In this analysis, he was preoccupied with navigation rather than collision avoidance, from [Paragraph 39] represented in clause (26). This is supported by evidence in [Paragraph 39].

There is an important difference between Conclusion, Analysis and Evidence diagrams and the Question, Options and Criteria notation. Options represent alternative design choices in QOC. In contrast, the analysis components of a CAE diagram support a single conclusion. They are not mutually exclusive. It should also be noted that the evidence shown in Figure 2 supports the analysis. It is possible to use dotted lines and '-' signs to indicate evidence which might contradict a particular line of enquiry.

CAE diagrams are not intended to replace the formal proof techniques that are used in their construction. The vernacular labels that represent the conclusions, analysis and evidence are open to the same problems of inconsistency and mis-interpretation that weaken the use of natural language in accident reports. For instance, it is perfectly possible to link a conclusion to a line of analysis that has little or no relationship to the conclusion. The use of formal proof techniques helps to ensure that this does not happen. It should also be emphasised that none of the techniques presented in this paper are intended to replace the use of natural language in accident report. Our use of discrete mathematics is similar to that of forensic scientists who frequently use continuous mathematical models, for instance of combustion. Both sorts of model can be used to represent and reason about the events leading to failure.

The United Kingdom Engineering and Physical Sciences Research Council has recently funded a three year investigation into the integration of formal reasoning and Conclusion, Analysis and Evidence diagrams for accident reports. We are particularly concerned to provide tool support for these techniques. For example, Figure 3 illustrates how CAE diagrams can be extended to represent the clauses that are used within the proof of a conclusion. This illustrates the mathematical relationship between the underlying evidence, at the bottom of the diagram, and the higher level conclusions. The task of constructing and maintaining such diagrams for complex accidents clearly requires some form of tool support, especially when one realises that key components of the proof, such as clause (27), are not shown. This problem could be addressed by exploiting hierarchical, graphical representations of formal proofs, such as tableaux . Hypertext display strategies might also be used to filter out the associated prose, or conversely, the mathematics at different stages during the analysis [13].



C: The proximiate cause of the casualty was the failure of Chief Officer Broekhoven, the person in charge of the watch on the NOORDAM at the time of the casualty, to maintain a vigilant watch in that he did not detect the presence of the MOUNT YMITOS visually or on the radar until the MOUNT YMITOS was less than 1 mile away and less than 2 minutes before the collision. [Conclusion 1]

forall t: not during(vigilant(broekhoven, t) iff
not(during(observe(broekhoven, mount_ymitos, visual), t), before(t, 2040)),
not(during(observe(broekhoven, mount_ymitos, arpa_radar), t), before(t, 2040)). (25)

A: The number of personnel (both watchstanding and non-watchstanding) on the bridge of the Noordam between 2020 and the time of the collision may have raised the complacency level and lowered the attentiveness of the bridge watchstanders with regards to maintaining a dedicated visual and radar watch. [Conclusion 5].

forall t:
not during(observe(broekhoven, mount_ymitos, visual), t)
iff during(position(kuiper, bridge(noordam), t),
during(position(veldhoen, bridge(noordam), t),
during(position(helmsman, bridge(noordam), t),
during(position(chief_officers_wife, bridge(noordam), t),
during(position(quartermaster_1, bridge(noordam), t),
during(position(quartermaster_2, bridge(noordam), t),
during(position(cadet, bridge(noordam), t),
during(position(broekhoven, bridge(noordam), t). (28)

A: Between 2030 and 2036, ... Veldhoean and Broekhoven used the 10-centimeter and centimeter radars to check the distance to the domino platforms, and particularly the bearing and range of the Racon 't'. They were using the radars for collision avoidance and observation of moving targets, and did not attempt to correlate every fixed target contact in the radar with fixed platforms observed visually to see if any were underway contacts rather than fixed platforms [Paragraph 39]

forall t:
during(perform(broekhoven, navigation_radar_check, t),
not(during(perform(broekhoven, correlate_radar_taregts, t) (26)

E: There were seven other persons on the bridge of the NOORDAM at this time (2037hrs) in addition to the chief officer, who was in control of the vessel - three licensed officers (one on duty and two off duty), one cadet, two quartermasters and the chief officer's wife [Paragraph 41].

exists t:
during(position(kuiper, bridge(noordam), t),
during(position(veldhoen, bridge(noordam), t),
during(position(helmsman, bridge(noordam), t),
during(position(chief_officers_wife, bridge(noordam), t),
during(position(quartermaster_1, bridge(noordam), t),
during(position(quartermaster_2, bridge(noordam), t),
during(position(cadet, bridge(noordam), t),
during(position(broekhoven, bridge(noordam), t),
before(2037, end(t)), before(begin(t), 2042). (38)

E: Between 2030 and 2036, ... Veldhoean and Broekhoven used the 10-centimeter and centimeter radars to check the distance to the domino platforms, and particularly the bearing and range of the Racon 't'. They were using the radars for collision avoidance and observation of moving targets, and did not attempt to correlate every fixed target contact in the radar with fixed platforms observed visually to see if any were underway contacts rather than fixed platforms [Paragraph 39]

exists t:
during(perfrom(broekhoven, navigation_radar_check, t),
not during(perform(broekhoven, correlate_radar_targets, t),
in(t, 2030, 2036) (15)

Figure 3: Conclusion, Analysis and Evidence diagram Integrating Formalisation and Informal Material

6 Conclusions and Further Work

This paper has argued that formal methods can support the use of natural language in accident reports. In particular, we are concerned to demonstrate that the recommendations of a report can be justified in terms of the events that are described in these documents.

6.1 Identifying Missing Information

A key point in this paper has been that informal argumentation is weakened by the omission of critical information. For example, the Noordam report did not explicitly state the reasons why Broekhoven failed to perform collision checks using the ARPA radar between 20:36 and 20:42. Similarly, it did not explain why 20:20 was cited as the critical moment when other crew members impaired operations on the bridge. Some of this detail can be inferred from other sections of the report. This is dangerous because incorrect inferences may lead readers to form inappropriate conclusions. For example, it would be wrong to assume that the number of crew members on the bridge at 20:20 should be used as a limit to the number of people allowed on the bridge of another ship [16]. Formal methods can be used to avoid such problems. We have shown how proof techniques can be used to establish that an accident report contains all of the evidence that is needed to support particular conclusions.

6.2 Structuring Critical Information

An additional benefit from the application of formal proof techniques is that they help analysts to present supporting evidence in a coherent format. By this we mean that there is a clear distinction between the axioms that model the accident and the theorems that represent the conclusions which can be made from that model. This is critical because many accident reports mix these two different types of information. For instance, the Noordam case study presents evidence about the number of people on the bridge in the body of the report [Paragraph 41]. It also presents further evidence about this in the closing sections of the report [Conclusion 5]. This gradual presentation of information forces the reader to piece together the events leading to the accident over the hundreds of pages that, typically, form an accident report. Formal proof techniques can help with this because they require that the model is built before inferences can be drawn.

6.3 Avoiding Ambiguity.

Natural language argumentation structures frequently make effective use of ambiguity. This is useful when it is difficult or impossible to provide exact information about particular events during an accident. For instance, the Coast Guard report does not represent the particular words that were uttered by Kuiper when he first observed the lights on the Mount Ymitos. It is important to note that this form of ambiguity can be modelled through the abstraction mechanisms of formal languages. Terms such as *curse* can be used in place of the precise words uttered by an operator. However, there are times when unnecessary ambiguity can have disastrous consequences for an accident report. A detailed analysis of the Noordam case study has shown considerable problems in identifying the seven individuals who were present on the bridge of the Noordam immediately before the collision [paragraph 41]. We initially identified eight people who contributed to the events leading up to the accident. This inconsistency can be explained in terms of the ambiguity of roles, such as *quartermaster*. They are ambiguous in the sense that they cannot easily be matched against the names of the individuals involved in the accident, such as the lookout Salyo, from lengthy natural language accounts. Formal analysis helps to avoid this problem by forcing analysts to explicitly represent the relationships between the terms of Table 1.

6.4 Defining Relevance

A final benefit from our formalisation is that it helps to define a notion of relevance for the material in an accident report. Information must be included if it is necessary in explaining the conclusions that are reached in the report. This is not to say that any details which are not called upon in the conclusions ought to be omitted. Much of the detail in an accident report helps to establish the context of failure rather that just the events that led to the accident. In the Noordam, this includes detailed consideration of the 'rules of the road' for maritime navigation. In contrast, we argue that information must be included if it supports the conclusions that are drawn from an investigation. This does not just relate to factual information, such as the missing timings mentioned in Section 6.1. It also refers to the supporting inferences that help to link the factual analysis to the conclusions. For example, in order to prove Conclusion 1, we were forced to explicitly state the reasons **why** Broekhoven failed to maintain a vigilant watch, see clause (23). Unfortunately, our work has shown that readers are often expected to infer the arguments that support the conclusions in accident reports.

6.5 Future Work

The techniques that are described in this paper can be applied to represent and reason about the bias that is often embodied in accident reports. This is achieved by examining the factual information and the inferences that are presented in each account. Formal analysis can then be used to identify the impact that factual omissions have for the reader. For example, very different interpretations might have been produced if the Coast Guard's report had removed information about the number of people on the bridge of the Noordam. Such an omission not only affects the timeline of events leading to failure, it also restricts the valid inferences that can be made about an accident. For instance, readers would have been forced to find alternative explanations for Broekhoven's failure to visually observe the Mount Ymitos. Formal analysis can identify these differences because it would no longer be possible to prove theorem (23). We have already demonstrated that this approach can be used to identify 'biases' in the reporting of nuclear power accidents [21]. It remains to be seen whether these techniques can be applied to accident reports from other domains. An important pre-requisite for this work is the tool support mentioned in the previous section. CAE diagrams quickly become unmanageable for the more complex proofs that are required to demonstrate the impact of bias between different accident reports.

It is important to emphasise that CAE diagrams and formal proof techniques are not intended to replace the natural language presentation of accident reports. In contrast, our intention is to provide a clear and coherent structure for the argumentation in accident reports [20]. This additional degree of precision is required if companies are to use these documents as a means of guiding future development and investment decisions.

Acknowledgements

The research reported in this paper has been supported by UK Engineering and Physical Sciences Research Council grants GR/J07686, GR/K55042 and

GR/K69148. I would like to thank the members of the Department of Computing Science, University of Glasgow for their support and encouragement in this work.

References

[1] Air Accidents Investigations Branch, Department of Transport. *Report On The Accident To Boeing 737-400 G-OBME Near Kegworth, Leicestershire on 8th January 1989*, number 4/90, Her Majesty's Stationery Office. London, United Kingdom, 1990.

[2] S. Austin and G.I. Parkin. Formal Methods: A Survey, Division Of Information Technology And Computing, The National Physical Laboratory, Sponsored by the United Kingdom Department of Trade and Industry, Teddington, United Kingdom, 1993.

[3] S. Buckingham Shum, Analysing The Usability Of A Design Rationale Notation. In T.P. Moran and J.M. Carroll (eds.) Design Rationale Concepts, Techniques And Use, Lawrence Erlbaum, Hillsdale, New Jersey, United States of America, 1995.

[4] Cullen, *Proceedings Of The Public Enquiry Into The Piper Alpha Disaster*. The Department of Energy, Her Majesty's Stationery Office, London, United Kingdom, 1990.

[5] A.J. Dix, *Formal Methods For Interactive Systems*, Academic Press, London, United Kingdom, 1991.

[6] D. Fennel, *Investigation Into The Kings' Cross Underground Fire*. Department of Transport, Her Majesty's Stationery Office, London, United Kingdom, 1988.

[7] C.W. Johnson, *A Probabilistic Logic For The Development of Safety-Critical Interactive Systems.* International Journal of Man-Machine Systems, 39(2):333-351, 1993.

[8] C.W. Johnson, *The Formal Analysis Of Human-Computer Interaction During Accident Investigations.* In G. Cockton, S.W. Draper and G.R.S. Weir, editors, People And Computers IX, 285-300. Cambridge University Press, Cambridge, 1994.

[9] C.W. Johnson, *The Application of Petri Nets to Represent and Reason about Human Factors Problems During Accident Analyses* In P. Palanque and R. Bastide, editors, The Specification And Verification Of Interactive Systems, 93-112, Springer Verlag, Berlin, 1995.

[10] C.W. Johnson, *Using Z to Support the Design of Interactive, Safety-Critical Systems* Software Engineering Journal, (10)2:49-60, 1995.

[11] C.W. Johnson, *Impact of Working Environment upon Human-Machine Dialogues: A Formal Logic for the Integrated Specification of Physical and Cognitive Ergonomic Constraints on User Interface Design.* Ergonomics (39)3:512-530, 1996.

[12] C.W. Johnson, *Literate Specification.* Software Engineering Journal, (11)4:225-237, 1996.

[13] C.W. Johnson, *Literate Specification: Using Design Rationale To Support Formal Methods In The Development Of Human-Machine Interfaces.* Human Computer Interaction Journal, (11)4;291-320, 1996.

[14] C.W. Johnson, J.C. McCarthy and P.C. Wright, *Using A Formal Language To Support Natural Language In Accident Reports.* Ergonomics, (38)6:1265-1283, 1995.

[15] P. Johnson, D. Diaper and J. Long, *Task, Skills And Knowledge: Task Analysis For Knowledge Based Descriptions.* In B. Shackel (ed.) Human-Computer Interaction - INTERACT' 84, Elsevier Science Publications, North Holland, Netherlands, 23-27, 1984.

[16] B.H. Kantowitz and P.A. Casper. *Human workload in aviation.* In E.L. Wiener and D.C. Nagel, editors, Human Factors In Aviation, 157-187. Academic Press, London, United Kingdom, 1988.

[17] National Transportation Safety Board, Aircraft Accident Report, American Airlines Inc. DC-10 at Chicago Illinois, May 25, 1979, NTSB AAR-79-17, Washington DC.

[18] P. Palanque and R. Bastide, Formal Specification and Verification of CSCW Using The Interactive Cooperative Object Formalism, In M.A.R. Kirby, A.J. Dix and J.E. Finlay (eds.), People and Computers X, 213-232. Cambridge University Press, Cambridge, 1995.

[19] F. Paterno, M.S. Sciacchitano and J. Lowgren, A User Interface evaluation, Mapping Physical User Actions to Task-Driven Formal Specifications. In P. Palanque and R. Bastide (eds.), Design, Specification and Verification of Interactive Systems '95, 35-53, Springer Verlag, Berlin, 1995.

[20] J. Reason, *Human Error*, Cambridge University Press, Cambridge, United Kingdom, 1990.

[21] A.J. Telford and C.W. Johnson, *Extending the Application of Formal Methods to Analyse Human*

Error and System Failure During Accident Investigations. Software Engineering Journal, (1106:355-365, 1996.

[22] United States Coast Guard, *Investigation into the Circumstances Surrounding the Collision Between the Passenger Vessel Noordam (NA) and the Freight Vessel Mount Ymitos (MT),* United States Department of Transport, Washington DC, 1993.

[23] N. Worley and J. Lewins (editors), *The Chernobyl Accident And Its Implications For The United Kingdom*, Report Number 19 Of The Watt Committee On Energy, Elsevier Science, North Holland, 1988.

Formalization and Analysis of the Separation Minima for Aircraft in the North Atlantic Region

Nancy A. Day
day@cs.ubc.ca
Department of Computer Science
University of British Columbia
2366 Main Mall, Vancouver, BC, Canada V6T 1Z4

Jeffrey J. Joyce, Gerry Pelletier
{jjoyce,gpelletier}@ccgate.hac.com
Hughes International Airspace Management Systems
13951 Bridgeport Rd,
Richmond, BC, Canada V6V 1J6

Abstract

The formalization and analysis of an air traffic control separation minima serves in this paper as an illustration of an approach that uses formal operational semantics to drive the automated analysis of specifications. This contrasts with the approach of translating one notation into the input format for an analysis tool, or hard-coding the semantics of a particular notation into the implementation of an analysis technique.

The semantic functions capture the structure of the specification and can be directly evaluated to map a notation to a rigourous mathematical foundation. This work contributes to a greater appreciation of how the structure of a specification (e.g., the organization of a table), not just the semantics, is an important input to many analysis functions. Building upon a common mathematical foundation, different notations can be combined to support an integrated approach to the analysis of a formal specification. A related issue is the importance of being able to reverse the effect of the semantic functions so that analysis results are provided to users at the same level of abstraction used in the input specifications.

The formalization of the separation minima combines the use of a tabular style of specification with predicate logic. This paper discusses how automated analysis functions were applied to the specification to check for the properties of consistency, completeness and symmetry. The benefit of doing this analysis is demonstrated by the discovery of an ambiguity in the separation minima.

1. Introduction

This paper describes work carried out at the University of British Columbia in collaboration with Hughes International Airspace Management Systems to formalize and validate a specification of the separation minima for aircraft in the North Atlantic (NAT) region. Our formalization is based on a description provided in a source document published by Transport Canada on behalf of ICAO[1]. This document describes the official North Atlantic Separation Minima as published by ICAO. This description provides guidance to air traffic controllers managing the region of oceanic airspace between Europe and North America. It is also used as the basis for the development of software based systems that support the management of the NAT region. For example, it would be used during the planning of a flight from New York to London to check whether the route is free from separation conflicts with other aircraft expected to be in the NAT region at the same time.

This collaboration has directly involved domain experts (not just formal methods experts) in the process of developing and analyzing a formal representation of a complex "real" description. The source document is an informal specification that has been scrutinized by the NATSPG (NAT Systems Planning Group) members who are ATC specialists from the NAT countries, and most of them maintain and use automated systems that implement these rules.

Our formal representation of these separation minima is given in a mixture of a tabular style of specification and a variant of higher order logic called "S" [11].

[1]This document, "Application of Separation Minima for the NAT Region" (3rd edition, effective December 1992), was published by Transport Canada on behalf the ICAO North Atlantic Systems Planning Group. ICAO is the International Civil Aviation Organization with headquarters in Montreal, Canada. This separation minima document was developed by the COMAG (Communications and ATM Automation Group), now called the CADAG (Communication, Automation and Data Link Applications Group).

Combining multiple notations makes it possible to choose the notation best suited to the various parts of the specification. The tabular style was chosen because the rules consist of complex decision logic describing predicates and functions. To unite the various parts, including environmental assumptions, in a common framework for analysis, the tables are considered a "style" of specification in the S notation. A "style" of specification includes associated semantic functions for the constructs that are introduced, which makes it possible to capture the structure of the specification and give meaning to the notation.

Once in a common environment, the specification can then be analyzed for various properties. These range from "style" independent properties, such as typechecking and symmetry, to properties particular to the notation being used. This paper describes the analysis of the completeness and consistency of the tabular specifications. Symmetry is a particularly desirable property of the separation minima, so we also describe how the same analysis mechanisms used for completeness and consistency are used to do this check. The most significant result of the analysis was the discovery of two tables in the specification with inconsistencies, where, for the same scenario, the specification indicated two different amounts of aircraft separation.

The formal specification and related analysis results can be found on-line at http://www.cs.ubc.ca/spider/day/Research/ SeparationMinima/SeparationMinima.html .

This work tests the doctoral thesis hypothesis of the first author, Day. This hypothesis is that explicit definitions of the operational semantics of a notation can be used directly in the analysis and that this method retains the domain knowledge captured by the structure of the specification, which can be exploited in analysis. This structure can be used to help convert the specification to a finite model and to determine the correct level of abstraction for presentation of the results of analysis. This paper outlines the framework for using operational semantics for analysis. We have implemented this framework, including the analysis techniques described in this paper in a tool called Fusion. The overall goal of this work is to make it possible to perform fast, automatic, lightweight checks to streamline the validation of specifications. The automated checks do not provide absolute assurance, but they isolate details that can be reviewed independently. The individual analysis checks described here usually took about 1 second of execution time on a Pentium-120 with 16 MB running Linux.

2. Related Work

2.1 Notation

Since the separation minima is a specification of combinations of conditions that produce different outcomes, a tabular style of specification seemed suitable. Previous successful efforts of using tables provide a good precedent for the readability of a tabular style of specification. These efforts include the AND/OR tables of the TCAS II project[12] and the Software Cost Reduction (SCR) notation used in the A-7 aircraft Operational Flight Program[9]. Initially, we considered using either AND/OR tables, or the style of tables presented by Parnas [16]. SCR tables are typically for system specifications that involve "modes" of operation and the separation minima does not have this characteristic.

An AND/OR table consists of a series of rows labeled by predicates. The columns to the right of the label contain "T" for true, "F" for false, or "." for "don't care". The cell is meant to represent the case where the condition given by the label is true or false. A "don't care" value means that the cell could contain either true or false. The table represents a predicate that is true if the conjunction of the cells in any column results in true.

A difficulty with AND/OR tables is that they only represent predicates. In the separation rules, sets of conditions are used to describe cases for different return values of functions.

The other approach considered was the tabular style presented by Parnas [16] which allows for the grouping of related conditions along a row. Grouping is achieved by allowing each different argument of the predicate (or function) represented by the table to have its own dimension. Hence, this style is best suited for capturing functions of a small number of dimensions which is not the case for the tables we expected to construct in our formal representation of the NAT separation minima.

2.2 Analysis

In the TCAS II specification, a table can be used to describe the condition for taking a transition in a state machine. In the completeness and consistency analysis carried out by Heimdahl and Leveson[6], the specification is considered complete if a transition is always enabled from a state. It is consistent if the specification is deterministic, i.e., if no two transitions can be enabled at the same time.

In the TCAS II AND/OR tables the cells in the rows can contain only true, false, or "don't care". In the analysis, a Boolean variable is associated with each row label. The meaning of each cell in the row is the condition of whether this Boolean variable is true or false. This allows for an efficient implementation using Binary Decision Diagrams (BDDs) [2]. Completeness analysis checks that the disjunction of the columns of all the tables used to describe transitions from a given state is a tautology. Consistency analysis checks that there is no overlap in the conditions between multiple tables describing transitions from the same state, i.e., the conjunction of the meaning of two tables is a contradiction. Checking if the BDD representation of an expression is a tautology or a contradiction takes constant time.

A difficulty with AND/OR tables is that related conditions such as "x < 280" and "x > 450" are listed on separate rows and therefore the structure of the table does not capture the relationship between these terms. Related conditions are associated with different Boolean variables. This can result in the analysis producing false negatives. For example, it might return a bogus result indicating that no table covers the case where both the conditions "x<280" and "x>450" are true. The tool created by Heimdahl and Leveson catches false negatives with respect to enumerated types, but not those arising from the use of mathematical functions. They are investigating linking their analysis with a theorem prover [6].

Although SCR tables are not applicable to the separation minima, their analysis techniques are relevant. Heitmeyer, Jeffords and Labaw [7,8] describe work on checking the completeness and consistency of condition tables given in the SCR notation. They also define completeness as a coverage property - that the disjunction of the conditions in a row is a tautology. Currently, they limit themselves to conditions ranging over Boolean values or those that have been converted by hand to Boolean variables. Expressions involving relations are also converted manually into Boolean variables. They are working on techniques to reason about conditions involving mathematical functions. Their analysis of the condition tables for the Operational Flight Program of the US Navy's A-7 aircraft found 17 legitimate errors in 36 tables with a total of 98 rows. Two false errors were found due to their strictly Boolean interpretation of the specification. Given the manual encoding to Boolean variables, these results must have been mapped by hand back to the correct level of abstraction for interpretation.

Both of these previous examples are control-oriented systems. In systems that contain a great deal of data complexity, such as these separation minima, it becomes more important to capture and utilize the relationships among data values.

Our work is similar to that carried out by Owre, Rushby, and Shankar where they have added a table construct to the PVS theorem prover[14,15]. The theorem prover is used to address some of the deficiencies of a strictly BDD-based approach. They follow the same approach as us of semantically embedding decision tables within higher order logic. The checks for completeness and consistency are carried out by proving type correctness conditions for the tables. This requires minimal theorem proving effort when the tables are complete and consistent but it appears that some effort is required to extract the cases not covered or the inconsistent cases.

Our effort documented here also attempts to address the difficulties of a strictly BDD-based approach while staying within the realm of lightweight, fully automatic techniques. In particular, we show how the structure of the table often can be utilized to eliminate the need for a more heavyweight tool such as a theorem prover. Our approach to executing the semantic definitions using symbolic functional evaluation as is done in functional programming languages also differentiates this work. Finally we demonstrate the value of including environmental constraints in the analysis process and present a simple approach for dealing with quantification to make this possible.

3. Specification Notation

At the beginning of this project, the first author was presented with an interpretation of the separation minima expressed as pseudo code (a draft documented dated 20 Sept 95). This interpretation was created by a third party to provide software developers with an algorithmic interpretation of the English text and diagrams contained in the NAT region separation minima specification. This pseudo code imposed an order of evaluation on the conditions as well as other implementation details. The imperative programming style of specification used involves

					Default
A.FlightLevel	_ <= 280	.	_ > 450	_ > 450	
B.FlightLevel	.	_ <= 280	_ > 450	_ > 450	
IsSupersonic (A)	.	.	_ = T	.	
IsSupersonic (B)	.	.	.	_ = T	
VerticalSeparationRequired (A,B)	**1000**	**1000**	**4000**	**4000**	**2000**

Figure 1: Vertical Separation

assignments of default values to variables followed by if-then-else statements to modify these variables, as well as procedure calls. Most of the conditions of the if-then-else statements were expressed in terms of English phrases.

Our goal became to formalize the separation minima using a notation that did not impose implementation constraints and that was amenable to analysis so we could determine which cases were being covered by the default values. This effort also required sorting out the variety of English phrases used to describe various conditions to yield a "dictionary" of primitives that were then introduced in the formal representation as uninterpreted functions and predicates.

Our review of previous work using tabular specifications led to the use of a variation of AND/OR tables. This variation allows related conditions to be captured within a row in a style closer to the idea of decision tables given in the structured analysis methodology of DeMarco [3]. A row isolates one dimension of the decision and the columns relate the different dimensions to produce a case. A table can also represent functions through the addition of a row of return values.

Figure 1 is an example of our tabular style of specification; this particular table specifies the minimum vertical separation (in feet) that must exist between two aircraft for them to be considered separated in the NAT region. The name of the function and the arguments to the function are given in the last row of the table which gives the return values of the function. Except for the last row of the table, the label of a row (the leftmost column) is an expression. The cells of the row are predicates that can be applied to this label to produce the condition that the cell represents. The parameter of the predicate is given by the "_" in the cell. A "." means "don't care", i.e., the predicate is always true. Then, as with AND/OR tables, the conjunction of the cells in any column is the case where the function returns the value in the last row for that column.

A semantically equivalent representation in S of the function given in Figure 1 is:

```
VerticalSeparationRequired(A:flight, B:flight) :=
        if (A.FlightLevel <= 280) then 1000
        else if (B.FlightLevel <= 280) then 1000
        else if ( (A.FlightLevel > 450) AND
                  (B.FlightLevel > 450) AND
                  (IsSupersonic (A) = T) ) then 4000
        else if ( (A.FlightLevel > 450) AND
                  (B.FlightLevel > 450) AND
                  (IsSupersonic (B) = T) ) then 4000
        else 2000;
```

The arguments A and B represent flights. In S, the "dot" notation as used in the expression "A.FlightLevel" is merely syntactic sugar for function application. "A.FlightLevel" is interpreted by an S parser as "FlightLevel (A)" to allow for the representation of static information about an item in the familiar "record" type of notation.

In addition to standard, "built-in" predicates such as "<=" and ">", the formalization also involved the introduction of uninterpreted types and constants. An uninterpreted constant has a type but no definition. For example, the following S declarations introduce an uninterpreted type[2] ("flight"), an uninterpreted function ("FlightLevel") and an uninterpreted predicate ("IsSupersonic"):

```
:flight;
FlightLevel : flight -> num;
IsSupersonic : flight -> bool;
```

[2] Uninterpreted types are analogous to "basic types" in a Z specification[17].

The use of uninterpreted terms allowed us to phrase the specification in domain terminology and to match the level of abstraction appropriate for this specification.

The table for vertical separation in Figure 1 specifies a function. In the case of a predicate (i.e., a function that returns a Boolean value) the bottom row of return values can be omitted and the cases designated by the columns are assumed to return true for the predicate. Any other cases are assumed to return false.

These tables also allowed us to match the modular nature of the decomposition of the separation minima. For example, longitudinal separation between two aircraft that are both turbojet depends on the current airspace (MNPS or WATRS[3]) of the aircraft. These cases are given in separate tables.

Since we were working within the S environment, we were also able to use definitions of functions in predicate logic rather than tables in some cases. These function could reference tables and tables could reference functions defined in S. This illustrated the benefits of being able to combine multiple notations.

4. Formalization Process

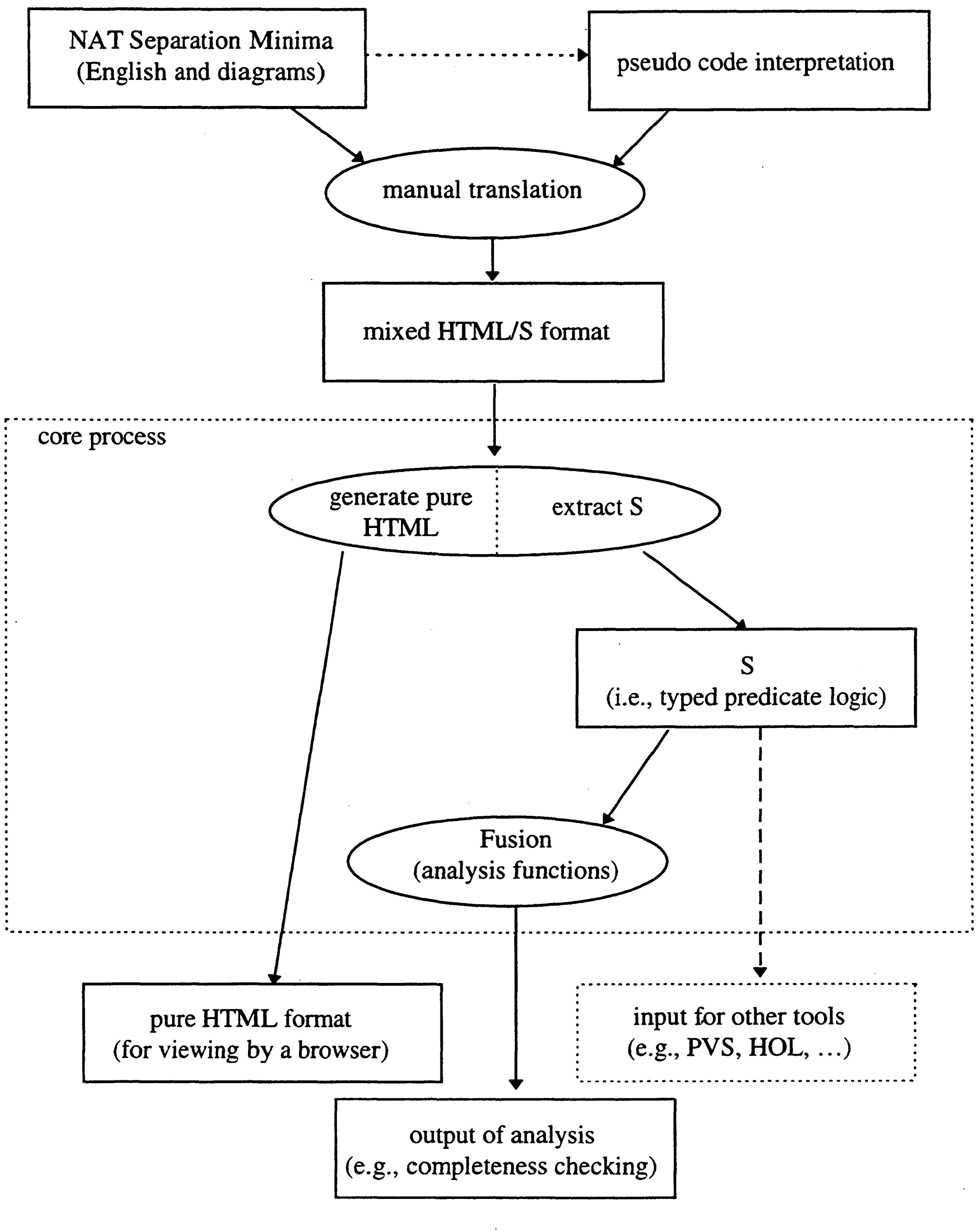
Figure 2 illustrates our formalization and analysis process. An important element in gaining industry acceptance of any formal method is the form in which the specification is presented for review to non formal methods experts. We chose to use HTML so that changes to the specification could be quickly viewed by all authors and so that cross references in the document between the use and definition of terms could be given using hyperlinks. This made it possible to give supplementary text to describe the formal tables. The top-down presentation given by this document also has advantages over the bottom-up order (i.e. declaring or defining terms before they are used) which is expected by analysis tools. However, it was also necessary to have a version of the specification that could be input to the analysis tools. To eliminate the difficulty of having to maintain versions of the specification existing in different forms, we created one document that is a mixture of HTML and formal notation. We used a preprocessor that produces the specification in pure HTML (using HTML tables for the formal tables) and automatically generates links from references to the declarations and definitions of terms[4]. It also produces a separate file containing only the representation in formal notation in the correct order which is used as input to the analysis tool.

The first draft of the formal specification was created by the first author based on a pseudo code representation of the separation minima. The first step in the formalization process was to determine the primitives of the specification and introduce these as uninterpreted functions and predicates in S. The pseudo code is modular so parts of it can be matched to individual tables. For each table, the relevant "inputs" were determined and used as the labels for the table rows. Columns in each table were then created as given by the logical combinations of these inputs. Typechecking was used to validate the first draft of our formal representation of the separation minima.

The first draft of the specification was handed off to the third author who is the domain expert. The only explanation of the tables that he was given was one paragraph of text with an example at the beginning of the document. From this, he edited the HTML version of the document and supplied the first author with a revised draft of the specification. This included smaller changes, such as terminology, and more meaningful changes, such as clarification of ambiguous phrases identified by the first author, such as, "a portion of the routes of both aircraft are within OR above OR below MNPS airspace". The correct interpretation of this phrase is that each aircraft is considered independently with respect to MNPS airspace, as opposed to both aircraft having to be within MNPS airspace or both having to be above MNPS airspace, etc. Relationships between various primitive terms were also identified.

[3] MNPS is Mininum Navigational Performance Specification. WATRS is West Atlantic Route System.

[4] This preprocesser was developed in the course of this work but is a separate tool that could be used for various specification notations. Information is available at http://www.cs.ubc.ca/spider/day/Research/hpp.html .



NAT Separation Minima (English and diagrams)
pseudo code interpretation
manual translation
mixed HTML/S format
core process
generate pure HTML
extract S
S (i.e., typed predicate logic)
Fusion (analysis functions)
pure HTML format (for viewing by a browser)
input for other tools (e.g., PVS, HOL, ...)
output of analysis (e.g., completeness checking)

Figure 2: Process Overview

The most notable absence from the pseudo code was the top level requirement stating what separation means. Two aircraft are separated if they satisfy the separation minima for at least one dimension, i.e. vertical, lateral, or longitudinal. The criteria for each of these is given in different units. The top level requirement as stated in S is given in Figure 3. In this requirement, "ABS" takes the absolute value of its argument. Vertical separation is measured in feet. Lateral separation is measured in miles (or equivalently in degrees of latitude). Longitudinal separation is measured in minutes. Two aircraft on opposing tracks cannot be considered longitudinally separated during a certain range of time when the aircraft are close to crossing. Vertical or lateral separation must exist during that time.

```
AreSeparated(A:flight,B:flight,t:time) :=
  /* A and B are vertically separated based on flight level */
  (ABS(A.FlightLevel- B.FlightLevel) > VerticalSeparationRequired(A,B))
  OR
  /* A and B are laterally separated based on either position in degrees of latitude or position in miles */
  (if (LatitudeEquivalent(A,B)) then
       (ABS(A.LateralPositionInDegrees  -  B.LateralPositionInDegrees)  >LateralSeparationRequiredInDegrees
(A,B))
   else
    (ABS(A.LateralPositionInMiles - B.LateralPositionInMiles) > LateralSeparationRequiredInMiles (A,B)))
  OR
  /* A and B are longitudinally separated based on time, depending on whether the two flights are in the
     approximate same or opposite direction */
  (if (AngularDifferenceGreaterThan90Degrees(A.RouteSegment,B.RouteSegment)) then
     /* opposite direction */
     NOT (WithinOppDirNoLongSepPeriod(A,B,t))
  else   /* same direction */
     ABS(A.TimeAtPosition - B.TimeAtPosition) >LongSameDirSepRequired(A,B));
```

Figure 3: Top level Specification of Separation

The addition of the top level requirement pointed out that the proper distinction had not made between minima for aircraft on opposing tracks and those on same direction tracks. The requirement on aircraft flying the same direction is a minimum number of minutes of separation. The requirement for aircraft flying in opposing directions is that some other form of separation must exist during the time period when the aircraft cross. This change mainly affected the statement of the top level requirement.

The resulting specification consisted of 15 tables, 16 definitions in S, and 47 uninterpreted constants. The largest table consisted of 8 rows and 6 columns.

5. Analysis

Our goal was to analyze the completeness, consistency, and symmetry of the tables in the separation minima specification. Completeness checking automatically determines the cases that are covered by the default column, or if no default is given, the cases that are not covered in the table. Consistency analysis returns pairs of columns with different return values that both include a set of conditions that can be true at the same time. Symmetry analysis determines if the table has the same meaning when its arguments (the pair of flights) are given in the opposite order.

All of these forms of analysis are based on the possible combinations of entries in the rows of the table. They cannot determine if some aspect of the decision given by the table has been omitted.

The framework for specification and analysis proposed in the thesis work of the first author is illustrated in Figure 4. Requirements specifications, possibly given in multiple notations are placed within a common logical framework using an embedding that closely matches the original notation and does not lose the structure of the specification. Semantic functions define the meaning of the embedded notation in logic. The semantic functions also indicate explicit join points for how multiple notations fit together, such as a predicate table being used to describe the condition on a transition in a statechart [5] similar to what is done in RSML (Requirements State Machine Language) [12].The keywords used in the embedding and their associated semantic functions are called a "style" of specification in S.

These semantic functions are executable in the sense that they map a structured specification, such as a table, into an expression in logic. The expression in logic is called the semantic representation in the diagram. One method of executing these semantics is to use rewrite rules within a theorem prover. However, this is a more general mechanism than is needed for functions known to be executable. Drawing on techniques from functional programming language implementations, Fusion includes a symbolic

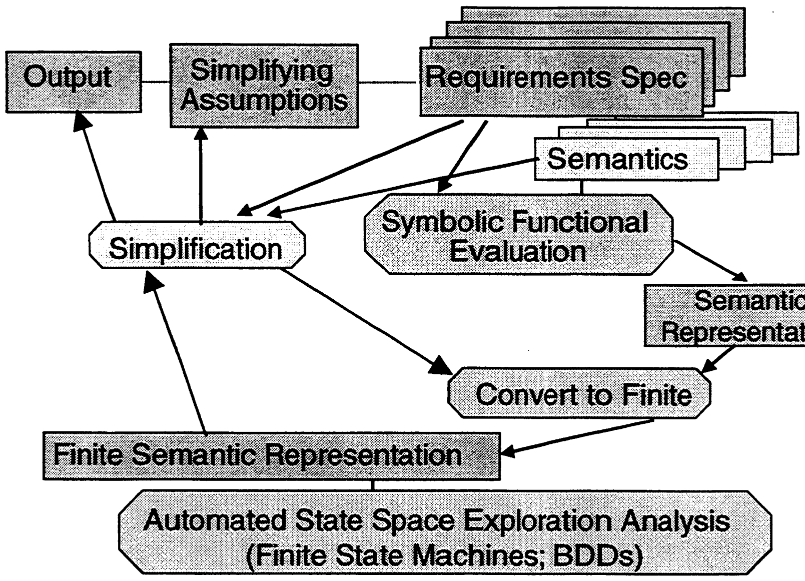

Output
Simplifying Assumptions
Requirements Spec
Semantics
Symbolic Functional Evaluation
Simplification
Semantic Representation
Convert to Finite
Finite Semantic Representation
Automated State Space Exploration Analysis (Finite State Machines; BDDs)

Figure 4: Specification and Analysis Framework

functional evaluator which can execute statements in S, stopping when it reaches uninterpreted constants. This engine carries out evaluation in place for efficiency, as is done in the implementation of a functional programming language.

Separately, clues given in the structure of the specification can be used to help determine simplifications to map the semantic representation into a finite model that can be efficiently represented using BDDs and analyzed using automatic means. Sometimes the simplification implied by the structure is not a valid abstraction. If the tool can not determine the validity of the simplifications these are stated to the user as "assumptions". This serves to the reduce the review process to one of evaluating isolated assumptions.

The results of the analysis map the simplifications back into the terms of the specification for the user to examine.

We regard tables to be a "style" of specification in the S language where the textual specification of the table in S is structurally close to the tabular presentation. Figure 5 shows the S representation of the table for "VerticalSeparationRequired" given in Figure 1. Our preprocessor turns this representation into an HTML table.

The keyword "Row" is a semantic definition that substitutes the label of each row into the list of predicates. The "_" describing the parameter of the predicate in the HTML version is given by a lambda variable. "DC" is a predicate that always returns the value true. This takes the place of the "." in cells in the HTML representation. "TRUE" is a predicate that states that its parameter must have the value "true". The result of applying the "Row" function is a list of elements with Boolean values.

The keyword "Table" gives meaning to the table, matching the conjunctions of values in the columns with return values. These semantic functions are defined in S and given in Appendix A.

```
VerticalSeparationRequired (A,B) := Table
        [Row (A.FlightLevel)    [(\x.x <= 280);      DC;        (\x. x > 450); (\x.x > 450)];
         Row (B.FlightLevel)    [  DC;         (\x.x <= 280); (\x. x > 450); (\x.x > 450)];
         Row (IsSupersonic (A)) [  DC;               DC;          TRUE;          DC     ];
         Row (IsSupersonic (B)) [  DC;               DC;          DC;            TRUE   ]]
   [1000;1000;4000;4000;2000];
```

Figure 5: VerticalSeparationRequired Table in S

A.FlightLevel	_<=280	(280 < __) AND (_ <= 450)	450 < __

Figure 6: Example Row

Having related conditions in a row uses the structure of the table to show how the user views the possible values of the row label. For example, in the table found in Figure 1 for vertical separation, the flight level of each aircraft is only important in how it compares to flight levels 280 and 450 for the purposes of the function given by the table. Therefore the analysis can assume that the specifier would like the analysis results given in these terms as well. This is in keeping with our goal of returning results at the same level of abstraction as the specification.

The simplification engine uses the entries in a row to partition the state space for the aspect of the problem given by the label of that row. This partition can then be encoded in Boolean variables just as an enumerated type can be encoded for automatic verification (as seen in [1,10]). For example, if a row contained the entries found in Figure 6, there are three conditions given by the cells in this row:

A.FlightLevel <= 280
(280 < A.FlightLevel) AND (A.FlightLevel <=450)
450 < A. FlightLevel

The flight levels of A have been divided into three ranges. To represent these ranges, we need two Boolean variables, say a1 and a2. We can use the encoding given in Figure 7. The use of structure eliminates the need for the more heavyweight reasoning of a theorem prover in this instance.

This encoding results in one leftover possible encoding for the Boolean variables of a1=true and a2=true. Since this does not correspond to any possible real case, we add this to the final expression to check in the analysis to ensure that this artifact does produce an extraneous result.

This technique does make the assumption that the partition the user provides is complete and consistent. This is not always the case. Some readers may have noticed that for the table "VerticalSeparationRequired" in Figure 1, using only the elements in the first row leaves out the case for when the flight level is between 280 and 450. An earlier version of the tool stated these assumptions about the partition to the user. This approach isolated details of the specification to review separately. The conditions given by the partition could also be evaluated by an automated decision procedure in a theorem prover such as PVS [13].

After noting that these assumptions are incorrect (i.e., a range is missing), one possible remedy is to modify the table. However, in all cases for the tables in this specification, the predicates over numeric values consist of a comparison to a concrete value. To improve the accuracy of the analysis results, we added a simple interval checker which checks the partition of the range given by the elements of the row and adds any ranges not mentioned explicitly in the row. After this addition, one table remained with overlapping ranges in the elements of the row. The interval checker identified this case and we modified the table to separate the ranges into multiple rows and used environmental assumptions to relate the rows.

There are many times when the possible values given by the element labeling the row are known. An example is a Boolean condition such as "IsSupersonic" found in the "VerticalSeparationRequired" table. This element can take on the values true and false even though the row for it in this particular table never has a false case. The checker automatically recognizes these situations and considers both true and false as possible values in the analysis. This is applicable for any values of finite types. Enumerated types can be declared using S declarations.

A.FlightLevel <= 280	NOT(a1) AND NOT(a2)
(280 < A.FlightLevel) AND (A.FlightLevel <=450)	a1 AND NOT(a2)
450 < A. FlightLevel	NOT(a1) AND a2

Figure 7: Example Encoding

```
>%include minima.s
Including rules.s
>%comp VerticalSeparationRequired
VerticalSeparationRequired is:
(
 (Table
  [
   ((Row (FlightLevel A))
    [(\x.(x <= 280));DC;(\x.(x > 450));(\x.(x > 450))]);
   ((Row (FlightLevel B))
    [DC;(\x.(x <= 280));(\x.(x > 450));(\x.(x > 450))])
   ;((Row (IsSupersonic A)) [DC;DC;TRUE;DC]);
   ((Row (IsSupersonic B)) [DC;DC;DC;TRUE])])
 [1000;1000;4000;4000;2000])

Invoking interval checker...

Interval checker partitions the range into:
((FlightLevel A) > 450)
((280 < (FlightLevel A)) AND ((FlightLevel A) <= 450))
((FlightLevel A) <= 280)

Invoking interval checker...

Interval checker partitions the range into:
((FlightLevel B) > 450)
((280 < (FlightLevel B)) AND ((FlightLevel B) <= 450))
((FlightLevel B) <= 280)

The following cases
yield the default value of 2000
Case 1
Row 1 : ((280 < (FlightLevel A)) AND ((FlightLevel A) <= 450))
Row 2 : ((280 < (FlightLevel B)) AND ((FlightLevel B) <= 450))
Row 3 : DC
Row 4 : DC

Case 2
Row 1 : ((FlightLevel A) > 450)
Row 2 : ((280 < (FlightLevel B)) AND ((FlightLevel B) <= 450))
Row 3 : DC
Row 4 : DC

Case 3
Row 1 : ((280 < (FlightLevel A)) AND ((FlightLevel A) <= 450))
Row 2 : ((FlightLevel B) > 450)
Row 3 : DC
Row 4 : DC

Case 4
Row 1 : ((FlightLevel A) > 450)
Row 2 : ((FlightLevel B) > 450)
Row 3 : ((IsSupersonic A) = F)
Row 4 : ((IsSupersonic B) = F)


Stats for VerticalSeparationRequired completeness checking:
Number of cases identified: 4
Total time: 1 sec
-----------------------------------------------------------
>
```

Figure 8: Completeness Checker Output

5.1 Completeness Checking

Checking the completeness of a table means determining if all possible cases are covered by the specification. The meaning of a column is the conjunction of the predicates in the row cells. For the tables, completeness checking involves checking whether the expression denoting the disjunction of the columns is a tautology. By applying the semantic function for the meaning of a predicate table and using the Boolean encodings identified by the simplification engine described above, we can evaluate this check efficiently.

Figure 8 shows a Fusion session in which the completeness analysis function "comp" is applied to the table "VerticalSeparationRequired". Every line shown in Figure 2 is generated by Fusion except for the two user commands which appear on the lines that begin with the user prompt, ">". The first command "%include minima.s" causes the S representation of the separation minima to be parsed and typechecked. The second command "%comp VerticalSeparationRequired" causes the completeness analysis function to be applied to the table "VerticalSeparationRequired".

The completeness analysis function first invokes the interval checker for the first two rows. It correctly determines the missing range for the possible values of the "FlightLevel" of the aircraft.

The completeness analysis function then generates a list of the cases covered by the default column. The analysis reveals four of these cases. In order to minimize the amount of output generated by the checker, the results are given in an approximation of minimal sum-of-products form. This can be seen in the use of "don't care" ("DC") values for some of the rows. It was important that these results were given in terms of the unexpanded row label in cases where the row label was an application of a function defined elsewhere. This meant the reviewer could easily match the cases given in the results to the original table in HTML after substituting the row label into the blank.

In reviewing this session output, the domain expert would decide whether the default value, 2000, is appropriate for the listed cases. The maximum number of default cases revealed by the completeness analysis for this specification was 50 - for a table of eight rows and six columns. A table is not necessarily flawed because it has default cases — but it is important for the default cases to be enumerated and reviewed by a domain expert. This kind of analysis would be performed by some means in a disciplined system development process. However, the use of an automated completeness analysis function, as illustrated here, streamlines and systematizes the review process by enumerating the default cases explicitly. A possible method of evaluating these would be to iteratively examine a single case, determine whether it is an error or not, and then add it, likely in a generalized format (i.e., with "don't care" values in some of the cells) to the table. This approach would mean that the default cases would gradually be fully specified. Thus, the use of Fusion for this purpose can also be seen as a way to measure the quality of a specification.

In Heimdahl and Leveson's work [6], they are able to draw conclusions about the overall completeness of a specification by referring to a functional definition of the semantics. For this specification, we can ask whether the completeness of individual tables ensures the completeness of the overall specification. Given that the tables represent functions, if the other parts of the specification (including the uninterpreted functions) represent total functions then we can conclude that the specification is complete, assuming the scope of each table is complete.

5.2 Environmental Assumptions

In reviewing the output of the completeness checker, our domain expert pointed out that some of the cases produced were impossible. These were situations where the rows within a table were related to each other. For example, an aircraft cannot satisfy both of the constraints "InCruiseClimb" and "IsLevel" at the same moment.

These constraints are information about the physical limitations of the items involved in the specification. They can be considered assumptions about the environment. We documented these in S using expressions such as:

```
forall (F:flight).
  mutually_exclusive(InCruiseClimb F, IsLevel F)
```

where "mutually_exclusive" is defined to mean only one of its arguments can be true.

These expressions are evaluated using the symbolic functional evaluator as for the tables. However in order to reduce these expressions to the terms used in the tables, we substituted any existing items of the correct type as the parameter in the "forall" expression. Most of the tables involve two flights, A and B. The above environmental assumption would evaluate to:

```
mutually_exclusive(InCruiseClimb A, IsLevel A)
AND
mutually_exclusive(InCruiseClimb B, IsLevel B)
```

Substitutions determined by the simplification engine for the table, along with some additional Boolean variables (since all environmental assumptions are not relevant to every table) are used to encode the environmental assumption.

The addition of environmental constraints slightly changed the method of evaluating the completeness of a table. Instead of checking whether the meaning of the table was a tautology, we had to check whether the environment conjoined with the negation of the meaning of the table was a contradiction.

In the output, the environmental assumptions are not listed. They are existentially quantified out of the results.

5.3 Consistency Checking

Consistency checking involves comparing each column of a table to all other columns within the table that have a different return value to see if the cases denoted by the columns overlap. The same evaluation

```
>cons otherSameDirLongSep env
otherSameDirLongSep is:
(
 (Table
  [((Row (ReportedOverCommonPoint (A , B))) [TRUE;DC]);
   ((Row (SameOrDivergingTracks (A , B))) [TRUE;DC]);
   ((Row ((AllOf [A;B]) (\x.((IsOnRoute Routes3) x))))
     [DC;TRUE])]) [15;20;30])

Columns 1 and 2 conflict in the following:
Case 1
Row 1 : ((ReportedOverCommonPoint (A , B)) = T)
Row 2 : ((SameOrDivergingTracks (A , B)) = T)
Row 3 : (((AllOf [A;B]) (\x.((IsOnRoute Routes3) x))) = T)


Stats for otherSameDirLongSep consistency checking:
Number of cases identified: 1
Total time: 0 sec
 ---------------------------------------------------
>
```

Figure 9: Output of Consistency Checker

and simplification process used for completeness checking is used for this analysis. However, here we check whether the conjunction of the meaning of the two columns is a contradiction. If the result is a contradiction, the checker indicates the two columns involved and lists the case(s) where they overlap in the same form as the output for completeness checking.

The results of analyzing the separation minima revealed that two tables are inconsistent. After consulting the official specification (i.e. not the pseudo code representation), our domain expert concluded that these are cases where the specification is ambiguous.

Figure 9 shows the result of analyzing one of the tables that is inconsistent. The table "otherSameDirLongSep" specifies the number of minutes of time that must exist between two aircraft (that are not both turbojet or both supersonic) flying in the same direction for them to be considered longitudinally separated. The checker identified that, for the case where two aircraft have reported over a common navigation point, are on the same or diverging tracks, and are both on a particular set of routes that have special criteria, the table is ambiguous as to whether there should be 15 or 20 minutes of separation between them.

The second table with inconsistencies describes requirements for lateral separation[5]. This table has eight rows and four columns. This case again involves special provisions for particular routes that overlap with the more general criteria. The results clearly reveal cases in the official specification that are ambiguous as to the amount of lateral separation required between aircraft.

5.4 Symmetry Checking

Symmetry is a desirable property of this specification. It is important that the separation criteria are the same regardless of the order of the parameters (i.e., the two flights) given to the functions and predicates that the tables describe.

[5] There were actually two other tables with inconsistencies but these two tables "LateralSeparationRequiredInMiles" and "LateralSeparationRequiredInDegrees" represent the same sets of conditions, but have different return values for the functions.

```
>%sym ssOppDirNoLongSepPeriod

ssOppDirNoLongSepPeriod is:
(
    (Table [((Row (ReportedOverCommonPoint (A , B)))
[TRUE;FALSE])]
    )
 [((ept (A , B)) , ((ept (A , B)) + 10));
  (((ept (A , B)) - 15) , ((ept (A , B)) + 15))])

The table is symmetric if the following condition(s) hold
(some conditions may overlap):

(
 ((ReportedOverCommonPoint ((FST (A , B)) , (SND (A , B)))) = T)
  =
 ((ReportedOverCommonPoint ((FST (B , A)) , (SND (B , A)))) = T)
 )

(
 ((ReportedOverCommonPoint ((FST (A , B)) , (SND (A , B)))) = F)
  =
 ((ReportedOverCommonPoint ((FST (B , A)) , (SND (B , A)))) = F)
 )

Total time: 1 sec
-----------------------------------------------------
>
```

Figure 10: Output of Symmetry Checker

To carry out symmetry checking, two versions of the table are created — one with each ordering of the parameters. The meaning of the disjunction of all columns within each table that return the same result is compared to the other table.

If these expressions are not equivalent, the symmetry checker returns constraints, that if satisfied, would mean the table is symmetrical. Figure 10 shows an example of the output of the symmetry checker applied to a simple table.

The initial results of this analysis (as seen in Figure 10) pointed out that the symmetry of a table is often dependent on the symmetry of the primitive terms used in the table. Environmental assumptions of the form,

forall A B.
  ReportedOverCommonPoint (A,B) =
      ReportedOverCommonPoint(B,A)

were added to make this analysis more accurate.

While this analysis did not reveal any errors in the specification, it did point out information about the primitive terms which might not be known by an implementor of the separation minima in software.

6. Future Work

This work is the first attempt to validate the thesis ideas of the first author. The operational semantics integrate the "style" of specification with the predicate logic environment, and are used directly in analysis to map the specification (possibly in multiple notations) into a form that can be automatically analyzed using state space exploration analysis techniques. The use of the explicit semantic definitions retains the structure of the specification for analysis so that structure can be used to help create a finite model for analysis. This was illustrated here in the use of predicate logic along with a tabular notation, and using the structure given in the rows to partition the state space. The continuation of this thesis work will look at how the techniques used in this example can be generalized for other notations and other state space exploration analysis techniques, such as model checking.

A particular issue in this work is the use of constants with semantic definitions as the keywords that capture the structure of the table, such as "Row" and "Table". This means that the notation associated with the semantics no longer has to be "lifted" from the base S notation. If the more traditional path of defining keywords like "Row" and "Table" as constructors had been chosen, any place one table references another table, the reference would need to be "wrapped" with its semantic function to refer to the meaning of the table. However, the execution of the semantic functions expands the definitions used in the specification. For the most efficient execution, evaluation must be done in place, which eliminates the original expression. This makes it difficult to return the appropriate level of abstraction in the results. In the case of the tables, the output had to be given in terms of the labels of the rows to be useful to the reviewer. We have already implemented a method of maintaining the original expression that works for the results given here. We are working on formally defining this method generally so that it is possible to retain the appropriate original expression while still taking advantage of evaluation in place.

Another important point in this work is the use of the general purpose interface language, predicate logic. State space exploration analysis techniques are geared toward model-oriented specification methods. While the form of the specification used in this example is functional, we have illustrated how data aspects of the problem can aid in the analysis. Having a general-purpose interface made it possible to integrate the functional and data aspects of the specification for analysis. Rather than taking the approach of only allowing specifications that definitely can be analyzed using finite means, we have started from general-purpose logic interface and applied particular techniques when the specification fit a particular form. This opens a door to the possibility of using formal specifications created primarily by domain experts as input to more specialized analysis techniques such as theorem proving performed by formal methods experts. It would be straightforward to convert the S representation of our tabular style of specification into input for other analysis tools such as PVS [13] and HOL [4].

Recently, a 4th edition of the document "Application of Separation Minima for the NAT Region" has been produced. This document has some significant additions including reduced vertical separation minima. It would be interesting to see if this document corrects the ambiguities found in this work through consistency checking.

7. Conclusion

This paper is perhaps most notable for the example of applying formal methods to the ICAO standard for separation between aircraft over the North Atlantic. The ease with which a domain expert was able to review and edit the specification and analysis results is a favourable data point in the struggle to make formal methods acceptable to industry. Beyond documenting this collaborative effort as an instance of the industrial use of formal methods, this work illustrates:

- the integration of the tabular style into a general-purpose predicate logic environment which allowed the specification of uninterpreted functions, and environmental assumptions;
- a framework for analysis which uses the explicit operational semantics directly, allowing different notations to be combined, and making it possible to exploit the structure of a given notation in analysis and to return results at the correct level of abstraction;
- the advantages of using a presentation format in HTML and how it is possible to integrate this with a format that can be used as input to analysis tools.

Although it rests upon a solid mathematical foundation, we believe that the approach illustrated in this paper could be fully integrated into an industrial system/software engineering process without causing the domain experts to be excluded or extensively retrained in formal methods. The automated analysis provided by Fusion streamlines the manual review process by automating some of the processing that would otherwise need to be done manually.

Acknowledgments

The authors would like to thank Richard Yates of MacDonald Dettwiler for comments on an early draft of the specification. The first author would also like to acknowledge the input of her thesis committee in focusing these ideas. Reviewers for this workshop made helpful comments. The university based component of this collaborative research is supported by funding from Hughes International Airspace Management Systems, MacDonald Dettwiler, and the BC Advanced Systems Institute.

References

[1] Joanne Marie Atlee. *Automated analysis of software requirements.* PhD thesis, University of Maryland, 1992.

[2] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-25(8):677-691, August 1986.

[3] Tom DeMarco. *Structured analysis and system specification.* Yourdon Press, Englewood Cliffs, New Jersey, 1979.

[4] M.J.C. Gordon and T.F. Melham. *Introduction to HOL.* Cambridge University Press, Cambridge, 1993.

[5]David Harel. Statecharts: A visual formalism for complex systems. *Science of Computing*, 8:231-274, 1987.

[6] Mats P.E. Heimdahl and Nancy G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363-377, June 1996.

[7] C.L. Heitmeyer and B.G. Labaw. Consistency checks for SCR-style requirements specifications. Technical Report NRL/FR/5540-93-9586, United States Naval Research Laboratory, Washington, D.C., December, 1993.

[8] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3): 231-261, July, 1996.

[9] K.L. Heninger. Specifying software requirements for complex systems: New techniques and their applications. *IEEE Transactions on Software Engineering*, 6(1):2-13, 1980.

[10] Alan J. Hu, David L. Dill, Andreas J. Drexler, and C. Han Yang. Higher-level specification and verification with BDDs. In *Computer-Aided Verification: Fourth International Workshop*, 1992.

[11] J. Joyce, N. Day, and M. Donat. S: A machine readable specification notation based on higher order logic. In *7th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, pages 285-299, Valletta, Malta, September, 1994.

[12] Nancy G. Leveson, Mats P.E. Heimdahl, Holly Hildreth, and Jon D. Reese. Requirements specification for process control-control systems. *IEEE Transactions on Software Engineering*, 20(9):684-707, September, 1994.

[13] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, pages 748-752, Saratoga, NY, 1992.

[14] Sam Owre, John Rushby, and Nataranjan Shankar. Analyzing Tabular and State-Transition Requirements Specifications in PVS. Technical Report CSL-95-12, Computer Science Laboratory, SRI International, Menlo Park, CA, April, 1996.

[15] Sam Owre, John Rushby, and Natarajan Shankar. Integration in PVS: Tables, Types, and Model Checking. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 336-383, Enschede, The Netherlands, Springer-Verlag Lecture Notes in Computer Science, Vol. 1217, April, 1997.

[16] David Lorge Parnas. Tabular representations of relations. Technical Report 260, Communications Research Laboratory, Faculty of Engineering, McMaster University, October 1992.

[17] J.M. Spivey. *Understanding Z.* Cambridge University Press, Cambridge, 1988.

Appendix A

The S notation is very similar to the syntax for the term language used in the HOL theorem prover [4]. But unlike HOL, S does not involve a meta-language as part of the specification format for declarations and definitions. Instead, the syntax for declarations and definitions is an extension of the syntax used for logical expressions. (In this respect, S more closely resembles Z and other similar formal specification notations.) For example, the symbol ":=" is used in S for a definition, e.g., "TWO := 2", in contrast to an assertion, e.g., "TWO = 2".

Another difference that will likely be noticed by readers familiar with HOL is the explicit type parameterization of constant declarations and definitions. Type parameters, if any, are given in a parenthesized list which prefixes the rest of the declaration or definition. This is illustrated in the definitions given below by the parameterization of "EveryAux" by a single type parameter, "ty".

Many of the definitions shown below are given recursively based on the recursive definition (not shown here) of the polymorphic type "list". These recursive definitions are given in a pattern matching style (similar to how recursive functions may be defined in Standard ML) with one clause for the "NIL" constructor (i.e., the non-recursive case) and another clause for the "CONS" constructor (i.e., the recursive case). Each clause in this style of S definition is separated by a "|". The functions "HD" and "TL" are standard library functions for taking the head (i.e., the first element) of a list and the tail (i.e., the rest) of a list respectively.

Type expressions of the form, ":ty1->ty2", are used in the declaration of parameters that are functions from elements of type "ty1" to elements of type "ty2". Similarly, type expressions of the form, ":(ty) list", indicate when a parameter is a list of elements of type "ty".

Lambda expressions are expressed in S notation as, "\x.E" (where E is an expression).

The semantic definitions for the tabular notation given in the S notation are shown below.

```
(:ty)
EveryAux (NIL) (p:ty->bool) := T |
EveryAux (CONS e tl) p :=
        (p e) AND EveryAux tl p;

(:ty)
Every (p:ty->bool) l := EveryAux l p;

(:ty)
ExistsAux (NIL) (p:ty->bool) := F |
ExistsAux (CONS e tl) p := (p e) OR ExistsAux tl p;

(:ty)
Exists (p:ty->bool) l := ExistsAux l p;

(:ty) UNKNOWN : ty;

(:ty)DC := \(x:ty).T;
TRUE := \x.x=T;
FALSE := \x.x=F;

(:ty1)
RowAux2 (CONS (p:ty1->bool) tl) label :=
        CONS (p label) (RowAux2 tl label) |
RowAux2 (NIL) label := NIL;

(:ty)Row label (plist:(A->bool)list) :=
        RowAux2 plist label;

Columns t :=
   if ((HD t)=NIL) then NIL
   else CONS
      (Every (HD) t)
      (Columns (Map t (TL)));

(:ty)
TableSemAux2 (NIL) (retVals:(ty)list) :=
   if (retVals=NIL) then UNKNOWN
   else (HD retVals) |
TableSemAux2 (CONS col colList) retVals :=
   if col
   then (HD retVals)
   else TableSemAux2 colList (TL retVals);

(:ty)
Table t (retVals:(ty)list) :=
   TableSemAux2 (Columns t) retVals;

PredicateTable t :=
   Exists (\x.x) (Columns t);
```

Modeling and Validating SAFER in VDM-SL

Sten Agerholm and Peter Gorm Larsen
IFAD
Forskerparken 10, DK-5230 Odense M, Denmark
Email: {sten,peter}@ifad.dk

Abstract

Formal methods can be applied with different levels of rigor. The more rigorously used, the more confidence is obtained in a formal model of a computer system. However, rigorous development using formal verification requires skilled personnel and is costly. Based on our experience of introducing formal specification to some European industrial companies, e.g. British Aerospace [7] and Aerospatiale [3], we believe that a less rigorous approach using validation by testing is a complement to formal verification, which engineers can use cost-effectively early in their formal methods careers. When they become more confident with constructing formal models, it would be natural to take the next step and introduce verification. In this paper we illustrate how testing-based validation can be applied to the SAFER example used throughout [9].

1 Introduction

Historically, NASA's involvement in formal methods has concentrated on formal verification using mechanical theorem provers [1, 2, 9]. In technology transfer projects supported by NASA such formal verification has been in focus. This is clearly understandable considering the criticality of the systems being developed in the avionics sector. However, we believe that the technology transfer process is more likely to have significant effect if the formal methods technology is incorporated in smaller "deltas" into the existing practice [5]. We only consider the transfer fully successful when the engineers normally developing the avionics systems, rather than the formal methods experts, feel that the technology is accessible to them and can be applied in a cost-effective manner.

With formal methods, engineers construct abstract models of computer systems before they start coding them. Among others, many of NASA's technology transfer projects have shown that benefits can be obtained already by formally specifying a system, but validation by testing and verification can increase confidence in models. Software engineering practice is heavily based on testing, and so this technique is well-known to engineers. We therefore advocate the use of testing to validate formal models. If animation is supported by tools, this prototyping activity also supports the presentation of models to others. A further advantage is that the formal specification can also be used as a basis for verifying interesting properties. Verification requires more skilled personnel than specification, and thus we see the technology transfer as a two step process. When the engineers feel confident in using formal specification and testing, it seems feasible to introduce verification.

In this paper we illustrate how a testing-based validation approach can be applied to the SAFER[1] example from [9] using VDM-SL [10] and the IFAD VDM-SL Toolbox [8, 4]. In [9], SAFER is specified in the PVS notation and properties are proved using the PVS theorem prover [11]. The VDM-SL notation is quite close to the PVS notation, but it is not the notational differences we find interesting. Instead, we focus on the kind of analysis that can be performed using a validation by testing approach rather than formal verification. This analysis reveals some interesting "unclarities" in [9]. To support the testing approach, we exploit the Dynamic Link facility of the Toolbox for rapid prototyping [6], linking graphical visualization models (compiled code) with our executable specification. We believe that this kind of animation support is very useful, in particular for presenting a specification to customers and non-trained staff members (e.g. management).

The remaining part of this paper starts with an overview of the SAFER system and the VDM-SL

[1]SAFER is an acronym for "Simplified Aid For EVA (Extravehicular Activity) Rescue".

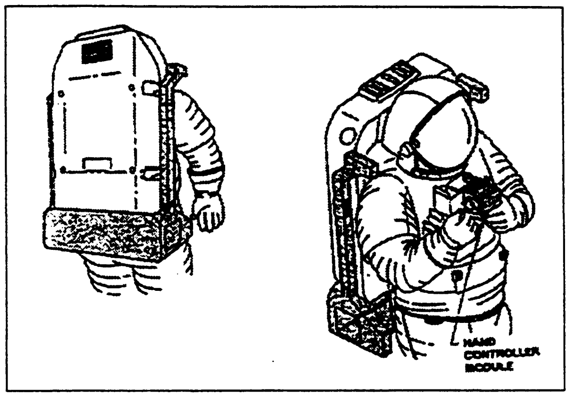


Figure 1: Front and back views of SAFER system worn by NASA crewmember.

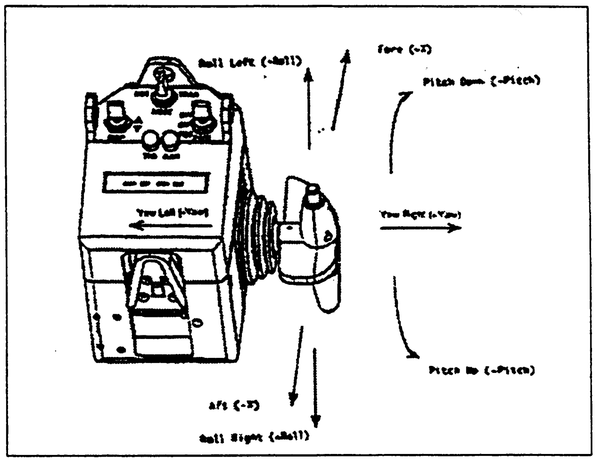


Figure 2: SAFER hand controller.

specification of SAFER. This is followed by a section in which we illustrate how a validation approach can be used to investigate different properties. Finally, a few concluding remarks are provided. The reader is assumed to have a basic knowledge of VDM-SL or a similar notation such as PVS.

2 Overview of the SAFER System

This overview of the SAFER system is based on, and partly copied from, the NASA report [9], which describes a cut-down version of a real SAFER system.

SAFER is a small, lightweight propulsive backpack system designed to provide self-rescue capabilities to a NASA space crewmember separated during an EVA (Extravehicular Activity). This type of contingency can arise if a safety tether breaks, or if it is not correctly fastened, during an EVA on a space station or on a Space Shuttle Orbiter docked to a space station. SAFER attaches to the underside of the Extravehicular Mobility Unit (EMU) primary life support subsystem backpack and is controlled by a single hand controller that is attached to the EMU display and control module (see Figure 1). SAFER provides an attitude hold capability and sufficient propellant power to automatically detumble and manually return a separated crewmember.

The SAFER system works by firing 24 gaseous-nitrogen (GN_2) thrusters, four in each of the positive and negative *X*, *Y* and *Z* directions. These are placed on the upper and lower edges of the grey part of the backpack in Figure 1 (see also Figure 13). The main focus of our specification is on the thruster selection logic, for example taking into account that translational hand controller commands are prioritized, with the priority order being *X*, *Y* and *Z*. Other aspects of the SAFER are ignored, e.g. the calculation of control output in the Automatic Attitude Hold (AAH), and various display units and switches on the hand controller which are not directly related to the selection of thrusters.

The hand controller is a four-axis mechanism with three rotary axes and one transverse axis using a certain hand controller grip (see Figure 2). A command is generated by moving the grip from the center null position to mechanical hardstops on the hand controller axes. Commands are terminated by returning the grip to the center position. The hand controller can operate in two modes, selected via a switch, either in translation mode, where *X*, *Y*, *Z* and *pitch* commands are available, or in rotation mode, where *roll*, *pitch*, *yaw* and *X* commands are available (see Figure 3). Note that *X* and *pitch* commands are available in both modes. *Pitch* commands are issued by twisting the hand grip around

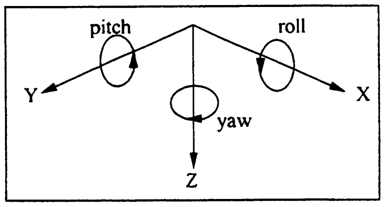


Figure 3: Six degree-of-freedom commands. Arrow direction is positive.

its transverse axis, while the other commands are obtained around the rotary axes.

A push-button switch on top of the grip initiates and terminates AAH according to a certain protocol. If the button is pushed down once the AAH is initiated, while the AAH is deactivated if the button is pushed down twice within 0.5 seconds.

As indicated above there are various priorities among commands. These make the thruster selection logic slightly complicated. Translational commands issued from the hand controller are prioritized to provide acceleration along a single translational axis, with the priority being X first, Y second, and Z third. When rotation and translation commands are present simultaneously from the hand controller, rotations take priority and translations are suppressed. Moreover, rotational commands from the hand grip takes priority over control output from the AAH, and the corresponding rotation axes of the AAH remain off until the AAH is reinitialized. However, if hand grip rotations are present at the time when the AAH is initiated, the corresponding hand controller axes are subsequently ignored, until the AAH is deactivated.

In Appendix C.2 of the NASA report, a number of requirement properties for SAFER are listed. Below we list a few relevant ones for our specification of the SAFER, and in Section 4.3 we discuss some scenarios which, among others, could be used to test that the specification satisfies the requirements.

(18) The pushbutton switch shall activate AAH when depressed a single time.

(19) The pushbutton switch shall deactivate AAH when pushed twice within 0.5 seconds.

(37) The avionics software shall disable AAH on an axis if a crewmember rotation command is issued for that axis while AAH is active.

(38) Any hand controller rotation command present at the time AAH is initiated shall subsequently be ignored until a return to the off condition is detected for that axis or until AAH is disabled.

(39) Hand controller rotation commands shall suppress any translation commands that are present, but AAH-generated rotation commands may coexist with translations.

(40) At most one translation command shall be acted upon, with the axis chosen in priority order X, Y, Z.

(41) The avionics software shall provide accelerations with a maximum of four simultaneous thruster firing commands.

These may seem straightforward and clear, but the discussion in Section 4.3 below will show that in fact they are not.

3 VDM-SL Specification of SAFER

In [9], a PVS specification is provided of a cut-down version of SAFER. We have translated this specification to VDM-SL, a relatively straightforward task since the two notations share many constructs. We have further simplified the model by cutting out parts which are irrelevant to the thruster selection logic, such as display units and interface functions to external sensors. These parts were only modeled in a very implicit way in PVS anyway, using e.g. uninterpreted functions.

We present some excerpts from the VDM-SL specification. We will make this presentation independent of the module structure of the specification, which contains five modules: *AUX* (auxiliary definitions), *HCM* (hand controller module), *AAH* (automatic attitude hold), *TS* (thruster selection) and *SAFER* (main control cycle). Modules may use definitions from other modules by prefixing with the module name, e.g. *HCM‘HandGripPosition* and *AUX‘RotCommand*.

3.1 Main Control Cycle

The SAFER system works by calculating thruster settings very frequently. As in the PVS model we assume that this happens by iteratively calling the main control operation, called *ControlCycle*. The main purpose of this operation is to calculate the thruster settings according to the hand controller commands and AAH control output, and to maintain the state of the AAH protocol. The operation is defined as shown in Figure 4[2].

VDM-SL supports both a functional subset, where a state can only be modeled indirectly in the signature of functions, and an imperative subset which supports states, operations and programming language statements directly. Here, *ControlCycle* is defined as an operation and not a function, since it depends on the AAH state (see Section 3.3)

[2]The notation mk-$R(\ldots)$ is used to construct an element of a record type R.

```
ControlCycle : HCM‘SwitchPositions × HCM‘HandGripPosition × AUX‘RotCommand $\overset{o}{\rightarrow}$
                TS‘ThrusterSet

ControlCycle (mk-HCM‘SwitchPositions (mode, aah), raw-grip, aah-cmd) ≜
  let grip-cmd = HCM‘GripCommand (raw-grip, mode),
      thrusters = TS‘SelectedThrusters (grip-cmd, aah-cmd, AAH‘ActiveAxes (), AAH‘IgnoreHcm ()) in
  (AAH‘Transition(aah, grip-cmd, clock) ;
   clock := clock + 1;
   return thrusters )
```

Figure 4: The *ControlCycle* operation.

```
SelectedThrusters : AUX‘SixDofCommand × AUX‘RotCommand × AUX‘RotAxis-set ×
                    AUX‘RotAxis-set → ThrusterSet

SelectedThrusters (hcm, aah, active-axes, ignore-hcm) ≜
  let mk-AUX‘SixDofCommand (tran, rot) = IntegratedCommands (hcm, aah, active-axes, ignore-hcm),
      mk- (bf-mandatory, bf-optional) = BFThrusters (tran (X), rot (PITCH), rot (YAW)),
      mk- (lrud-mandatory, lrud-optional) = LRUDThrusters (tran (Y), tran (Z), rot (ROLL)),
      bf-thr = if rot (ROLL) = ZERO
               then bf-optional ∪ bf-mandatory
               else bf-mandatory,
      lrud-thr = if rot (PITCH) = ZERO ∧ rot (YAW) = ZERO
                 then lrud-optional ∪ lrud-mandatory
                 else lrud-mandatory in
  bf-thr ∪ lrud-thr
```

Figure 5: The *SelectedThrusters* function.

and the SAFER state, which is simply a natural number valued clock (as in [9] we do not use a real-time clock):

```
state SAFER of
  clock : ℕ

  init s ≜ s = mk-SAFER (0)
end
```

The result of a call to *ControlCycle* is a set of thruster settings, i.e. a set of thruster names to be turned on. The arguments of *ControlCycle* are: (1) switch positions on the hand controller, telling whether the mode is translation or rotation and whether the AAH button on the grip is up or down, (2) hand grip positions, a record containing four fields corresponding to the transverse axis and the three rotation axes, and (3) external input from the AAH control laws whose calculation is based on data measured by sensors. This AAH control function in (3) is not modeled either in the PVS specification of [9], though it is mentioned in a very implicit way.

The control cycle first transforms the "raw" grip commands to translation and rotation commands. Next it calculates the thruster settings and then the AAH state is updated in the body of the let-statement. We treat each of these steps in separate subsections below. Note that values of AAH state variables are fetched by calling the operations *ActiveAxes* and *IgnoreHcm*, which both return sets of rotation axes.

3.2 Thruster Selection

The six degree-of-freedom of the translation and rotation commands is modeled using a record type:

```
SixDofCommand :: tran : TranCommand
                 rot  : RotCommand;
```

whose two fields are finite maps, i.e. a kind of tables, from translation and rotation axes respectively to axis commands. For example, the type of translation commands is defined as follows:

```
BFThrusters : AUX`AxisCommand × AUX`AxisCommand × AUX`AxisCommand →
              ThrusterSet × ThrusterSet
BFThrusters (A, B, C) △
  cases mk-(A, B, C) :
    mk-(NEG, ZERO, ZERO)  → mk-({B1, B4}, {B2, B3}),
    mk-(ZERO, ZERO, ZERO) → mk-({}, {}),
    mk-(POS, NEG, ZERO)   → mk-({F1, F2}, {}),
    ...                   → ...
  end;

LRUDThrusters : AUX`AxisCommand × AUX`AxisCommand × AUX`AxisCommand →
                ThrusterSet × ThrusterSet
LRUDThrusters (A, B, C) △
  cases mk-(A, B, C) :
    mk-(NEG, NEG, ZERO)  → mk-({}, {}),
    mk-(NEG, ZERO, ZERO) → mk-({L1R, L3R}, {L1F, L3F}),
    mk-(POS, ZERO, POS)  → mk-({R2R}, {R2F, R4F}),
    ...                  → ...
  end;
```

Figure 6: Extracts from *BFThruster* and *LRUDThrusters*.

$TranCommand = TranAxis \xrightarrow{m} AxisCommand$

inv $cmd \triangleq$ dom $cmd = \{X, Y, Z\}$;

where the invariant ensures that command maps are total. The type of rotation commands is defined similarly. Enumerated types are used for axis commands and translation and rotation axes:

$AxisCommand = \text{NEG} \mid \text{ZERO} \mid \text{POS}$;

$TranAxis = X \mid Y \mid Z$;

$RotAxis = \text{ROLL} \mid \text{PITCH} \mid \text{YAW}$;

In the *SelectedThrusters* function in Figure 5 grip commands from the hand controller (with six-degree-of freedom) are integrated with the AAH control output. Then thrusters for back and forward accelerations and left, right, up and down accelerations are calculated by two separate functions. These represent a kind of look-up tables, modeled using cases expressions. Note that they return two sets of thruster names, representing mandatory and optional settings respectively. Cut-down versions of the functions are presented to save space, see Figure 6. Note that thrusters are named according to which direction they provide acceleration for, and the number indicates which of four quadrants they are positioned in. The third letter indicates whether the thruster is positioned to the rear or front in a quadrant (see Figure 13 below).

The first case in *LRUDThrusters* yields two empty sets since it is a "Not Applicable" case.

The more interesting parts of the specification can be found in the *IntegratedCommands* function and the auxiliary functions it uses. Together these define the selection logic, modeling for instance the various priorities among translation and rotation axes. Below, null translation and rotation commands map all axes to ZERO.

The *IntegratedCommands* function is presented in Figure 7. The function treats two cases, depending on whether or not the set of active axes in the AAH is empty (tested in *AllAxesOff*). If there are rotation commands from the hand controller then these take priority over translation commands using the *PrioritizedTranCmd* function (see Figure 7). Otherwise translation commands are prioritized in the order *X*, *Y*, *Z*.

If there are rotation commands from both the AAH and the hand controller then these must be combined such that the hand controller commands take priority, unless they were also issued when the AAH was initiated (recorded in *ignore-hcm*). This is done by the *CombinedRotCmds* function (see Figure 7). Note that the $\uplus$ operator used in the *CombinedRotCmds* function is simply a union between maps. *AUX`rot-axis-set* is defined as the set of all rotation axes. This completes the description of the thruster selection logic.

```
IntegratedCommands : AUX`SixDofCommand × AUX`RotCommand × AUX`RotAxis-set ×
                     AUX`RotAxis-set → AUX`SixDofCommand

IntegratedCommands (mk-AUX`SixDofCommand (tran, rot), aah, active-axes, ignore-hcm) $\triangleq$
  if AAH`AllAxesOff (active-axes)
  then if RotCmdsPresent (rot)
       then mk-AUX`SixDofCommand (AUX`null-tran-command, rot)
       else mk-AUX`SixDofCommand (PrioritizedTranCmd (tran), AUX`null-rot-command)
  else if RotCmdsPresent (rot)
       then mk-AUX`SixDofCommand (AUX`null-tran-command,
                                  CombinedRotCmds (rot, aah, ignore-hcm))
       else mk-AUX`SixDofCommand (PrioritizedTranCmd (tran), aah);


PrioritizedTranCmd : AUX`TranCommand → AUX`TranCommand

PrioritizedTranCmd (tran) $\triangleq$
  if tran (X) $\neq$ ZERO
  then AUX`null-tran-command † {X $\mapsto$ tran (X)}
  elseif tran (Y) $\neq$ ZERO
  then AUX`null-tran-command † {Y $\mapsto$ tran (Y)}
  elseif tran (Z) $\neq$ ZERO
  then AUX`null-tran-command † {Z $\mapsto$ tran (Z)}
  else AUX`null-tran-command;


CombinedRotCmds : AUX`RotCommand × AUX`RotCommand × AUX`RotAxis-set → AUX`RotCommand

CombinedRotCmds (hcm-rot, aah, ignore-hcm) $\triangleq$
  let aah-axes = ignore-hcm $\cup$ {a | a $\in$ AUX`rot-axis-set · hcm-rot (a) = ZERO} in
  {a $\mapsto$ aah (a) | a $\in$ aah-axes} $\uplus$ {a $\mapsto$ hcm-rot (a) | a $\in$ AUX`rot-axis-set \ aah-axes};
```

Figure 7: The *IntegratedCommands* function and its auxiliary functions.

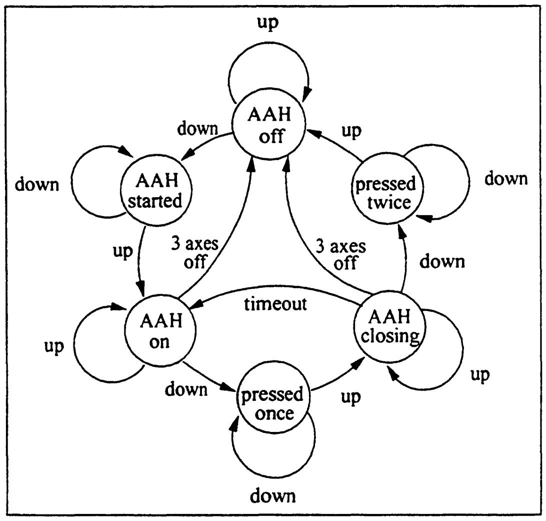


Figure 8: The AAH protocol state machine.

```
Transition : HCM`ControlButton × AUX`SixDofCommand × ℕ ⥲ ()
Transition (button-pos, hcm-cmd, clock) ≜
  let engage = ButtonTransition (toggle, button-pos, active-axes, clock, timeout),
      starting = (toggle = AAH_OFF) ∧ (engage = AAH_STARTED) in
  (active-axes := {a | a ∈ AUX`rot-axis-set · starting ∨
                                              (engage ≠ AAH_OFF ∧ a ∈ active-axes ∧
                                               (hcm-cmd.rot (a) = ZERO ∨ a ∈ ignore-hcm))};
   ignore-hcm := {a | a ∈ AUX`rot-axis-set · (starting ∧ hcm-cmd.rot (a) ≠ ZERO) ∨
                                              (¬ starting ∧ a ∈ ignore-hcm)};
   timeout := if toggle = AAH_ON ∧ engage = PRESSED_ONCE
              then clock + click-timeout
              else timeout;
   toggle := engage);
```

Figure 9: The *Transition* operation.

3.3 AAH Transitions

Finally, we present excerpts from the AAH module. The AAH push-button protocol can be viewed as a state machine, presented in Figure 8. In VDM-SL, the state of the AAH is represented by:

```
state AAH of
  active-axes : AUX`RotAxis-set
  ignore-hcm  : AUX`RotAxis-set
  toggle      : EngageState
  timeout     : ℕ
  init s ≜ s = mk-AAH ({}, {}, AAH_OFF, 0)
end
```

where the actual state machine states are presented as an enumerated type:

```
EngageState = AAH_OFF | AAH_STARTED |
              AAH_ON | PRESSED_ONCE |
              AAH_CLOSING | PRESSED_TWICE
```

The main operation in the AAH module is *Transition* which is defined in Figure 9. It uses the function *ButtonTransition*, a direct encoding of Figure 8, to calculate the next state. Note that *Transition* uses a *starting* predicate for the situation when the AAH is initiating. The starting predicate is used to calculate the active axes and axes for which the hand controller must be ignored.

4 Validation Using Testing

The NASA guidebook [9] focuses mainly on formal verification as a method for analysis of requirements and high-level design. However, we find that alternative approaches to validating specifications based on testing techniques are also beneficial, though they should not replace verification. From a technology transfer viewpoint, testing is cheaper to introduce, but in most situations it is also cheaper to apply than verification and would therefore, in particular, be suitable for the early stages of analysis like model construction.

In this section we illustrate the use of testing techniques to validate the VDM-SL model of SAFER presented above. Our specification is written in the executable subset of VDM-SL, which is supported by the interpreter of the IFAD VDM-SL Toolbox. We shall illustrate the use of a test coverage facility of the Toolbox, and we shall try to use testing techniques and the Dynamic Link facility to investigate the properties listed in Section 2.

4.1 Test Coverage of Specifications

In addition to basic facilities for checking specifications, such as syntax and type checking, the IFAD Toolbox supports a large executable subset of VDM-SL. This means that the Toolbox supports validation techniques such as testing which are usually not encouraged by theorem provers. We shall not turn this into an exercise in test case selection, but limit our attention to just a few test cases for illustration.

In testing, the focus is often on more concrete properties than in verification; given some input we test whether some function computes the desired result. This kind of testing, which to a large extent also could be performed by theorem provers, e.g. if rewriting is available, is not considered nor mentioned in [9]. However, it does make sense to test, for instance, that thrusters are fired correctly according to hand grip commands. The actual selection is based on some fairly large tables which have

$$\begin{array}{l}
\mathit{SAFER}`\mathit{ControlCycle}\,(\mathsf{mk}\text{-}\mathit{HCM}`\mathit{SwitchPositions}\,(\textsc{Tran},\textsc{Up}),\\
\qquad\qquad \mathsf{mk}\text{-}\mathit{HCM}`\mathit{HandGripPosition}\,(\textsc{Zero},\textsc{Pos},\textsc{Zero},\textsc{Zero}),\mathit{AUX}`\mathit{null\text{-}rot\text{-}command})
\end{array}$$

$$\begin{array}{l}
\{\mathsf{mk}\text{-}(\mathit{mode},a,b,c,d) \mapsto \mathit{SAFER}`\mathit{ControlCycle}\,(\mathsf{mk}\text{-}\mathit{HCM}`\mathit{SwitchPositions}\,(\mathit{mode},\textsc{Up}),\\
\qquad\qquad\qquad\qquad \mathsf{mk}\text{-}\mathit{HCM}`\mathit{HandGripPosition}\,(a,b,c,d),\mathit{AUX}`\mathit{null\text{-}rot\text{-}command})\\
\mid \mathit{mode} \in \{\textsc{Tran},\textsc{Rot}\},\ a,b,c,d \in \{\textsc{Neg},\textsc{Pos},\textsc{Zero}\}\}
\end{array}$$

Figure 10: Compact test expressions.

$$\begin{array}{l}
\mathit{LRUDThrusters} : \mathit{AUX}`\mathit{AxisCommand} \times \mathit{AUX}`\mathit{AxisCommand} \times \mathit{AUX}`\mathit{AxisCommand} \to\\
\qquad\qquad \mathit{ThrusterSet} \times \mathit{ThrusterSet}\\
\mathit{LRUDThrusters}\,(A,B,C) \triangleq\\
\quad \mathsf{cases}\ \mathsf{mk}\text{-}(A,B,C):\\
\qquad \mathsf{mk}\text{-}(\textsc{Neg},\textsc{Neg},\boxed{\textsc{Zero}}) \to \boxed{\mathsf{mk}\text{-}(\{\},\{\})},\\
\qquad \mathsf{mk}\text{-}(\textsc{Neg},\textsc{Zero},\textsc{Zero}) \to \mathsf{mk}\text{-}(\{\mathrm{L1R},\mathrm{L3R}\},\{\mathrm{L1F},\mathrm{L3F}\}),\\
\qquad \mathsf{mk}\text{-}(\textsc{Pos},\textsc{Zero},\textsc{Pos}) \to \mathsf{mk}\text{-}(\{\mathrm{R2R}\},\{\mathrm{R2F},\mathrm{R4F}\}),\\
\qquad \ldots \qquad\qquad\qquad\quad \to \ldots\\
\quad \mathsf{end};
\end{array}$$

Figure 11: Extracts from *LRUDThrusters* with test coverage information.

been translated by hand to the specification, and of course such translation could have introduced errors.

Test expressions can be typed in manually for the interpreter. For example, in the first expression in Figure 10 a call of *ControlCycle* with a forward acceleration command is issued. The interpreter will immediately evaluate this to the correct result, which is the set {F1,F2,F3,F4}.

If one wishes to test all hand grip commands without typing all combinations in by hand, then a compact test expression could be written in VDM-SL using a map comprehension expression as shown as the second expression in Figure 10. Assuming that the AAH is switched off, this will cover all hand grip translational and rotational commands, in total $2 * 3^4 = 162$ cases. This is executed in seconds. The result is a large map from variable values to results (thruster settings). Interestingly, this test only yields 17 different settings and does not cover the functions *BFThrusters* and *LRUDThrusters* very well; recall these convert six degree-of-free commands to thruster names. A test coverage facility is provided with the Toolbox, which computes basic statistical information and colors uncovered parts of specifications as illustrated in Figure 11. The two thruster functions are both called 162 times, but they are only covered 41% and 46% respectively.

The reason for this bad coverage is due to priorities for hand controller translational axes and for rotation and translation commands from the hand controller. A larger test where the AAH pushbutton and rotation command output are also variable (8748 cases, executed in 7 minutes) yields 189 different thruster settings and covers the *BFThrusters* function 100%. However, the *LRUDThrusters* function is still only 72% covered. But the uncovered parts are precisely those that have the "Not Applicable" label in the requirement specification from [9], so this is good. The test coverage coloring shows this very clearly in the boxes in Figure 11 (we have cut the specification down to the same cases as in Section 3.2). The sub-expressions written in grey have never been reached in the interpretation of the test argument(s). Note that in principle it would be possible to manually inspect the 8748 test results to check that the thruster settings are correct, but this may not be feasible in all situations. Also note that the tests mentioned above only test the AAH state machine protocol in an arbitrary way, and therefore the AAH transition function is not fully covered by this kind of testing.

4.2 Using Dynamic Link Modules

Such tests as described above could be difficult to understand for staff members who have not been trained in the notation used for the formal specification. The value of the model would also increase

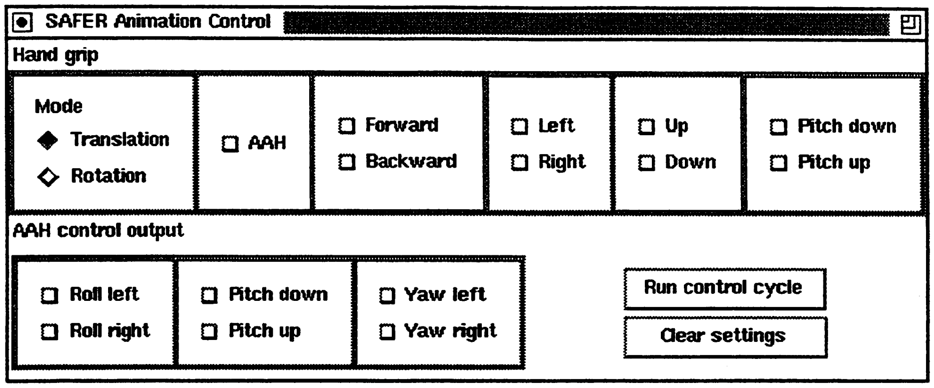


Figure 12: The interface model of the hand controller.

in the discussion with someone outside the development team, such as a customer, if some kind of prototype could be produced where the interface to the formal model is easier to understand.

In order to ease this kind of testing we can exploit the Dynamic Link facility of the IFAD Toolbox for combining compiled code and VDM-SL specifications [6]. We have made a simple interface model of the hand controller using Tcl/Tk and linked this with the SAFER specification. Figure 12 shows a screen dump of the interface, where the mode (translational or rotational) and the AAH switch together form the first parameter to *ControlCycle*. The hand grip positions form the second parameter, and the AAH control outputs form the third parameter simulating the external input from the AAH control laws whose calculation would be performed from sensor data of movements of the astronaut.

Moreover, a simplified model of the SAFER backpack in a 3D Graphics tool called Geomview[3] is also linked to the specification and used to visualize thruster settings (see Figure 13). This simple figure could naturally be enhanced significantly, but our purpose here is simply to illustrate the feasibility of this approach. For example, if we move the hand grip forward, we expect to see the forward thrusters fire in the backpack model. Hence, we can very easily make basic tests of the SAFER thruster selection logic. This animation approach is useful for testing many requirement properties of SAFER (see Section 4.3).

If effort would be put into calculating how the astronaut would be moving depending upon the thrusters fired then naturally a more advanced Dynamic Link module could be made to show the movement in 3D. We have a naive solution to this based on a simple model of astronaut movements in space.

[3]Geomview is available at `http://www.geom.umn.edu/docs/software/viz/geomview/geomview.html`.

4.3 Validating Properties

We shall now discuss a testing approach to validating, i.e. checking but not verifying, that some of the requirement properties listed in Section 2 hold in our VDM-SL model; we do not have room in this paper to consider all of them. The testing raised interesting questions regarding the requirements and found various omissions/problems in our specification as well as in the PVS specification.

4.3.1 Automatic Checking of Property (41)

Recall property (41):

> The avionics software shall provide accelerations with a maximum of four simultaneous thruster firing commands.

Consider also:

E1 Thruster firing consistency: No two selected thrusters should oppose each other, i.e., have canceling thrust with respect to the center of mass. (Mentioned in Section C.4.1 of the NASA report.)

These are properties that should hold for all thruster settings output from the SAFER *ControlCycle* and therefore they can conveniently be stated as a post-condition on *ControlCycle*, see Figure 14.

The Toolbox interpreter can be requested to automatically check that the post-condition holds for

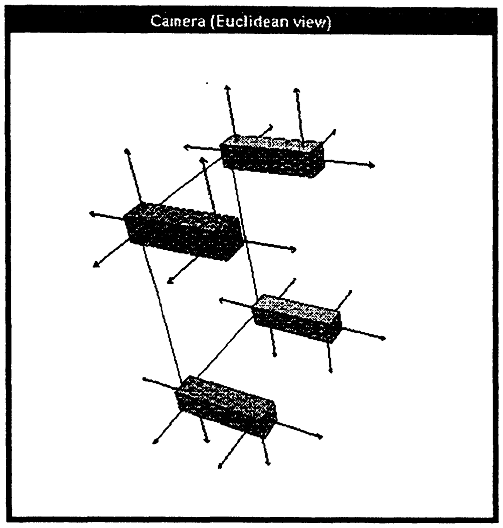


Figure 13: The interface model of the SAFER backpack.

```
$ControlCycle : HCM`SwitchPositions \times HCM`HandGripPosition \times AUX`RotCommand \xrightarrow{o} TS`ThrusterSet$

$ControlCycle\ (\text{mk-}HCM`SwitchPositions\ (mode, aah), raw\text{-}grip, aah\text{-}cmd) \triangleq$
  let $grip\text{-}cmd = HCM`GripCommand\ (raw\text{-}grip, mode)$,
      $thrusters = TS`SelectedThrusters\ (grip\text{-}cmd, aah\text{-}cmd, AAH`ActiveAxes\ (), AAH`IgnoreHcm\ ())$ in
  $(AAH`Transition(aah, grip\text{-}cmd, clock)$ ;
   $clock := clock + 1$;
   return $thrusters$ )
post card $RESULT \leq 4 \wedge$
     $ThrusterConsistency\ (RESULT)$ ;


$ThrusterConsistency : TS`ThrusterName\text{-set} \rightarrow \mathbb{B}$

$ThrusterConsistency\ (thrusters) \triangleq$
  $\neg (\{B1, F1\} \subseteq thrusters) \wedge$
  $\neg (\{B2, F2\} \subseteq thrusters) \wedge$
  $\neg (\{B3, F3\} \subseteq thrusters) \wedge$
  $\neg (\{B4, F4\} \subseteq thrusters) \wedge$
  $\neg (thrusters \cap \{L1R, L1F\} \neq \{\} \wedge thrusters \cap \{R2R, R2F\} \neq \{\}) \wedge$
  $\neg (thrusters \cap \{L3R, L3F\} \neq \{\} \wedge thrusters \cap \{R4R, R4F\} \neq \{\}) \wedge$
  $\neg (thrusters \cap \{D1R, D1F\} \neq \{\} \wedge thrusters \cap \{U3R, U3F\} \neq \{\}) \wedge$
  $\neg (thrusters \cap \{D2R, D2F\} \neq \{\} \wedge thrusters \cap \{U4R, U4F\} \neq \{\})$
```

Figure 14: The *ControlCycle* operation and the *ThrusterConsistency* function.

all results from *ControlCycle* (this feature is optional). Hence, if a result failed the check, the interpreter would give a run-time error with position information (line and column) in the specification document. This never happened in any of our tests (including the big test mentioned above).

In [9] a proof of the maximum thruster property (41) is given. The validation made here is not as strong as that, because we would have to test it with all values in a model-checking fashion. However, the validation carried out here has definitely increased our confidence in the maximum thruster property being satisfied with our model.

Point: In cases where a property does not hold, the validation technique can be an efficient way of discovering counter-examples. It can also increase confidence in a model.

4.3.2 Property (18)

> The pushbutton switch shall activate AAH when depressed a single time.

This property is related to the protocol of the AAH given in Figure 8, and thus it may be a good idea to revisit this figure to be able to follow the possible test scenarios. Assume the hand grip is in center position throughout this test; this means that no translation or rotation commands are issued from the hand grip. Initially, the AAH is in the "off state". Thus, initially any AAH control rotation output must be ignored. Using various arbitrary examples we can test that the thruster settings are empty. The AAH is activated by pushing the AAH button down. The AAH goes to the "started state", but it does not go to the "on state" until the button is released. We can run the same and other AAH rotation commands to see that they are now taken into account (and are performed correctly). Note that property (38) would affect this test if a rotation command from the hand grip was issued at the same time as the AAH was initiated.

Point: In this case the test has increased our confidence that property (18) is satisfied, such that it is sufficient to push the AAH button without releasing it again.

4.3.3 Property (19)

> The pushbutton switch shall deactivate AAH when pushed twice within 0.5 seconds.

This property is also related to the AAH protocol so Figure 8 must be used again. The timeout of 0.5 seconds is for test purposes represented as 10 control cycles in our specification. As in [9] we do not represent a real-time clock. We first turn the AAH on by pressing and then releasing the AAH button. We then press and release the AAH button twice within 10 cycles and check that the AAH is turned off by running a number of tests with rotation commands from the AAH. Turn AAH on again, press and release the AAH button once, wait at least 11 cycles, press and release again and check that the AAH still works. Finally, press, release and press the AAH button within 10 cycles. The report [9] is unclear here: Strictly speaking, property (19) says that this should deactivate the AAH while the PVS specification and the AAH state diagram does not deactivate the AAH until the button is released.

Point: Since the word "pushed" (rather than "pushed and released") is used in property (19) our validation have discovered a discrepancy between requirement (19) and both the PVS model and the state transition diagram from Figure 8.

4.3.4 Property (37)

> The avionics software shall disable AAH on an axis if a crewmember rotation command is issued for that axis while AAH is active.

This property is related to the combination of the AAH protocol and the thruster selection logic for rotation commands. We can test this by first inspecting how SAFER behaves when the AAH is switched on and the AAH control output wishes to rotate over an axis. Afterwards we can inspect what happens if the hand grip controller issues a rotation command on the same axis. In this case property (37) requires SAFER to disable the AAH for that axis.

One possible first test scenario for this property could be: Assume the hand grip is in center position (all directions are ZERO). Turn the AAH on (button down and then up) and issue an AAH rotation (control output) on an axis, e.g. roll left. Keep this rotation while making a rotation command on the hand grip, e.g. roll right (in rotation mode). Check the output to see that this takes precedence over AAH. Keep the AAH rotation but center the hand grip. The AAH axis should remain off, as stated explicitly in Section C.4.1 of the NASA report.

However, when executing the test scenario on our specification, this appears not to be the case. The test shows that the AAH axis still has influence on the selected thrusters, in fact, the result is exactly the same as before the axis was deactivated. The problem is that our model (like the PVS model) implicitly assumes that the AAH control output does not provide a rotation command for an axis which is not active. In our manually generated AAH output, this may however be the case. We could capture this problem using a pre-condition on the SAFER *ControlCycle* operation. In the PVS specification, an obvious place to take this into account would be in a post-condition of the `AAH_control_out` function, which could be done using the subtype facility in PVS, but in [9] this function is implicitly defined and does not give any information whatsoever about implementation requirements. When translating the PVS specification to VDM-SL, we cut out the very implicit `AAH_control_out` and modeled this as the third argument of *ControlCycle* instead. A post-condition on the PVS `AAH_control_out` would therefore have been translated to a pre-condition on *ControlCycle* in our model.

As a consequence of the implicitness concerning the AAH control output, it is not possible to verify property (37) in a black-box fashion in PVS, as discussed above. However, a white-box approach of showing that the "active" flag is properly reset and stays off is possible. But in the NASA report it is not documented anywhere that the `AAH_control_out` should take this flag into account, which then seems to be an omission, at least. As mentioned above, this requirement would be easy to specify in a post-condition, which would enable black-box verification. The verification of other properties is also weakened by this omission. For example:

> Once AAH is turned off for a rotational axis it remains off until a new AAH cycle is initiated.

which appears in Section C.4.1 of the NASA report.

Point: In validating this property we discovered that neither our model nor the PVS model in [9] is sufficiently strong to prove certain desirable properties.

4.3.5 Property (38)

> Any hand controller rotation command present at the time AAH is initiated shall subsequently be ignored until a return to the off condition is detected for that axis or until AAH is disabled.

A question raised immediately here when trying to design a test scenario was: what do they mean by "a return to the off condition"? If one carefully reads the PVS specification, and in particular looks at the way that *ignore-hcm* is updated in *Transition*, it seems clear that the only way for a hand controller rotation command to become reconsidered is by manually disabling the AAH.

Point: This is an example where seeking to come up with a test case for a certain situation discovered an ambiguity in the meaning of this requirement.

5 Concluding Remarks

In this paper we have illustrated the analysis of a formal specification using a testing-based approach to validation. We believe that the ratio between the insight gained by this kind of analysis and the effort spent on the analysis is promising, in particular in a technology transfer context, because less skilled engineers are required for this kind of validation than if formal verification was applied. Moreover, many errors can be found (relatively) cheaply using testing which otherwise might require a great deal of effort to single out using formal verification.

The kind of tool support demonstrated in this paper, including the combined use of a specification executor and Dynamic Link modules, can in our opinion help in making the formal methods technology accessible to more engineers. Specifically with respect to the SAFER example, we believe that the prototyping and animation facilities provide an easy way for a specifier to demonstrate the consequences of the thruster selection logic to someone unfamiliar with formal notations. Moreover, in the SAFER example there is very little trade-off in going from the PVS model to an executable VDM-SL model, because the PVS model is already relatively concrete and essentially executable. Generally speaking, models for verification might be more abstract than executable models to ease verification, and so there could be a trade-off.

We see validation using simulation and formal verification as complementary techniques which can be fruitfully applied in the same project. Given that the system is sufficiently critical to justify the costs, we imagine that the most productive approach would be to use the validation technology first, and to continue with a formal verification of the properties which cannot simply be tested, after the worst bugs in the model have been removed. These different kinds of analyzes tend to discover different kinds of problems. Hence, though some verification of the PVS specification of SAFER had already been carried out in [9], we were still able to detect a few points that need further clarification using the validation approach (see Section 4.3).

Throughout this paper, we have assumed that verification is costly. However, for certain restricted domains automatic verification using model checking has proved to be feasible, and hence potentially very cheap to apply. One might view the use of such automatic verification tools as a compromise towards fully verified models. Typically, the price for this compromise is a more restrictive notation increasing the effort required to produce models.

In fact, we have continued the experiment above together with Arne Borälv from Logikkonsult, in order to investigate the potential power of automatic verification. Arne Borälv has taken our VDM description and manually translated it to a model in NP-Tools [12], using only propositional logic extended with integer arithmetic and enumerated types. Using this model it was possible to automatically prove the maximum thruster and the thruster consistency requirements, as well as some other properties. Proof execution times were within the order of seconds (45 seconds for the hardest requirement, maximum thruster). For rather finite systems like the SAFER example we envisage that it would be possible to automatically translate VDM models to models in NP-Tools and in this way be able to automatically verify properties fast. This approach seems appealing and will be further investigated.

Acknowledgments

We would like to thank Erik Toubro Nielsen, Ole Storm Pedersen and Anders Søndergaard for developing the different parts of the Dynamic Link modules. We are also grateful for the constructive comments we got on earlier versions of this paper from Hanne Carlsen, Benny Graff Mortensen, Paul Mukherjee and Anne Berit Nielsen. Finally PGL would like to thank John Kelly for asking for review comments on [9].

References

[1] Ricky W. Butler, James L. Caldwell, Victor A. Carreno, C. Michael Holloway, Paul S. Miner, and Ben L. Di Vito. Nasa langley's research and technology transfer program in formal methods. In *Tenth Annual Conference on Computer Assurance (COMPASS 95)*. Gaithersburg, MD, June 1995. (expanded version available from http://atb-www.larc.nasa.gov/fm.html).

[2] James L. Caldwell. Formal methods technology-transfer: a view from nasa. In S. Gnesi and D. Latella, editors, *Proceedings of the ERCIM Workshop on Formal Methods for Industrial Critical Systems*, Oxford England, March 1996.

[3] Lionel Devauchelle, Peter Gorm Larsen, and Henrik Voss. PICGAL: Lessons Learnt from a Practical Use of Formal Specification to Develop a High Reliability Software. In *DASIA'97*. ESA, May 1997.

[4] René Elmstrøm, Peter Gorm Larsen, and Poul Bøgh Lassen. The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications. *ACM Sigplan Notices*, 29(9):77–80, September 1994.

[5] J.S. Fitzgerald and P.G. Larsen. Formal specification techniques in the commercial development process. In M. Wirsing, editor, *Position Papers from the Workshop on Formal Methods Application in Software Engineering Practice, International Conference on Software Engineering (ICSE-17), Seattle*, April 1995. ftp://ftp.ifad.dk/pub/papers/icse.ps.gz.

[6] Brigitte Fröhlich and Peter Gorm Larsen. Combining VDM-SL Specifications with C++ Code. In Marie-Claude Gaudel and Jim Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 179–194. Springer-Verlag, March 1996.

[7] Peter Gorm Larsen, John Fitzgerald, and Tom Brookes. Applying Formal Specification in Industry. *IEEE Software*, 13(3):48–56, May 1996.

[8] Paul Mukherjee. Computer-aided Validation of Formal Specifications. *Software Engineering Journal*, pages 133–140, July 1995.

[9] NASA. Formal methods, specification and verification guidebook for software and computer systems - a practitioner's companion. Technical Report Draft 2.0, Washington, DC 20546, USA, November 1996.

[10] P. G. Larsen and B. S. Hansen and H. Brunn N. Plat and H. Toetenel and D. J. Andrews and J. Dawes and G. Parkin and others. Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language — Part 1: Base language, December 1996.

[11] PVS World Wide Web page. `http://www.csl.sri.com/pvs/overview.html`.

[12] Gunnar Stålmarck. A System for Determining Propositional Logic Theorems by Applying Values and Rules to Triplets that are Generated from a Formula, 1989. Swedish Patent No. 467 076 (approved 1992), U.S. Patent No. 5 276 897 (1994), European Patent No. 0403 454 (1995).

Requirements Analysis of Real-Time Control Systems using PVS

Bruno Dutertre Victoria Stavridou

bruno@dcs.qmw.ac.uk, victoria@dcs.qmw.ac.uk
Department of Computer Science,
Queen Mary and Westfield College,
University of London

Abstract

This paper presents a practical application of the PVS theorem prover involving requirements analysis of real-time control systems. This work was conducted within the SafeFM project and relied on a real world avionics case study. We show how PVS was used to formalize the software requirements for the system and to verify safety-related properties. We also present the main result of the experiment. We give an overview of PVS libraries which were developed after the case study experiment and are intended to facilitate the specification and verification of similar systems.

1 Introduction

The SafeFM project[1] investigated the practical application of formal methods to the development and assessment of high integrity systems [17]. Within the project, we investigated the use of formal methods and theorem proving in requirement analysis of real-time control systems. A major part of this work involved an experiment applying the PVS theorem prover to the analysis of an avionics control system. The case study was a substantially complex example based on an existing system developed by GEC-Marconi, one of our SafeFM partners.

This paper presents the main results of SafeFM in the domain of formal requirement specification and analysis. Section 2 gives an overview of the verification method. Section 3 presents the formal notations we used and shows how PVS can provide mechanical support to formal requirements analysis. Section 4 is an overview of the case study experiment; it describes the successive phases of the formalization and verification using PVS, and presents the main experimental results. Section 5 describes further work dealing with one of the limitations of PVS identified during the case study: the lack of general purpose libraries.

2 Methodology

2.1 Applications Considered

SafeFM focused on a specific class of high-integrity systems, namely digital controllers. In particular, we were interested in medium size applications encountered in avionics such as air-data computers or store managers. These applications have several important characteristics:

- They are usually fault tolerant systems with a redundant architecture. Because of the need for high reliability, avionics controllers incorporate multiple processors so that hardware failures can be tolerated.
- They are real-time applications. Digital controllers interact with an active environment which imposes timing constraints. For example, input signals have to be sampled and processed at a sufficient rate for the system to maintain an accurate image of its environment. Commands have to be produced at the right time and may have to accommodate various externally defined mechanical constraints.
- They are hybrid systems. They may receive input both from discrete or from analogous sources and have to implement complex control laws which mix logical and numerical computations.

All these characteristics are related to a major element of all control applications: the system under control. Real time digital controllers cannot be understood and analyzed without taking into consid-

[1]This work was partially funded under the UK Department of Trade and Industry SafeIT programme by EPSRC Grant No GR/H11471 under DTI Project No IED/1/9013.

eration the controlled system, its behavior, and its properties.

2.2 Method of Analysis

Our primary objective within SafeFM was to design a methodology for the formal analysis of software requirements for real-time digital controllers. Our investigation was guided by the SafeFM case study and we started from the following point of view:

- The control system consists of various processing units and the general architecture of the system (processors, communication links, interface, etc.) is given.
- The requirements describe functional modules to be implemented by each of the processors. Typically, each module defines a control function or some other task such as failure detection or monitoring. The requirements may include real-time and temporal constraints as well as purely functional aspects.

In order to get confidence in the validity of requirements, we want first to check that the description of each module is internally consistent. This assumes that the requirements can be specified formally and that various consistency checks can be performed on the formal specification. Verification might include type checking, the detection of out-of-range values, or the proof of general semantic properties. The preservation of invariants in state-based formalisms such as B[1] or VDM[10] is a typical example of such semantic verification. Although it is not exactly a consistency property, checking total coverage of the input domain is another important example of semantic validation [9, 15].

The class of applications we consider are often safety critical or safety related. In this context, checking only the internal consistency of functional requirements is not sufficient. We also need to be able to verify that critical properties are satisfied. Such properties are global constraints on the behavior of the system under control. Verifying that a system satisfies these high-level properties requires more than knowing the functional requirements of each module. Additional information, such as the architecture of the controller and a model of the system under control, is necessary.

In summary, we assume that specifications for real-time control applications can be structured into three broad classes: the functional requirements, a list of assumptions about the controlled system, and the critical properties to be verified. Coherence can be demonstrated by applying various consistency checks to individual functional modules and by proving that the critical properties are satisfied by the functional requirements.

3 Requirements Analysis with PVS

In order to perform the different kinds of verification mentioned above, some form of mechanical support is necessary. The tool must provide a rich formal notation able to cover the various classes of requirements of real-time control systems. It must also support reasoning about numerical as well as logical properties.

PVS was the tool chosen for SafeFM. PVS offers both a very expressive specification language based on higher order logic and a powerful theorem prover. A description of the system and examples of applications of PVS can be found at the PVS world-wide web site[2] and elsewhere [4, 13, 14, 16].

PVS is a general system and we had to define a specification approach for real-time systems requirements. We used a straightforward and easy to implement approach with explicit time. Time is modeled by the non-negative real numbers, time varying quantities are manipulated explicitly as functions of time, and temporal properties are written using explicit time indices.

There are two kinds of time dependent variables. The first kind is used to specify the requirements of discrete components; functions are defined on a subset of the time domain which represents a clock. The second is total function of time and models the continuous variables of an application.

3.1 Functional Requirements

In order to specify functional requirements, we used a data flow approach inspired from the synchronous languages LUSTRE and SIGNAL [8, 11]. Every module is assumed to be activated at regular intervals by a clock of known frequency. In PVS, such clocks are defined as follows:

```
clocks[K:posreal]: THEORY
  BEGIN

  IMPORTING time
```

[2]http://www.csl.sri.com/pvs.html

```
t: VAR time
n: VAR nat

clock: TYPE =
   {t | EXISTS n : t = n * K}.
```

A clock is a parameterized subtype of `time`, characterized by a positive real `K` – the period of the clock. An element `x` is of type `clock[K]` if it is a multiple of the period.

The specification for each functional module includes a clock and a list of input, output and internal variables. All these are time dependent and are represented as functions of the module's clock. For example, an output `WSCMD` of a module of clock `H` is of type

```
WSCMD : [H -> ws_range].
```

The elements of `H` represent the instant when the module is activated, the input signals processed, and the output produced. The clock forms an increasing sequence of instant

$$t_0 < t_1 < t_2 < \ldots < t_i < \ldots$$

and `WSCMD`(t_i) is the command computed at the i-th activation.

With this approach, requirements can be specified as a list of function definitions. For example, we could have the following definition for `WSCMD`:

```
t: VAR H

WSCMD(t): ws_range = max(X(t), Y(t)),
```

where `X` and `Y` are other variables of the same module. Since we want the specifications to be implementable in software, certain restrictions are imposed on the form of the definitions. Roughly, we require that the value of an internal or output variable `X` at time t_i only depends on values of other variables at t_i or at t_{i-1} and possibly on $X(t_{i-1})$. Intuitively, in case where $X(t_i)$ depends on $X(t_{i-1})$, the variable X is kept in memory for use in the subsequent step; the corresponding PVS definition uses recursion. For example a definition such as

```
F(t): RECURSIVE real =
   IF init(t) THEN A(t)
   ELSE F(pre(t)) + A(t)
   ENDIF
 MEASURE rank
```

means that `F` accumulates the successive values of `A`. The predicate `init` and the functions `pre` and `rank` are generic and defined for every clock. `init`(t) is true if t is the first element of the clock, `pre`(t_i) is equal to t_{i-1} and `rank`(t_i) is equal to i. The measure clause in the definition is required by PVS for every recursive function and is used to ensure that the function is well defined so that the recursion always terminates.

In most points, this model is similar to an abstract state machine, as used in [3, 5] for example. The state at time t_i is a vector $X_1(t_i), \ldots, X_n(t_i)$ where $X_1, \ldots, X_n$ are variables of the module. The transition function and the initial state are implicitly contained in the definition of all these variables. The restricted form of recursive definition shown above ensures that the state at time t_i depends on the value of input variables at t_i and on the state at t_{i-1}.

Using this form of data flow specifications, the functional requirements are all expressed in a purely definitional style. This is strong evidence concerning the consistency of requirements. When type checking the specifications, PVS may generate various proof obligations known as TCCs. For example, a PVS specification such as

```
cmd_range : TYPE =
  {x: real| 0 <= x AND x < M }

CMD(t): cmd_range = A(t) * A(t),
```

defines a data flow `CMD` of range `cmd_range`. When analyzing this specification, PVS will generate a TCC to ensure that the expression `A(t) * A(t)` is effectively within the allowed range:

```
CMD_TCC1: OBLIGATION
  FORALL t:
    0 <= A(t) * A(t) AND A(t) * A(t) < M.
```

Provided all such TCCs are discharged, we know that all the data flows are well defined functions; there is no risk of inconsistency. Since PVS requires that all functions are total, this style of specification also ensures the total coverage of input domains.

3.2 Assumptions and Critical Properties

Assumptions about the system under control and the critical properties to be verified are written as PVS axioms and conjectures, respectively. Unlike the functional requirements, assumptions and properties may be related to non-discrete quantities such as external physical parameters or the position of mechanical components. For example, assuming a system receives input signals `Pt` and `Ps` from a probe measuring atmospheric pressure then the two inputs might be declared as

```
Pt, Ps: [time -> pressure_range]
```

and we can write assumptions such as

```
pressure_constraint: AXIOM
   FORALL (t:time): Ps(t) <= Pt(t).
```

In the same way, a critical property is written as a conjecture using explicit time indices and quantifiers. For example, a typical response property could look like:

```
response: CONJECTURE
   FORALL t : P1(t) IMPLIES
     EXISTS u :
        t <= u AND u <= t + D AND P2(t);
```

P2 is expected to hold within a delay D after P1 is true.

3.3 Verification

Once formalized, the requirements for a real-time controller consist of one or several functional modules specified as a set of data-flow definitions, a collection of axioms which describe assumptions about the controller's environment, and a list of conjectures which specify critical properties. There are also other elements such as the communication between the modules or between functional modules and the controlled system.

The verification of requirements is then a theorem proving exercise; we have to prove the conjectures using the axioms and the data flow definitions. In practice, the proofs of critical properties can be complex and require the introduction of many intermediate theorems and lemmas.

It is also useful to prove simple properties of the specifications (so-called *putative theorems*). Such properties are intended to check that the formalization of assumptions or functional requirements is reasonable. They are typically simple properties one expects to be true of the system which may or may not help prove the main results.

4 The Case Study

This methodology and formalization approach have been applied within SafeFM to a realistic case study. The system is based on an air-data computer which controls the flaps and variable geometry wings of an aircraft. The example is typical of the target applications. It is a real-time system with a redundant architecture and it performs complex control functions as well as failure detection tasks. Originally, the requirements were expressed in a mixture of English description, mathematical formulas and various graphs. The requirements were produced by software engineers with previous experience with systems similar to the case study and were inspired from a large subset of an existing air-data computer.

The remainder of this section gives an overview of the air-data computer and of its formalization and verification. A more detailed description of the case study appears in [7].

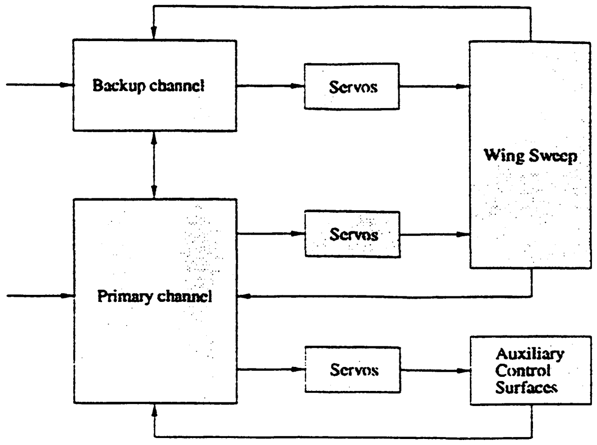


Figure 1: ADC architecture

4.1 Architecture

The architecture of the controller is shown in Fig. 1. The system includes two channels with different clocks, each channel being composed of up to four functional modules. A primary channel performs all the controller's function during normal operation and a backup channel with restricted functionality takes over when the primary fails. The two channels do not synchronize; they have two independent clocks of different frequencies. The only cross channel communication is a discrete signal which indicates failures of the primary channel to the backup channel.

The commands from the two channels are transmitted to servos which control the wing sweep actuators. Feedback signals from the servos and actuators are continuously monitored by the two channels in order to detect mechanical failures. The controller receives other input signals from the pilot, from various sensors and probes, and from other avionics systems in the plane.

```
wing_sweep_primary[
     (IMPORTING time, types)
     ALTITUDE : [time-> altitude_range],
     MACH     : [time-> mach_range],
     ...] : THEORY

  BEGIN
  ...

  t : VAR clock[PRIMARY_PERIOD]

  ...

  AUTO_MODE(t) : RECURSIVE bool =
      if      DESELECT_AUTO(t)
      then    FALSE
      elsif   SELECT_AUTO(t) or init(t)
      then    TRUE
      else    AUTO_MODE(pre(t))
      endif
    MEASURE rank
  ...

  WSCMD1(t) : RECURSIVE ws_range = ...

  END wing_sweep_primary.
```

Figure 2: Example of functional module

4.2 Functional Requirements

The PVS formalization of the functional requirements consists of six modules describing the six main functions of the controller. Each of these functional module is specified as a separate PVS theory. Fig. 2 gives an overview of such a functional module. The theory specifies the wing sweep control function performed by the primary channel. The theory parameters ALTITUDE, MACH, etc. specify the input signals received by the module. The theory contains a list of data-flow definitions which are all functions of the same clock, the clock of the primary channel. The figure shows two examples of data-flows; AUTO_MODE corresponds to an internal boolean variable and WSCMD1 specifies the wing sweep commands produced by the primary channel.

4.3 Assumptions and Safety Properties

The assumptions give a (crude) model of mechanical and electro-mechanical components of the controlled system (sensors, actuators, and interlocks). For example, the axiom cmd_wings below relates the wing sweep angle (WSPOS) and the wing sweep commands (CMD) when certain mechanical interlocks are not active. The value CMD(t) depends on the wing sweep command produced by the active channel.

```
cmd_wings : AXIOM
    constant_in_interval(CMD,t,t+eps)
 and
    not wings_locked_in_interval(t,t+eps)
 implies
    CMD(t) = WSPOS(t+eps)
  or
    CMD(t) < WSPOS(t+eps) and
    WSPOS(t+eps) <= WSPOS(t) - eps*min_rate
  or
    CMD(t) > WSPOS(t+eps) and
    WSPOS(t+eps) >= WSPOS(t) + eps*min_rate
```

Other axioms describe the evolution of the wing sweep angle when the locks are active, the initial position of the wings, and various contraints about other components of the system.

The air-data computer has to satisfy two main safety properties related to constraints on the position of the wings when various sets of flaps are extended. Our first attempt to verify these properties failed because of a lack of information about the system under control. In particular, our first model did not include mechanical interlocks which ensure that the flaps do not extend at the wrong time. Other mechanical locks limit the extension of the wings when the flaps are not retracted.

After several iterations and consultation with GEC-Marconi engineers, we reformulated the initial safety requirements in order to take the mechanical locks into account. We defined the *safe states* of the two channels as those where the wing sweep commands issued cannot force the wings against the mechanical interlocks. We then specified three safety properties:

- While the primary channel is in control, the system stays in a safe state.
- After the backup channel assumes control, it converges to a safe state within a specified period of time.
- If the backup channel is in control and in a safe state, it will stay in a safe state.

These three constraints are expressed easily in PVS. The first property ensures that the system is in a safe state until a possible failure of the primary channel. The second corresponds to a transitory period from the instant the primary channel fails to the instant the backup channel reaches a safe state. The third property ensures that once the backup channel has reached a safe state, the system remains in a safe state.

The second property indicates that the backup channel may not be immediately in a safe state when the primary channel fails. This is due to the absence of synchronization between the two channels. Because the two subsystems have different clocks and receive input from separate sources, there can be a substantial difference between their internal states and between the wing sweep commands they compute. If the backup channel takes control while the difference is large, it may momentarily produce commands which force the interlocks but the latter ensure that the wings remain in a safe position.

4.4 Results

The formalization of the system consisted of approximately 4500 lines of PVS specifications. This included basic theories and definitions for time and clocks, a collection of general purpose functions and theorems (*background knowledge*), and the specification of the case study proper. The amount of effort involved in formalization and verification is estimated at around 18 person months. Approximately half of this time was spent on the verification of the three main properties. As a whole, a total of 385 proofs were performed. These include 106 TCCs most of which were discharged automatically by the prover; the rest was proved by hand. The verification of the three top-level critical properties required the proof of a total of 124 propositions. The other properties we verified were putative theorems whose proofs did not pose any major difficulty.

During the proof of a putative theorem, an error was found in our formalization of the functional requirements. This corresponded to an unexpected situation where a "lower limit" gets larger than an "upper limit" and the resulting commands are wrong. This error is triggered by an exceptional combination of input parameters. It can be traced back to the original informal requirements where the possible inversion of the two limits is completely overlooked. This was the only error discovered in the functional requirements. After a simple correction, all the putative theorems were proved.

The main difficulty we encountered was a lack of information about the system under control and the imprecision of the informal safety requirements. There was only limited information about these aspects in the original requirements which were destined primarily for software engineers and focused on functionality. As a result, our first attempts to prove the safety properties were unsuccessful. The solution was to consult with systems engineers who have a wider view of the system including hardware and software as well as safety mechanisms. We also got more information about the expected behavior of the controller from various documents including the pilot's manual. The interaction with systems engineers and the new sources of information helped us clarify the safety requirements and formulate the correct assumptions. Once the precise requirements were established, we were able to prove the safety properties.

The experiment clearly demonstrated the advantage of the PVS specifications over the original informal document; the formal requirements are concise, precise, and unambiguous. In this respect, the SafeFM case study has confirmed other authors' conclusions about the value of formal requirements specifications [3, 5].

However, the main benefits of the formal approach were realised during the validation stages. Proofs are an essential means of detecting errors in requirements. The proof process requires a thorough analysis and gives a profound understanding of the system behavior. In particular, proofs can help understand the subtle interactions between the components of a complex control system.

An important conclusion of the case study is that formal methods can be applied in practice to the requirement analysis of real industrial systems. By the size of its specification and the number of proofs performed, our experiment represents a major effort in formal verification. Such large scale verification is feasible and beneficial provided adequate tool support is available. In this respect, SafeFM has corroborated other reports on industrial uses on PVS, such as [12].

However, another lesson of the experiment was that the proof process can be expensive and difficult to estimate. It took us eighteen months to complete the work instead of the six we had originally planned. Some of these delays can be attributed to our lack of familiarity with PVS. Other factors can also explain this time overrun:

- There was little guidance on how to apply PVS effectively to real-time control applications. A non-negligible part of the work was taken up in writing PVS theories defining the basic notions of clocks and time.

- We had to spent time in developing and proving various general purpose results. For example, we had to prove properties of sets of real numbers or of finite sets.
- The initial requirements document was written from a software engineering point of view and contained little information about the system under control. Several iterations were required before formulating the correct assumptions and getting the precise safety requirements.

5 Building PVS Libraries

Some of the difficulties encountered were due to a lack of maturity of the PVS prover. In order to reduce the effort involved in formal specification and in verification, it is necessary to provide general purpose PVS libraries. Fortunately, this need has been identified and the current version of PVS comes with more predefined notions and with better support for libraries[3]. An important issue for the PVS community is to develop and make available further libraries.

In order to contribute to this effort, we have developed various PVS libraries during the last months of the SafeFM project. The main one includes basic results of real analysis [6]. Some aspects of the case study specification could have been simpler to model if such a library had been available. Other PVS developments were related to finite sets and have been incorporated in a more general library [2]. In more recent work, we defined a smaller set of PVS theories for manipulating roots and square roots of real numbers[4].

The main objectives when building such libraries are generality and usability. We need to define notions as generally as possible to make them applicable to many classes of problems. We also need to present the results in a convenient form to ease their use in PVS proofs.

In developing the SafeFM libraries, we used the following principles:

- For the notion of continuity to be useful in practice, we cannot restrict ourselves to the case of total functions from $\mathbb{R}$ to $\mathbb{R}$; we must be able to consider continuous functions defined on subtypes of the reals.
- The most convenient and powerful way of using lemmas in proofs is using the rewriting capabilities of PVS. The mechanism takes lemmas which have the syntactic form of a "rewrite rule" or of a "conditional rewrite rule". The user can either apply these rules selectively or install them as automatic rules which will be used internally by the theorem prover.

  Another important capability is the general proof commands PVS provides for reasoning by induction. These commands use (explicitly or implicitly) lemmas which must have a particular form. We need to provide such induction rules when necessary.
- PVS relies on a powerful type system and type checking mechanism. The latter can generate proof obligations to ensure that specifications are consistent. As far as possible we need to reduce the number of proof obligations that might be generated when using lemmas and functions from the library. A new feature of PVS – *the judgements* – can be of great help in this respect.
- Taking advantage of the decision procedures is another way of making lemmas and theorems simpler to use; we need to provide results in a form such that the decision procedures can be applied.

  This can sometimes work against the principle of generality. For example, the following important property is proved in the continuity library: from any sequence of reals, one can extract a monotonic subsequence. This property is also true for sequences of any type T, provided T is totally ordered. Unfortunately, manipulations of order relations on general, non-numerical types are extremely tedious because no decision procedures apply.

There are simple pragmatic rules one can adopt which can make libraries both easier to use and widely applicable:

- Generality can be achieved by using parameterized theories and the subtyping structure. For example, parameters allowed us to define continuity of functions of type `[T -> real]` where `T` is any subtype of the reals.

[3]Most of the SafeFM work was done with an older version which did not include these facilities.

[4]These PVS libraries are accessible at http://www.cs.rhbnc.ac.uk/research/formal/safefm-pvs.html or ftp://ftp.cs.rhbnc.ac.uk/pub/safefm/pvs/roots.dmp.gz

- Usability can be improved by writing lemmas and properties in certain syntactic forms in order to make them usable as rewrite or induction rules. The possible forms of rewrite rules are presented in [16]; a few examples are given below

```
sqrt_def : LEMMA
   sqrt(x) = y IFF y * y = x

square_sqrt : LEMMA
   sqrt(x) * sqrt(x) = x

sqrt_square : LEMMA
   sqrt(x * x) = x.
```

  The two variables x and y are non-negative reals.

- It is convenient to offer several variants of the same theorem, even if they look redundant. For example, it does no harm to have the following lemmas about square roots even though anything provable with the last three lemmas is provable from the first one alone.

```
both_sides_sqrt_lt : LEMMA
   sqrt(x) < sqrt(y) IFF x < y

both_sides_sqrt_le : LEMMA
   sqrt(x) <= sqrt(y) IFF x <= y

both_sides_sqrt_gt : LEMMA
   sqrt(x) > sqrt(y) IFF x > y

both_sides_sqrt_ge : LEMMA
   sqrt(x) >= sqrt(y) IFF x >= y.
```

  With these four variants, many properties can be proved by automatic rewriting. A single lemma would require much more tedious manual intervention to instantiate the variables.

- The *judgement* clauses are extremely useful for simplifying type checking and reducing the number of proof obligations. This mechanism is described in [13, 4]; it allows us to specify for example that the square root of a positive number is positive:

```
JUDGEMENT sqrt
   HAS_TYPE [posreal -> posreal].
```

- It may be sometimes necessary to reduce generality in order to improve usability. This is in particular the case when notions are defined on types too general for the decision procedures to apply.

All the above rules provide only partial answers to the difficult problem of building usable libraries. It is hard to anticipate what results and lemmas can be useful in practice and to know how to present them in a convenient form. Such issues can only be resolved by acquiring more experience in library development and by learning from users of libraries.

6 Conclusion

One main objective of SafeFM was to investigate practical methods of producing coherent software requirements for digital controllers, using a formal approach and theorem proving. The case study experiment has confirmed the potential benefits of formal requirements analysis. A formal specification provides clear advantages such as clarity and precision, but the main benefit in our experiment was the feasibility of thorough analysis via proofs.

The project showed that PVS provides adequate tool support for such an analysis. A relatively large and complex system was completely formalized and verified with PVS. We believe that the theorem proving technology is mature enough to be applied to real-life, industrial applications. The main practical issue is to provide guidance on the use of theorem provers to specific classes of problems and in developing libraries to facilitate specification and verification.

Acknowledgements

The work described in this paper would not have been possible without the support of the project partners. We are indebted to Tim Boyce, Jonathan Draper, Bob Smith and our other GEC-Marconi colleagues who have had to educate us in avionics systems.

References

[1] J. R. Abrial. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[2] R. W. Butler. The NASA Langley PVS Library, March 1996. This note and the library it describes can be accessed via `http://atb-www.larc.nasa.gov/ftp/larc`.

[3] J. Crow and B. Di Vito. Formalizing Space Shuttle Software Requirements. In *ACM SIGSOFT Workshop on Formal Methods in Software Practice*, January 1996.

[4] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. In *WIFT'95 Workshop on Industrial-Strength Formal Specification Techniques*, April 1995.

[5] B. Di Vito. Formalizing New Navigation Requirements for NASA's Space Shuttle. In *FME'96*, March 1996.

[6] B. Dutertre. Elements of Mathematicl Analysis in PVS. In *Theorem Proving in Higher Order Logics: 9th Intermational Conference, TPHOLs'96*, pages 141–156. Springer-Verlag, LNCS 1125, August 1996.

[7] B. Dutertre and V. Stavridou. Formal requirements analysis of an avionics control system. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[8] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Data Flow Programming Language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1321, September 1991.

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

[10] Cliff B. Jones. *Systematic Software Development using VDM.* Prentice-Hall International, 1986. 2nd Edition.

[11] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming Real-Time Applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[12] S. P. Miller and M. Srivas. Formal Verification of the AAMP5 Microprocessor: A Case Study in the Industrial Use of Formal Methods. In *WIFT'95 Workshop on Industrial-Strength Formal Specification Techniques*, April 1995.

[13] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[14] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language.* Computer Science Lab., SRI International, April 1993.

[15] D. L. Parnas. Some theorems we should prove. In *Proc. of 1993 International Meeting on Higher Order Logic Theorem Proving and Its Applications*, pages 156–163. The University of British Columbia, Vancouver, BC, August 1993.

[16] N. Shankar, S. Owre, and J. M. Rushby. *The PVS Proof Checker: A reference Manual.* Computer Science Lab., SRI International, March 1993.

[17] V. Stavridou, A. Boothroyd, T. Boyce, P. Bradley, J. Draper, B. Dutertre, and R. Smith. Developing and Assessing Safety Critical Systems with Formal Methods: the SafeFM Way. *Journal of High Integrity Systems*, 1(6):541–545, 1996.

Reuse of a Formal Model for Requirements Validation

Robyn R. Lutz*

rlutz@cs.iastate.edu
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109-8099

Abstract

Abstract

This paper reports experience from how a project engaged in the process of requirements analysis for evolutionary builds can reuse the formally specified design model produced for a similar, earlier project in the same domain. Two levels of reuse are described here. First, a formally specified generic design model was generated on one project to systematically capture the design commonality in a set of software monitors onboard a spacecraft. These monitors periodically check for faults and invoke recovery software when needed. The paper summarizes the use of the design model to validate the software design of the various monitors on that first project. Secondly, the paper describes how the formal design model created for the first project was reused on a second, subsequent project. The model was reused to validate the evolutionary requirements for the second project's software monitors, which were being developed in a series of builds. Some mismatches due to the very different architectures on the two projects suggested changes to make the model more generic. In addition, several advantages to the reuse of the first project's formal model on the second project are reported.

1 Introduction

In some application domains, successive software projects tackle many of the same problems. In such applications, software design from prior projects in the same domain or product family is sometimes used to guide the requirements for a current project. At the informal level this occurs when "Lessons Learned" on earlier projects are collected and considered during the requirements phase of a subsequent project. In other cases, reuse of existing software components or design patterns from a previous project may be mandated on a new project, with the reuse sometimes driving the requirements [20].

This paper investigates how a project engaged in the process of requirements analysis can exploit the formal design modeling and analysis done on a similar past project in the same application domain. The approach is outlined in Figure 1.

The paper describes two applications of reuse:

1. A formally specified design model was generated on Project 1 to systematically capture the design commonality in eighteen software monitors. This generic model was then reused to validate the software design of each of the eighteen monitors.

2. The formal model created for Project 1 (the Cassini spacecraft) was subsequently reused to analyze the requirements for similar software (i.e., fault monitors) on a second project in the same application domain.

   Each element (data item or function) of the generic formal model produced for Project 1 was either traced to the requirements for the monitors in Project 2 (the Deep Space-1 spacecraft) or the discrepant element was noted and investigated.

In addition to the anticipated benefit of validating requirements in the current build of Project 2, the work also clarified the allocation of requirements among the software elements, provided a structured way to capture design constraints and design assumptions during

*Mailing address is Dept. of Computer Science, Iowa State University, Ames, IA 50011. The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with NASA.

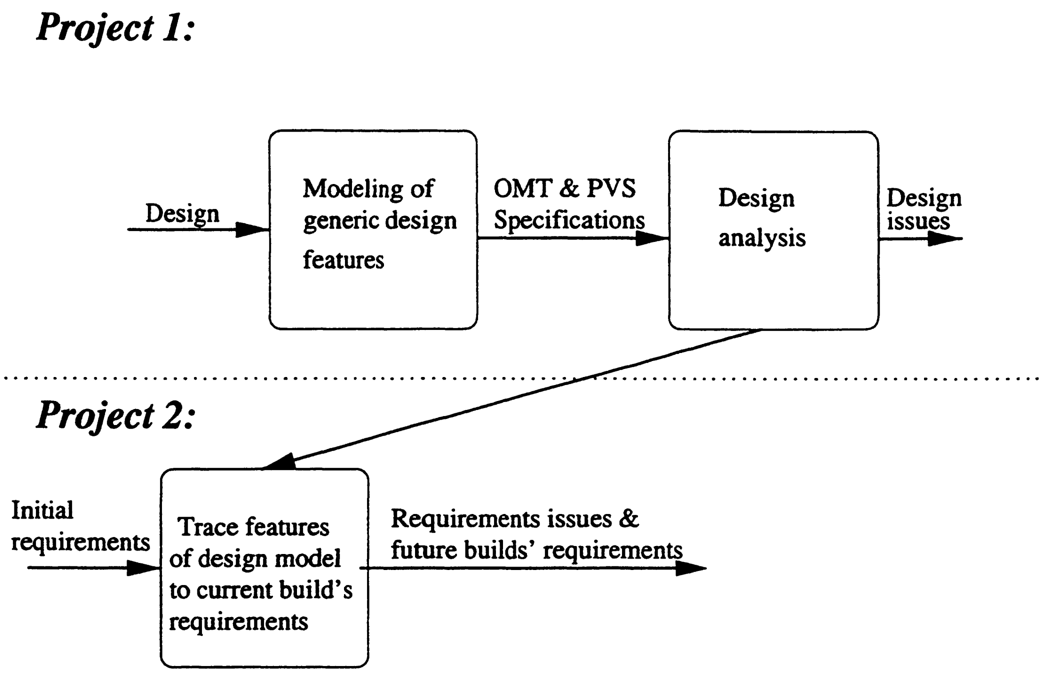

Project 1:
Design
Modeling of generic design features
OMT & PVS Specifications
Design analysis
Design issues
Project 2:
Initial requirements
Trace features of design model to current build's requirements
Requirements issues & future builds' requirements

Figure 1: Reuse of Project 1's Formally Specified Design Model for Requirements Validation on Project 2

requirements analysis, and guided the identification of requirements to be added in later builds during the evolutionary software development process.

The need for rapid, low-cost requirements analysis and the planned, steady evolution of requirements on the new project motivated the reuse of the earlier design model. The goal was to import some of the lessons learned about system-level fault protection monitor design on the earlier project into the subsequent project in a structured but informal way. The results show that, although the software architectures and the development processes for the two systems are very different (see Section 4), the design model from the earlier project provided some guidance for validating a specific build's requirements and identifying future builds' requirements in the second project. By tracing each element of the earlier formal design model to the preliminary requirements for the later system, additional insights into the assumptions underlying the requirements, the design constraints of the new system, and the criteria for design choices were gained.

The assumption underlying the experimental reuse of the first project's formal model on the second project's requirements is that similar behavior and similar data must occur in each monitor in this domain. This assumption turned out to be largely true. The data items and functions in the formal specification can often be mapped directly to a data or behavioral requirement in the second project's monitors, adding some assurance that the requirements are complete and correct. This mirrored the experience applying the generic formal model to the eighteen monitors in the first project. In that case, flagging deviations of particular monitors from the norm (i.e., the generic model) was a quick way to identify areas of concern for additional analysis or testing. Similarly, identification of instances in which the second project's monitors deviated from the first project's generic formal model provided insights into currently missing requirements (to be required in later builds) and into implied design constraints.

Both projects involved fault-monitoring software in the same domain. Project 1 is the Cassini spacecraft, set for an October, 1997 launch to Saturn and its moon, Titan. Project 2 is the Deep Space-1 (DS-1) spacecraft, which will be launched in 1998 to rendezvous with an asteroid and a comet [7]. The software described here was, on both projects, the system-level fault-protection monitoring software. In the spacecraft domain, a monitor is software that periodically checks for malfunctions and initiates a process leading to recovery when appropriate. The monitors are the "eyes and ears" of the spacecraft [15].

In both projects the software monitors are required to display similar behavior (e.g. ignoring transient faults) and to use similar data (e.g., fault thresholds against which the input data is measured to determine if a fault exists). A fault is defined to be either "a defect in a hardware device or component" or "an incorrect step, process, or data definition in a computer program" [9]. Monitors are safety-critical software in that they must autonomously detect onboard threats to the spacecraft's health and mission. Since fault monitoring software in other applications' control systems frequently displays many of the same behavior and data dependencies represented in the formal model here, the approach described in this paper may have application beyond the spacecraft domain.

The rest of the paper is organized as follows. Section 2 describes the formal specification and analysis of the generic model. Section 3 summarizes the results from its use on the first project. Section 4 surveys some relevant commonalities and differences between the two projects in terms of their software development processes (waterfall vs. spiral) and their architecture (centralized vs. remote agents). Section 5 describes the results from the application of the design model to the requirements analysis of the software builds on the second project. Section 6 briefly discusses related work and future directions. Section 7 summarizes the lessons learned from the experience reported here.

2 The Formal Model

In previous work we used two technologies, formal methods and object-oriented modeling, to analyze the software design for portions of the Cassini spacecraft's software [1, 12]. The two tools that were used were OMT, the Object Modeling Technique [17], and PVS, the Prototype Verification System tool (SRI). PVS is an integrated environment for developing and analyzing formal specifications using support tools and a theorem prover [18]. These tools allowed the modeling, formal specification, and analysis of the monitors' design in the Cassini system-level fault protection software [11].

There are eighteen monitors in the system-level fault protection onboard the Cassini spacecraft. Eight of these are overtemperature monitors that are nearly identical in their logic. The other ten fault monitors detect loss of commandability (uplink), loss of telemetry (downlink), heartbeat loss (i.e, communication between computers), overpressure, overtemperature, undervoltage, and selected other failures.

These monitors share many of the same functions and attributes. One of the roles of the OMT models in the design analysis was to explicitly represent these common features in a way that could be readily reviewed by the Cassini engineers. This approach worked well. The OMT approach provides three viewpoints from which to represent the software design. The design of the fault protection monitor was thus represented in three OMT design diagrams [11]:

- The object-oriented design approach was represented in a object diagram. At the design phase, the object model of the monitor provided insight into the common behavior, properties, and relationships that the various monitors share. The OMT diagrams allowed the similarities in the monitors to be compactly represented.

- The functional viewpoint was represented in a data flow diagram. The data flow diagram characterized the data and data transformations common to all Cassini system-level fault protection monitors.

- The dynamic viewpoint was represented in a state diagram (Fig. 2). The state diagram for the design contained a sequence of six states that an active monitor can reach. Monitors commonly (1) test the validity of the input measurements that they receive from the sensors, (2) detect the existence of a fault condition in the various input data, (3) decide whether a fault condition in fact exists (perhaps by voting), (4) disregard transient anomalies, (5) determine whether a recovery response is appropriate, and (6) update state data, including possibly a request for a recovery response.

The formal specification in PVS of the design for the monitor software drew on the OMT diagrams to guide the formal specification of the design model. This was consistent with our earlier experience that creating OMT diagrams prior to formally specifying the requirements enhanced the accuracy of the initial formal specifications and reduced the effort required to produce them [12]; see also [2]. The formal specification in PVS of the design for the monitor consists of two theories (five pages of typechecked PVS specifications). The first theory, called **mon**, specifies the design of a system-level fault protection monitor (Fig. 3). Voting behavior (among inputs on whether a fault exists) was not represented in the formal model, although it would be a desirable addition.

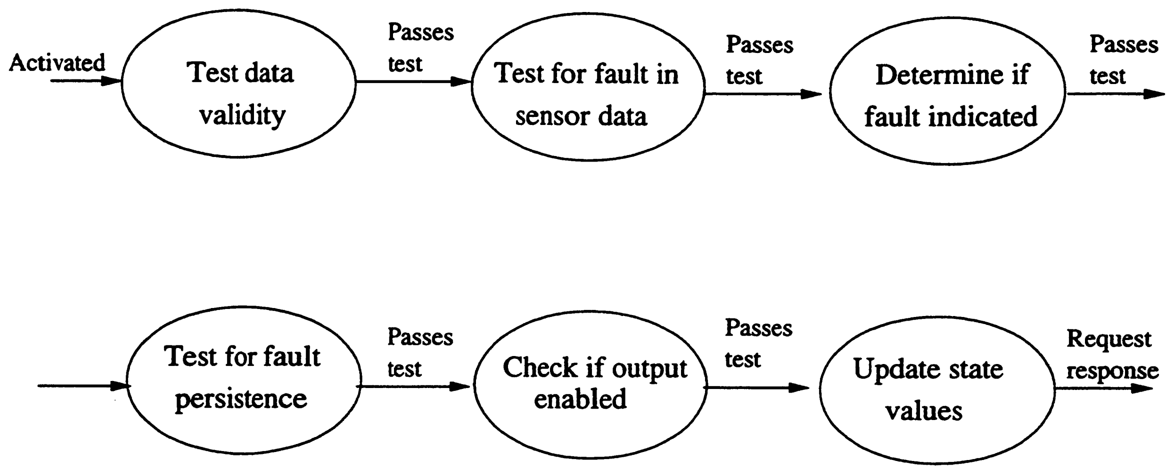


Figure 2: Dynamic Model

```
% Generic Cassini System-Level Fault Protection Monitor
% Monitor requests recovery response.

request_response (i, x, threshold, low_filter, high_filter, enabled,
                  prev_persist_ctr, persist_limit, sensor_input): bool =
                    valid_data_exists (i, low_filter, high_filter)
                       AND cond_exists  (i, x, threshold, low_filter,
                          high_filter, sensor_input)
                       AND cond_persists (i, x, threshold, low_filter,
                          high_filter, prev_persist_ctr, persist_limit,
                          sensor_input)
                       AND enabled
```

Figure 3: Excerpt from the Formal Model

The second theory, called `monlem`, states seven lemmas that specify the monitor's behavior. Two of these lemmas describe basic safety properties that the monitor is expected to obey, as documented in the project requirements ("A response is requested by a monitor only if the detected fault is not a transient fault" and "If a fault is not detected by a monitor, then the monitor doesn't request a response"). The other five lemmas concern the monitor's interfaces, i.e., the validity of the input data it uses to make a control decision as to whether to request a recovery attempt. The seven lemmas were proven, several by Martin Feather.

The conformity of the OMT representation and the formal specifications to the actual, final software design was checked against the eighteen system-level fault protection monitors in the post-Critical Design Review document [3, 4]. One step in evaluating that the model accurately represented the design was to classify the Data Lists provided in [3] for each of the monitors, and then to map those data classifications to the model. Toward this goal, the 162 data items in the Data Lists were classified into eight categories. In descending order of frequency, the eight categories were: Measurements (input data from sensors), Enabled/Disabled flags (monitors can be disabled), Thresholds (limits beyond which a fault condition exists), Filters (persistence requirements so that transient faults are ignored), State Updates (e.g., "high-water"-the highest measurement seen to date), Validity ranges (measurements outside these ranges are assumed to be from failed sensors), Heartbeat/Messages (from other software), and State Currently Commanded (information about the current configuration). The eight categories of data found in the document were reflected in the OMT design model and the formal specifications.

3 Reuse of the Formal Model on Project 1

The design analysis involved in constructing the model, in formally specifying the design of the monitor, in stating and proving the lemmas, and in confirming the accuracy of the model and specifications, identified eight deviations of Project 1's individual fault monitors from the formal specification of the generic design. Four of the eight discrepancies were found through the design analysis needed to develop the formally specified generic model. The other four of the eight discrepancies were found by comparing the Data Lists for each of the eighteen fault monitors with the data in the formal specification. The issues found were reported back to the Cassini project. In each case, either a decision was made to retain the current design for the reason listed below or the issue was still being worked when this phase of the analysis ended.

None of these discrepancies involved errors in the design logic, but several yielded discussions about the most robust way to design the software or how best to flag these items for attention during testing. Among the more interesting issues found by the analysis were:

- Unnecessary coupling. In a single case, a monitor cancels a response. This appears to be unnecessary coupling of the monitor and the response, since in all other cases, the Fault Protection Manager performs this supervisory function. Conversations with the software designers indicated that, since there is no known problem with the current unique design for this monitor, that it will be retained.

- Unique design. In a single case, a monitor disables itself (permanently or until ground operators interfere) if it even once receives bad data. In all other cases, the response disables its related monitor. The design ratonale is that past flight data indicates that once a transducer (the hardware component providing the data) provides unhealthy data, that it is likely to be unrelative from then on. However, other monitors do not cease to accept data from transducers that have once provided unhealthy data.

- Missing data validity checks? Inputs from the transducers on the propellant tanks and temperature transducers are checked for validity (i.e., whether the values are outside a range of credible values, which would indicate a failed transducer); however, inputs from other sensors do not undergo similar validity checks. At the time the study ended, this design issue was being worked as an action item by the design team.

- Misleading data name. In one monitor, the monitor can update the value of two global variables even when the monitor's "Enabled flags" indicate that its output is disabled. Thus, in this one case, a disabled monitor can produce output, contrary to what the flags' names suggest.

Several other discrepancies involved inconsistent documentation practices (e.g., a passed variable was in one case inaccurately listed as a global variable). By raising these issues while the design was still being defined, the Cassini design team was able to resolve them (by change or by documenting a rationale for the inconsistency) prior to implementation and testing.

The key benefit of abstracting from the documentation of individual monitors to model and formally specify the design of a general-purpose monitor in this study was to support design analysis. Flagging the design deviations of specific monitors from the general pattern was useful to the design verification process. These deviations are more likely to be design errors, more likely to be implemented incorrectly (because they are exceptions to the norm), and more likely to be overlooked in the selection of test cases than other software. For example, Software FMEA's (Failure Modes and Effects Analyses) that were being performed at the same time on the Cassini system-level fault protection software incorporated the design deviations found via the formal methods analysis into the SFMEA process [13].

4 Similarities and Differences in the Two Projects

The fault monitors on the two spacecraft have much in common. In both systems the fault protection software is divided into software monitors, that detect when a fault condition exists, and software responses, that take autonomous action to command the spacecraft to a known-safe state.

Despite the similarities in the responsibilities of the fault monitors in the two systems, there are major differences in both the software architectures and in the development processes. System-level fault protection on Cassini is managed by a Fault Protection Executive that runs in a separate virtual machine. This is consistent with the basic fault protection design of prior spacecraft from which Cassini inherited portions of its fault-protection architecture. DS-1, currently under development, instead uses an innovative architecture based on recent advances in remote agent software, artificial intelligence, and robotics to monitor and recover from faults.

The software development process on Cassini is essentially a waterfall software development process tailored to the needs and constraints of the overall system development. The software development process on DS-1 uses an evolutionary model, similar to the spiral process model, with rapid development of an initial system and three subsequent builds to add incremental functionality. The requirements for a next build are derived in large part from the testing of the previous build. Consequently, the distinction between requirements and design is blurred. Risk identification during testing, and risk resolution during the definition of the next build's goals and constraints, drive the evolution of the requirements.

This evolutionary development of requirements, together with the shift from a familiar to a relatively new spacecraft architecture, motivated the decision to reuse the Cassini design model. Concerns with requirements completeness and consistency for each build's baseline could be addressed in part by reference to the previous project's formally specified generic model.

On DS-1, two monitors have been established as C++ classes which can be reused by various subsystems [5, 14, 15, 16]. The two monitors are a Threshold Monitor, which closely parallels the functionality of the generic Cassini monitor in detecting when a persistent fault occurred, and a Transaction Monitor, which reports on the status of a transaction (e.g, the successful or failed attempt to take a picture). On Cassini the generic monitor model was created strictly for independent design validation; on DS-1 the monitor class is incorporated directly into the implementation. The two DS-1 monitors are specified by means of state transition diagrams, supplemented by descriptions of the data. It is these specifications of monitors to which the Cassini generic design model was mapped.

One difficulty in mapping the generic formal model to DS-1's requirements was that the states in DS-1's monitors represent the software's knowledge of the hardware device it is monitoring, whereas the states in Cassini's monitors represent the monitor's own state of execution. This difference complicated the mapping since the same condition may lead to different states in the two systems. For example, on Cassini when the predicate "sensor surpasses threshold" becomes true, the software changes state from "Determine if fault indicated" to "Test for fault persistence." In DS-1 the same predicate causes the state to change from, e.g., the nominal state ("Looks-OK") to the state indicating that a fault may exist.

5 Reuse of the Formal Model on Project 2

Nine threshold monitors and seven transaction monitors are included in a recent, intermediate build of DS-1 [15, 16]. An early estimate was that there will be at least thirty monitor instances in the final launch code [6]. In addition, aggregate monitors will be composed from several Threshold Monitors.

The formal specification in PVS of the generic design model of the Cassini system-level fault protection monitors was used to build two tables, one for the data items in the formal model and one for the functions in the formal model. In these tables each element of the formal generic model (data and functions) was traced to the current DS-1 build's requirements for the Threshold and Transaction monitors.

The tables list nineteen elements of the design model (the external inputs to the monitor, the prior state inputs and next state outputs, the external outputs, and the functions). Some simple data items and functions needed only for correct PVS specification or to simplify proofs (e.g., a variable that specifies the ordinal number of an input within a set), but that were not present in the OMT models, are excluded from the tables. Figure 4 shows an example from the Data Table of the input data item, "persistence_limit". Figure 5 shows a sample function, "condition_persists". In the tables, "S" is a structure holding the internal monitor state, with S.Persistence being the limit on the number of unacceptable values, beyond which an error is declared; and S.Count being the number of unacceptable values [15]. (The actual tables are turned sideways with each row of the table describing a data item or function, and the Transaction and Threshold monitors described separately, but are reformatted here for readability of the excerpts.) in a paper.)

Those elements that are present in the Cassini design model but are neither in a current DS-1 build nor have a clear rationale for being excluded are candidates for software requirements for future DS-1 builds. On the other hand, because the monitors in DS-1 have less spacecraft redundancy to manage, some data and behavior present in Cassini monitors are not required on DS-1. For example, the DS-1 monitors do not receive redundant sensor data, so do not need the "num_sensors" data item present in the Cassini design model.

The following types of issues arose during the development and inspection of the tables:

1. *Evolutionary requirements.* The tables were most useful in identifying some requirements that needed to be added in later builds. The tables allowed the current status of the requirements to be tracked against the generic design model for similar software. The tables thus serve as a partial checklist against which the evolving requirements can be measured. The tables also allow some possible requirements for future builds to be inferred. For example, the collection of data required for ground diagnosis of the monitor's state was explicitly deferred to a later build (e.g., the collection by the Threshold Monitors of the highest and lowest values that each monitor ever sees for downlinked telemetry). The tables capture this intent for later builds' requirements.

2. *Validation of requirements in the current build.* Instances in which the DS-1 monitors were not required to exhibit behaviors that were present in the generic model were fed back to the project for confirmation. In most cases investigation yielded a rationale for the exclusion (e.g., there is not the same redundancy in sensor input on DS-1 as on Cassini, so DS-1 monitors do not need the "number of sensors" data item that the Cassini monitor used) or a better understanding of the spacecraft interfaces (e.g., that on DS-1 some lower-level filtering of the data occurs before the data are made available to the monitors). In a few cases deviations of a DS-1 monitor from the generic model led to continued discussions of some requirements decisions. For example, the use of dual thresholds ("too-high" and "too-low") in the Threshold Monitor raised questions about whether a single persistence counter suffices, since it masks whether the upper threshold, lower threshold, or some combination of both thesholds has been repeatedly surpassed.

3. *Clarifications of requirements allocation.* Given DS-1's innovative architecture and increased autonomy, there is an ongoing focus by the project on providing clean interfaces among software components and on avoiding the possibility that requirements drop through the cracks. The process of systematically either mapping each element of Cassini's design model elements to DS-1's monitors or of documenting why that element was not needed in DS-1 assists in this effort. The production of the tables prompted several questions about requirements allocation that crossed subsystem boundaries, leading to useful clarification by project personnel. Examples are where validity checks and noise filtering on input data are performed (externally or internally to the monitor); how outdated sensor inputs are handled; and where and under what conditions various data items should be reinitialized (e.g. should one good reading reset the fault counter to zero?).

4. *Design constraints/rationales.* The tables which trace Cassini's generic formal model of the fault monitor to DS-1's requirements for the monitors

Information for each Data Element	*Example of a Data Entry*
Data Item in Formal Specification	persist_limit
Data Type in Formal Specification	nat
Explanation	Fault must persist for a specified duration
Cassini Design Rationale/Assumptions	Transient faults should be ignored
Data Item in DS-1 Transaction Monitor	S.Persistence
Data Item in DS-1 Threshold Monitor	S.Persistence
DS-1 Build in which Implemented, and/or Rationale for Inclusion/Exclusion	Build x

Figure 4: Reuse of Formal Model for Requirements Validation of Data

Information for each Function	*Example of a Function Entry*
Function Name in Formal Specification	cond_persists
Cassini Design Rationale/Assumptions	Ignores transient fault condition
Function in DS-1 Transaction Monitor	S.Count >= S.Persistence
Function in DS-1 Theshold Monitor	S.Count > S.Persistence
DS-1 Build in which Implemented and/or Rationale for Inclusion/Exclusion	Build y; note that difference in trigger mechanism reflects individual requirements of the two monitors on DS-1

Figure 5: Reuse of Formal Model for Requirements Validation of Functions

document both the rationale and assumptions made by Cassini in adopting their design, and the reason for deviations of DS-1's requirements from that baseline. For example, on Cassini there was a design decision to allow monitors to remain enabled but to disable their output under some circumstances. The table notes that this distinction has no meaning for DS-1, where the design is constrained to handle monitors as subsystem function calls.

Initially an effort was made to trace each element in the OMT diagrams to the DS-1 monitors. This was useful for better understanding the DS-1 monitors, especially with regard to requirements allocation. However, the subsequent tracing of the PVS elements to the DS-1 monitors subsumed this earlier work. Moreover, the imprecision and repetition in the OMT diagrams was removed by focusing on the PVS specifications. The tracing of the OMT diagrams to the DS-1 monitors is thus omitted here.

6 Discussion

The discussion up to this point has primarily described the reuse of the formally specified generic model from the first project on the requirements validation of the second project's fault monitors. This effort at reuse was also instructive for validating the formal model itself and for pointing toward possible areas of future work.

6.1 Evaluating the formal model

There were data items in DS-1's monitors that did not appear in the generic design model, namely dual upper and lower thresholds for detecting fault conditions. On Cassini only one threshold was tested in any single monitor–e.g., overpressure and underpressure would be tested in separate monitors since different responses would be triggered. However, the deviation of the DS-1 Threshold Monitor from this pattern was a reminder that any extension of this generic design monitor to other applications should also handle a design decision to test dual thresholds in a single monitor.

Similarly, although the value of the fault persistence counter in Cassini never exceeds the value of the persistence limit, in DS-1's Transaction Monitor the counter can increase beyond this limit. Again, any further generalization of the generic design monitor should take this possibility into account. Most interesting is the inclusion in the DS-1 monitors of a confidence level, a number against which the number of successful or non-failed iterations of the monitor is compared. Too few successful iterations lead the monitor to report that it knows too little to accurately report the situation, i.e., the status is "unknown" [16].

Several of these mismatches between the second project's requirements and the model could be resolved by revisions or extensions to the formal model. Such work is currently underway by others in anticipation of reuse on subsequent projects.

Other mismatches are not readily resolvable due to the very different architectures on the two projects. For example, the second project's transaction monitors must report all changes, good or bad, rather than just faults, due to the increased on-board state modeling required by the remote agents. The formal model did a better job of validating the requirements for the Threshold Monitor on DS-1 than the Transaction Monitor on DS-1, since the Threshold Monitor's requirements were closer to the behavior of the Cassini monitors. This suggests that a set of generic models of fault monitors, rather than the single monolithic model developed for Cassini, may be needed if the generic model is to be reused for design (rather than requirements) validation.

Despite these limitations, the reuse of the formal model performed well in providing an early reasonableness check on the completeness and the correctness of the requirements for the second project's monitors. At the fairly high level of abstraction in the model, a high degree of correspondence between the required behaviors or responsibilities of the two project's monitors, and between the data they need to do their jobs, was evident. Required functionality currently deferred to later builds in the second project became evident and could be made explicit in the tables.

Architectural differences, which will lead to very different implementations of the fault monitors on the two spacecraft are, for the most part, masked at the model's high level of abstraction. A more detailed formal model might be less appropriate for reuse in requirements validation of the second project since it might be more likely to contain architectural dependencies.

Of the seven lemmas that were formally specified and proved in PVS for the Cassini generic design model, five involved the validity of the input data used for the control decision of whether to request a recovery response. All five of the properties specified in these lemmas (e.g., "The valid data from a sensor is within the range low-filter to high-filter") are required behavior of the lower-level "reflex" or fault detection behavior incorporated into each component in DS-1. One of the remaining two lemmas ("A response is requested by a monitor only if the detected fault is not a transient fault") describes required behavior of the Threshold and Transaction Monitors. The other lemma ("If a fault is not detected by a monitor, then the monitor doesn't request a response") describes required behavior of the Threshold Monitor but not of the Transaction Monitor. This is because the Transaction Monitor reports any change in status, including the first successful transaction after a string of failed transactions.

6.2 Related Work

Recent work in design patterns contains much in common with the process of defining and reusing the generic model described in this paper. Both approaches emphasize the specification of "the core of a solution to a recurring problem" [10]. However, the use of the generic model differs in two significant ways from the use of design patterns.

First, the reuse of the generic model provided a mechanism for requirements validation rather than a design mechanism. The formally specified and validated components of the generic model were traced to the components of the second project to check for gaps in functionality, robustness, and environmental assumptions. The generic model thus provided a reasonableness check on requirements rather than a design pattern for the new project's software.

Secondly, design patterns are often tightly linked to an architectural model. In contrast, one of the interesting features of the reuse of the generic model is that the second project had a significantly different architecture (remote agents) from the first project (centralized control). This is discussed in more detail in Section 4.

6.3 Future Work

Future work on generic monitors may involve identifying and specifying one or more multimission fault protection monitor for use in future spacecraft development. Since monitoring software similar to that on the spacecraft is part of many other safety-critical control systems, the monitor is also being investigated as a possible design pattern [19]. Possible benefits of such an effort include:

- Reducing design complexity. The specification of a generic design supports functional abstraction by identifying shared properties. It supports data abstraction by identifying the common objects and classes, and operations on them. By organizing similiar objects into classes and similar classes into superclasses, a generic design can help uncover underlying similarities and promote generalization and inheritance of shared attributes. In general, a common design specification keeps the internal logic of the individual modules as simple and general-purpose as possible.

- Encouraging gradual design refinement. Use of a generic design may encourage hierarchical design development (successive refinement). For example, some monitors vote on whether a fault exists, but the voting strategy (2 of 3, etc.) varies among the monitors. Design updates often change the voting strategy in a particular monitor. With a generic design the details of the different voting strategies can be cleanly deferred to the detailed design stage of each monitor.

- Tracking and evaluating design changes. The existence of a model and formal specification may allow more rapid response to proposed design changes by keeping the program structure evident. This might help avoid "design recovery" problems in the maintenance of existing software.

- Reducing test time by reducing coupling and increasing cohesion. General-purpose designs keeps the interfaces simple and the interdependence among modules minimal. The attention paid to abstraction creates more tightly bound modules with a clear sequence of tasks.

Work is underway to investigate whether a similar strategy for requirements reuse in a software product family offers these same advantages.

7 Conclusion

Formally modeling and analyzing the design of a generic fault protection monitor articulated the many commonalities among the data and functions of the eighteen Cassini software monitors. Having a formally specified design of a general-purpose fault protection monitor then allowed the occasional design deviations of individual monitors from the general pattern to be readily flagged for further analysis. This was useful because the discrepancies (1) may be design errors, (2) may need additional documentation of their design rationale (to preserve project awareness of their uniqueness), and (3) may require special attention during testing (since erroneous implementation of these exceptions from the pattern is easy).

The formally specified design model provided a baseline against which to measure the completeness of the requirements for similar fault-monitoring software on DS-1. Tracing data items and functions in the PVS design model to the requirements for the monitors in the second project helped (1) validate requirements in the current build, (2) identify requirements for future builds, (3) clarify the allocation of requirements among software components, and (4) document possible design rationales and constraints.

Acknowledgments

The author thanks Martin Feather, Gary Leavens, Judy Crow, and Jack Callahan for their insightful comments on an earlier draft of this paper, and Sarah Gavit, John Kelly, and Nicolas Rouquette for their assistance and suggestions. The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Reference herein to any specific commercial product, process, or service by tradename, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

References

[1] Y. Ampo and R. Lutz, "Evaluation of Software Safety Analysis Using Formal Methods", *Workshop for Foundation of Software Engineering (FOSE)*, Hamana-Ko, Japan, Dec, 1995.

[2] R. H. Bourdeau and B. H. C. Cheng, "A Formal Semantics for Object Model Diagrams," *IEEE Transactions on Software Engineering*, 21(10): pp. 799-821, October, 1995.

[3] *Cassini Orbiter Functional Requirements Book, System Fault Protection Algorithms*, CAS-3-331, Jet Propulsion Laboratory, June 7, 1995.

[4] *Cassini System Fault Protection Final Design Review*, Jet Propulsion Laboratory, Pasadena, CA, June, 1995.

[5] D. Dvorak, N. Rouquette, Q. Vu, "Monitors: How To Design, Build, Test," JPL internal posting, August, 1996.

[6] L. Fesq and D. Bernard, "DS-1 Fault Protection," in *New Millenium Interim Design Concurrence DS1 Autonomy/FSW,"* June, 1996.

[7] L. Fesq, A. Aljabri, C. Anderson, R. Connerton, R. Doyle, M. Hoffman, and G. Man, "Spacecraft Autonomy in the New Millenium," *Proceedings of the 19th AAS Guidance and Control Conference, Advances in the Astronautical Sciences*, ed. R. D. Culp, Breckinridge, CO, February, 1996.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[9] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std 610.12-1990, IEEE, New York, 1990.

[10] N. Islam and M. Devarakonda, "An Essential Design Pattern for Fault-Tolerant Distributed State Sharing," *CACM, Special Issue on Software Patterns*, 39(10): pp. 65-71, October, 1996.

[11] R. Lutz, "Design Analysis of Cassini Fault-Protection Montiors Using Formal Methods," JPL Document D-13431, May 1, 1996.

[12] R. Lutz and Y. Ampo, "Experience Report: Using Formal Methods for Requirements Analysis of Critical Spacecraft Software," *Proceedings of the 19th Annual Software Engineering Workshop*, NASA Goddard Space Flight Center, pp. 231-248, Greenbelt, MD, December, 1994.

[13] R. Lutz and R. Woodhouse, "Requirements Analysis Using Forward and Backward Search," *Annals of Software Engineering, Special Volume on Requirements Engineering*, forthcoming, 1997.

[14] "Mode Identification, Reconfiguration, and Monitoring: Problem Statement ," JPL internal posting.

[15] N. Rouquette, JPL internal posting.

[16] N. Rouquette, "R2S3 Monitors Design Review," July, 1996, JPL internal document.

[17] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[18] N. Shankar, S. Owre, and J. M. Rushby, *The PVS Specification and Verification System*, SRI, March, 1993.

[19] A. Shiflet, "Draft: MonitorReport Pattern," JPL internal document, July 24, 1996.

[20] Software Productivity Consortium, *Reuse-Driven Software Processes Guidebook*, SPC-92019-CMC, v. 02.00.03, November 1993.

Applying the SCR Requirements Method to a Simple Autopilot*

Ramesh Bharadwaj and Constance Heitmeyer
Center for High Assurance Computer Systems (Code 5546)
Naval Research Laboratory
Washington, DC 20375
{ramesh,heitmeyer}@itd.nrl.navy.mil
http://www.itd.navy.mil/ITD/5540/personnel/heitmeyer.html

Abstract

Although formal methods for developing computer systems have been available for more than a decade, few have had significant impact in practice. A major barrier to their use is that developers find formal methods difficult to understand and apply. One exception is a formal method called SCR for specifying computer system requirements which, due to its easy-to-use tabular notation and demonstrated scalability, has achieved some success in industry.

To demonstrate and evaluate the SCR method and tools, we recently used SCR to specify the requirements of a simplified mode control panel for the Boeing 737 autopilot. This paper presents the SCR requirements specification of the autopilot, outlines the process we used to create the SCR specification from a prose description, and discusses the problems and questions that arose in developing the specification. Formalizing and analyzing the requirements specification in SCR uncovered a number of problems with the original prose description, such as incorrect assumptions about the environment, incompleteness, and inconsistency. The paper also introduces a new tabular format we found useful in understanding and analyzing the required behavior of the autopilot. Finally, the paper compares the SCR approach to requirements with that of Butler [5], who uses the PVS language and prover [14] to represent and analyze the autopilot requirements.

1 Introduction

Although formal methods for developing computer systems have been available for more than a decade, few of these methods have had significant impact in the development of practical systems. A major impediment to the use of formal methods in industrial software development is the widespread view that the methods are impractical. Not only do developers regard most formal methods as difficult to understand and apply; in addition, they have serious doubts about the scalability and cost-effectiveness of the methods.

A promising approach to overcoming these problems is to hide the logic-based notation associated with most formal methods and to adopt a notation, such as a graphical or tabular notation, that developers find easy to use. Specifications in the more "user-friendly" notation can be translated automatically to a form more amenable to formal analysis. In addition, the formal method should be supported by powerful, easy-to-use tools. To the extent feasible, the tools should detect software errors automatically and provide easy-to-understand feedback useful in tracing the cause of an error.

By providing a "user-friendly" tabular notation with demonstrated scalability, a formal method called SCR for specifying the requirements of computer systems has already achieved some success in practice. Since the publication more than 15 years ago of the requirements specification for the A-7 aircraft's Operational Flight Program (OFP) [12, 1], many industrial organizations, including Rockwell-Collins, Lockheed, Grumman, and Ontario Hydro, have used SCR to specify requirements. To support the SCR method, we have recently developed a formal state machine model to define the SCR semantics [9, 11] and a set of integrated software tools to support validation and verification of SCR requirements specifications [8, 10, 4]. The tools include an *editor* for creating and modifying a requirements specification, a *simulator* for symbolically executing the specification, a *consistency checker* which checks the specification for well-formedness (e.g., syntax and type correctness, no missing cases or unwanted nondeterminism), and a *verifier* based on model checking for analyzing the specification for application properties.

*This work was supported by the Office of Naval Research.

To demonstrate and evaluate the SCR method and tools, we recently used SCR to specify the requirements of a simplified mode control panel for the Boeing 737 autopilot based on a description in a report by Butler [5]. Butler initially presents an incomplete prose description of the autopilot, and then adds prose to clarify the description. He also represents the required behavior in the PVS language [14] and verifies certain properties of the model using the PVS prover. This paper outlines the process we used to create the SCR requirements specification of the mode control panel, presents the SCR specification, and discusses the problems and questions that arose in developing the specification. Formulating the requirements specification in SCR exposed a number of problems with the prose description of the requirements, such as missing initial values, missing type definitions, missing units of measurement, lack of specificity, incorrect requirement, and several instances of inconsistency. The paper also introduces a new tabular format we found useful in understanding and analyzing the behavior of the autopilot. Finally, the paper compares the SCR approach to requirements with Butler's PVS approach.

2 The SCR Method: Background

2.1 SCR and Other Approaches

A recent article by Shaw [16] presents and discusses a number of different "specifications" of an automobile cruise control system. Each is constructed to satisfy different objectives. For example, Atlee and Gannon use a language based on logic to model the required behavior of a cruise control system [3] and a model checker to detect violations of selected properties. Below, we refer to their logic-based description as an *abstract model.*

The abstract model in [3] differs from an SCR specification in an important respect—namely, in the specific information it contains about the required behavior. Because its purpose is verification, the abstract model omits many details. For example, it does not describe the system outputs. Omitting this information in an abstract model is appropriate because the properties analyzed in [3] are independent of the system outputs and because a model useful in verification should only include information needed to reason about selected properties. Eliminating irrelevant information is especially important in verification. Without dramatic reductions in the size of the state space to be analyzed, model checking is usually infeasible. Moreover, the elimination of irrelevant facts is also beneficial in mechanical theorem proving where the model to be analyzed should only include those facts needed to establish the properties of interest.

In contrast to the abstract model of the system described in [3], the SCR requirements specification is a repository for all information that developers will need to construct the system software. Hence, it is necessarily more detailed and less abstract than a model useful in verification. An advantage of the SCR approach to requirements is that it not only provides detailed guidance on exactly what information belongs in a requirements document, but in addition provides a conceptual model of the system to be developed as well as special language constructs for representing the system requirements. This detailed guidance, system model, and language constructs specialized for requirements specification are lacking in more generic languages such as Statecharts [7] and PVS which, unlike SCR, are not customized for requirements specification.

2.2 The SCR Model

In the SCR approach, the system requirements are specified as a set of relations that the system must maintain between quantities of interest in its environment. In SCR, a requirements specification provides a "black box" description of the required behavior as two relations, REQ and NAT, from *monitored variables*, representing environmental quantities that the system monitors, to *controlled variables*, representing environmental quantities that the system controls [15]. NAT describes the natural constraints on the system behavior, such as constraints imposed by physical laws and the system environment. REQ describes the relation the system must maintain between the environmental quantities represented by the monitored and controlled variables. In SCR, these relations are specified concisely using a tabular notation.

To provide a precise and detailed semantics for the SCR method, the SCR model represents a system as a finite state automaton and describes the monitored and controlled variables and other constructs that make up an SCR specification in terms of that automaton [11, 9]. To concisely describe the required relation between the monitored and controlled variables, the model uses four constructs—modes, terms, conditions, and events. A *mode class* is a partitioning of the system states. Each equivalence class in the partition is called a *system mode* (or simply *mode*). A *term* is any function of monitored variables, modes, or other terms. A *condition* is a predicate defined on a system state. An *event* occurs when the value of any system variable changes (a system variable is a monitored or controlled variable, a mode class, or a term). The notation "`@T(c) WHEN d`" denotes a *conditioned event*, defined as

$$\texttt{@T(c) WHEN d} \overset{\text{def}}{=} \neg c \wedge c' \wedge d,$$

where the unprimed conditions c and d are evaluated in the "old" state, and the primed condition c' is evaluated in the "new" state. Informally, this denotes the event "predicate c becomes *true* in the new state when predicate d holds in the old state". The notation "`@F(c)`" denotes the event `@T(NOT c)`. During the operation of the system, the environment changes a monitored variable, causing an *input event*. In response, the system updates terms and mode classes and changes controlled variables.

3 Developing the SCR Requirements

Figure 1 illustrates the simplified mode control panel for the Boeing 737 as described in [5]. The autopilot monitors the aircraft's altitude (ALT), flight path angle (FPA) and calibrated air speed (CAS) and controls three displays which, depending on the mode, show either the current or desired value of the aircraft's altitude, its flight path angle, and its airspeed. The pilot enters (i.e., preselects) a new value into a display by using one of three knobs next to the displays and engages or disengages the autopilot by pressing one of four buttons at the top of the panel. Appendix A contains a prose description of the system adapted from [5]. The reader should note that the prose presented by Butler in [5] was intended as an example and is therefore (intentionally) incomplete. In the prose presented in this paper, we have (to the best of our knowledge) eliminated all intended incompleteness. Also, the variable names have been changed slightly to conform to the naming conventions of SCR specifications.

3.1 Environmental Variables

In the autopilot specification, we use the prefix "m" to indicate the names of monitored variables. The *type* of a monitored variable indicates the range of values that may be assigned to the variable. The autopilot system monitors the current altitude (represented by monitored variable **mALTcurrent**), the current flight path angle (**mFPAcurrent**), and the current calibrated air speed (**mCAScurrent**). Each of these monitored variables is of type integer. We assume that the autopilot measures the altitude **mALTcurrent** in **feet**, the flight path angle **mFPAcurrent** in **degrees**, and the calibrated air speed **mCAScurrent** in **feet per second**. The monitored variables **mALTsw**, **mATTsw**, **mCASsw**, and **mFPAsw** represent the positions of the four buttons, and each is either **on** or **off**. Finally, the monitored variables **mALTdesired**, **mCASdesired**, and **mFPAdesired** represent the values indicated by the three knobs and range over the integers.

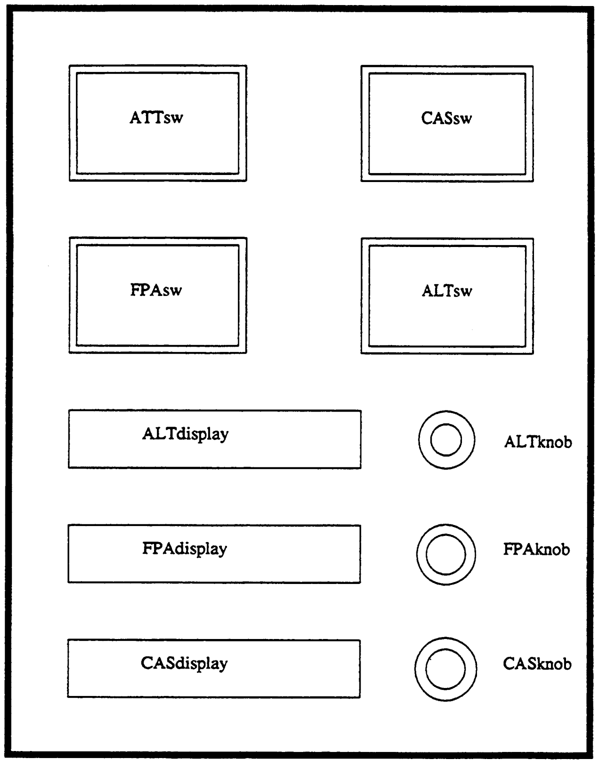


Figure 1: Mode Control Panel

The controlled variables are assigned names with the prefix "c". Just as for monitored variables, each controlled variable has an assigned type. As in [5], we only model the mode control panel, omitting commands sent to the flight control computer. We represent the three controlled quantities of the mode control panel as **cALTdisplay**, **cFPAdisplay**, and **cCASdisplay** and assume each is of type integer. We further assume that **cALTdisplay** displays the altitude in **feet**, **cFPAdisplay** displays the flight path angle in **degrees** and **cCASdisplay** displays the calibrated air speed in **feet per second**.

3.2 System Modes

The SCR specification includes a single mode class **mcStatus** containing modes in the set {**ALTmode**, **ATTmode**, **FPAarmed**, **FPAunarmed**}. When the system is in FPA mode and the altitude engage mode is "armed", we say the system is in mode `FPAarmed`. When the system is in the "normal" flight path angle selected mode, we say the system is in mode `FPAunarmed`. Thus, the system is in `FPAmode` when `mcStatus` is either `FPAarmed` or `FPAunarmed`. Because the system can be in the calibrated air speed mode independently of whether it is in `ALTmode`, `ATTmode`, or `FPAmode`, we exclude the calibrated air speed mode from `mcStatus` and use a term to describe whether the system is in this mode (see below).

3.3 Terms

Terms are assigned names with the prefix "`t`". The autopilot specification contains five terms, each of type boolean. The terms `tALTpresel`, `tCASpresel`, and `tFPApresel` indicate whether the pilot has preselected the altitude, the calibrated air speed, or the flight path angle using one of the three knobs. The term `tNear` denotes when the difference between the desired altitude and the current altitude is less than or equal to 1200 feet, i.e., `mALTdesired` − `mALTcurrent` $\leq$ 1200. Finally, the term `tCASmode` indicates whether the system is in the calibrated air speed mode.

3.4 Relation REQ

The relation REQ is specified by a set of tables, one for each controlled variable, term, and mode class.

Mode Transition Table. The *mode transition table* in Figure 2 specifies the behavior of the mode class `mcStatus`. In the table, the expression `CHANGED(x)` denotes the event "variable `x` has changed value". The table defines all events that change the value of the mode class `mcStatus`. For example, the fourth row of the table states: "If `mcStatus` is `ALTmode`, and `mATTsw` is switched `on`, or the setting of knob `mALTdesired` is changed, then `mcStatus` changes to `ATTmode`." An assumption is that events omitted from the table do not change the value of the mode class. For example, when the system is in `ALTmode`, pressing the button labeled "ALTsw" (that is, the occurrence of the input event `@T(mALTsw=on)`) does not change the value of `mcStatus`.

Each row in the mode transition table in Figure 2 corresponds to certain parts of the prose description in Appendix A. We describe this correspondence below by associating each row of the table with the number of a paragraph in the prose description in Appendix A. In some cases, two rows of the table are derived from the same paragraph; for example, rows R3 and R5 are both derived from paragraph 1.

R1 *The pilot engages a mode by pressing the corresponding button on the panel* (paragraph 1), i.e., pressing `ALTsw` engages `ALTmode`. *However, the altitude must be preselected before* `ALTsw` *is pressed* (paragraph 4). *If the pilot dials an altitude that is more than* 1200 *feet above* `ALTcurrent` *and then presses* `ALTsw`, *then* `ALTmode` *will not directly engage* (paragraph 3).

R2 *If the pilot dials into* `ALTdesired` *an altitude that is more than* 1200 *feet above* `ALTcurrent` *and then presses* `ALTsw`, *then* `ALTmode` *will not directly engage. Instead, the altitude engage mode will change to "armed" and* `FPAmode` *is engaged* (paragraph 3).

R3 *The pilot engages a mode by pressing the corresponding button on the panel* (paragraph 1), i.e., by pressing `FPAsw` the pilot engages `FPAunarmed`.

R4 *The pilot engages a mode by pressing the corresponding button on the panel* (paragraph 1), i.e., pressing `ATTsw` should engage `ATTmode`, **OR** *if the pilot dials in a new altitude while* `ALTmode` *is engaged, then* `ALTmode` *is disengaged and* `ATTmode` *is engaged* (paragraph 7).

R5 Same as row R3.

R6 Combination of scenarios for rows (4) and (3) above.

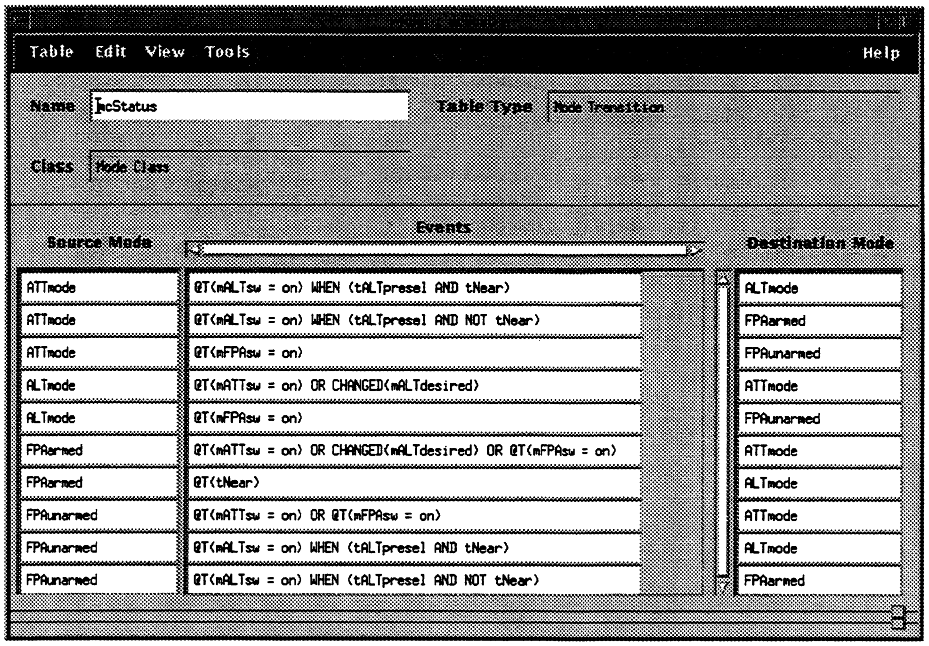
R7 `FPAmode` *will remain engaged until the aircraft is within* 1200 *feet of* `ALTcurrent`, *then* `ALTmode` *is automatically engaged* (paragraph 3).

R8 *The pilot engages a mode by pressing the corresponding button on the panel* (paragraph 1), i.e., by pressing `mATTsw` the system enters `ATTmode`, **OR** `FPAsw` *toggles on and off every time it is pressed* (paragraph 5).

R9 Same as row R1.

R10 Same as row R2.

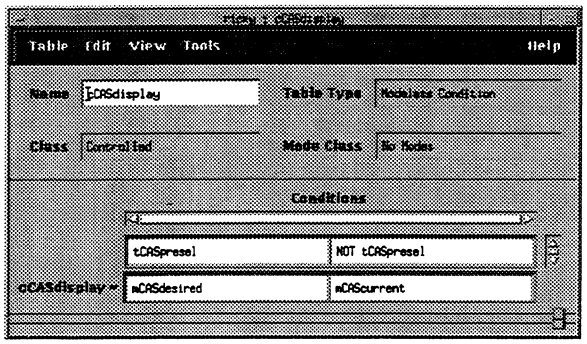
Term and Controlled Variable Tables. Each of the three terms, `tALTpresel`, `tCASpresel`, and `tFPApresel`, is *true* when the corresponding display, `cALTdisplay`, `cCASdisplay`, or `cFPAdisplay`, shows the "preselected" value and is *false* when the corresponding display shows the "current" value.



Name: mcStatus Table Type: Mode Transition
Class: Mode Class

Source Mode	Events	Destination Mode
ATTmode	@T(mALTsw = on) WHEN (tALTpresel AND tNear)	ALTmode
ATTmode	@T(mALTsw = on) WHEN (tALTpresel AND NOT tNear)	FPAarmed
ATTmode	@T(mFPAsw = on)	FPAunarmed
ALTmode	@T(mATTsw = on) OR CHANGED(mALTdesired)	ATTmode
ALTmode	@T(mFPAsw = on)	FPAunarmed
FPAarmed	@T(mATTsw = on) OR CHANGED(mALTdesired) OR @T(mFPAsw = on)	ATTmode
FPAarmed	@T(tNear)	ALTmode
FPAunarmed	@T(mATTsw = on) OR @T(mFPAsw = on)	ATTmode
FPAunarmed	@T(mALTsw = on) WHEN (tALTpresel AND tNear)	ALTmode
FPAunarmed	@T(mALTsw = on) WHEN (tALTpresel AND NOT tNear)	FPAarmed

Figure 2: Mode Transition Table for **mcStatus**



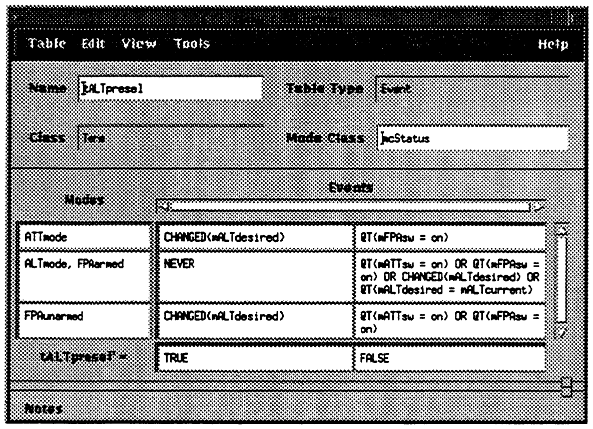
Name: cCASdisplay Table Type: Modeless Condition
Class: Controlled Mode Class: No Modes

	Conditions	
	tCASpresel	NOT tCASpresel
cCASdisplay =	mCASdesired	mCAScurrent

Figure 3: Condition Table for **cCASdisplay**

Figure 3 is a *condition table* which specifies the behavior of the display **cCASdisplay**. This table states: "If **tCASpresel** is *true*, then **cCASdisplay** has the value **mCASdesired**; otherwise, it has the value **mCAScurrent**". The behavior of displays **cALTdisplay** and **cFPAdisplay** are specified similarly (see Appendix B).

The event table in Figure 4 specifies the behavior of **tALTpresel**. Like mode transition tables, event tables make explicit only those events that cause the variable defined by the table to change. For example, the first entry in the first row states: "If **mcStatus** is **ATTmode** and **mALTdesired** changes value, then **tALTpresel** becomes *true.*" The entry "**NEVER**" in an event table means that no event can cause the variable defined by the table to assume



Name: tALTpresel Table Type: Event
Class: Term Mode Class: mcStatus

Modes	Events	
ATTmode	CHANGED(mALTdesired)	@T(mFPAsw = on)
ALTmode, FPAarmed	NEVER	@T(mATTsw = on) OR @T(mFPAsw = on) OR CHANGED(mALTdesired) OR @T(mALTdesired = mALTcurrent)
FPAunarmed	CHANGED(mALTdesired)	@T(mATTsw = on) OR @T(mFPAsw = on)
tALTpresel' =	TRUE	FALSE

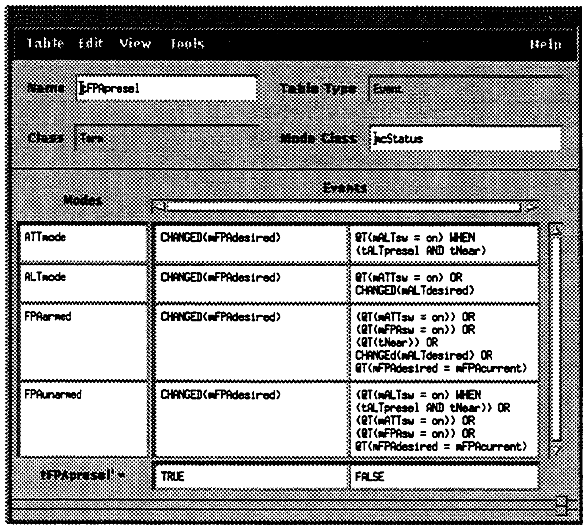
Notes

Figure 4: Event Table for **tALTpresel**

the value in the same column as the entry; thus, the entry "**NEVER**" in row 2 of the table means, "When **mcStatus** is **ALTmode** or **FPAarmed**, then no event can cause **tALTpresel** to become *true.*" Figure 5 shows the event table for **tFPApresel**.

4 Questions and Issues

Developing the SCR specification raised a number of questions about the required behavior of the mode control panel described by the prose in [5]. In a few cases (noted below), the problems were corrected in Butler's PVS formulation. These questions arose because applying the SCR method exposes instances of incompleteness and inconsistency



Name: tFPApresel
Table Type: Event
Class: Term
Mode Class: mcStatus

Modes	Events	
ATTmode	CHANGED(mFPAdesired)	@T(mALTsw = on) WHEN (tALTpresel AND tNear)
ALTmode	CHANGED(mFPAdesired)	@T(mATTsw = on) OR CHANGED(mALTdesired)
FPAarmed	CHANGED(mFPAdesired)	(@T(mATTsw = on)) OR (@T(mFPAsw = on)) OR (@T(tNear)) OR CHANGEd(mALTdesired) OR @T(mFPAdesired = mFPAcurrent)
FPAunarmed	CHANGED(mFPAdesired)	(@T(mALTsw = on) WHEN (tALTpresel AND tNear)) OR (@T(mATTsw = on)) OR (@T(mFPAsw = on)) OR @T(mFPAdesired = mFPAcurrent)
tFPApresel' =	TRUE	FALSE

Figure 5: Event Table for **tFPApresel**

in the specification. This section presents these problems and describes how we resolved them. In resolving each problem, we made an educated guess about the actual requirement. Appendix B contains our revised SCR specification. For this specification to be acceptable, however, our decisions would need to be reviewed by system engineers with expert knowledge of the Boeing 737 autopilot.

4.1 Incompleteness

Developing the SCR specification exposed numerous instances of incompleteness in the prose description. First, the prose provides no information about the types, ranges, and units of measurement of the monitored variables that represent the current and desired altitude, (**mALTcurrent** and **mALTdesired**), the current and desired flight path angle (**mFPAcurrent** and **mFPAdesired**), and the current and desired calibrated air speed (**mCAScurrent** and **mCASdesired**). In addition, the ranges and units of measurement of the three controlled variables **cALTdisplay**, **cFPAdisplay**, and **cCASdisplay** are also omitted. This information is also missing in Butler's PVS model due to its abstract level. As noted above, the SCR specification represents these quantities as integers. In the final specification, even more precise information about the types would be required. For example, what are the minimum and maximum values of each monitored quantity? Are negative integers acceptable?

In addition, the prose description does not indicate the initial state of the autopilot. Butler's PVS description, however, does provide this information. Our SCR specification states that in the initial state **mcStatus** is **ATTmode**, **tCASmode = false**, the desired and current altitude are 0, etc. Appendix B shows the initial values and types of the monitored and controlled variables, terms, and mode class in the SCR specification. (In a specification produced by our toolset, the variable and mode class dictionaries, omitted here due to space limitations, would contain this information.)

4.2 Lack of Specificity

Paragraph 5 of the prose description states: "**FPAsw** *toggles on and off every time it is pressed.*" Paragraph 1 states: "*One of the three modes* **ATTmode**, **FPAmode**, *or* **ALTmode** *should be engaged at all times.*" From these two sentences one can infer that if **FPAmode** is toggled off by **FPAsw**, then one of **ATTmode** or **ALTmode** is engaged; *but not which one.* In our specification as well as Butler's, the decision is to engage **ATTmode**.

4.3 Wrong Interpretation

Paragraph 3 of the prose description states: "*If the pilot dials an altitude that is more than* 1200 *feet above* **ALTcurrent** *and then presses* **ALTsw**, *then* **ALTmode** *will not directly engage. Instead, the altitude engage mode will change to* armed *and FPAmode is engaged.*" This sentence is either incorrect, or is misinterpreted by Butler—the PVS model in [5] sets the altitude engage mode to *armed* also in the case where the pilot dials a value for desired altitude that is more than 1200 feet *below* **ALTcurrent** and then presses **ALTsw**.

4.4 Incorrect Requirement

Paragraph 6 of the prose description states: "*Whenever a mode other than* **CASmode** *is engaged, all other preselected displays should return to current.*" However, consider the scenario where **CASmode** is engaged, **CASdisplay** shows the preselected value, and the pilot engages **ATTmode**. Clearly, returning **CASdisplay** to show the current value would be wrong in this situation since **CASmode** remains engaged. Therefore, the above sentence should read instead, "*Whenever a mode other than* **CASmode** *is engaged,* **ALTdisplay** *and* **FPAdisplay**, *if preselected, should return to current.*" The PVS model does the right thing for this scenario; however, Butler does not point out the error in the prose.

4.5 Inconsistent Requirement

Paragraph 6 of the prose description states: "*Whenever a mode other than* **CASmode** *is engaged,* ALTdisplay *[...], if preselected, should return to current.*" For the scenario of paragraph 7 "*If the pilot dials in a new altitude while* ALTmode *is engaged or the altitude engage mode is "armed", then* ALTmode *is disengaged and* ATTmode *is engaged.*" This suggests that tALTpresel is set to *false* (because ATTmode is engaged). On the other hand, the sentence "*However, the pilot can enter a new value into a display by dialing in the value using the knob next to the display*" of paragraph 2 suggests that tALTpresel is set to *true* for this scenario (because the pilot has dialed-in a new altitude). We resolved this inconsistency by setting tALTpresel to *false*. Butler does not point out this inconsistency in the prose. Unlike the SCR specification, his PVS model resolves the inconsistency by preselecting the display.

5 Consistency Checking, Simulation

After creating the requirements specification, we used our automated consistency checker [8, 9] to check for proper syntax, type correctness, missing cases, nondeterminism, and other application-independent properties. Then, we used our simulator to symbolically execute the requirements specification [10] to ensure that the specification captures (what we assume is) the "customers' intent". For the autopilot specification, our consistency checker detected three instances of inconsistent requirements. Whereas we detected the inconsistency described in Section 4.5 by inspection, we overlooked the following three cases of inconsistency. Butler's PVS model, being too abstract, failed to detect any of these problems.

1. Consider the two sentences in paragraph 2 of the prose description: (1) *However, the pilot can enter a new value into* [FPAdisplay] *by dialing in the value using the knob* [FPAdesired] and (2) *Once the target value is achieved* [...], *the display reverts to showing the "current" value.* These sentences are inconsistent in the situation where the pilot enters a new value into FPAdisplay that is the same as FPAcurrent. In this situation, should the display show the dialed-in value or the current value?

   Butler[1] answers this question as follows: *The phrase ... "will this affect the preselected value (i.e., change it to current)" is difficult to interpret. I assume you meant "will this affect the status of the corresponding display (i.e., change it to current)". Interestingly in this case the status distinction is an artifact of the formalism because the target and current are the same value. So the "status" is merely a matter of choice/taste.*

   While we agree with Butler that the status distinction does not affect the *current* value of the display, we note that it does affect the *future* values displayed. For instance, suppose FPAcurrent proceeds to diverge from FPAdesired immediately following the above scenario. The sentence marked (1) specifies FPAdisplay to continue to show FPAdesired, whereas the sentence marked (2) specifies that FPAdisplay should track the "current" value. We resolved this issue by assuming that sentence (2) takes precedence over sentence (1). The table defining tFPApresel in Appendix B reflects this decision.

2. A similar scenario may be constructed for the calibrated air speed display. We resolved the issue in the same way as above. The table defining tCASpresel in Appendix B reflects this decision.

3. The first sentence of paragraph 7 states: *If the pilot dials in a new altitude while* [...] *the altitude engage mode is "armed", then* ALTmode *is disengaged and* ATTmode *is engaged.* However, dialing in a new altitude in mode FPAarmed can cause tNear to simultaneously become *true*, which leads to inconsistency. We ignore the event @T(tNear) in this situation. The revised specification for mcStatus in Appendix B reflects this decision.

6 Application Properties

After applying the consistency checker and the simulator, we wanted to check the requirements specification for critical application properties, such as safety properties. Verification may be carried out using an interactive theorem prover such as PVS [14, 5], or by using "lightweight" analysis tools such as model checkers. The SCR toolset supports proof of safety properties of requirements specifications using model checking based on state exploration [4]. The following sentences in paragraph 1 of the prose description are examples of properties of the autopilot mode control panel:

1. Only one of the three modes ALTmode, ATTmode, or FPAmode can be engaged at any time.

[1] Private communication.

2. One of the three modes, **ALTmode**, **ATTmode**, or **FPAmode** should be engaged at all times.

3. Engaging any of the three modes will automatically cause the other two to be disengaged since only one of these three modes can be engaged at a time.

4. The mode **CASmode** can be engaged at the same time as any of the other modes.

The type definition of mode class **mcStatus** is {**ALTmode**, **ATTmode**, **FPAarmed**, **FPAunarmed**}, and by definition, the system is in **FPAmode** if **mcStatus** is **FPAarmed** or **FPAunarmed**. We denote the system being in **CASmode** by the boolean term **tCASmode** whose value is independent of **mcStatus**. By this choice of the domain for mode class **mcStatus**, and the definition of **tCASmode**, the above properties are trivially satisfied (and verified automatically by a type checker).

5. Whenever the altitude engage mode is "armed", **FPAmode** is engaged.

For the SCR autopilot specification, this follows directly from the definition of **FPAmode**, i.e., the system is in **FPAmode** if **mcStatus** is **FPAarmed** or **FPAunarmed**. Therefore, if **mcStatus** is **FPAarmed**, the system is in **FPAmode**.

We used the Spin model checker to verify the requirements specification for two additional properties stated in [5] which are listed below. These properties could not be checked using simple type checking.

P1 When **FPAmode** is disengaged, the FPA display reverts to showing the "current" value.

P2 When **ALTmode** is disengaged, the ALT display reverts to showing the "current" value.

We currently check two classes of properties: *state invariants*, which assert the truth of a predicate formula for all reachable states of a system, and *transition invariants*, which assert the truth of a predicate formula (on two states) for all pairs of consecutive states of a system. Both P1 and P2 are transition invariants. Model checking can be ineffective in practice due to *state explosion*. By their very nature, the number of reachable states of practical systems is usually very large in relation to their logical representation. Several techniques have been proposed in the literature for limiting state explosion. The technique we use is *abstraction*—instead of model checking the whole SCR specification, we model check a smaller, more abstract model. To obtain the abstraction, we exploit the structure of the formula and the structure inherent in all SCR specifications. We use the two correctness preserving reductions of [4] to derive the abstract specification. Finally, we translate the abstract specification to PROMELA, the language of Spin [13], and run Spin on the PROMELA model.

When we initially attempted to check property P1, Spin detected a violation. The counterexample generated by Spin had 4867 states (which translates to 811 SCR "steps"). The shortest counterexample had only 73 states (12 SCR "steps"). By running the counterexample through the simulator of our toolset, we were able to pinpoint the cause of the property violation—a typographical error in the mode transition table for **mcStatus**.

7 A New Tabular Format

In [5], Butler presents two different PVS models of the panel requirements. In his initial model, he identifies the different input events (e.g., changing the ALT button to on, setting the knob labeled ALT to a new value, etc.) and then specifies the required behavior by describing the state changes and the changes in the displays that each input event causes. Thus, his initial model is organized by the input events. His second model defines a set of modes and describes the required behavior in terms of those modes. Butler claims that the latter organization, which is the organization used in SCR specifications, results in "a more complex formulation for this example problem." Moreover, he notes that his initial model is smaller, containing 373 words in contrast to 761 words.

We agree that understanding the overall system behavior can be difficult when that behavior is specified in numerous tables. This problem is overcome to some extent by the dependency graph produced by our toolset (see Appendix B), which shows all the variables in the specification and their dependencies. In initially creating the SCR specification, we developed individual tables for the mode class, the terms, and the three control variables. To enhance our understanding, we introduced a new tabular format that combines several smaller tables into a larger table. This larger table, which specifies the values of several variables each defined by either a mode transition table or an event table, is similar to the "selector table" used in the original A-7 requirements specification[2].

[2]A similar tabular format was also used in Lockheed's SCR specification of the OFP for the C-130J aircraft.

The specification in Appendix B contains two examples of the new tabular format. The first example defines the values of the terms **tCASmode** and **tCASpresel**. The other defines the values of the mode class **mcStatus** and the terms **tALTpresel** and **tFPApresel**. The new tabular format is useful because it collects in a single place all events that change a set of related variables. For example, the table defining mode class **mcStatus** and the terms **tALTpresel** and **tFPApresel** shows that when the system is in mode **FPAarmed**, pressing either **mATTsw** or **mFPAsw** causes the system to enter **ATTmode** and sets both **tALTpresel** and **tFPApresel** to *false*. Although the new table format is useful, the three separate tables defining **mcStatus**, **tALTpresel**, and **tFPApresel** are also useful, but for answering a different set of questions: for example, the table defining the mode class **mcStatus** identifies all events that can change the current mode but does so without the clutter of the extra information in the new table. Further, in a large application, merging tables together into a single table could produce very large tables that would be difficult to understand. As a result, we plan to support both our original tables, which define individual variables, and the new table, which defines two or more variables—and to automatically generate the larger table from selected tables that define individual variables.

In our view, the SCR specification in Appendix B is both easy to understand and concise. The complete specification is contained in two pages. Moreover, the SCR specification has an advantage over Butler's PVS model: it is easier to change. For example, our initial version of the specification defined a term called **tArmed** and only three modes, **FPAmode**, **ALTmode**, and **ATTmode**, in the mode class **mcStatus**. Our revised version removed the term **tArmed** and replaced mode **FPAmode** with two modes **FPAarmed** and **FPAunarmed**, thus producing a specification that was more concise and easier to understand. Making the change was quite straightforward—we simply eliminated the table for **tArmed**, revised the table for **mcStatus**, and modified the tables for **tALTpresel** and **tFPApresel** to describe the required behavior in modes **FPAarmed** and **FPAunarmed**. The tables defining the displays as well as all rows of tables that did not involve **FPAmode** were unchanged.

8 SCR versus PVS

Although PVS was not specifically designed to specify requirements, Butler advocates the use of the PVS language and prover for requirements specification [5]. In his report [5], Butler presents two PVS models of the mode control panel and verifies properties of the model organized by inputs using the PVS prover. As noted in the introduction, different formal models serve different purposes. While the PVS model of the panel allowed Butler to verify certain properties of interest, in our view, PVS is not a good notation for expressing requirements specifications to be used by software developers. This is because PVS, a language based on higher-order logic, produces specifications that are less readable by practitioners than specifications in alternative, more user-friendly languages. Moreover, because PVS is not part of a requirements method but is a general-purpose language designed to specify mathematical models, most PVS models omit (abstract away) information needed in a requirements specification—that is, PVS models are usually incomplete. Many of the questions that arose in our development of the SCR specification emerged because an SCR specification requires information that is lacking in Butler's PVS model.

Below, we compare the SCR approach to requirements specification and verification with the approach used by Butler. Although some of the problems we discuss are intrinsic to PVS, others are the result of decisions Butler made in developing the PVS model.

8.1 What are the requirements?

In the SCR approach, a requirements specification is complete when the specification contains all the information software developers will need to design and implement the software. To accomplish this, the specification must identify the quantities of interest in the system's environment, in particular, the monitored and controlled quantities, and specify the required relation between them. Butler's PVS model does not clearly identify the environmental quantities of interest. Nor does the PVS model clearly delineate monitored and controlled quantities. The result is that one cannot infer from the PVS model the required relationships among these quantities.

The PVS model makes use of *actions* or *events* as undefined (primitive) elements. In SCR, in contrast, the system inputs and outputs are modeled as variables, thus capturing more semantic information about the system behavior. This semantic information can be exploited in analyzing the specification for errors. For example, the PVS model assumes that certain input events are mutually disjoint, which results in the omission of an input event from the model (see Section 8.2). Since the SCR specification explicitly defines the environmental quantities of interest, this incompleteness in the specification was automatically detected during consistency checking.

The requirements specification presented in Appendix B is closer than the PVS model to a "real" requirements specification useful to developers. We note, however, that it is still incomplete in at least three respects. First, the I/O devices (or subsystems) that the autopilot uses to measure and compute the monitored and controlled quantities must be specified. Second, the required timing and accuracy of the system is yet unspecified. Third, the constraints imposed by NAT need to be specified. Once these aspects of the required software behavior are provided, developers would have all the information necessary to design, implement, and test the system.

8.2 What are the constraints?

To be complete, a requirements specification (including ours) should model the constraints that physical laws and the system's environment impose on the environmental quantities. For example, changes in altitude are limited by physical laws (e.g., the laws of gravity) and by the maximum rate at which the Boeing 737 can gain or lose altitude. Inclusion of such constraints in the specification can be used later in software development to do sanity checks on the specification—and the code—and to indicate when some fault has occurred (e.g., a sensor measuring altitude has failed).

Relations NAT and REQ are specified separately in the SCR method (see Section 2.2). The PVS model, on the other hand, does not distinguish relationships that arise from existing physical or other constraints (relation NAT of SCR) and relationships that are to be enforced by the system (relation REQ of SCR). This gap leads to two related problems: *overspecification* and *incompleteness*.

Overspecification. It is useful (and simpler) to ignore impossible situations (i.e., situations ruled out by NAT) in the definition of relation REQ. For example, the PVS model defines the following events:

`alt_reached`	the altitude reaches the preselected value
`alt_gets_near`	the altitude is now near, but not equal to the preselected value

On page 12 of [5], Butler addresses the scenario where the system is in mode `FPAarmed` (i.e., predicate `tNear` is *false*) and the event `alt_reached` occurs without `alt_gets_near` occurring first. This situation is clearly ruled out by relation NAT (if `tNear` is *false*, the current altitude cannot reach the preselected value without `tNear` becoming *true* first). Therefore, when specifying relation REQ, one need not deal with the above scenario.

Incompleteness. The PVS model considers the following events to be mutually disjoint:

`input_alt`	the action of dialing a value using `mALTdesired`
`alt_reached`	the altitude reaches the preselected value
`alt_gets_near`	the altitude is now near, but not equal to the preselected value

But, these events are *not* disjoint. For example, the action of dialing a value (denoted by event `input_alt`) can simultaneously cause either of the other two events to occur. The PVS model fails to consider these cases.

8.3 What is the level of abstraction?

The PVS description of the autopilot may be viewed as an abstract model of the mode control panel. For example, the monitored quantity `ALTcurrent` is denoted abstractly by two boolean variables `alt_reached` and `alt_gets_near`; boolean variable `input_alt` abstractly denotes the pilot dialing in the desired altitude using knob `ALTdesired`; etc. Once the model is constructed, one can use deductive reasoning to check that the model satisfies specified properties of interest, such as the application properties described above. Although such an approach is a good way to detect errors in the system requirements, our claim is that such abstract models must be transformed into a more concrete *requirements specification*, such as the one we present in Appendix B. Without a requirements specification, one cannot determine the monitored and controlled quantities of interest, and the required relationship between them. If the correspondence between the abstract model and the requirements specification is informal (and the required relation REQ is never specified explicitly), developers may misinterpret the requirements.

8.4 Role of tool support

The major strength of PVS is verification: using PVS, a user can analyze a specification for complex properties. In another task, we have used PVS to detect serious errors in the specification of a hybrid, real-time system [2]. However, because PVS was not designed for specifying and analyzing require-

Problem Description	Location	Phase			Reference
		Formulating SCR Spec	Consistency Checking	Model Checking	
Missing initial values	prose	many			4.1
Missing ranges, types, and units of measurement	prose, PVS	many			4.1
Lack of specificity	prose	1			4.2
Incorrect requirement	prose	1			4.4
Inconsistent requirements	prose, PVS	1	3		4.5, 5
Transcription error	SCR			1	6
Wrong interpretation	PVS	1			4.3
Overspecification	PVS	1			8.2
Incompleteness	PVS	1			8.2

Table 1: Detected Problems

ments, it lacks a requirements method. Without such a method, users have little guidance in developing a requirements specification. In contrast, the SCR method has been specifically designed to produce precise and complete requirements specifications. In our experience, tools that support a specific conceptual model and method are more effective than general-purpose tools. If a formal model lacks a strong underlying method, the benefits of automation are likely to be minimal ([6] provides more details). Since the SCR method focuses on a limited class of systems and standardizes the conceptual model, the notation, and the process, significant automated tool support is possible.

9 Analysis

Table 1 summarizes the problems we detected in applying the SCR method to the autopilot specification and identifies the specification in which the problem occurs. Some of the problems listed (missing initial values, lack of specificity, and incorrect prose) were corrected by Butler. All the other problems, except the typographical error, were additional problems detected when we applied the SCR method to Butler's (presumably correct) prose specification. All but four of them were detected in formulating the SCR specification. Of the remaining four problems, the typographical error was detected by model checking and the remaining three cases of inconsistency were detected by our consistency checker.

It is clear from Table 1 that merely specifying a system using the SCR method *without any automated analysis* can expose many problems. It may be argued that some of these problems might eventually be detected by the PVS prover. To do this, however, users must formulate properties that will expose the problems. It has been our experience that formulating *correct* properties for a large requirements specification can be non-trivial. Moreover, in our opinion, the effort needed to analyze the specification with PVS would be significantly greater than the effort involved in formulating the specification in SCR and analyzing it with the SCR tools.

Our experience can be compared to that of Miller[3], who reports that the SCR method helped uncover 18 errors in a specification of an autopilot at Rockwell-Collins. Of the errors, one-third of them were detected during formulation of the SCR specification, one-third by the consistency checker, and one-third during simulation. (Since we used the simulator to a much smaller degree than Miller, we found no significant errors using the simulator.) Our two efforts clearly demonstrate that lightweight methods are highly effective in uncovering errors in requirements specifications. It is also important to note that most of these errors, including those that were detected by formulating the specification in SCR, would have probably gone undetected without appropriate tool support.

10 Conclusions

In this paper, we outlined a process for creating an SCR requirements specification of a simplified mode control panel for the Boeing 737 autopilot, based on the prose description of the system presented in [5]. Developing an SCR specification of the autopilot uncovered a number of problems that were undetected in Butler's formalization of the problem using PVS. While PVS is useful for verifying deep properties of specifications, this study

[3]Private communication.

provides evidence that PVS is not well suited for formulating and analyzing requirements specifications, especially during the initial stages. This is due to a number of factors: the logic-based PVS language which software developers find difficult to apply, the mathematical sophistication and theorem proving skills that developers need to verify properties using the PVS prover, and the lack of a requirements method for PVS.

We envision a process for developing high-quality requirements specifications that combines the SCR technology and a mechanical prover, such as PVS. This process would rely on the light-weight SCR tools during the initial part of the requirements process—specification using a formal yet "user-friendly" notation to capture the requirements, automated consistency checking and model checking to detect violations of simple properties, and simulation to ensure that the specification captures the customers' intent. Once sufficient confidence in the specification is developed, a mechanical proof system, such as PVS, may be used to verify deep properties of the complete requirements specification or, more likely, safety-critical components. While software developers themselves will have the skills needed to apply the light-weight SCR tools, applying heavy-duty theorem proving is likely to require formal methods experts with the requisite mathematical sophistication and theorem proving skills.

Acknowledgments

We gratefully acknowledge Ricky Butler for providing helpful insights and for his prompt answers to all our questions about the autopilot mode control panel. We thank Todd Grimm and Bruce Labaw for implementing support for our verification method in the SCR toolset. We also acknowledge the very helpful comments of our colleagues Myla Archer and Bruce Labaw on earlier drafts of this paper.

References

[1] T. Alspaugh, S. Faulk, K. Britton, R. Parker, D. Parnas, and J. Shore. Software Requirements for the A-7E Aircraft. Technical Report NRL-9194, NRL, Washington DC, 1992.

[2] M. Archer and C. Heitmeyer. Verifying hybrid systems modeled as timed automata: A case study. *Proc. 1997 International Workshop on Hybrid and Real-Time Systems (HART'97)*, Grenoble, France, March 1997.

[3] J. M. Atlee and J. Gannon. State-Based Model Checking of Event-Driven System Requirements. *IEEE Transactions on Software Engineering*, pp 22–40, January 1993.

[4] R. Bharadwaj and C. Heitmeyer. Verifying SCR requirements specifications using state exploration. In *Proc. First ACM SIGPLAN Workshop on Automatic Analysis of Software,* Paris, France, January 14, 1997.

[5] R. W. Butler. An Introduction to Requirements Capture Using PVS: Specification of a Simple Autopilot. NASA Technical Memorandum 110255. NASA Langley Research Center, May 1996.

[6] S. R. Faulk. Software Requirements: A Tutorial. Technical Report *NRL/MR/5546-95-7775*, Naval Research Laboratory, Washington DC, 1995.

[7] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming,* 8(3), 231-274, June 1987.

[8] C. Heitmeyer, B. Labaw, and D. Kiskis. Consistency checking of SCR-style requirements specifications. In *Proc. 1995 Int'l Symposium on Requirements Engg.*, York, England, March 1995.

[9] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated Consistency Checking of Requirements Specifications. *ACM Trans. on Software Engg. and Methodology,* 5(3), 231–261, July 1996.

[10] C. Heitmeyer, J. Kirby, and B. Labaw. Tools for formal specification, verification, and validation of requirements. In *Proc.* 12th *Annual Conference on Computer Assurance*, NIST, Gaithersburg MD, June 1997.

[11] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Tools for analyzing SCR-style requirements specifications: A formal foundation. Technical Report, Naval Research Laboratory, Washington DC, 1997. In preparation.

[12] K. L. Heninger. Specifying software requirements for complex systems: New techniques and their applications. *IEEE Transactions on Software Engineering* SE-6(1), Jan 1980.

[13] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[14] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In 11th *International Conference on Automated Deduction*, LNCS-607, pp 748–752, 1992.

[15] D. L. Parnas and J. Madey. Functional documents for computer systems. *Science of Computer Programming,* 25(1), pp 41–62, Oct 1995.

[16] M. Shaw. Comparing architectural design styles. *IEEE Software*, November 1995.

A Description of the autopilot

1. *The mode-control panel contains four buttons for selecting modes and three displays for dialing in or displaying values, as shown in Figure 1. The system supports the following four modes: attitude control wheel steering (*`ATTmode`*), flight path angle selected (*`FPAmode`*), altitude engage (*`ALTmode`*), and calibrated air speed (*`CASmode`*).*

   Only one of the first three modes can be engaged at any time. The mode `CASmode` *can be engaged at the same time as any of the other modes. The pilot engages a mode by pressing the corresponding button on the panel. One of the three modes,* `ATTmode`, `FPAmode`, *or* `ALTmode` *should be engaged at all times. Engaging any of the first three modes will automatically cause the other two to be disengaged since only one of these three modes can be engaged at a time.*

2. *There are three displays on the panel: altitude (*`ALTdisplay`*), flight path angle (*`FPAdisplay`*), and calibrated air speed (*`CASdisplay`*). The displays usually show the current values of altitude (*`ALTcurrent`*), flight path angle (*`FPAcurrent`*), and air speed (*`CAScurrent`*) of the aircraft. However, the pilot can enter a new value into a display by dialing in the value using the knob next to the display (*`ALTdesired`, `FPAdesired`, *or* `CASdesired`*). This is the target or "pre-selected" value that the pilot wishes the aircraft to attain. For example, if the pilot wishes to climb to* 25,000 *feet, he will dial* 25,000 *(using the knob* `ALTdesired`*) into* `ALTdisplay` *and then press* `ALTsw` *to engage* `ALTmode`*. Once the target value is achieved or the mode is disengaged, the display reverts to showing the "current" value.*

3. *If the pilot dials into* `ALTdesired` *an altitude that is more than* 1,200 *feet above the current altitude (*`ALTcurrent`*) and then presses* `ALTsw`*, then* `ALTmode` *will not directly engage. Instead, the altitude engage mode will change to "armed" and* `FPAmode` *is engaged. The pilot must then dial in, using the knob* `FPAdesired`*, the desired flight-path angle into* `FPAdisplay`*, which will be followed by the flight-control system until the aircraft attains the desired altitude.* `FPAmode` *will remain engaged until the aircraft is within* 1,200 *feet of* `ALTcurrent`*, then* `ALTmode` *is automatically engaged.*

4. `CASdesired` *and* `FPAdesired` *need not be pre-selected before the corresponding modes are engaged—the current values displayed will be used. The pilot can dial-in a different target value after the mode is engaged. However, the altitude must be pre-selected before* `ALTsw` *is pressed. Otherwise, the command is ignored.*

5. `CASsw` *and* `FPAsw` *toggle on and off every time they are pressed. For example, if* `CASsw` *is pressed while the system is already in* `CASmode`*, that mode will be disengaged. However, if* `ATTsw` *is pressed while* `ATTmode` *is already engaged, the command is ignored. Likewise, pressing* `ALTsw` *while the system is already in* `ALTmode` *has no effect.*

6. *Whenever a mode other than* `CASmode` *is engaged, all other pre-selected displays should return to current.*

7. *If the pilot dials in a new altitude while* `ALTmode` *is engaged or the altitude engage mode is "armed", then* `ALTmode` *is disengaged and* `ATTmode` *is engaged. If the altitude engage mode is "armed" then* `FPAmode` *should be disengaged as well.*

B SCR Specification of the autopilot

Monitored Variables:

`mALTcurrent`, `mCAScurrent`, `mFPAcurrent` : Integer **initially** all 0;
`mALTsw`, `mATTsw`, `mCASsw`, `mFPAsw` : {on, `off`} **initially** all `off`;
`mALTdesired`, `mCASdesired`, `mFPAdesired` : Integer **initially** all 0;

Controlled Variables:

`cALTdisplay`, `cCASdisplay`, `cFPAdisplay` : Integer **initially** all 0;

Terms:

`tALTpresel`, `tCASpresel`, `tFPApresel` : Boolean **initially** all `false`;
`tCASmode` : Boolean **initially** `false`;
`tNear` $\overset{\text{def}}{=}$ `mALTdesired` $-$ `mALTcurrent` ≤ 1200;

Mode Class:

`mcStatus` : {`ALTmode`, `ATTmode`, `FPAarmed`, `FPAunarmed`} **initially** `ATTmode`;

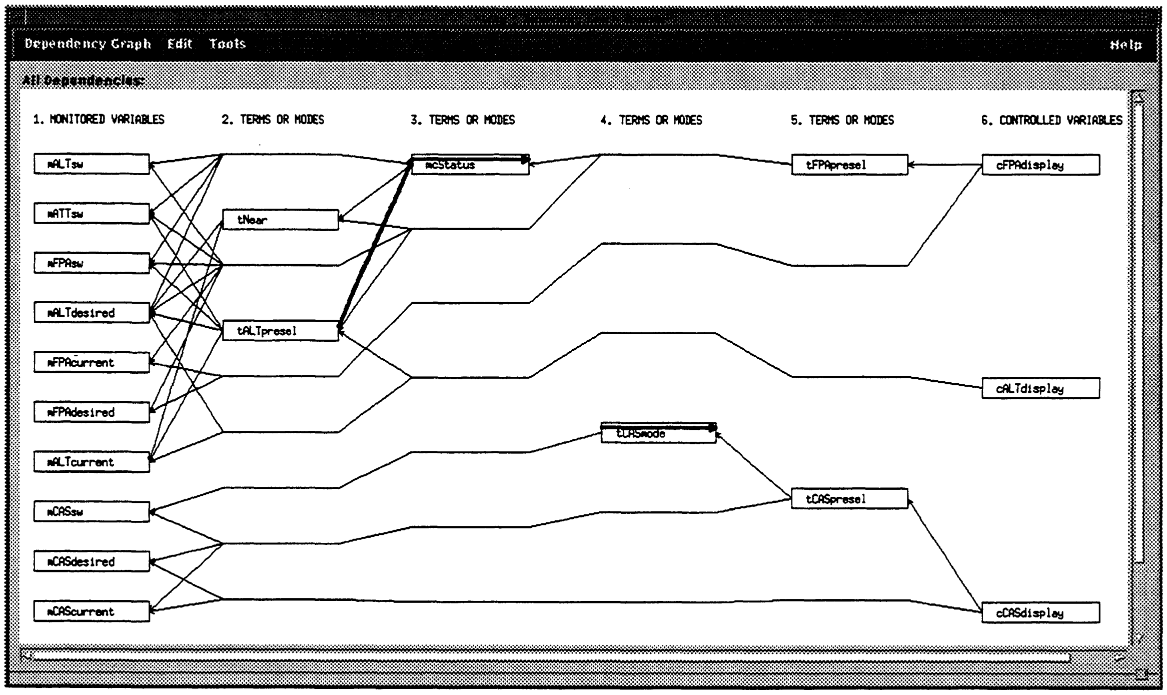


Figure 6: Variable Dependency Graph

Term = tCASmode			
	Events	tCASmode	tCASpresel
NOT tCASmode	@T(mCASsw=on)	true	
	CHANGED(mCASdesired)		true
tCASmode	@T(mCASsw=on)	false	false
	@T(mCASdesired=mCAScurrent)		false
	CHANGED(mCASdesired) AND mCASdesired' ≠ mCAScurrent'		true

Mode Class = mcStatus				
Old Mode	Events	New Mode	tALTpresel	tFPApresel
ATTmode	@T(mALTsw=on) WHEN (tALTpresel AND tNear)	ALTmode		false
	@T(mALTsw=on) WHEN (tALTpresel AND NOT tNear)	FPAarmed		
	@T(mFPAsw=on)	FPAunarmed	false	
	CHANGED(mALTdesired)		true	
	CHANGED(mFPAdesired)			true
ALTmode	@T(mATTsw=on)	ATTmode	false	false
	@T(mFPAsw=on)	FPAunarmed	false	
	CHANGED(mALTdesired)	ATTmode	false	false
	CHANGED(mFPAdesired)			true
	@T(mALTdesired = mALTcurrent)		false	
FPAarmed	@T(mATTsw=on) OR @T(mFPAsw=on)	ATTmode	false	false
	CHANGED(mALTdesired)	ATTmode	false	false
	CHANGED(mFPAdesired) AND mFPAdesired' ≠ mFPAcurrent'			true
	@T(tNear) AND mALTdesired = mALTdesired'	ALTmode		false
	@T(mFPAdesired = mFPAcurrent)			false
FPAunarmed	@T(mALTsw=on) WHEN (tALTpresel AND tNear)	ALTmode		false
	@T(mALTsw=on) WHEN (tALTpresel AND NOT tNear)	FPAarmed		
	@T(mATTsw=on) OR @T(mFPAsw=on)	ATTmode	false	false
	CHANGED(mALTdesired)		true	
	CHANGED(mFPAdesired) AND mFPAdesired' ≠ mFPAcurrent'			true
	@T(mFPAdesired = mFPAcurrent)			false

cALTdisplay =

Conditions	
tALTpresel	NOT tALTpresel
mALTdesired	mALTcurrent

cCASdisplay =

Conditions	
tCASpresel	NOT tCASpresel
mCASdesired	mCAScurrent

cFPAdisplay =

Conditions	
tFPApresel	NOT tFPApresel
mFPAdesired	mFPAcurrent

A Tabular Language for System Design

Steven D. Johnson*

sjohnson@cs.indiana.edu
Computer Science Department
Indiana University
Bloomington Indiana 47405-4101

Abstract

A tabular language for describing synchronous behaviors is developed as a visual representation for formalized design derivation. A sketch of *behavior table* syntax and semantics is given. An example illustrates the kinds of formal manipulations investigated by the research. Evidence is accumulating that tables are perspicuous for specification, design, and verification, but graphical support is essential to their effective use.

1 Introduction and background

The tabular specification language described in this article emerged as a visual representation for interactive system design. We started using tables in a casual way, generating them from the underlying expressions of a formal system for *design derivation* [13]. *Behavior tables* emerged as a bridging notation between control oriented and architecture oriented modes of description.

With better graphical support, we think tables such as these can assume a prominent role in system specification, verification, and synthesis. In our case studies of design derivation, we began using tables to visualize formal transformations on design expressions. Over time, notational features evolved that we have not found in other hardware description languages. Even though we intimately understand the underlying formalism, we believe that the tables are more expressive than the underlying modeling expressions they represent because they offer additional visual structure.

This realization prompted us to consider our tables more seriously as a formal design notation, and we began exploring features that are useful in system design applications [28, 25]. We comtemplated direct realizations in VLSI [20]. Other encouraging influences have been the emerging tools and techniques for using tables in requirements specification, verification, and synthesis. These are reviewed in the next section.

In Sections 3 through 5 we present a syntax and representative semantics for the tables we use. We think of behavior tables as denoting persistent, communicating processes rather than procedures or functions. The substance of the difference is that behavior tables cannot themselves be entries in other behavior tables. Instead, they are composed by connecting their I/O ports. Thus, behavior tables represent typed, synchronous transition systems, which we believe are closer to the intended high-level synthesis models than the "synthesizable subsets" of VHDL and Verilog now in use.

Section 6 illustrates the kind of manipulations we perform in design derivation. The example was constructed manually, but a corresponding derivation was performed using an existing transformation tool. Section 7 reviews additional syntactic features contemplated and topics of further research, including

*This research is supported, in part, by the National Science Foundation under Grants MIP-9208745 and MIP96-10358.

manipulations that we have mathematically formalized but not automated. We believe that the most urgent task of this line research is to develop graphics interface facilities.

An apology about terminology. The term "behavior table" arose spontaneously in our laboratory. In Section 3 we adopt "decision table" from [9] and "action table" from [18] for fragments of our forms. But these fragments are not identical to the previously published objects. Furthermore, there are other objects in the literature with similar names, including "behavior table," that have different, possibly incompatible forms and interpretations. We hope this terminology will stabilize in the future.

2 Related work

The work on decision tables by Hoover, Chen, and others [9, 8] inspired us to think more seriously about the behavior tables developed in our case studies of design derivation. Their *Tablewise* specification tool was developed for avionics software development, but clearly applies to reactive systems in general. In addition to a graphical front end, there are functions for verifying exclusivity and completeness of decision tables and for performing structural analyses to aid in obtaining these properties. Future topics mentioned in [9] include connections to state-machine and statechart based specification. This connection is the focus of our interest.

The *Software Cost Reduction* system of Naval Research Laboratories is also a requirements specification tool set with graphics support and aids for analysis and verification [7]. A formalization in PVS by a group at SRI is based on SCR* constructs and also contains an extensive review of tabular specification notations [19]. Their treatment is a shallow embedding, supported by tabular syntax contained in the PVS surface language. One immediate benefit of this approach is exploitation of the PVS type system, in particular, its management of *type correctness conditions.* Our experience integrating design derivation with PVS verification suggests a somewhat deeper embedding will needed to support reasoning about transformations. One reason for this is that the underlying semantic domain of *streams* is not well founded [14].

Leveson's *Requirements State Machine Language* [16] is based on Harel's state charts [6], but uses *and-or tables* to specify hyper-edges. She echos Hoover's observation that decision tables are readily accepted and used by practicing engineers.

Li and Gupta introduce *timed decision tables* as an HDL [18, 17]. Their results on optimizations exploiting don't care entries are directly applicable to the forms we use in our work. Their work is also evidence of the utility of a tabular specification language for CAD tool development. Behavior tables have been proposed as an interchange format by Gajski, Dutt, et. al. [5, 4]. The intermediate synthesis language BLIV-MV contains a very rich syntax for the tabular specification of multi-valued boolean functions [15]. We find it very encouraging that research in high-level synthesis and formal methods finds common ground in these tabular representations; it reflects new opportunities for synergy between communities that frequently encounter problems with each others' notations.

3 Syntax of behavior tables

Behavior tables are arrays of *terms* over an amalgamated abstract type giving ground syntax for constants and operations and equational laws for reasoning about them. Our examples will involve commonly understood types, but a type system is intended to support conceptual hierarchies, parameterization, and other structuring capabilities. A useful tool must have built-in reasoning for concrete types, but must also have facilities for reasoning about and between user specified type complexes.

The notion of term evaluation used here is standard. The value of a term, t, is written $\sigma[\![t]\!]$, where σ is an *assignment* or association of values to variables. A generic *don't-care* constant is written as '$\natural$'.

A finite extension of propositional logic is assumed—Hoover calls it *finite logic* [8]. Arbitrary collections of enumerated values, or *tokens*, can be formed. These finite sets come with a polymorphic selection operation. A behavior table can be thought of as a parallel composition of selection expressions.

Behavior tables are closed expressions whose terms contain variables from three disjoint sets: I (inputs), S (data state), and C (combinational signals). Fix these sets for the remainder of this section. We will write ISC for $I \cup S \cup C$ and SC for $S \cup C$. We use the term 'register' for an element of S, but this is a euphemism that should be interpreted very abstractly. There is no assumption that these variables denote finite values, nor are tables intended only for register-transfer specification. The form of a behavior table is

Name: *Inputs* $\rightarrow$ *Outputs*	
Conditions	*Registers and Signals*
$\vdots$	$\vdots$
Guard	*Computation Step*
$\vdots$	$\vdots$

Inputs is a list of input variables and *Outputs* is a set of terms over ISC; for simplicity, assume $O \subseteq ISC$. *Conditions* is a set P of predicates over ISC, that is, terms ranging over finite types, such as truth values, token sets, etc. A *guard* is a set of constants indexed by the condition set P: $g = \{c_p\}_{p \in P}$. We say g *holds* for an assignment σ to ISC when, for each $p \in P$, either $c_p = \natural$ or $\sigma[\![p]\!] = c_p$.

A *decision table* $\mathbf{D} = [P, G]$, consists of a condition set and a list of guards. Following [9], we say a decision table is *functional* when G describes a proper partitioning of the possible assignments to ISC. In other words, the guards are "exclusive and exhaustive." A *computation step* or *action* is a set of terms, one for each register and signal: $a = \{t_v\}_{v \in SC}$. An *action table* is an indexed set of actions.

A *behavior table for* $I \rightarrow O$ consists of a decision table, $\mathbf{D}$, with guards $G = \{g_1, \ldots g_n\}$, and an action table indexed by G, $\mathbf{A} = \{t_{v,k} \mid v \in SC \text{ and } g_k \in G\}$.

4 Synchronous semantics

A behavior table $[\mathbf{D}, \mathbf{A}]$ for $O \subseteq ISC$ denotes a relation between infinite input and output sequences. We call these sequences *streams* because in prior work we obtain a semantics by interpreting a table as a (co)recursive system of stream-defining equations [13]. More directly, suppose we are given a set of initial values for the registers, $\{x_s\}_{s \in S}$ and a stream for each input variable in I. Construct a sequence of assignments, $\langle \sigma_0, \sigma_1 \ldots \rangle$ for ISC as follows:

(a) $\sigma_n(i)$ is given for all $i \in I$ and all n.

(b) For each $s \in S$, $\sigma_0(s) = x_s$.

(c) $\sigma_{n+1}(s) = \sigma_n[\![t_{s,k}]\!]$ if guard g_k holds for σ_n.

(d) For each $c \in C$, $\sigma_n(c) = \sigma_n[\![t_{c,k}]\!]$ if guard g_k holds for σ_n.

The stream associated with each $o \in O$ is $\langle \sigma_0(o), \sigma_1(o), \ldots \rangle$. This semantic relation is well defined if there are no circular dependencies among the combinational actions $\{t_{c,k} \mid c \in C,\ g_k \in G\}$. The relation is a function (i.e. deterministic) if decision table $\mathbf{D}$ is functional.

This semantics is at odds with both TDTs and Tablewise (Section 2), but the differences are reconcilable, and are by no means special to tabular notations. We think of behavior tables as denoting persistent, communicating processes rather than procedures to be invoked.

In other words, behavior tables cannot themselves be entries in other behavior tables, but instead are composed by interconnecting their I/O ports. Compositions give rise to hierarchical network descriptions in which the "leaves" are tables. This is closer to the intended high-level synthesis models than the "synthesizable HDL subsets" now in use. For example, Borrione, et.al., have recently proposed *hierarchical finite state machines* (HFSMs) as a common basis for HDL interoperability [1]. The semantic relationship between HFSMs and behavior tables is very close.

Composition is specified by giving a connection map that is faithful to each component's arity. Valid maps must preserve I/O directionality, excluding both combinational cycles and output conflicts. In our function-oriented modeling methodology, such compositions are expressed as recursive systems of equations,

$$\begin{aligned}
&\lambda(U_1,\ldots,U_n).(V_1,\ldots,V_m) \text{ where} \\
(X_{11},\ldots,X_{1q_1}) &= \mathcal{T}_1(W_{11},\ldots,W_{1\ell_1}) \\
&\vdots \\
(X_{p1},\ldots,X_{pq_p}) &= \mathcal{T}_p(W_{p1},\ldots,W_{p\ell_p})
\end{aligned}$$

in which the defined variables X_{ij} are all distinct, each $\mathcal{T}_k$ is the name of a behavior table or other composition, and the outputs V_k and internal connections W_{ij} are all simple variables coming from the set $\{U_i\} \cup \{X_{jk}\}$.

Provided they are well formed, deterministic systems are readily animated in modeling languages that allow recursive stream networks to be expressed [10]. As long as each register has an initial value, the streams are constructed head-first as a fixed-point computation. Translation to event-based simulation languages is also relatively straightforward for systems expressed over concrete types.

A synchronous semantics is simple and suited to the clocked implementation models most high-level synthesizers use. In fact, behavior tables will acquire a range of semantics, depending on their applications, just as HDLs and programming languages do. Even with a variety of interpretations, their inherent structure helps reduce the mathematical bookkeeping that often obscures semantic definitions.

5 Examples of behavior tables

The behavior table shown in Figure 1 describes a process that computes the Fibonacci function: The inputs are control signal go and data input in; the outputs are control signal done* and the data signal v. The '*' is a notational convention for distinguishing combinational signals from state-holding registers. Three other representations in Figure 1 depict various aspects of the design. The labeled transition diagram is keyed to the table's rows; its labels consist of a condition under which the transition is taken, the outputs associated with the transition, and an update to the data state; the same information as a row of the table. The control automaton is represented in the table by the now register. Throughout this paper, we reserve the name now for this purpose. A timing diagram shows the interface abstraction. The textual expression of the algorithm at the lower left of Figure 1 describes is the well-known iterative computation of $\mathit{fib}(n)$, where

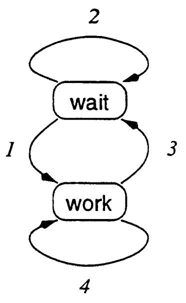
$$\begin{aligned}
\mathit{fib}(0) &= 0 \\
\mathit{fib}(1) &= 1 \\
\mathit{fib}(k+2) &= \mathit{fib}(k) + \mathit{fib}(k+1)
\end{aligned}$$

The behavior table in Figure 2 describes the garbage collector of a list processing computer [2]. It is representative of the tables we work with in our case studies. Its level of specification is more abstract, with two of the registers of type *memory*. An implementation of this table was realized in about 5,000 ACTEL FPGA cells, of which 1,500 4-input MUX elements compute the behavior. A behavior table for the same computer's CPU is about twice as big, when expressed at a level where garbage collection is an abstract operation. A table closer the the register transfer level, as in Figure 2, would be more than ten times larger, but even at that scale the tables are useful in our derivation methodology—and would be even more useful with better display automation.

As these examples may illustrate, behavior tables are not the best representation for understanding the specification of an algorithm. However, they seem (in our experience) to serve well as a bridging notation for simultaneously contemplating control and architecture. Furthermore, studies (e.g. [16]) suggest that complicated control functions are clearer when presented as decision tables. In hardware design, the intuitive sense of control flow is quickly overwhelmed when processes are composed.

6 Table manipulations

Let us explore some basic transformations, starting with the table in Figure 1. As in any derivation, the order of presentation is not necessarily the order in which the transformations were conceived. In practice, backtracking is involved as the architectural goals develop and concrete representations are intro-



	(go, in) → (done*, v)							
	now	go	u=0	now	done*	u	v	w
1	wait	*true*	♮	work	*false*	in	0	1
2	"	*false*	♮	wait	*true*	♮	♮	♮
3	work	♮	*true*	wait	*true*	♮	v	♮
4	"	♮	*false*	work	*false*	u-1	w	v+w

```
**await** $go$;
  $u, v, w$ := $input(in), 0, 1$;
**while** $(u \neq 0)$ do
  $\{v = fib(in_0 - u) \wedge w = fib(in_0 - u + 1)\}$
  $u, v, w := u - 1, w, v + w$;
$output(v)$;
**assert** $done*$
```

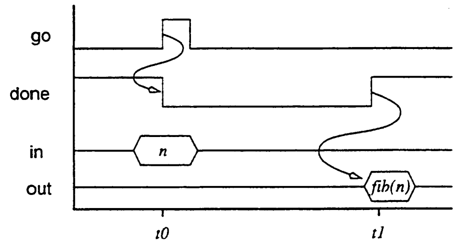


Figure 1: A behavior table and related diagrams

duced. The final derivation is just a residual proof of the design.

This example was carried out and formatted by hand, but Tuna was able to mimic the entire derivation [12] using the DRS mechanized transformation system [3] which operates on recursive systems of functional expressions. That exercise exposed one significant error in the manual derivation.

The now and done* columns suggest an assignment of concrete values 0 to work and 1 to wait. To reduce clutter, let us also assign 1 to true and 0 to false.

(go, in) → (done*, v)							
now	go	u=0	now	done*	u	v	w
1	1	♮	done*	¬go	in	0	1
"	0	♮	"	¬go	♮	♮	♮
0	♮	1	"	u=0	♮	v	♮
"	♮	0	"	u=0	u-1	w	v+w

With these changes, the first and second rows have become identical up to don't-cares, so we can merge them to obtain

(go, in) → (done*, v)							
now	go	u=0	now	done*	u	v	w
1	♮	♮	done*	¬go	in	0	1
0	♮	1	"	u=0	♮	v	♮
"	♮	0	"	u=0	u-1	w	v+w

The predicate go has become irrelevant will be removed. We note in passing that the last two rows could be merged by replacing the term for v with select(u=0,v,w). Behavior tables seem especially useful for this kind of interplay between control and computation—all the more so with the provisions for *indirection* discussed in Section 7.

Next is a scheduling transformation on the third row that puts the arithmetic terms u-1 and v+w into different computation steps. The goal is to assign these operations to a single arithmetic component.

go, in → done*, v						
now	u=0	now	done*	u	v	w
1	♮	done*	¬go	in	0	1
0	1	"	u=0	♮	v	♮
0	0	2	false	u-1	v	w
2	♮	done*	u=0	u	w	v+w

	NOW	RQ	(= U A)	(pointer? H)	(bvec-h? H)	(eq? 'forward (tag d))	(tag H)	(= C 0)	(= C 1)	NOW	OLD:*mem*	NEW:*mem*	H:*cont*	D:*cont*	C:*addr*	U:*cont*	A:*addr*	AK
1	idle	1	♮	♮	♮	♮	♮	♮	♮	driver	OLD	NEW	H	D	C	(rd OLD H)	0	0
2	"	0	♮	♮	♮	♮	♮	♮	♮	idle	OLD	NEW	H	♮	♮	♮	♮	1
3	driver	♮	1	♮	♮	♮	♮	♮	♮	idle	NEW	OLD	0	D	C	U	A	1
4	"	♮	0	♮	♮	♮	♮	♮	♮	nextobj	OLD	NEW	(rd NEW U)	D	C	U	A	0
5	nextobj	♮	♮	1	♮	♮	♮	♮	♮	objtype	OLD	(wt NEW U (cell H A))	H	(rd OLD H)	C	U	A	0
6	"	♮	♮	0	1	♮	♮	♮	♮	driver	OLD	NEW	H	(rd OLD H)	C	(+ U (btow-u (pr-pt H)) (cin 1))	A	0
7	"	♮	♮	0	0	♮	♮	♮	♮	driver	OLD	NEW	H	(rd OLD H)	C	(+ U (const 0) (cin 1))	A	0
8	objtype	♮	♮	♮	♮	1	♮	♮	♮	driver	OLD	(wt NEW U (cell H d))	H	D	C	(+ U (const 0) (cin 1))	A	0
9	"	♮	♮	♮	♮	0	fixed	♮	♮	copy	(wt OLD H (cell forward A))	NEW	(cell H (add1-ptr (pr-pt H)))	D	(add1-2 (fixed-size H))	U	A	0
10	"	♮	♮	♮	♮	0	vec	♮	♮	vec	(wt OLD H (cell forward A))	NEW	(cell H (add1-ptr (pr-pt H)))	D	(btow-c (pr-pt d))	U	A	0
11	"	♮	♮	♮	♮	0	bvec	♮	♮	vec	(wt OLD H (cell forward A))	NEW	(cell H (add1-ptr (pr-pt H)))	D	(btow-c (pr-pt d))	U	A	0
12	vec	♮	♮	♮	♮	♮	♮	1	♮	driver	OLD	(wt NEW A d)	(cell H (add1-ptr (pr-pt H)))	D	C	(+ U (const 0) (cin 1))	(add1-a A)	0
13	"	♮	♮	♮	♮	♮	♮	1	♮	copy	OLD	(wt NEW A d)	(cell H (add1-ptr (pr-pt H)))	(rd OLD H)	C	U	(add1-a A)	0
14	copy	♮	♮	♮	♮	♮	♮	♮	1	driver	OLD	(wt NEW A d)	H	D	C	(+ U (const 0) (cin 1))	(add1-a A)	0
15	"	♮	♮	♮	♮	♮	♮	♮	0	copy	OLD	(wt NEW A d)	(cell H (add1-ptr (pr-pt H)))	(rd OLD H)	(sub1 C)	U	(add1-a A)	0

Figure 2. Behavior Table for a Garbage Collector

The newly created control token, '2', induces a type mismatch with boolean done* in the now column. This is a problem to be resolved by underlying type inference system. In addition to implicit coercions, this transformation's validity depends *both* on the fact that the sequence of two steps preserves the original computation *and* the fact that the surrounding synchronization protocol is preserved. Verifying the latter of these conditions is not automatic, in general. In this case, we are relying on the interface specification of Figure 1, which says that the result is ready only when done* is asserted.

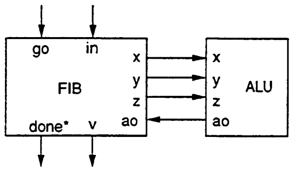
The next table is a simple example of *system factorization*, a decomposition technique that is central to the derivation formalism. As desired, terms u-1 and v+w are allocated to a single combinational arithmetic component, called ALU.

FIB: (go, in, ao) → (done*, v, x*, y*, z*)

now	u=0	now	done*	u	v	w	x*	y*	z*
1	♮	done*	¬go	in	0	1	♮	♮	♮
0	1	"	u=0	♮	v	♮	♮	♮	♮
0	0	2	false	ao	v	w	sub	u	1
2	♮	done*	u=0	u	w	ao	add	v	w

ALU:(x,y,z) → ao*

x	ao*
add	x+y
sub	x-y



A system factorization encapsulates a set of subject terms in a new table and generates residual interface signals [11]. Here, the interface signal x* generates instruction tokens, sub and add, telling ALU which operation to perform. The transformation tool keeps track of the connectivity. In particular, factorizations preserve well-formedness even when one of the factors is entriely combinational, as is ALU in this case.

To finish the example, we make some assignments to the don't-care entries whose ultimate effect is to isolate control. As a second example of system factorization, we decompose into a control process generating an encoded command signal, cmd, to the data path DP, as shown in Figure 3.

7 Directions

We are encouraged by the number of recent papers centering on tabular specification languages. It is usually reported that such tables are a good "engineering notation," and they seem also to be relatively easy to represent formally. If this consensus grows, the most urgent task may be the mundane one of building tools to manipulate table syntax. It is our hope that such graphics tools will be general enough to accommodate the range of applications tables are finding in the design community.

At this stage, we are investigating a number of additions to and variations of behavior table syntax.

7.1 Assertions

Both Tablewise and TDTs (Section 2 have provisions for *assertions* that are not yet in our behavior tables, but which should be included in any graphics support. Tablewise incorporates type-declarative fields that we would defer to a background type system in our applications. TDTs contain time parameters used in optimization. In Tablewise the primary intent seems to be the verification of invariant properties, but assertions could also be used to state constraints, measures, and, for that matter, computational actions.

The algorithmic specification in Figure 1 contains a loop invariant that might be attached to row 4 of the corresponding behavior table. It is interesting to contemplate how subsequent manipulations, especially decompositions, might affect this assertion. Since system design verification often involves liveness, safety, and other eventualities, assertions would likely take the form of temporal logic predicates on the current state (i.e., row).

7.2 Decompositions

Our notion of system factorization, involving both data abstraction and interface specification, has yet to be fully reflected in our behavior tables. The underlying ideas are more general than the example shows, having evolved over several years of research.

CTL: (go, u) → (done*, cmd*, x*)

now	u=0	now	done*	cmd*	x*
1	♮	done*	¬go	0	♮
0	1	"	u=0	1	sub
0	0	2	false	1	sub
2	♮	done*	"	2	add

DP: (cmd, in, ao)
→ (u, v, y*, z*)

cmd	u	v	w	y*	z*
0	in	0	1	♮	♮
1	ao	v	w	v	w
2	u	w	ao	u	1

ALU:(x,y,z) → ao*

x	ao*
add	y+z
sub	y-z

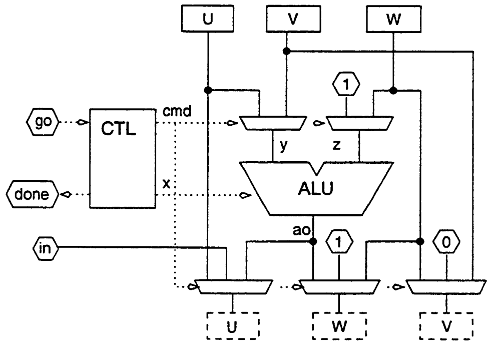


Figure 3: Final decomposition of the example

As an illustration, let us consider a two-phase ALU that takes operands sequentially.

ALU2:(op,in) → (phase, out*)

phase	op	phase	hold	out*
1	♮	2	in	♮
2	add	1	in	hold + in
2	sub	1	in	hold - in

We wish to use ALU2 to determine a factorization of the table below, a variant of the specification in Figure 1. A *reset* input, r, has been added to illustrate why tables are sometimes better than algorithmic languages at expressing features of global control flow. Row 1 of the table says that whenever r is asserted the FSM moves to state A.

FIB:(r, go, in) → (d*, v)

	r	now	go	u=0	now	d*	u	v	w
1	0	♮	♮	♮	A	0	♮	♮	♮
2	1	A	1	♮	B	0	in	0	1
3	"	"	0	♮	A	1	♮	♮	♮
4	"	B	♮	T	A	1	♮	v	♮
5	"	"	♮	F	B	0	u-1	w	v+w

A targeted factorization has to instantiate the protocol ALU2 expects, synchronizing with its phases, and presenting the operands sequentially. First, we serialize the arithmetic as before:

FIB:(r, go, in) → (d*, v)

	r	now	go	u=0	now	d*	u	v	w
1	0	♮	♮	♮	A	0	♮	♮	♮
2	1	A	1	♮	B	0	in	0	1
3	"	"	0	♮	A	1	♮	♮	♮
4	"	B	♮	T	A	1	♮	v	♮
5a	"	"	♮	F	C	0	u	w	v+w
5b	"	C	♮	♮	B	0	u-1	v	w

To decompose according to ALU2, add a wait state to get into phase and graft the addition and subtraction paths into the control flow. One possible factorization is shown in Figure 4. We have have investigated several constructive approaches to this family of decompositions [21, 23, 30, 29, 24]. We cannot yet claim a universal construction, but we do have transformations general enough to handle many common interface specifications [22]. Performing simultaneous decompositions—which is necessary for practical application of formal derivations—remains a topic of research.

FIB:(r, go, in) ⟶ (d*, v)

r	now	go	u=0	now	d*	u	v	w
0	♮	♮	♮	A	0	♮	♮	♮
1	A	1	♮	B	0	in	0	1
"	"	0	♮	A	1	♮	♮	♮
"	B	♮	T	A	1	♮	v	♮
"	"	♮	F	B	0	u-1	w	v+w

÷

ALU2:(op,in) ⟶ (phase, out*)

phase	op	phase	hold	out*
1	♮	2	in	♮
2	add	1	in	hold + in
2	sub	1	in	hold - in

⇒

FIB:(r, go, in, ph, alu) ⟶ (d*, v, op, opd)

	r	now	go	u=0	ph	now	d*	u	v	w	op	opd
1	0	♮	♮	♮	♮	A	0	♮	♮	♮	♮	♮
2a	1	A	1	♮	2	B	0	in	0	1	♮	♮
2b	"	"	1	♮	1	D	0	in	0	1	♮	♮
3	"	"	0	♮	♮	A	1	♮	♮	♮	♮	♮
6	"	D	♮	♮	♮	B	0	u	v	w	♮	♮
4	"	B	♮	T	♮	A	1	♮	v	♮	♮	♮
5aa	"	"	♮	F	♮	C	0	u	v	w	♮	v
5ab	"	C	♮	♮	♮	E	0	u	w	alu	add	w
5ba	"	E	♮	♮	♮	F	0	1	v	w	♮	u
5bb	"	F	♮	♮	♮	B	0	alu	v	w	sub	u

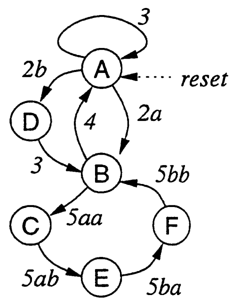


Figure 4: The factorization developed in Section 7.2

7.3 Other syntax

The tabular languages we have seen exhibit a great variety of abbreviation techniques. Typically, these serve to condense decision conditions by specifying sets of values. *BLIF-MV*, for example, allows subrange, subset, and complementation expressions in its table specifications [15].

We have added syntax for *bounded indirection* which often significantly reduces the size of action tables [25] and is novel for hardware description languages. If r is a signal or register, then #r denotes a token referring to r. If register s contains such a token, then @s denotes the entity to which s refers; that is,

$$@s \equiv \mathsf{case}\ s\ \mathsf{of} \ldots \#r{:}\,r \ldots$$

In [28] we show how a behavior table describing a bus reduces to one row when indirection is used to specify sources and destinations. In [27] we explore control indirection.

The *behavior FSMs* proposed by Takach, Wolf, and Leeser contain constructs to constrain events to occur within sets of transitions. BFSM are intended to serve as specification models for high-level synthesis. Any implementation of a BFSM refines this constraint by assigning particular transition to each event, subject to the constraints [26]. As the example of Section 6 suggests, one row of an action table can represent a number of transitions at a finer time scale. However, BFSMs are more expressive than behavior tables in the sense that unstructured programs are more expressive than structured ones. This suggests to us that action tables may need provisions to relax their output behaviors.

References

[1] Dominiquie Borrione, Fredrik Vestman, and Hakim Bouamama. An approch to Verilog-VHDL interoperability for synchronous designs. In *Procedings of the IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'97)*. To appear.

[2] Robert G. Burger. The scheme machine. Technical Report 413, Indiana University, Computer Science Department, August 1994. 59 pages.

[3] Derivation Systems, Inc., Carlsbad, California. *DRS: Derivational Reasoning System*, 1.2.1 edition, December 1995. Contact `drs@derivation.com`.

[4] Nikil D. Dutt and Daniel D. Gajski. Exel: A language for interactive behavioral synthesis. In John A. Darringer and Franz J. Rammig, editors, *Computer Hardware Description Languages and their Applications*, pages 3–18, 1989.

[5] D. Gajski, N. Dutt, A. Wu, and S. Lin. *High-level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[6] D. Harel. Statecharts: a visual formalism for complex systems. *The Science of Computer Programming*, 8:231–274, 1987.

[7] Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. SCR*: a toolset for specifying and analyzing requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS '95)*, pages 109–122, 1995.

[8] D. N. Hoover and Zewei Chen. Tbell: A mathematical tool for analyzing decision tables. Contractor Report 195027, NASA/LRC, Hampton VA 23681-0001, November 1994.

[9] D. N. Hoover, David Guaspari, and Polar Humenn. Applications of formal methods to specification and safety of avionics software. Contractor Report 4723, NASA/LRC, Hampton VA 23681-0001, November 1994.

[10] Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations*. MIT Press, Cambridge, 1984.

[11] Steven D. Johnson. Manipulating logical organization with system factorizations. In Leeser and Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, volume 408 of *LNCS*, pages 260–281. Springer, July 1989.

[12] Steven D. Johnson. A tabular language for system design, Appendix A. Technical Report 485, Indiana University Computer Science Department, June 1997.

[13] Steven D. Johnson and Bhaskar Bose. A system for mechanized digital design derivation. In *IFIP and ACM/SIGDA International Workshop on Formal Methods in VLSI Design*, 1991. Available as Indiana University Computer Science Department Technical Report No. 323 (December 1990).

[14] Steven D. Johnson and Paul S. Miner. integrated reasoning support in system design: design derivation and theorem proving. In *IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'97)*, 1997. To appear.

[15] Yuji Kukimoto. BLIF-MV. `http://www-cad.eecs.berkeley.edu/~/vis/`.

[16] Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specifiation for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.

[17] Jian Li. Timed decision tables: A behavioral model for embedded system specification and optimization. Technical Report UIUCDCS-R-96-1971, Univeristy of Illinois Department of Computer Science, 1304 West Springfield Ave, Urbana IL 61801, 1996.

[18] Jian Li and Rejash K. Gupta. HDL optimization using timed decision tables. In *33rd ACM/IEEE Design Automation Conference*, 1996.

[19] Sam Owre, John Rushby, and N. Shankar. Integration in PVS: Tables, types, and model checking. In *Tools and Algorithms for the Construction and Analysis of Systems TACAS '97*, Springer *LNCS*. To appear.

[20] K. Rath, I. Celis, and R. M. Wehrmeister. RTBA: A generic bit-sliced bus architecture for

datapath synthesis. Technical Report 321, Department of Computer Science, Indiana University, December 1990.

[21] Kamlesh Rath. *Sequential System Decomposition.* PhD thesis, Computer Science Department, Indiana University, USA, 1995. Technical Report No. 457, 90 pages.

[22] Kamlesh Rath, Bhaskar Bose, and Steven D. Johnson. Derivation of a DRAM memory interface by sequential decomposition. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 438–441. IEEE, October 1993.

[23] Kamlesh Rath, Venkatesh Choppella, and Steven D. Johnson. Decomposition of sequential behavior using interface specification and complementation. *VLSI Design Journal*, 3(3-4):347–358, 1995.

[24] Kamlesh Rath and Steven D. Johnson. Toward a basis for protocol specification and process decomposition. In D. Agnew, L. Claesen, and R. Camposano, editors, *Proceedings of IFIP Conference on Hardware Description Languages and their Applications*, pages 157–174. Elsevier, April 1993.

[25] Kamlesh Rath, M. Esen Tuna, and Steven D. Johnson. Behavior tables: A basis for system representation and transformational system synthesis. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 736–740. IEEE, November 1993.

[26] Andreś Takach, Wayne Wolf, and Miriam Leeser. An automaton model for scheduling constraints in synchronous machines. *IEEE Transactions on Computers*, 44(1):1–12, January 1995.

[27] M. Esen Tuna, Steven D. Johnson, and Bob Burger. Continuations in hardware-software codesign. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 264–269. IEEE, October 1994.

[28] M. Esen Tuna, Kamlesh Rath, and Steven D. Johnson. Specification and synthesis of bounded indirection. In *Proceedings of the Fifth Great Lakes Symposium on VLSI*, pages 86–89. IEEE, March 1995.

[29] Zheng Zhu and Steven D. Johnson. Automatic synthesis of sequential synchronizations. In D. Agnew, L. Claesen, and R. Camposano, editors, *Proceedings of IFIP Conference on Hardware Description Languages and their Applications*, pages 285–301. Elsevier, April 1993.

[30] Zheng Zhu and Steven D. Johnson. Capturing synchronization specifications for sequential compositions. In *Proceedings of the 1994 IEEE International Conference on Computer Design (ICCD 94)*, pages 117–121. IEEE, October 1994.

Verifying Communication Related Safety Constraints in RSML Specifications*

Mats P.E. Heimdahl
University of Minnesota, Institute of Technology
Department of Computer Science, 4-192 EE/CS Bldg.
Minneapolis, MN 55455
heimdahl@cs.umn.edu

Abstract

Languages based on hierarchical finite state machines, such as, Statecharts, SCR (Software Cost Reduction), and the Requirements State Machine Language (RSML), are suitable for specification of software for embedded systems. The languages are relatively easy to use, allow automated verification of properties such as completeness and consistency, and support execution and dynamic evaluation of the specifications. However, the support to rigorously specify and analyze the communication between physically distinct components in a system is currently not well supported in any of the approaches.

We know that the interfaces between the software and the embedding environment are a major source of costly errors. For example, Lutz reported that 20% - 35% of the safety related errors discovered during integration and system testing of two spacecraft were related to the interfaces between the software and the embedding hardware.

In this paper we introduce a formal approach to the specification of system level inter-component communication and show how this formalism can be used to prove safety constraints and a limited notion of liveness constraints. The interface definitions and the constraints are translated to PVS (Prototype Verification System) proof obligations, and the proofs of compliance with the constraints are performed in the PVS domain. To demonstrate the feasibility of the approach we have implemented a prototype tool and used the tool to prove some desirable properties of the inter-component communication of an avionics system.

*This work has been partially supported by NSF grants CCR-9624324 and CCR-9615088, and University of Minnesota Grant in Aid of Research 1003-521-5965.

1 Introduction

Writing and validating software requirements for embedded systems present particularly difficult problems, for example, the software is required to interact with a variety of analog and digital components in its environment, the software must be able to detect and recover from error conditions in the environment, and the software is often subject to rigorous safety and performance constraints.

Languages based on hierarchical finite state machines, for instance, Statecharts [9, 10, 11], SCR (Software Cost Reduction) [15, 17], and the Requirements State Machine Language (RSML) [20], are powerful modeling languages suitable for specification of software for these types of systems. The languages are relatively easy to use, allow automated verification of properties such as completeness and consistency, and support some execution and dynamic evaluation of the specifications [7, 8, 11, 14, 15, 17, 20]. However, the support for rigorous specification and analysis of the communication between physically distinct components in a system is currently not well supported in any of the approaches.

We know that the interfaces between the software and the embedding environment are a major source of costly errors. For example, Lutz reported that 20% - 35% of the safety related errors discovered during integration and system testing of two spacecraft were related to the interfaces between the software and the embedding hardware [22, 23]. The problems often involve, for example, misunderstandings about how the hardware operates, failure to detect and respond to inputs outside the normal operating regime, and failure to prevent undesirable outputs from being generated [18, 19, 22, 23, 24]. Thus, it is imperative that a requirements specification for an embedded software system rigorously captures the interfaces and the communication between the software and its embedding environment.

The specification of high-level inter-component communication poses many interesting challenges. Since the components may represent software, digital hardware components, or analog hardware components, the interface descriptions must be able to capture, for example, expected arrival rate, expected minimum and maximum values, and the unit the value represents. Furthermore, a specification language should support encapsulation of communication specific aspects of a model. Encapsulation in this context serves two purposes, (1) it helps shield the rest of the model from the inevitable changes in the embedding system and (2) it captures communication related properties in one place for ease of inspection and ease of analysis.

In this paper, we introduce a formal approach to the specification of system level inter-component communication and show how this formalism can be used to prove certain types of safety and liveness constraints related to the communication. The formalism is influenced by our previous experiences with using RSML to capture the requirements of a large avionics system [20]. We encapsulate information about the physical properties of the communication in an interface specification, for example, properties such as timing assumptions, and encapsulate the definition of how incoming and outgoing messages are treated in communication handlers associated with the interfaces, for instance, under which conditions we are allowed to generate a specific output.

The formality of the specification allows us to automatically verify certain types of communication related constraints. Since many of the properties related to communication are encapsulated in the interface definitions, we attempt to prove that the constraints are satisfied by only considering the information in the interface definitions. This approach greatly simplifies the proofs of safety constraints and simple liveness constraints in complex state-based models.

The interface definitions and the constraints are translated to PVS [6, 25, 26] proof obligations and the proofs of compliance with the constraints are performed in the PVS domain. To demonstrate the feasibility of the approach we have implemented a prototype tool and used the tool to prove some desirable properties of the inter-component communication of an avionics system called TCAS II. TCAS II is an airborne, collision-avoidance system required on all commercial aircraft carrying more than 30 passengers through U.S. airspace.

1.1 Previous and Related Work

Both SCR and RSML allow a specification to be automatically checked for consistency (there are no conflicting requirements) and a notion of completeness (all possible input scenarios are handled by the specification) [14, 15]. Although this analysis has been used to detect problems in large specifications, the analysis procedures are rather limited. For example, the procedures used to determine if large Boolean formulas are mutually exclusive do not interpret the terms in the formula and, thus, may generate large numbers of spurious error reports. To overcome such problems, many approaches to static analysis enforce restrictions on the modeling language to facilitate accurate analysis, such as restricting variables to Boolean [4, 5] or enumerated types [16]. Recently, we have addressed this problem in RSML by using PVS for the analysis and in that way enable interpretation of all terms in the Boolean expressions, including terms using linear and non-linear arithmetic. In our work we want to avoid enforcing unnecessary confining restrictions on our modeling language. RSML was successfully used to model a complex avionics system [20] and our experience from that effort convinced us that enforcing restrictions, such as the restrictions mentioned above, will limit the usability of the modeling language to a point where practitioners will find the language too restrictive to use.

Nevertheless, the approaches discussed above only address the issues of completeness and consistency in state-based models. To assure that more complex properties hold in a specification, more powerful analysis approaches are needed. Several groups have attempted to apply model checking techniques to RSML and SCR.

Atlee *et al.* reported success with applying the SMV model checker to SCR style specifications [3, 2]. In a similar effort, Anderson *et al.* have applied the SMV model checker to a part of the TCAS II RSML specification [1]. Both efforts, however, either limit the analysis to models only containing enumerated input and output variables or limit the analysis to subsets of the model not involving, for example, arithmetic expressions.

In this investigation we take a different approach to analysis. We encapsulate some behavior related to communication in the input and output interface definitions, define assertions we are interested in verifying in an easy to read notation, and automatically generate proof obligations for PVS. If the specification is properly structured, the assertions can be proven by only considering the communication specifications and we can disregard the rest of the model.

This approach to enforcement of constraints is not unique to our work. Leveson *et al.* discussed the use of a *safety kernel* to enforce safety policies in safety critical systems [21]. The kernel centralizes the enforcement of safety policies and detection/recovery of safety violation in one small easily verifiable component. The notion of safety kernels responsible for policy enforcement has been further discussed by Wika and Knight [29, 30]. These approaches, however, mainly address the design and implementation stages of development and do not discuss verification of safety properties in the high-level specification stage.

Rushby has provided a detailed and formal discussion of the suitability of a kernel approach for safety enforcement [27]. He concluded that a kernel architecture is most suited to enforce negative properties, for example, that certain actions are not taken in some situations. A kernel approach is more limited when it comes to enforcing positive properties, for instance, that an action is always performed under certain conditions. The work presented in this paper uses the communication definitions as a simple kernel architecture and is largely inspired by Rushby's discussion.

Our approach is built on our previous work using PVS to prove consistency and completeness in RSML specifications [13]. In that project we developed a tool that automatically generates PVS proof obligations for the completeness and consistency criteria in RSML. In the project described in this paper, we have extended our tool to accept assertions related to the communication with the external world and to generate proof obligations to verify that the assertions hold in the RSML specification. These assertions can express both safety constraints, for example, that a certain output can never be produced under certain conditions, and simple liveness properties, such as, that a certain input will always lead to a shutdown event.

The paper is organized as follows. The next section provides an overview of RSML. Section 3 presents our approach to system-level inter-component communication and Section 4 outlines how safety and simple liveness constraints can be expressed and verified. Section 5 illustrates how we generate PVS proof obligations from the RSML specification and from the constraints. Section 6 provides a summary and conclusions.

2 RSML Overview

RSML was developed as a requirements specification language specifically for embedded systems. The language is based on hierarchical finite state machines and is in many ways similar to David Harel's Statecharts; for example, RSML supports parallelism, hierarchies, and guarded transitions borrowed from Statecharts (Figure 1) [9, 12].

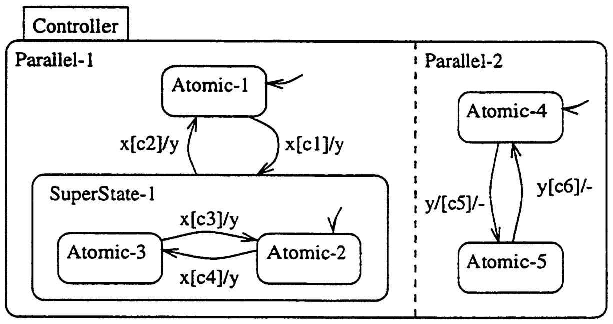


Figure 1: An example of an hierarchical state machine.

One of the main design goals of RSML was readability and understandability by non computer professionals such as, in our case, pilots, air frame manufacturers, and FAA representatives. During the TCAS project, we discovered that the guarding conditions required to accurately capture the requirements were often complex. The prepositional logic notation traditionally used to define these conditions did not scale well to complex expressions and, thus, quickly became unreadable. To overcome this problem, we introduced a tabular notation for defining the guarding conditions (Figure 2). We call these tables AND/OR tables. The tables are read columnwise and were found to be very readable. To further increase the readability of the specification, we introduced many other syntactic conventions in RSML; for example, we allow expressions used in the predicates to be defined as mathematical functions (e.g., Other-Tracked-Relative-Alt-Rate$_{\text{f-246}}$), and familiar and frequently used conditions to be defined as macros (e.g., 100-Ft-Crossing$_{\text{m-195}}$)[1]. A macro is simply a named AND/OR table defined elsewhere in the document. A detailed description of the full notation can be found in [20].

[1]The subscript is used to indicate the type of an identifier (f for functions, m for macros, and v for variables) and gives the page in the TCAS II requirements document where the identifier is defined.

Transition(s): $\boxed{\text{ESL-4}} \longrightarrow \boxed{\text{ESL-2}}$

Location: Own-Aircraft ▷ Effective-SL$_{\text{s-30}}$

Trigger Event: Auto-SL-Evaluated-Event$_{\text{e-279}}$

Condition:

		OR		
AND	Auto-SL$_{\text{s-30}}$ **in state** ASL-2	T	T	·
	Auto-SL$_{\text{s-30}}$ **in one of** {ASL-2,ASL-3,ASL-4,ASL-5,ASL-6,ASL-7}	·	·	T
	Lowest-Ground$_{\text{f-241}}$ = 2	·	·	T
	Mode-Selector = **one of** {TA/RA,TA-Only,3,4,5,6,7}	T	·	T
	Mode-Selector$_{\text{v-34}}$ = TA-Only	·	T	·

Output Action: Effective-SL-Evaluated-Event$_{\text{e-279}}$

Figure 2: A transition definition from TCAS II with the guarding condition expressed as an AND/OR table.

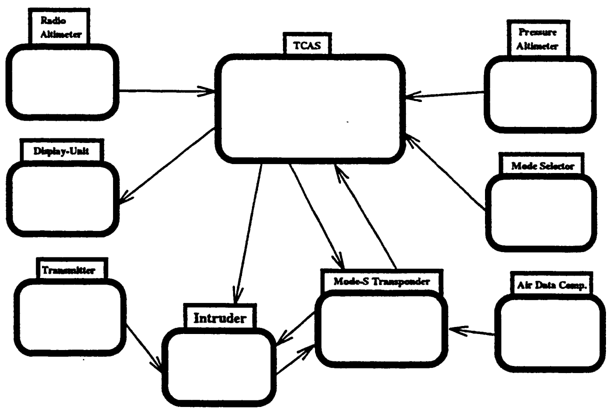


Figure 3: The components and channels in an avionics system.

In RSML we view a system as a collection of physically distinct *components* communicating over unidirectional communication *channels*. The components represent physically separate pieces of the system, for example, a software controller, sensors and actuators (analog or digital), and physical processes. A graphical representation (using the RSML notation) of a collection of system components and communication channels can be seen in Figure 3. We define the behavior of the components in the system using the state machines discussed above. The same language can be used either to capture the required behavior of a component, for example, the TCAS component, or it can be used to model assumed behavior of components, such as, altimeters and display units communicating with the software.

The components are connected to the channels through *interfaces* and can send *messages* over the channels. A message is a collection of *fields* holding the atomic pieces of information communicated between the components.

In the next section we provide an overview of the communication mechanism we are using in in this investigation.

3 Communication Model

In our formal definition of the RSML communication mechanisms we use a layered approach. We use Alan Shaw's Communicating Real-Time State Machines [28] to define the semantics of a collection of low-level RSML communication primitives. We then provide a high-level notation that supports the encapsulation of the inter-component communication in interface specifications. Due to space constraints, we cannot included the full definition of all primitives; we are limited to showing how interrupt driven communication is defined.

3.1 Low-Level Foundation

Shaw's notation is based on communicating finite state machines. Transitions in the state machines are defined as guarded commands; the guard is a Boolean expression and the command is an IO event (send or receive data) or an internal command (variable assignment or computation, see Figure 4). The guarded commands and synchronous communication mechanism are simplified versions of CSP. The communi-

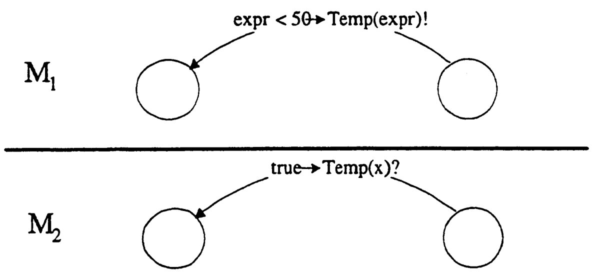


Figure 4: Synchronous communication example.

cation occurs through channels between components. The channels are considered perfect (no loss and no propagation delay) and model 1-1 communication between two state machines. The guarded commands used to control the state transitions have the syntax

$$\langle \textit{guard} \rangle \rightarrow \langle \textit{command} \rangle$$

where $\langle \textit{guard} \rangle$ is a Boolean expression and $\langle \textit{command} \rangle$ can be an input, output, or internal command. A transition is taken if the guard evaluates to true and the command can be executed. The $\langle \textit{command} \rangle$ may be coupled to a time-limit. The time-limit is expressed as $\langle \textit{command} \rangle[t_1, t_2]$, meaning that the command must be performed at the earliest t_1 time units after it was enabled and at the latest t_2 units after it was enabled. Input and output are synchronous and modeled directly after CSP. An input command has the syntax:

$$\langle \textit{channel-name} \rangle(\langle \textit{target} \rangle)?$$

where $\langle \textit{channel-name} \rangle$ is a channel and $\langle \textit{target} \rangle$ is a list of input variables that is compatible with the fields in the message passed over the channel. Output has the syntax:

$$\langle \textit{channel-name} \rangle(\langle \textit{message-components} \rangle)!$$

An IO event can occur only when the names of the communication channels match and the message components are compatible with the variable(s) in the target. For example, M_1 sends an output *Temp(expr)*! and M_2 issues an input command *Temp*(x)? (see Figure 4). This communication is possible if x and *expr* are of the same type. Note that any of the machines may have to wait until communication is possible. The result of the IO is equivalent to $x := expr$ in the input machine M_2.

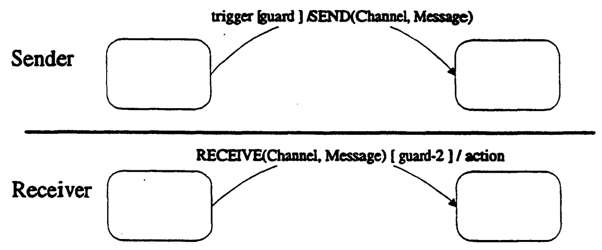
Relationship to Hierarchical State Machines: RSML has a *trigger[condition]/action* defining the semantics of the transitions (as described in Section 2). The guarded command notation in Shaw's model can be used to model the *trigger[condition]/action* semantics. The events can be viewed as Boolean variables, and the *trigger[condition]/action* transition predicate rewritten as $(\textit{trigger} \wedge \textit{condition}) \rightarrow \textit{action}$ in Shaw's notation (assuming that we always can perform the action). Thus, we can use the well understood communication mechanisms from Shaw and CSP to define the semantics of the directed inter-component communication in RSML.

Inter-Component Communication in RSML: Two basic types of communication are commonly encountered in physical systems. First, asynchronous communication with a non-blocking send and no buffering. Second, asynchronous communication with non-blocking send but with the information being persistent on the channel. That is, the sender can send data at any time, the data is buffered (buffer size one), and the receiver can read the data from the buffer at any time. With these two basic communication mechanisms we can implement any other communication scheme we may be interested in, for example, stimulus response. This paper is limited to the discussion of the first communication mechanism.

In RSML, we are using two primitives to model non-persistent (or interrupt driven) communication; SEND($\langle \textit{channel} \rangle$, $\langle \textit{message} \rangle$) and RECEIVE($\langle \textit{channel} \rangle$, $\langle \textit{message} \rangle$). In terms of RSML's and Statecharts' *trigger-action* semantics, SEND is an action that sends the message $\langle \textit{message} \rangle$ over the channel $\langle \textit{channel} \rangle$. RECEIVE is a trigger event that occurs when the message $\langle \textit{message} \rangle$ is received over the channel. The use of these two primitives is illustrated in Figure 5. The behavior of the SEND-RECEIVE pair is defined using Shaw's formalism (Figure 6). The constant d defines how long the message is available on the channel and can be arbitrarily defined depending on the system being modeled.

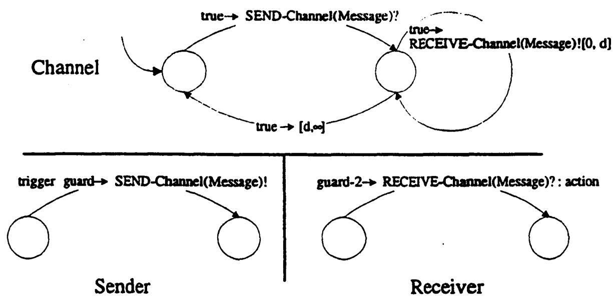
3.2 Readable Communication Specifications

The RSML communication primitives introduced in the previous section, together with the other constructs in RSML, are adequate to fully model system level inter-component communication. Indiscriminate use of the communication primitives, however, may lead to unstructured and difficult to understand models. A state-based model that has its communi-



Sender
trigger [guard] /SEND(Channel, Message)
Receiver
RECEIVE(Channel, Message) [guard-2] / action

Figure 5: The SEND-RECEIVE action-event pair.

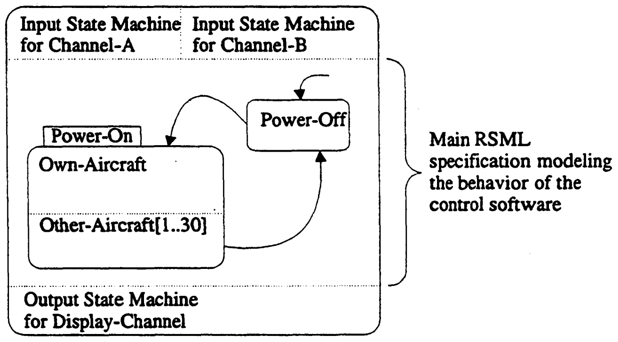


Channel
true→ SEND-Channel(Message)?
true→ RECEIVE-Channel(Message)![0, d]
true → [d,∞]
trigger guard→ SEND-Channel(Message)!
guard-2→ RECEIVE-Channel(Message)? : action
Sender
Receiver

Figure 6: SEND-RECEIVE formally modeled using Shaw's notation.

cations with the environment distributed throughout the model can be very difficult to understand and maintain. Thus, the communication with the environment should be encapsulated in well defined communication modules within each component. For example, in TCAS II, all communication with the RA-Display should be confined to a small state machine dedicated to this task. By encapsulating the communication in dedicated state machines, the main parts of an RSML specification will be shielded from the inevitable changes in the embedding environment (Figure 7).

To facilitate ease of specification and encapsulation we supply a high-level language based on our communication primitives. As a high-level interface description language we chose to use simple textual forms (Figure 8). Leveson *et al.* successfully used a similar approach when specifying the communication mechanisms for TCAS II [20]. The definitions in this paper are an extension and refinement of their approach.



Input State Machine for Channel-A
Input State Machine for Channel-B
Power-On
Power-Off
Own-Aircraft
Other-Aircraft[1..30]
Output State Machine for Display-Channel
Main RSML specification modeling the behavior of the control software

Figure 7: Dedicated communication state machines shield the RSML specification.

3.3 Send-Receive Communication

The interface definition in Figure 8 is adopted from TCAS II and defines how the TCAS box communicates with the pilot's display.

Interface definitions consist of two parts, (1) a physical interface definition that captures properties related to the physical aspects of the communication, for example, the channel name and simple timing assumptions, and (2) a collection of handlers that determine under which conditions we can send/receive messages over this channel (the example in the figure only has one handler, the example will be extended to three handlers later in the paper). The physical interface definition is used to assure that components connected together have compatible properties, for example, that the expected arrival rate at the RECEIVE side is greater than or equal to the expected send rate at the SEND side.

Our interface definition is an abstraction of a simple state machine using the basic RSML communication primitives. Figure 9 shows how the textual definition in Figure 8 is defined with a state machine. Naturally, the state machine could be directly used to specify the inter-component communication. But, since the state machine is very simple and only adds visual noise to the graphical RSML model, we have chosen to use a purely textual notation to abstract away the simple state machines defining the communication. Also, the textual notation forces encapsulation of all communication information in the textual interface definitions.

The output interface in Figure 8 is interpreted as follows. When a state machine in the main part of the specification generates the interface's trigger event and the handler guarding condition is satisfied, the output action in the handler is performed. In this ex-

Output Interface: Display-Unit-Interface

Channel: Display-Channel
Trigger: Send-Traffic-Event[i]
Max Separation: 1.2 second
Min Separation: 0.8 second

Handler-1

Condition:
For all j **in** {1..30}:

		OR	
A	$i \neq j$	T	·
N	Traffic-Display-Status[i] **in state** Waiting-To-Send	T	T
D	Traffic-Display-Status[j] **in state** Waiting-To-Send	F	·
	Traffic-Score(Other-Aircraft[i]) $\geq$ Traffic-Score(Other-Aircraft[j])	·	T

Action: SEND(Advisory-Code[i])

Figure 8: Original definition of the communication with the pilot's display.

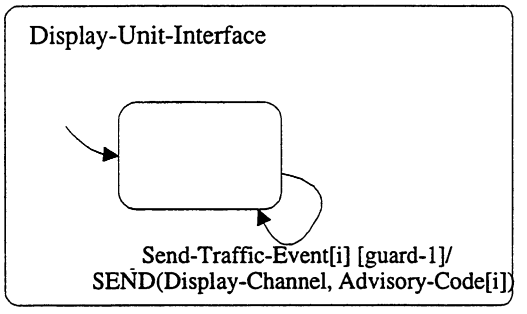

Display-Unit-Interface
Send-Traffic-Event[i] [guard-1]/
SEND(Display-Channel, Advisory-Code[i])

Figure 9: Definition of the communication with the pilot display using an RSML state machine.

ample taken from TCAS II, the state machine model was required to model 30 intruding aircraft (modeled with state machines named Other-Aircraft). The model of each Other-Aircraft contains a state machine called Traffic-Display-Status. When TCAS has detected an intruder and has determined that the pilot needs to be notified, the state machine Traffic-Display-Status associated with that intruder will enter the state Waiting-To-Send. This indicates that TCAS is ready to send an advisory regarding this particular intruder to the pilot's display[2]. If TCAS tracks several intruders and needs to notify the pilot about more than one intruder (more than on Other-Aircraft model is in state Waiting-To-Send), the intruder model with the highest priority (Traffic-Score) takes precedence. The advisory relating to an intruder is contained in the variable Advisory-Code. The communication handler in Figure 8 is parameterized. Any Other-Aircraft model can generate a trigger event for this handler. The handler will simply be instantiated with the index of that intruder (the index is indicated with the *i* in the definition). Thus, the interface in Figure 8 tells us that Other-Aircraft[i] can only send an advisory to the pilot if there are no Other-Aircraft models ready to send (column 1) or there are no Other-Aircraft models with a higher traffic score (column 2).

4 Constraints and Constraint Verification

Since all communication is encapsulated in the interfaces, the guarding condition in a handler is effectively a precondition for the handler's communication to take place. This encapsulation acts as a simple *kernel architecture*; through these preconditions we can assure that no undesired outputs leave our model and that no damaging inputs enter our model. The formality of the communication definition allows us to (1) assure that the input and output definitions are consistently and completely defined and (2) prove that communication related safety assertions and simple liveness assertions hold in the model.

[2]An advisory is a notification to the pilot, for example, if the intruder is very close a resolution advisory will be displayed

4.1 Completeness and Consistency

In a previous investigation, we defined a collection of analysis procedures that assures that an RSML specification is complete and consistent [14]. The notion of completeness and consistency extends to the interface definitions. In this paper it suffices to state the completeness and consistency rules informally:

1. Within an interface definition, every pair of handlers must have mutually exclusive guarding conditions; exactly one handler can be used at any time.
2. The logical OR of the guarding conditions on all handlers within an interface definition must form a tautology; if an input arrives on a channel, it is always defined how this input will be handled

Analysis procedures assuring that the criteria are satisfied are straightforward to automate: the guarding conditions on any two distinct handlers must be contradictory and the disjunction of the guarding conditions on the handlers within each interface must form a tautology. Clearly, the interface in Figure 8 is incomplete since it only handles the case when we are actually allowed to send an advisory. On the other hand, the interface is by definition consistent since there is only one handler. A refined interface definition that is both complete and consistent can be seen in Figure 12. Our tool automatically generates the proof obligations for completeness and consistency. This, however, is not the focus of the paper and the interested reader must be referred to [13, 14] for a rigorous treatment of this topic.

4.2 Safety and Liveness Verification

In TCAS, a safety constraint may be that we cannot remove a Resolution-Advisory from the pilot's display as long at the intruder that caused the advisory to be generated is declared to be a Threat (Other-Aircraft **in state** Threat)[3]. An intruding aircraft is declared to be a threat when a near mid air collision (NMAC) is considered imminent. We may, however, display a Resolution-Advisory against an intruder that is not a threat. An example of such a situation would be when a resolution advisory has only been displayed for a short time, the intruder passes and is no longer considered a threat, but we want to keep the advisory for a few more seconds to provide a sense of continuity to the pilot.

The safety constraint above can be formalized as in Figure 10. Informally, the constraint states that if we

[3]Note that, although a reasonable constraint, this constraint was created for illustration only.

Output Invariant:

The following output: Advisory-Code[i]
can **only** be sent if
Condition:

AND		OR	
	Other-Aircraft[i] **in state** Threat	F	T
	Advisory-Code[i] = Resolution-Advisory	·	T

Figure 10: Safety constraint limiting when we can remove a resolution advisory from the pilot's display.

attempt to output an advisory regarding an intruder that is a threat to our own aircraft, that advisory must be a resolution advisory. If all interactions with the environment are encapsulated in the interfaces, we will be able to verify constraints of this type by only considering the interface specifications.

The verification approach progresses in two simple steps. First, we determine which handlers that can output the variable we are interested in. Second, we show that the guarding condition (g) in those handlers imply the constraint (c), that is, that ($g \Rightarrow c$).

A similar approach can be used to prove simple liveness constraints. For example, if a certain input arrives and its value is outside the expected boundaries, we may always want to initiate a system shutdown or some recovery procedure. An example of a liveness constraint derived from TCAS II can be seen in Figure 11. All aircraft are supposed to have a unique transponder identification number known as the Mode-S-Address. The address is assigned by a central regulatory agency before an aircraft is taken into operation. Unfortunately, some aircraft, for example, prototypes in test flight, may not have a proper address and fly with the default address assigned to the transponder by the manufacturer. This default address is commonly all zeros or all ones (0 or MaxID). If such an aircraft is detected, it should always receive special treatment. The invariant in Figure 11 captures this simple liveness property; if TCAS receives an invalid Mode-S-Address, it will always generate an exception event. This is an admittedly weak assertion, but we found places in the TCAS II specification where such assertions would have been helpful. In fact, failure to properly detect and handle invalid Mode-S-Addresses was a problem with early TCAS II implementations.

Verification of liveness is similar to the verification of the safety constraints discussed above. In this case, however, we want to show that all handlers that have

Input Invariant:

The following input: Mode-S-Address[i]
when
Condition:

		OR	
AND	Other-Mode-S-Address[i] = 0	T	·
	Other-Mode-S-Address[i] = MaxID	·	T

always leads to:
Action: Invalid-Intruder-ID-Event

Figure 11: Safety constraint indicating that when we receive an invalid Mode-S-Address we must always raise an exception.

Mode-S-Address as an input and have a guarding condition g that is implied by the constraint c always generate the Invalid-Intruder-ID-Event as an action ($c \Rightarrow g$)

To evaluate our approach, we have augmented an existing execution environment and analysis tool for RSML with the capability to take communication related safety and liveness assertions as input. The tool generates proof obligations based on the rules discussed in this section. We generate proof obligations in the PVS specification language and use the PVS theorem prover to perform the proofs. The next section gives an example of the translation approach.

5 Generating Proof Obligations for PVS

The Prototype Verification System (PVS) is a verification system that provides an interactive environment for the development and analysis of formal specifications [25, 26]. PVS consists of a specification language, a parser, a type-checker, an interactive theorem prover, and various browsing tools.

To illustrate our approach, consider the interface definition in Figure 8 and the assertion in Figure 10. Clearly, we cannot prove the assertion from the information provided in this interface specification. Furthermore, as mentioned above, the interface is incomplete. Figure 12 shows the same interface extended to handle the normal case when we are allowed to send an advisory, the case where we are not allowed to send an advisory, and the case where we have a safety violation. The rest of this section illustrates how we prove that this interface complies with the assertion.

Our tool generates a PVS theory for each handler and assertion in an RSML specification. We do this in a two stage process. First, we define each predicate in the AND/OR table as a predicate in the PVS specification language[4]. Second, a predicate representing the full guarding condition (or assertion) is built from the individual predicates defined in the first stage. In PVS, the assertion in Figure 10 would be defined by the theory shown in Figure 13. The constants in the system are defined as a separate theory and imported to the theory defining the assertion (or handler).

Since we are interested in proving that an Advisory-Code[i] with the value Resolution-Advisory can only be generated under certain circumstances, our analysis algorithm will identify Handler-1 in Figure 12 and generate the PVS theory in Figure 14.

The actual proof obligations are generated based on the criteria described in the previous section. Figure 15 shows the proof obligation for this example. In this case we want to show that the precondition (guarding condition) for the communication implies that the assertion is true (Figure 15). PVS proved this property automatically with one predefined strategy.

6 Summary and Conclusion

In this paper we discussed a formal approach to the specification of inter-component communication in RSML specifications. The approach is based on communicating finite state machines. The formalism allows encapsulation of communication related properties in well defined interface specifications. The encapsulation enables us to use the interface specifications as simple *safety kernels* and enforce certain safety and liveness constraints in these kernels

Furthermore, we described how safety and liveness constraints related to inter-component communication can be formalized using a simple and easy to understand constraint language. To formally verify that the constraints are satisfied in an RSML model, we attempt to prove that the constraints are satisfied by only looking at the interface specifications. If the constraints are enforced in the interface definitions, the proofs are relatively small and easy to perform. If we did not encapsulate the communication related properties in the interfaces and instead had the communication distributed throughout the model, verification of the constraints could be overwhelmingly complex. We illustrated the approach with an example from TCAS II.

To evaluate the potential of the approach, we

[4]A predicate in PVS is a function with return type Boolean.

Output Interface: Display-Unit-Interface

Channel: Display-Channel
Trigger: Send-Traffic-Event[i]
Max Separation: 1.2 second
Min Separation: 0.8 second

Handler-1

Condition:
For all j **in** {1..30}:

			OR		
	$i \neq j$	T	T	·	·
A	Traffic-Display-Status[i] **in state** Waiting-To-Send	T	T	T	T
N	Traffic-Display-Status[j] **in state** Waiting-To-Send	F	F	·	·
D	Traffic-Score(Other-Aircraft[i]) $\geq$ Traffic-Score(Other-Aircraft[j])	·	·	T	T
	Other-Aircraft[i] **in state** Threat	F	T	F	T
	Advisory-Code[i] = Resolution-Advisory	·	T	·	T

Action: SEND(Advisory-Code[i])

Handler-2

Condition:
Exists at least one j **in** {1..30}:

				OR			
	$i \neq j$	T	T	F	F	·	·
A	Traffic-Display-Status[i] **in state** Waiting-To-Send	F	F	F	F	T	T
N	Traffic-Display-Status[j] **in state** Waiting-To-Send	F	F	·	·	T	T
D	Traffic-Score(Other-Aircraft[i]) $\geq$ Traffic-Score(Other-Aircraft[j])	·	·	·	·	F	F
	Other-Aircraft[i] **in state** Threat	F	T	·	F	F	·
	Advisory-Code[i] = Resolution-Advisory	·	T	T	·	·	T

Action: None

Handler-3

Condition:

A N	Other-Aircraft[i] **in state** Threat	T
D	Advisory-Code[i] = Resolution-Advisory	F

Action: Assertion-Violation-Event

Figure 12: Modified definition of the communication with the TCAS display. This description is complete, consistent, and enforces the assertion in Figure 10

```
%------------------------------------------------%
% Definition of the safety invariant 1 for the     %
% output variable Advisory-Code[i]                 %
%------------------------------------------------%

Output_Invariant: THEORY
BEGIN
IMPORTING TypeDefs
AdvisoryCode: VAR AdvisoryCodeType
OtherAircraft: VAR OtherAircraftType
i: VAR i_type
pred1?(OtherAircraft, i): bool = Threat?(OtherAircraft(i))
pred2?(AdvisoryCode, i): bool = ResolutionAdvisory?(AdvisoryCode(i))
OutputInvariant?(AdvisoryCode, OtherAircraft, i): bool =
      ( NOT pred1?(OtherAircraft, i)
         OR ( pred1?(OtherAircraft, i) & pred2?(AdvisoryCode, i) ))
END Output_Invariant
```

Figure 13: A PVS theory for the safety assertion in Figure 10.

have looked at the interfaces and possible safety constraints in TCAS II. From this limited study we identified several cases in TCAS II (two of which are mentioned in this paper) where the capability to capture and prove simple communication related assertions would have been helpful.

Although we believe the approach holds great potential, there are several questions that must be addressed. First, a more thorough investigation is needed to determine how useful the assertions we are capable of expressing really are. As mentioned above, our experience with TCAS II indicate that they are quite useful, but a more thorough case study is clearly needed. Second, since the assertions must be enforced in the interface definitions, the interface definitions may become unnecessarily complex. We do not want to sacrifice readability, clarity, and ease of use of the interface definitions for the sole purpose of verification. The effect of encapsulation of communication constraints in the interfaces is currently not clear and, again, further case studies are needed.

In summary, we believe the verification approach outlined in this paper will be effective, useful, and practical. However, we need to conduct further case studies to better evaluate the expressive power of the constraints and the effect of the encapsulation of the communication constraints on the overall readability of RSML specifications.

Acknowledgment

I would like to thank Barb Czerny for developing the RSML to PVS translator and for performing the proofs in this investigation.

References

[1] R.J. Anderson, P.Beame, S. Burns, W. Chan, F. Modugno, Notkin D, and J.D. Reese. Model checking large software specifications. In D. Garlan, editor, *Proceedings of the Fourth ACM SIGSOFT Symposium on the foundations of Software Engineering (SIGSOFT'96)*, October 1996.

[2] J.M. Atlee and M.A. Buckley. A logic-model semantics for SCR software requirements. In S.J. Zeil, editor, *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'96)*, pages 280–292, January 1996.

[3] J.M. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.

[4] M. Chechik and J. Gannon. Automatic verification of requirements implementations. In *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–14, 1994.

[5] M. Chechik and J. Gannon. Automatic analysis of consistency between implementations and requirements: A case study. In *Proceedings of the Tenth Annual Conference on Computer Assurance*, pages 123–131, 1995.

```
Handler1_OutputInterface: THEORY
  BEGIN
  IMPORTING TypeDefs
  AdvisoryCode: VAR AdvisoryCodeType
  TrafficDisplayStatus: VAR TrafficDisplayStatusType
  OtherAircraft: VAR OtherAircraftType
  TrafficScore: VAR TrafficScoreType
  j: VAR int
  i: VAR i_type

  pred1?(i, j): bool = NOT (i = j)
  pred2?(TrafficDisplayStatus, i): bool = WaitingToSend?(TrafficDisplayStatus(i))
  pred3?(TrafficDisplayStatus, j): bool = WaitingToSend?(TrafficDisplayStatus(j))
  pred4?(TrafficScore, OtherAircraft, i, j): bool =
    TrafficScore(OtherAircraft(i)) >= TrafficScore(OtherAircraft(j))
  pred5?(OtherAircraft, i): bool = Threat?(OtherAircraft(i))
  pred6?(AdvisoryCode, i): bool = ResolutionAdvisory?(AdvisoryCode(i))

  Handler1?(AdvisoryCode, TrafficDisplayStatus,
            OtherAircraft, TrafficScore, i): bool =
    FORALL (j: int | j >= 1 AND j <= 30):
      (  (pred1?(i, j)
         & pred2?(TrafficDisplayStatus, i)
         & NOT pred3?(TrafficDisplayStatus, j)
         & NOT pred5?(OtherAircraft, i))
       OR
         (pred1?(i, j)
         & pred2?(TrafficDisplayStatus, i)
         & NOT pred3?(TrafficDisplayStatus, j)
         & pred5?(OtherAircraft, i)
         & pred6?(AdvisoryCode, i))
       OR
         (pred2?(TrafficDisplayStatus, i)
         & pred4?(TrafficScore, OtherAircraft, i, j)
         & NOT pred5?(OtherAircraft, i))
       OR
         (pred2?(TrafficDisplayStatus, i)
         & pred4?(TrafficScore, OtherAircraft, i, j)
         & pred5?(OtherAircraft, i)
         & pred6?(AdvisoryCode, i)))
  END Handler1_OutputInterface
```

Figure 14: A PVS theory for the first handler in Figure 12.

```
%-----------------------------------------------------%
% Proof obligation for safety invariant 1             %
%-----------------------------------------------------%

OutputInterface: THEORY
  BEGIN

  IMPORTING TypeDefs, Handler1_OutputInterface,
            Output_Invariant

  AdvisoryCode: VAR AdvisoryCodeType
  TrafficDisplayStatus: VAR TrafficDisplayStatusType
  OtherAircraft: VAR OtherAircraftType
  TrafficScore: VAR TrafficScoreType
  i: VAR i_type

Handler1_implies_OutputInvariant: CONJECTURE
      Handler1?(AdvisoryCode, TrafficDisplayStatus,
              OtherAircraft, TrafficScore, i)
       IMPLIES
         OutputInvariant?(AdvisoryCode, OtherAircraft, i)

  END OutputInterface
```

Figure 15: A PVS theory the proof obligation for the constrint in figure 10.

[6] J. Crow, S. Owre, J. Rushby, et al. A tutorial introduction to PVS. In *WIFT 95: Workshop on Industrial-Strength Formal Specification Techniques*, 1995.

[7] S. Gerhart, D. Craigen, and T. Ralston. Experience with formal methods in critical systems. *IEEE Software*, vol-11(1):21–39, January 1994.

[8] S. Gerhart, D. Craigen, and T. Ralston. Formal methods reality check: Industrial usage. *IEEE Transactions on Software Engineering*, 21(2):90–98, February 1995.

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

[10] D. Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[11] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[12] D. Harel and A. Pnueli. On the development of reactive systems. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. Springer-Verlag, 1985.

[13] M. P.E. Heimdahl and B. C. Czerny. Using PVS to analyze hierarchical state-based requirements for completeness and consistency. In *Proceedings of the IEEE High Assurance Systems Engineering Workshop*, 1996.

[14] M. P.E. Heimdahl and N.G. Leveson. Completeness and Consistency Analysis of State-Based Requirements. *IEEE Transactions on Software Engineering*, TSE-22(6):363–377, June 1996.

[15] C. L. Heitmeyer, R.D. Jeffords, and B. L. Labaw. Consistency checking of SCR-style requirements specifications. *ACM Transactions on Software Engineering and Methodology*, vol-5(3):231–261, July 1996.

[16] C. L. Heitmeyer, B. L. Labaw, and D. Kiskis. Consistency checking of SCR-style requirements specifications. In *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, March 1995.

[17] K. L. Heninger. Specifying software for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering*, 6(1):2–13, January 1980.

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

[19] N. G. Leveson. Software safety: What, why, and how. *ACM Computing surveys*, 18(2), June 1986.

[20] N. G. Leveson, M. P.E. Heimdahl, H. Hildreth, and J. D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):694–707, September 1994.

[21] N.G. Leveson, T.J. Shimeall, J.L. Stolzy, and J.C. Thomas. Designing for safe software. In *Proceedings of the AIAA 21st Aerospace Sciences Meeting*, pages 1–5, 1993.

[22] R. Lutz. Targeting safety-related errors during software requirements analysis. In *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1993.

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

[24] B.E. Melhart. *Specification and Analysis of the Requirements for Embedded Software with an External Interaction Model.* PhD thesis, University of California, Irvine, July 1990.

[25] S. Owre, N. Shankar, and J.M.Rushby. *The PVS Specification Language.* Computer Science Laboratory; SRI International, Menlo Park, CA 94025, beta release edition, April 1993.

[26] S. Owre, N. Shankar, and J.M.Rushby. *User Guide for the PVS Specification and Verification System.* Computer Science Laboratory; SRI International, Menlo Park, CA 94025, beta release edition, March 1993.

[27] J. Rushby. *Safe and Secure Computing Systems*, chapter 13, Kernels for Safety?, pages 210–220. Blackwell Scientific Publications, 1989.

[28] A. C. Shaw. Communicating real-time state machines. *IEEE Transactions on Software Engineering*, 18(9):805–816, September 1992.

[29] K.G. Wika. *Saefty Kernel Enforcement of Software Safety Policies.* PhD thesis, University of Virginia, May 1995.

[30] K.G. Wika and J.C. Knight. On the enforcement of software safety policies. In *COMPASS Proceedings*, pages 83–93, 1995.

Towards High-Assurance High-Performance Program Synthesis

Michael Lowry
NASA Ames Code IC
M. S. 269-2
Moffett Field, CA 94035
lowry@ptolemy.arc.nasa.gov

Steven Roach
NASA Ames Code IC
M. S. 269-2
Moffett Field, CA 94035
sroach@ptolemy.arc.nasa.gov

Jeffrey Van Baalen
NASA Ames Code IC
M. S. 269-2
Moffett Field, CA 94035
jvb@ptolemy.arc.nasa.gov

Abstract

Domain-specific automatic program synthesis tools, also called application generators, are playing an ever-increasing role in software development. However, high-performance application generators require difficult manual construction, and are very difficult to verify correct. This paper describes research and an implemented system that transforms program synthesis tools based on deductive synthesis into high-performance application generators. Deductive synthesis uses theorem-proving to construct solutions when given problem specifications. The verification condition for a deductive synthesis tool is essentially the soundness of the implemented inference rules.

TOPS (theory operationalization for program synthesis) synergistically combines reformulation, automated mathematical classification, and compilation through partial deduction to decision procedures. It transforms general-purpose deductive synthesis, with exponential performance, into efficient special-purpose deductive synthesis, with near-linear performance. This paper describes our experience with and empirical results of PD(TH) - theory-based partial deduction - in which partial deduction of a set of first-order formulae is performed within the context of a background theory. The implemented TOPS system currently performs a special variant of PD(TH) in which the compilation process results in the transformation of a set of first-order formulae into the theory of an instantiated library decision procedure augmented by a compiled unit theory.

1. Motivation

Application generators have been used for over a decade to greatly improve the speed and productivity of software development. As early as the mid nineteen-eighties, over half of the COBOL code generated annually was synthesized by application generators. The source language of these tools target a narrow application domain, as opposed to a general-purpose compiler-based language. Visual programming front-ends typically facilitate end-user development of the source specifications. The output of these tools is usually code in a general-purpose language (e.g., C, C++, or COBOL). Examples of these application generators include database application development tools (also called fourth generation languages) and GUI builders. Application generators are also being increasingly used in aerospace applications. As just one example, several NASA space projects, such as the Delta Clipper, are using the MatrixX code generator to synthesize control system software.

The current generation of commercial application generators suffer from two defects. The first defect is the difficulty of constructing high-performance application generators. Application generators which perform significant semantic transformations or optimizations from source to object level require extensive expert hand-coding. This limits the cost-effective use of these tools to stable application domains with a large number of potential users. Tools for building application generators themselves (i.e., application generator-generators [3]) are limited in the semantic power of the source-to-object level transformations they can be used to develop. The second, more important defect is the reliability of the application generators themselves, thereby limiting the level of assurance of the code they generate. This is particularly a problem for high-assurance domains typified by the aerospace industry. In these domains, where test and validation consume over half of software costs, the potential of application generators to increase software productivity is limited by the lack of verification of the code-generator.

We have been pursuing research to address these limitations of commercial application generators. Program synthesis tools based on automated theorem-proving (i.e., deductive synthesis) are intrinsically high-assurance, but have been limited by their inefficiency. As source specifications grow linearly, the time required to generate a solution usually grows exponentially. Deductive synthesis tools are in principle easy to construct, because the semantics of a domain can be expressed declaratively. Furthermore, in principle, the semantics of a domain can be built up out of reusable components related to mathematical abstractions. In practice they are quite difficult to construct, because considerable expertise in general-purpose theorem proving is required even to formulate the semantics of a domain in a manner that makes deductive synthesis feasible. The paradigm employed by most users of general-purpose theorem provers consists of problem specification followed by an iterative activity in which an expert reformulates the background theory and tunes parameters of the theorem prover. The behavior of the theorem prover is observed on each iteration to make adjustments for the next iteration. This can be an extremely difficult and time consuming process that often yields less than satisfactory results.

We have developed a suite of domain-oriented deductive synthesis systems generically called AMPHION. These systems can be used by non-programmer end-users, with no experience in automated theorem proving, to generate programs consisting of hundreds of lines of code. In the past, AMPHION systems have been constructed by experts in deductive synthesis, with substantial tuning of deductive synthesis for each new domain (but no additional tuning for new problems). In the future, we anticipate these systems will be constructed by domain experts using META-AMPHION [11], which is a suite of tools for generating AMPHION systems. The major impediment for domain-expert construction of an AMPHION system is the tuning of deductive synthesis. The TOPS (theory operationalization for program synthesis) tool automates this tuning by transforming an easy-to-validate but inefficient deductive synthesis system based on general-purpose inference rules into an efficient program synthesis system based on decision procedures. Decision procedures are special-purpose inference rules, which can be implemented very efficiently precisely becaus of their limited applicability.

The research described in [16] has resulted in new algorithms for TOPS; empirical results are presented in this paper. These algorithms are based on an enhanced theoretical understanding of the search interaction of decision procedures for deductive synthesis. By exploiting properties specific to deductive synthesis, as opposed to automated theorem proving in general, known difficulties in using decision procedures have been circumvented. In particular, there are sufficient conditions, which are less restrictive than those in [15], under which exponential search in deductive synthesis can be avoided through decision procedures. A new algorithm to compile decision procedures from a domain theory has been developed based on these sufficient conditions. In addition to generating decision procedures, this algorithm reformulates parts of the domain theory to achieve separation of decision procedures. This seperation limits the potentially combinatorial interaction between decision procedures and hence avoids exponential search. New empirical results are presented in this paper that demonstrate these algorithms result in more efficient program synthesis tools than expert manual tuning of a deductive synthesis system.

2. Background

AMPHION is a generic program synthesis system that greatly facilitates end-user use of domain-oriented software libraries. AMPHION enables a user to state a problem in an abstract, domain-oriented vocabulary using a graphical notation; rather than requiring the user to construct a solution by manually composing software components. AMPHION automatically generates a program consisting of calls to library components that implements a solution to a problem specification. On average, it takes an order of magnitude less time for a user to develop a domain-oriented problem specification with AMPHION than to manually generate and debug a program. Equally important, a user does not need to learn the details of the components in a software library, thereby removing a significant barrier to the use of software libraries. Three AMPHION systems have been developed at NASA, for the domains of space science observation geometry, space shuttle navigation, and computational fluid dynamics. AMPHION is described in detail in ([10,17]); an overview is presented here as background to understanding the search control issues and empirical results.

AMPHION consists of a specification acquisition subsystem and a program synthesis subsystem, both of which are generic across domains; and a domain-specific subsystem. The graphical user interface enables a user to interactively build a diagram representing a formal problem specification in first-order logic. Diagrams are equivalent to specifications of the following form :

$\forall(inputs)\ \exists\ exists\ (outputs)\ \ \exists\ (intermediates)\ matrix$

The matrix is expressed in the abstract specification language (see below), except for atoms which express the relation between concrete input/output variables and the abstract variables they represent.

The program synthesis subsystem consists of an applicative program generator and a translator into the target programming language. An applicative program is generated through deductive synthesis [12]. AMPHION uses the SNARK resolution theorem prover [17], to generate a proof that the specification is a theorem of the domain theory. During a proof, substitutions are generated for the existential variables through unification and equality replacement. The substitutions for the output variables are constrained to be terms in the target language whose function symbols correspond to the components of the library. The applicative program is translated into a target programming language, such as FORTRAN or C++, through program transformations.

The domain-specific subsystem consists of a domain theory and theorem-proving tactics. An AMPHION domain theory has three parts: an abstract theory whose language is suitable for problem specifications, a concrete theory that includes the target component specifications, and an implementation relation between the abstract and concrete theory. The implementation relation is axiomatized in the style of Hoare [6], through abstraction maps. The following commutative diagram provides a useful way of visualizing AMPHION domain theories, specifications, and later in this paper, generated decision procedures:

$$\begin{tikzcd}
\text{Abstract Input} \arrow[r] & \text{Abstract Output} \\
\text{Concrete Input} \arrow[u] \arrow[r] & \text{Concrete Output} \arrow[u]
\end{tikzcd}$$

Figure 1: Basic AMPHION Commutative Diagram

The vertical arrows are abstraction maps between the concrete level and the abstract level. An AMPHION problem specification is formulated at the abstract level, augmented with abstraction maps from the definitions of the concrete inputs and outputs (i.e., the representation and format of data for the input and output variables). The lower horizontal arrow is not defined in an AMPHION problem specification. In essence, AMPHION generates this lower horizontal arrow through deductive synthesis: this arrow is the desired program that maps the concrete inputs to the concrete outputs such that the whole diagram commutes, i.e., the generated program is correct with respect to the specification. The generated decision procedures described later in this paper perform special-purpose deductive synthesis (i.e., fills in the lower horizontal arrow); each over small portions of the domain theory.

The theorem-proving tactics for any particular AMPHION domain theory guide SNARK from abstract, specification-level constructs towards concrete, implementation-level constructs. The tactics are implemented in part by defining an agenda ordering function, which is used to choose the next descendent of the goal (specification) clause to operate on in an overall set-of-support strategy. The tactics are highly effective, reducing program synthesis times from hours (sometimes days) to minutes. The graph below shows the time (in seconds) required for deductive synthesis plotted against the number of literals in a problem specification. The tactics do not need to be tuned for individual problems, but often do require expert manual tuning when a domain theory is modified. In figure 2, the white boxes are deductive synthesis times using generic abstract-to-concrete tactics that have not been manually tuned to the particular domain theory (in this case, the domain of space observation geometry). As the number of literals in a specification increase, computation times (in seconds) grow exponentially and preclude finding a solution in acceptable times for specifications exceeding forty literals. In contrast, the black boxes are deductive synthesis times after manual tuning of the tactics for the space observation geometry domain theory. Computation times scale well with increasing numbers of literals in a problem specification.

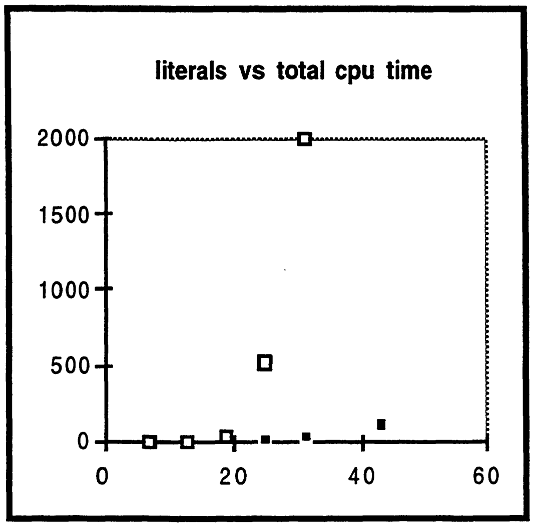


Figure 2: Performance of Tuned versus Untuned Theorem-Proving Tactics

Some of the AMPHION domain theories have gone through dozens of major revisions. Manually tuning the tactics when a domain theory is modified can be quite difficult, since the tactics interact significantly with the formulation of a domain theory. It requires significant expertise and labor to experiment with and adjust the tactics. This is a large impediment to transitioning AMPHION from a few application domains in the research laboratory to a plethora of application domains in the field.

One approach around these difficulties would be to automatically learn or generate theorem-proving tactics, through techniques such as genetic programming or machine learning for control rules, as exemplified by Multi-Tac [13]. However, we believe the approach of automated induction of tactics is limited in its ability to scale up, due to the complexity of the tactical knowledge required to efficiently guide general-purpose inference rules through long deduction chains. We note that tactics can be quite complex, in fact, tactical complexity was the original motivation for many of the innovative language features of ML.

A promising alternative to the tactic approach that is achieving increasing success in automated deduction is to specialize the behavior of a theorem prover by interfacing special-purpose decision procedures to the general-purpose theorem prover [1,4,5,9,17]. This approach has the advantage of identifying decision procedures to use based on axioms present in the domain theory, rather than tuning tactics based on the observed performance of the theorem prover. For example, axioms for linear arithmetic can be subsumed by a decision procedure for linear arithmetic. As a result, the identification of appropriate decision procedures is often much easier than developing tactics that guide general-purpose deduction. As reported in this paper, identifying and specializing the appropriate decision procedures can be automated through a process closely related to deductive synthesis, and hence is correct by construction (assuming the correctness of the library of parameterized decision procedures). Furthermore, once the original AMPHION domain theory is validated, then the automatically specialized synthesis system is also valid.

A difficulty with the decision procedure approach is that interfacing a decision procedure to a general-purpose theorem prover can be an ad-hoc and very difficult design and development project in its own right. Hence, even though identifying decision procedures that can speed theorem proving can be straightforward, interfacing those decision procedures to the theorem prover can be problematic, and in previous efforts has led to the loss of efficiency [1]. We have circumvented these difficulties by exploiting the particular context of deductive synthesis in AMPHION, specifically the separation between the specification level and the implementation level, as described in section 4.

3. Empirical Results of TOPS

TOPS automatically operationalizes a domain theory for efficient deductive synthesis. The approach is analogous to applying AMPHION at the meta-level: given a meta-theory of program synthesis and a domain theory for a particular application domain, TOPS constructs an efficient deductive synthesis system by composing and instantiating parameterized decision procedures from a library and then instantiating the generic AMPHION architecture. The meta-theory axiomatizes the applicability conditions of the decision procedures, which replace axioms in the domain theory. This meta-theory is a hierarchy of parameterized theories that describe applicability conditions of decision procedures. The decision procedures are similar to ones previously developed for DRAT [20,21], but for deductive synthesis have been extended to include a parameterized portion which is compiled by TOPS. The algorithms reported in [16], which compile an instantiation for the parameterized portion of a decision procedure, are significant adaptations and extensions of partial deduction algorithms [8].

The compilation is highly effective, and produces results that are better than manual tuning of theorem proving tactics. This section describes experimental results of applying TOPS to one release of the NASA NAIF domain theory. The NAIF domain theory is continuously being augmented, and thus provides a good real-world example for testing the effectiveness of our approach to domain theory maintenance and operationalization. The release of the NAIF domain theory consists of 330 axioms in first order logic that collectively define:

1. An abstract specification language for space observation geometry, which is essentially an augmentation of Euclidean geometry.
2. Axiomatizations for pre- and post-conditions for a set of FORTRAN routines in the NAIF (Navigation and Ancillary Information Facility) tool kit. These routines access data formats defined by NAIF (such as trajectory data for planets and spacecraft) and routines in analytic geometry (such as the intersection of a ray denoting an observation vector with an ellipsoid denoting a planet).
3. Definitions of abstraction mappings between concrete sorts and abstract sorts.

The search that is incurred during deductive synthesis through resolution refutation is due to the resolution inference rule and the paramodulation inference rule. Each of these rules counts as one step in the search tree. SNARK performs a full term simplification (demodulation) with each of these rules. Likewise, when SNARK is augmented with decision procedures compiled by TOPS, the theory resolution interface runs the decision procedures to quiescence with each resolution or paramodulation step. The total amount of CPU time is thus roughly linearly proportional to the total number of resolution and paramodulation steps.

To test the TOPS algorithms, we used three configurations of the AMPHION system for the NAIF domain: SNARK with just the generic abstract-to-concrete tactic, SNARK with an augmented manually-tuned tactic specific to the NAIF domain, and SNARK with decision procedures compiled by TOPS (and the generic abstract-to-concrete tactic). A set of specifications of varying number of literals were developed to test these configurations.

Figure 3 plots the time it takes (in seconds, the y-axis) for each configuration to synthesize programs from specifications with varying numbers of literals (the x-axis). The diamonds labeled Series 3 are data points for the configuration with only a generic tactic, while Series 1 are the data points for the manually-tuned tactic. The series 2 triangles are the data points for the configuration generated by TOPS. Note that the generic tactic results in exponential performance, while the manually-tuned and TOPS-generated configurations scale well. At this scale for time, the latter two are indistinguishable.

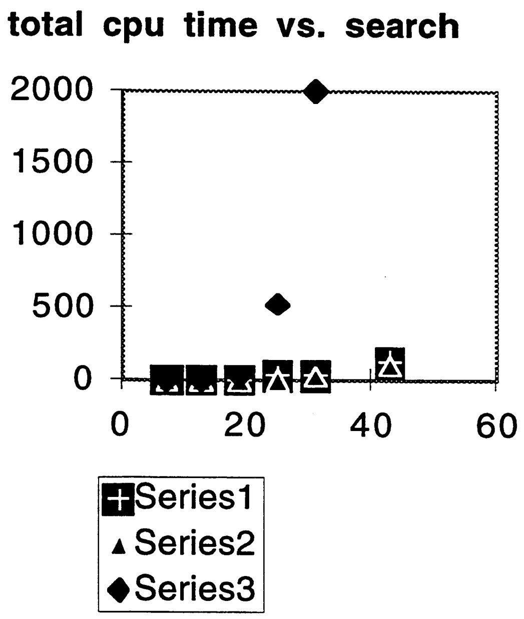


Figure 3: Plot of CPU times for 3 Program Synthesis Configurations

Figure 4 blows up the scale for manually-tuned and TOPS-generated configurations, and also substitutes the total number of resolution/paramodulation steps for the time axis. Figure 4 demonstrates that the decision procedures effectively eliminate much of the search engendered by the resolution inference rule and the paramodulation inference rule.

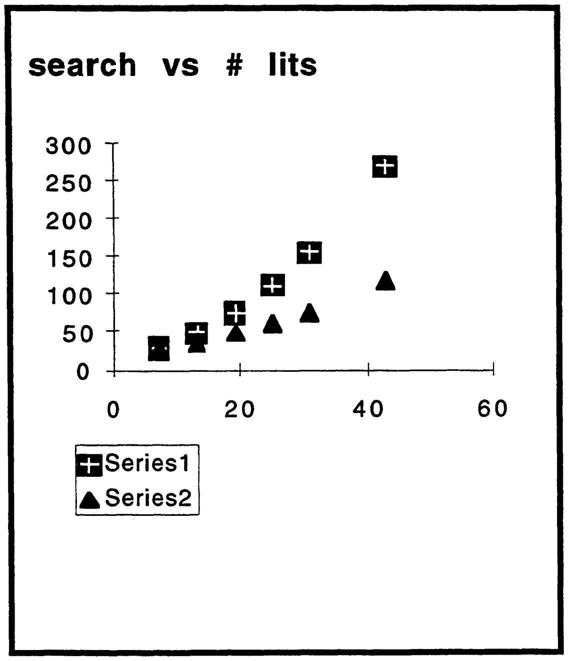


Figure 4: Total steps in Proof Search versus Literals in a Specification

4. Summary

Commercial application generators are limited by both their level of assurance and their ability to be adapted to a changing target domain. Historically, program synthesis tools based on deductive synthesis are high assurance but have been severely limited by their inefficiency and inability to scale up to larger specifications. Manual methods for increasing their efficiency by tuning of tactics and strategies makes it very difficult to maintain them in the context of a changing target domain.

The use of decision procedures for general automated theorem proving has in the past had mixed results for increasing efficiency. It can be difficult to sufficiently separate the decision procedures from each other and the theorem prover to avoid a very high overhead in communication, thus largely negating the efficiency of the decision procedures. Classical sufficient conditions for achieving separation [14] are overly restrictive in the context of deductive synthesis.

Building on past work, in this paper we describe empirical results based on more relaxed sufficient conditions for separation of decision procedures that are specific to the context of deductive synthesis in AMPHION-structured domain theories. Using these results, we overviewed an implemented improved algorithm for automatically compiling domain theories into decision procedures. This algorithm is validated experimentally, and results in high-assurance deductive synthesis systems whose efficiency is superior to manually-tuned deductive synthesis systems. We believe this paradigm is suitable for generating high-assurance high-performance program synthesis systems for changing target domains. This hypothesis will be validated in future work.

Acknowledgments

Sincere thanks to NASA Code Q and NASA Code S for funding this work.

References

[1] R. Boyer and J. Moore, "Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study with Linear Arithmetic," *Machine Intelligence* 11, Oxford University Press, 1988.

[2] H.-J. Burckert. A resolution principle for a logic with restricted quantifiers. In *Lecture notes in Artificial Intelligence,* Springer-Verlag, 1991.

[3] J.C. Cleaveland and C. Kintala, "Tools for Building Application Generators." In *AT&T Technical Journal,* Vol. 67, No. 4, 1988, pp. 46-58.

[4] J.H. Gallier, *"Logic for computer science, Foundations of Automatic Theorem Proving,"* Harper & Row, 1986.

[5] F. Giunchiglia, P. Pecchiari, C. Talcott, "Reasoning Theories: Towards an Architecture for Open Mechanized Reasoning Systems," Stanford CS Technical Report CS-TN-94-15, 1994.

[6] C.A.R. Hoare, "Proof of Correctness of Data Representations," *Acta Informatica* 1973, pp. 271-281.

[7] R. Jullig and Y.V. Srinivas, "Diagrams for Software Synthesis," *KBSE 1993.*

[8] J. Komorowski, "Synthesis of Program in the Partial Deduction Framework," in *Automating Software Design,* Lowry and McCartney (Eds), MIT, 1991.

[9] D. W. Loveland. *Automated Theorem Proving: a logical basis.* North Holland, 1978.

[10] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood, "A Formal Approach to Domain-Oriented Software Design Environments," In *KBSE 1994.*

[11] M. Lowry and J. Van Baalen. META-AMPHION: Synthesis of Efficient Domain-Specific Program Synthesis Systems. *Automated Software Engineering,* March 1997.

[12] Z. Manna and R. Waldinger, "Fundamentals of Deductive Program Synthesis," *IEEE Transactions on Software Engineering,*(18) 8, August 1992, pp. 674-704.

[13] S. Minton and S. Wolfe, S. ``Using Machine Learning to Synthesize Search Programs", In *KBSE 1994*

[14] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions of Programming Languages and Systems,* pages, 1:245-257, 1979.

[15] G. Nelson, "Combining Satisfiability Procedures by Equality Sharing," in *Automated Theorem Proving after 25 Years,* Bledsoe and Loveland (Eds), American Mathematical Society, 1984.

[16] S. Roach, "TOPS: Theory Operationalization for Program Synthesis," PhD Thesis at University of Wyoming, 1997.

[17] M. Stickel, "Automated Deduction by Theory Resolution," *Automated Reasoning,* Vol. 1, 1985, pp. 333-355.

[18] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood, "Deductive Composition of Astronomical Software from Subroutine Libraries," In *CADE-12,* 1994.

[19] E.H. Tyugu, *Knowledge-Based Programming,* Turing Institute Press, Glasgow, Scotland, 1988.

[20] J. Van Baalen. The Completeness of DRAT, a technique for automatic design of satisfiability procedures. In *Proceedings of International Conference on Knowledge Representation and Reasoning,* 1991.

[21] J. Van Baalen. Automated design of specialized representations. *Artificial Intelligence,* 54:121-198, 1992.

On the Automatic Discovery of Loop Invariants*

Andrew Ireland† and Jamie Stark‡
Department of Computing & Electrical Engineering
Heriot-Watt University
Edinburgh, Scotland, U.K. EH14 4AS

Abstract

We present a technique for automating the discovery of loop invariants based upon the analysis of failed proof attempts. Previously we have shown how failure analysis may be used productively in the search for inductive proofs. This work had direct application to the verification of functional programs. Here we show how these ideas can also play an important role in the formal verification of imperative programs. While presented as an automatic technique we believe that our approach may be easily integrated within an interactive proof environment.

1 Introduction

The notion of an invariant is a well established technique for specifying the behaviour of computer programs. The discovery and verification of suitable invariants remains, however, a significant challenge to the formal verification community. While some believe that the challenge is too great [9] we feel there is still significant progress to be made in terms of mechanizing the wealth of heuristic knowledge which exists within the programming methodology literature [1, 8, 16, 22].

We focus here on the problems of reasoning about imperative programs, and in particular, the search control issues associated with the discovery and verification of loop invariants. In terms of formal verification the discovery of a loop invariant is typically seen as a eureka step. This is reflected in the fact that some of the foremost verification environments, *e.g.* GYPSY [12] and PENELOPE [17], rely on the user to supply all loop invariants.

We build upon *proof plans* [3], a meta-level reasoning technique for automating proof search. The technique is implemented in the CLAM theorem proving system [5]. Given a conjecture, CLAM attempts to automatically generate a LCF style tactic [15] guided by its available proof plans. If successful then the tactic is used to control proof construction in an object-level theorem prover. Currently CLAM is coupled to a tactic based theorem prover called OYSTER[1]. By explicitly separating the meta- and object-levels, CLAM provides an effective framework for developing and sharing proof strategies. That is, CLAM can potentially be used to support proof construction in any theorem proving environment which supports a tactic or proof script mechanism, *e.g.* LARCH [11] and HOL[2] [13].

Our approach to discovering loop invariants is based upon the analysis of failed proof attempts. Proof *critics* [19], an extension to the basic idea of a proof plan, support the analysis of proof failures. Previously this approach to automating formal reasoning has been applied very successfully within the context of inductive proof [20, 21]. This success exploited the constraints of *rippling* [4], a proof plan designed to guide step case proofs. The systematic analysis of partial success of rippling enabled us to

*The research reported in this paper was supported by EPSRC grant GR/L/11724 and ARC grant 438.

†A.Ireland@hw.ac.uk

‡jamie@cee.hw.ac.uk

[1] A Prolog implementation of NUPRL [7].

[2] Currently CLAM is being coupled to HOL through a collaborative research project between the Artificial Intelligence Department in Edinburgh and the Computer Laboratory in Cambridge funded by EPSRC grant GR/L/14381.

bridge gaps within the formal reasoning process, *e.g.* to automate the discovery of missing lemmata and generalizations.

Our contribution here is two fold. Firstly, we demonstrate that rippling can also play a role in automating the verification of loop invariants. Secondly, we show that the critics developed for inductive proof provide a basis for automating the discovery of loop invariants through the analysis of failed verification proofs.

Background material on reasoning about the behaviour of imperative programs is covered in §2 where we focus in particular on the while-loop construct. Building on the close connection between induction and iteration we show in §3 how a generalization of the ripple method enables its application to loop invariant proofs. Exploiting this connection further, we demonstrate in §4 how an existing generalization critic developed for inductive proof can be used as a basis for guiding the discovery of loop invariants automatically. Our implementation and experimental results are described in §5. Related work and future extensions are discussed in §6 and §7 respectively.

2 Reasoning about programs

We consider the problem of reasoning about the behaviour of imperative programs in the context of a Floyd-Hoare-style logic [10, 18]. Specifications in this logic take the form of a triple, *i.e.* $\{P\}C\{Q\}$, where P and Q denote first order assertions which express the pre- and post-conditions with respect to C, an imperative program. To illustrate, consider **exp1** given in figure 1 for computing exponentiation[3]. Note that we adopt the convention of using calligraphic letters to denote values while lower case letters are used to denote program variables. The process of verification is typically divided into three steps as described below:

1. **Assertion propagation:** using the given assertions and the axiomatic definition of the language constructs, intermediate program assertions are calculated. To illustrate, consider the rule defining the while-loop:

$$\frac{\vdash \{P \wedge S\}\ C\ \{P\}}{\vdash \{P\}\ \textbf{while}\ S\ \textbf{do}\ C\ \{P \wedge \neg S\}}$$

[3]Definitions are given in appendix A.

Note however that in order to apply this rule we must discover some property P which is invariant with respect to the execution of C, the body of the loop. In the case of the exponentiation algorithms an appropriate *loop invariant* takes the form:

$$r * exp(x, y) = exp(\mathcal{X}, \mathcal{Y}) \qquad (1)$$

This choice of P, in the case of **exp1**, gives rise to the following intermediate assertions:

```
{$r * exp(x,y) = exp(\mathcal{X},\mathcal{Y}) \wedge y > 0$}
begin
    $r := r * x$;
    $y := y - 1$
end
{$r * exp(x,y) = exp(\mathcal{X},\mathcal{Y})$}
```

Propagation of assertions is complete once all atomic program statements are assigned pre- and post-conditions.

2. **Verification condition generation:** given the pre- and post-conditions for an atomic program statement a purely mathematical statement of partial correctness is extracted. Such statements are called *verification conditions.* The verification condition generated by the assignment command, *i.e.*

$$\{P\}V := E\{Q\}$$

takes the form $P \rightarrow Q[E/V.]$ Based upon this schema the verification condition generated for the loop-body given above takes the form:

$$r * exp(x, y) = exp(\mathcal{X}, \mathcal{Y}) \wedge (y > 0) \rightarrow$$
$$(r * x) * exp(x, (y - 1)) = exp(\mathcal{X}, \mathcal{Y}) \qquad (2)$$

3. **Theorem proving:** verification of the overall program is based upon the verification of the atomic program statements. The verification task requires a theorem prover which can deal with implications such as (2).

In terms of calculating intermediate assertions a loop invariant represents a major bottle-neck to the formal verification process. Once the propagation of assertions is complete the generation of verification conditions is mechanical. Our overall approach is to use failure productively at step 3 in order to refine the choice of intermediate assertions, *i.e.* loop invariants, at step 1.

```
exp1:  {$x = \mathcal{X} \wedge y = \mathcal{Y}$}
       $r := 1$;
       while ($y > 0$) do
       begin
           $r := r * x$;
           $y := y - 1$
       end
       {$r = exp(\mathcal{X}, \mathcal{Y})$}
```

```
exp2:  {$x = \mathcal{X} \wedge y = \mathcal{Y}$}
       $r := 1$;
       while ($y > 0$) do
       begin
           if $odd(y)$ then $r := r * x$;
           $y := y \; div \; 2$;
           $x := x * x$
       end
       {$r = exp(\mathcal{X}, \mathcal{Y})$}
```

Figure 1: Algorithms for computing exponentiation

3 Verifying loop invariants

Proofs which share commonality of structure will also share common methods of proof. This is the premise on which the notion of a proof plan is based. Loop invariant proofs and the proofs of inductive conjectures share commonality of structure. Schematically, proving a loop invariant gives rise to a goal of the form

$$P \rightarrow P[E/V]$$

where E denotes the term assigned to the variable V. Now compare this with the schematic structure of a step case goal, *i.e.*

$$\frac{\ldots \; P \rightarrow P[E/V]}{\forall V. \; P(V)}$$

where E denotes the induction term and V the induction variable. Note that in both goals there exists a strong syntactic similarity between antecedent and consequent. The method of proof known as rippling was designed for step case goals. The commonality between step case and loop invariant proofs observed above led us to investigate the role in which rippling may play in guiding loop invariant proofs. Below we outline rippling and illustrate its applicability to step case and loop invariant proofs.

3.1 Rippling: the general pattern

Rewriting often involves the manipulation of a goal formula so that a target formula, *e.g.* a hypothesis or previously established result, can be applied. Rippling exploits syntactic similarities between the goal and target in guiding the selective application of rewrite rules. In particular it identifies term structure within the goal which prevents a match with the target. Such term structures are called *wave-fronts.* Conversely any term structure within the goal which corresponds to the target is called *skeleton.* In general, embedded within each wave-front will be parts of the skeleton term structure, these are known as *wave-holes.*

Wave-fronts, wave-holes and skeletons are meta-level notions. Given a schematic goal of the form $f(g(c(x), y), z)$ and a target of the form $f(g(x, y), z)$ then $c(\ldots)$ denotes a wave-front. We use a box and an underline to represent wave-fronts and wave-holes respectively, *e.g.* an annotated version of the goal given above takes the form:

$$f(g(\boxed{c(\underline{x})}^{\uparrow}, y), z)$$

The arrow is used to indicate the direction in which a wave-front can be moved with respect to the skeleton term structure. Directed wave-fronts enable the termination of rippling to be guaranteed [2]. The movement of wave-fronts is performed by *wave-rules*, a syntactic class of rewrite rule which preserves the skeleton while making progress to eliminating wave-fronts. Wave-rules are derived automatically from definitions and lemmata. Example wave-rules are provided in the following sections. Proofs guided by rippling can be classified in terms of the direction in which wave-fronts are moved with respect to the skeleton term structure. The three basic strategies for rippling are summarised in figure 2. For a completely formal account of rippling see [2, 4]. In general a proof may require a *hybrid* form of rippling. The

rippling-out:

$$\underset{\text{before}}{f_1(\ldots(f_n(\boxed{c_1(\underline{\ldots})}^{\uparrow}))\ldots)} \qquad \underset{\text{after}}{\boxed{c_n(\underline{f_1(\ldots(f_n(\ldots))\ldots)})}^{\uparrow}}$$

rippling-sideways:

$$\underset{\text{before}}{f_1(\boxed{c_1(\underline{\ldots})}^{\uparrow},\ldots,f_i(\ldots),\ldots)} \qquad \underset{\text{after}}{f_1(\ldots,\ldots,\boxed{c_i(\underline{f_i(\ldots)})}^{\downarrow},\ldots)}$$

rippling-in:

$$\underset{\text{before}}{\boxed{c_n(\underline{f_1(\ldots f_n(\ldots)\ldots)})}^{\downarrow}} \qquad \underset{\text{after}}{f_1(\ldots f_n(\boxed{c_1(\underline{\ldots})}^{\downarrow})\ldots)}$$

Figure 2: The three basic rippling strategies

notion of sideways and hybrid rippling is crucial to our work on invariant discovery. We begin by illustrating how rippling can guide both step case and loop invariant proofs in the following sections.

3.2 Rippling for step cases

Consider the following inductive conjecture:

$$\forall A, B, C : nat.\ mult(A, B, C) = (A * B) + C \qquad (3)$$

where $*$ and *mult* are defined in appendix A. Note that *mult* is tail-recursive. A proof of (3) may be constructed by structural induction on A giving rise to a step case goal in which we have an induction hypothesis of the form:

$$\forall B, C : nat.\ mult(a, B, C) = (a * B) + C \qquad (4)$$

The annotated version of the associated conclusion takes the form:

$$mult(\boxed{\underline{a}+1}^{\uparrow}, \lfloor b \rfloor, \lfloor c \rfloor) = (\boxed{\underline{a}+1}^{\uparrow} * \lfloor b \rfloor) + \lfloor c \rfloor \qquad (5)$$

A winning strategy in this context involves rippling-sideways. The motivation for rippling-sideways in this context is to direct wave-fronts towards term structures within the conclusion which correspond to universally quantified variables in the hypothesis. We call such term structures *sinks*, since they may "absorb" wave-fronts through the specialization of the induction hypothesis. We delimit sinks by the meta-level annotation $\lfloor \ldots \rfloor$. Note that the occurrences of b and c within (5) denote sinks since they correspond to the universal variables B and C within (4). The proof of (5) guided by rippling is given below:

$$\begin{aligned} mult(\boxed{\underline{a}+1}^{\uparrow}, \lfloor b \rfloor, \lfloor c \rfloor) &= (\boxed{\underline{a}+1}^{\uparrow} * \lfloor b \rfloor) + \lfloor c \rfloor \\ mult(a, \lfloor b \rfloor, \left\lfloor \boxed{b+\underline{c}}^{\downarrow} \right\rfloor) &= (\boxed{\underline{a * \lfloor b \rfloor} + b}^{\uparrow}) + \lfloor c \rfloor \\ mult(a, \lfloor b \rfloor, \left\lfloor \boxed{b+\underline{c}}^{\downarrow} \right\rfloor) &= (a * \lfloor b \rfloor) + \left\lfloor \boxed{b+\underline{c}}^{\downarrow} \right\rfloor \quad (6) \end{aligned}$$

Note that by instantiating B and C to b and $b + c$ respectively then (4) and (6) match which completes the proof. The wave-rules which are required[4] for this proof are as follows[5]:

$$\begin{aligned} \boxed{\underline{X}+1}^{\uparrow} * Y &\Rightarrow \boxed{\underline{(X * Y)} + Y}^{\uparrow} & (7) \\ mult(\boxed{\underline{X}+1}^{\uparrow}, Y, Z) &\Rightarrow mult(X, Y, \boxed{Y + \underline{Z}}^{\downarrow}) & (8) \\ \boxed{\underline{X}+Y}^{\uparrow} + Z &\Rightarrow X + \boxed{Y + \underline{Z}}^{\downarrow} & (9) \end{aligned}$$

Wave-rules (7) and (9) ripple the right-hand-side and wave-rule (8) is used on the left-hand-side. Here wave-rules (7), (8) and (9) are derived from equations (23), (25) and (26) respectively.

[4]Note that for a given theory all possible wave-rules are automatically derived and stored by CLAM.

[5]We use $\Rightarrow$ to denote rewrite rules and $\rightarrow$ to denote logical implication.

3.3 Rippling for loop invariants

We demonstrate the applicability of rippling to loop invariant proofs by considering verification condition (2) which gives rise to an invariant hypothesis of the form:

$$r * exp(x, y) = exp(\mathcal{X}, \mathcal{Y}) \tag{10}$$

The associated annotated version of the conclusion takes the form:

$$\boxed{\underline{r} * x}^{\uparrow} * exp(x, \boxed{\underline{y} - 1}^{\uparrow}) = exp(\mathcal{X}, \mathcal{Y}) \tag{11}$$

Again the winning strategy involves rippling-sideways. However, in the context of loop invariant proofs there are no universally quantified variables within the invariant hypothesis to exploit. The absence of such quantification precludes the use of the meta-level notion of a sink but not sideways rippling. Here the motivation to ripple-sideways is to achieve a match with the invariant hypothesis through wave-front cancellation, *i.e.* destructive interference. The proof of (11) guided by rippling is as follows:

$$\begin{aligned}
\boxed{\underline{r} * x}^{\uparrow} * exp(x, \boxed{\underline{y} - 1}^{\uparrow}) &= exp(\mathcal{X}, \mathcal{Y}) \\
r * \boxed{x * \underline{exp(x, \boxed{\underline{y} - 1}^{\uparrow})}}^{\downarrow} &= exp(\mathcal{X}, \mathcal{Y}) \\
r * exp(x, y) &= exp(\mathcal{X}, \mathcal{Y}) \qquad (12)
\end{aligned}$$

Note that (10) matches (12) and completes the proof. The wave-rules required for the proof of (11) are as follows:

$$\boxed{\underline{X} * Y}^{\uparrow} * Z \Rightarrow X * \boxed{Y * \underline{Z}}^{\downarrow} \tag{13}$$

$$Y > 0 \rightarrow \boxed{X * \underline{exp(X, \boxed{\underline{Y} - 1}^{\uparrow})}}^{\downarrow} \Rightarrow exp(X, Y) \tag{14}$$

Wave-rule (13) comes from (27) while (14) is derived from (24), the definition of *exp*. Note that (14) provides the destructive interference of wave-fronts mentioned above and is an example of a hybrid wave-rule.

3.4 Generalizing rippling

Two alternative preconditions for rippling-sideways were demonstrated above. While sinks do not play a role within loop invariant proofs the notion of wave-front cancellation is applicable to both domains. For instance when proving an inductive conjecture in which both constructor and destructor style definitions are intermixed then wave-front cancellation will be crucial. Consequently, a more general precondition, than described in [4], is required which takes into account the potential for term cancellation and hypothesis specialization. This is reflected in the generalized version of the rippling preconditions given in figure 3.

4 Discovering loop invariants

The full benefits of a proof plan become apparent when a proof attempt fails. The declarative nature of method preconditions provides a basis for using failure productively. As mentioned in §1 we analyse failure by means of proof critics. Experimental evidence demonstrating the merits of this approach within the context of inductive proof can be found in [21]. This work documented a family of critics associated with the ripple method. Here we focus on one of these critics, a critic for generalizing inductive conjectures. In the same way that rippling has been shown to be applicable to loop invariant proofs we now show how the generalization critic can be transferred to this new domain.

4.1 A generalization critic

The generalization critic is triggered by the failure of precondition 4 of rippling (see figure 3). In the context of a step case proof this corresponds to a *missing sink*. To illustrate consider the following modified version of conjecture (3):

$$\forall A, B : nat.\ mult(A, B, 0) = A * B \tag{15}$$

Consider an attempt to prove (15) by structural induction on A. Using rippling to guide the associated step case proof we obtain an induction hypothesis of the form:

$$\forall B : nat.\ mult(a, B, 0) = a * B$$

Using wave-rule (7) the annotated conclusion ripples to give:

$$mult(\underbrace{\boxed{\underline{a} + 1}^{\uparrow}}_{\text{blocked}}, b, 0) = \underbrace{\boxed{\underline{a * b} + b}^{\uparrow}}_{\text{blocked}} \tag{16}$$

No further rippling is possible on either side of the equation. Wave-rule (8) fails to apply on the left-hand side because of a missing sink. This suggests

Input sequent:

$$H \vdash G[f_1(\boxed{c_1(\underline{\ldots})}^{\uparrow}, f_2(\lfloor\ldots\rfloor), f_3(\boxed{c_2(\underline{\ldots})}^{\uparrow}))]$$

Method preconditions:

1. there exists a subterm T of G which contains wave-front(s), *e.g.*

   $f_1(\boxed{c_1(\underline{\ldots})}^{\uparrow}, f_2(\lfloor\ldots\rfloor), f_3(\boxed{c_2(\underline{\ldots})}^{\uparrow}))$

2. there exists a wave-rule which matches T, *e.g.*

   $C \to f_1(\boxed{c_1(\underline{X})}^{\uparrow}, Y, Z) \Rightarrow \boxed{c_5(\underline{f_1(X, \boxed{c_3(\underline{Y})}^{\downarrow}, \boxed{c_4(\underline{Z})}^{\downarrow})})}^{\uparrow}$

3. the wave-rule condition follows from the context, *e.g.*

   $H \vdash C$

4. resulting inward directed wave-fronts are potentially removable, *e.g.*

   $\ldots\boxed{c_3(\underline{f_2(\lfloor\ldots\rfloor)})}^{\downarrow}\ldots$ (sinkable) or

   $\ldots\boxed{c_4(\underline{f_3(\boxed{c_2(\underline{\ldots})}^{\uparrow})})}^{\downarrow}\ldots$ (cancellable)

Output sequent:

$$H \vdash G[\boxed{c_5(\underline{f_1(\ldots, \boxed{c_3(\underline{f_2(\lfloor\ldots\rfloor)})}^{\downarrow}, \boxed{c_4(\underline{f_3(\boxed{c_2(\underline{\ldots})}^{\uparrow})})}^{\downarrow})})}^{\uparrow}]$$

Figure 3: Generalized preconditions for rippling

Blockage:

$$mult(\boxed{\underline{a}+1}^{\uparrow}, \lfloor b \rfloor, 0) = \ldots$$

Critic preconditions:

- preconditions 1, 2 and 3 of rippling succeed, *i.e.*
  1. there exists a subterm which contains wave-front(s), *e.g.*

     $mult(\boxed{\underline{a}+1}^{\uparrow}, \lfloor b \rfloor, 0)$
  2. there exists a wave-rule which matches, *e.g.*

     $mult(\boxed{\underline{X}+1}^{\uparrow}, Y, Z) \Rightarrow mult(X, Y, \boxed{Y+\underline{Z}}^{\downarrow})$
  3. the wave-rule condition follows from the context, *e.g.*

     no condition
- precondition 4 fails, *i.e.*
  4. resulting inward directed wave-fronts are potentially removable, *e.g.*

     $mult(a, \lfloor b \rfloor, \underbrace{\boxed{b+\underline{0}}^{\downarrow}}_{\text{no sink}})$

Patch specification:
Coerce precondition 4 by introducing a new universally quantified variable into the conjecture, *e.g.*

$$mult(\boxed{\underline{a}+1}^{\uparrow}, \lfloor b \rfloor, F_1(\lfloor c \rfloor)) = \ldots$$

where F_1 denotes a second-order meta-variable.

Figure 4: Patch: introduction of sinks (primary)

Blockage:

$$\ldots = \boxed{a * \underline{\lfloor b \rfloor} + b}^{\uparrow}$$

Critic preconditions:

- precondition 1 of rippling succeeds, *i.e.*
    1. there exists a subterm which contains wave-front(s), *e.g.*

    $$\boxed{a * \underline{\lfloor b \rfloor} + b}^{\uparrow}$$

- precondition 2 of rippling partially succeeds, *i.e.*
    2. there exists a partial match with a sideways wave-rule, *e.g.*

    $$\boxed{a * \underline{\lfloor b \rfloor} + b}^{\uparrow}$$

    $$\boxed{\underline{X} + Y}^{\uparrow} + Z \Rightarrow X + \boxed{Y + \underline{Z}}^{\downarrow}$$

Patch specification:
Coerce precondition 2 by introducing additional skeleton term structure into the conjecture such that preconditions 3 and 4 will also potentially succeed, *i.e.*

$$\ldots = F_2(\boxed{\underline{a} + 1}^{\uparrow} * \lfloor b \rfloor, \lfloor c \rfloor)$$

where F_2 denotes a second-order meta-variable.

Figure 5: Patch: introduction of sinks (secondary)

patching the proof by introducing a sink occurrence within the third argument position of *mult*. On the right-hand side no wave-rules match. The blocked wave-front, however, does match with the wave-front associated with (9). This partial match with a sideways wave-rule suggests patching the proof by introducing additional skeleton term structure which contains a sink occurrence. We previously referred to these as *primary* and *secondary* ripples [20]. The associated critics are presented in figures 4 and 5 respectively. The resulting patches give rise to the following schematic induction hypothesis:

$$\forall B, C : nat.\ mult(a, B, F_1(C)) = F_2(a * B, C)$$

and a schematic conclusion of the form:

$$mult(\boxed{\underline{a} + 1}^{\uparrow}, \lfloor b \rfloor, F_1(\lfloor c \rfloor)) = F_2(\boxed{\underline{a} + 1}^{\uparrow} * \lfloor b \rfloor, \lfloor c \rfloor)$$

where F_1 and F_2 denote second-order meta-variables. Space prevents us from providing the complete proof, however, the constraints of rippling instantiate F_1 to be $\lambda X.X$ and F_2 to be $\lambda X.\lambda Y.X + Y$. For a detailed description of the constraint mechanism see [20]. The result of this patched proof attempt is to generalize (15) where the generalized conjecture corresponds to (3).

4.2 A loop invariant critic

The basic ideas underlying our generalization critic can be used to suggest loop invariants. We show how each of the failure patterns presented above maps onto a heuristic for discovering loop invariants.

Consider again **expl** given in figure 1. Previously we verified that (1) is invariant with respect to the associated while-loop. We now consider how this property can be discovered automatically.

4.2.1 Replacing constants by variables

As our starting point we use the following equation as a first approximation[6] to the loop invariant:

$$r * exp(x, \mathcal{Y}) = exp(\mathcal{X}, \mathcal{Y})$$

[6]This kind of invalid invariant might be supplied by the user of a development environment. The ability for a theorem prover to correct invalid invariants automatically would be of great benefit in such a context.

The corresponding verification condition gives rise to an annotated conclusion of the form:

$$\underbrace{\boxed{\underline{r} * x}^{\uparrow}}_{\text{blocked}} * exp(x, \mathcal{Y}) = exp(\mathcal{X}, \mathcal{Y})$$

No rippling is possible. The pattern of this failure is almost identical to the generalization critic given in figure 4. The patch differs in that we introduce an opposing wave-front rather a sink occurrence. The full critic is presented in figure 6 where the patch specified gives rise to a schematic invariant hypothesis of the form:

$$r * exp(x, F_1(r, x, y)) = exp(\mathcal{X}, \mathcal{Y}) \qquad (17)$$

The rippling of the associated schematic conclusion proceeds as follows:

$$\boxed{\underline{r} * x}^{\uparrow} * exp(x, F_1(\boxed{\underline{r} * x}^{\uparrow}, x, \boxed{\underline{y} - 1}^{\uparrow})) = exp(\mathcal{X}, \mathcal{Y}) \qquad (18)$$

$$r * \boxed{x * \underline{exp(x, F_1(\boxed{\underline{r} * x}^{\uparrow}, x, \boxed{\underline{y} - 1}^{\uparrow}))}}^{\downarrow} = exp(\mathcal{X}, \mathcal{Y})$$

$$r * exp(x, y) = exp(\mathcal{X}, \mathcal{Y})$$

As before the proof is achieved by wave-rules (13) and (14). However, as a side-effect of this ripple, F_1 is instantiated to be $\lambda X.\lambda Y.\lambda Z.Z$. Note that the wave-front annotations are used to constrain the instantiation of the second-order meta-variables. Propagating this instantiation through (17) gives rise to a revised invariant hypothesis which is identical to (1), the required invariant. Note that this form of invariant patching corresponds to the *replacement of constants by variables* heuristic which appears in the literature [16, 22].

4.2.2 Tail-invariants

We now use the given post-condition for **expl** (see figure 1) as our first approximation to the loop invariant, *i.e.*

$$r = exp(\mathcal{X}, \mathcal{Y})$$

The corresponding verification condition gives rise to an annotated conclusion of the form:

$$\underbrace{\boxed{\underline{r} * x}^{\uparrow}}_{\text{blocked}} = exp(\mathcal{X}, \mathcal{Y})$$

Blockage:

$$\boxed{\underline{r} * x}^{\uparrow} * exp(x, \mathcal{Y}) = \ldots$$

Critic preconditions:

- preconditions 1, 2 and 3 of rippling succeed, *i.e.*
  1. there exists a subterm which contains wave-front(s), *e.g.*

     $\boxed{\underline{r} * x}^{\uparrow} * exp(x, \mathcal{Y})$
  2. there exists a wave-rule which matches, *e.g.*

     $\boxed{\underline{X} * Y}^{\uparrow} * Z \Rightarrow X * \boxed{Y * \underline{Z}}^{\downarrow}$
  3. the wave-rule condition follows from the context, *e.g.*

     no condition
- precondition 4 fails, *i.e.*
  4. resulting inward wave-fronts are potentially removable, *e.g.*

     $\underbrace{\boxed{x * \underline{exp(x, \mathcal{Y})}}^{\downarrow}}_{\text{no opposing wave-front}}$

Patch specification:
Coerce precondition 4 by introducing new structure into the conjecture which contains an opposing wave-front, *e.g.*

$$\boxed{\underline{r} * x}^{\uparrow} * exp(x, F_1(\boxed{\underline{r} * x}^{\uparrow}, x, \boxed{\underline{y} - 1}^{\uparrow})) = exp(\mathcal{X}, \mathcal{Y})$$

where F_1 denotes a second-order meta-variable.

Figure 6: Patch: replacement of constants by variables

Blockage:

$$\boxed{\underline{r} * x}^{\uparrow} = exp(\mathcal{X}, \mathcal{Y})$$

Critic preconditions:

- precondition 1 of rippling succeeds, *i.e.*
    1. there exists a subterm which contains wave-front(s), *e.g.*

    $\boxed{\underline{r} * x}^{\uparrow}$

- precondition 2 of rippling partially succeeds, *i.e.*
    2. there exists a partial match with a sideways wave-rule, *e.g.*

    $\boxed{\underline{r} * x}^{\uparrow}$

    $\boxed{\underline{X} * Y}^{\uparrow} * Z \Rightarrow X * \boxed{Y * \underline{Z}}^{\downarrow}$

Patch specification:
Coerce precondition 2 by introducing additional skeleton term structure into the conjecture such that preconditions 3 and 4 will also potentially succeed, *i.e.*

$$F_1(\boxed{\underline{r} * x}^{\uparrow}, x, \boxed{\underline{y} - 1}^{\uparrow}) = exp(\mathcal{X}, \mathcal{Y})$$

where F_1 denotes a second-order meta-variable.

Figure 7: Patch: tail-invariant

Again no rippling is possible. This pattern of failure corresponds to the generalization critic given in figure 5. As previously the patch differs in that we introduce additional skeleton term structure in which an opposing wave-front rather a sink occurs. The complete critic is presented in figure 7 where the specified patch gives rise to a schematic invariant of the form:

$$F_1(r, x, y) = exp(\mathcal{X}, \mathcal{Y}) \tag{19}$$

The rippling of the associated schematic conclusion takes the form:

$$\begin{aligned} F_1(\boxed{\underline{r} * x}^{\uparrow}, x, \boxed{\underline{y} - 1}^{\uparrow}) &= exp(\mathcal{X}, \mathcal{Y}) \quad (20) \\ r * \boxed{x * \underline{F_2(\boxed{\underline{r} * x}^{\uparrow}, x, \boxed{\underline{y} - 1}^{\uparrow})}}^{\downarrow} &= exp(\mathcal{X}, \mathcal{Y}) \\ r * exp(x, y) &= exp(\mathcal{X}, \mathcal{Y}) \end{aligned}$$

As before this proof exploits wave-rules (13) and (14). This time, however, rippling instantiates F_1 to be $\lambda X.\lambda Y.\lambda Z.X * F_2(X, Y, Z)$ while F_2 is instantiated to be $\lambda X.\lambda Y.\lambda Z.exp(Y, Z)$. Propagating these instantiations through (19) gives rise to a refined invariant hypothesis which again is identical to (1), the required invariant. This property is a special case of what is known as a *tail-invariant*. Tail-invariants exploit the strong connection which exists between tail-recursive functions and while-loops. Although *exp* is not tail-recursive, as Kaldewaij [22] observes, any function definition which fits the following pattern strongly suggests the need for a tail-invariant:

$$\begin{aligned} b(x) \rightarrow g(x) &= a \\ \neg b(x) \rightarrow g(x) &= h(x) \oplus g(d(x)) \end{aligned} \tag{21}$$

where $\oplus$ is associative. Note that (21) gives rise to a hybrid wave-rule, *i.e.*

$$\neg b(x) \rightarrow \boxed{h(x) \oplus \underline{g(\boxed{d(\underline{x})}^{\uparrow})}}^{\downarrow} \Rightarrow g(x) \tag{22}$$

which is exactly the form of wave-rule required in order to achieve a successful sideways ripple in the context of a loop invariant proof.

5 Implementation and results

The work presented here has been implemented within the CLAM proof planner [5]. The implementation makes use of the higher-order features of λ-Prolog [24]. For our application we require only second-order unification. We build upon the technique described in [20] which uses rippling to constrain the unification process. Here we have generalized the technique so that it can deal with hybrid wave-rules. Previously it could deal with only single wave-fronts.

Our initial testing of the proof critics for loop invariant discovery have provided some promising results which are documented in table 1. We draw the reader's attention to the branching rate statistics. These statistics clearly show the significance of rippling in constraining the search space when rewriting in the presence of second-order meta-variables.

6 Related work

Early research into Automatic Programming investigated heuristic based methods for discovering loop invariants. Wegbreit [25] developed an approach in which a candidate invariant was incrementally refined using both domain-independent and domain-specific heuristics. The choice of refinement was guided by the satisfiability and validity of the current candidate invariant. The theorem proving and heuristic components were only loosely coupled. This was reflected in other heuristic approaches at the time [23]. Wegbreit hinted, however, that a closer relationship between the heuristic guidance and the theorem prover would be desirable. The proof planning framework in which our heuristics are expressed enables this close relationship to be exploited.

Our approach, like all heuristic based techniques, is not complete. A complete approach has been designed [6] based upon a novel unskolemization technique for deriving logical consequences of first-order formulae. Completeness, however, comes with a price. In practice this means that any inductive lemmas required for a particular verification task must be provided by hand. A strength of our approach is that we can attempt to automate the discovery of such inductive lemmas. I return to this point in §7.

As noted above the patch specified in figure 7 corresponds to the tail-invariant heuristic found in the literature. The heuristic is based upon observing recursive definitions which match the form of equation (21). Such definitions provide exactly the hybrid form of wave-rule which is required to achieve a successful sideways ripple. Wave-rules are not restricted,

ex	*loop invariant*		*critic*		*lemma*	*branching rate*			
	initial	*final*	*i*	*ii*		*MW*	*TW*	*MR*	*TR*
exp1	$r * exp(x, \mathcal{Y}) = exp(\mathcal{X}, \mathcal{Y})$	$r * exp(x, y) = exp(\mathcal{X}, \mathcal{Y})$	●		27	1	133	4	22
exp1	$r = exp(\mathcal{X}, \mathcal{Y})$	$r * exp(x, y) = exp(\mathcal{X}, \mathcal{Y})$		●	27	2	133	28	22
sum	$r + sum(\mathcal{X}) = sum(\mathcal{X})$	$r + sum(x) = sum(\mathcal{X})$	●		26	1	82	6	21
sum	$r = sum(\mathcal{X})$	$r + sum(x) = sum(\mathcal{X})$		●	26	3	82	25	21
fac	$r * fac(\mathcal{X}) = fac(\mathcal{X})$	$r * fac(x) = fac(\mathcal{X})$	●		27	1	124	6	28
fac	$r = fac(\mathcal{X})$	$r * fac(x) = fac(\mathcal{X})$		●	27	2	124	32	28
div	$r + div(\mathcal{X}, y) = div(\mathcal{X}, \mathcal{Y})$	$r + div(x, y) = div(\mathcal{X}, \mathcal{Y})$	●		28	2	106	5	19
div	$r = div(\mathcal{X}, \mathcal{Y})$	$r + div(x, y) = div(\mathcal{X}, \mathcal{Y})$		●	28	4	106	28	19
sumd	$r + sumd(\mathcal{X}) = sumd(\mathcal{X})$	$r + sumd(x) = sumd(\mathcal{X})$	●		26	1	135	4	27
sumd	$r = sumd(\mathcal{X})$	$r + sumd(x) = sumd(\mathcal{X})$		●	26	3	135	30	27
exp2	$r * exp(\mathcal{X}, \mathcal{Y}) = exp(\mathcal{X}, \mathcal{Y})$	$r * exp(x, y) = exp(\mathcal{X}, \mathcal{Y})$	●		27,29	1	264	14	27
exp2	$r = exp(\mathcal{X}, \mathcal{Y})$	$r * exp(x, y) = exp(\mathcal{X}, \mathcal{Y})$		●	27,29	4	264	37	27

Algorithms **exp1** and **exp2** are given in figure 1. The **sum** algorithm computes the summation of a natural number while **sumd** computes the sum of the digits of a natural number using the quotient and remainder operators for integer division. Repeated subtraction is used as the basis for **div** which computes integer division while **fac** uses repeated multiplication to compute factorials. The *loop invariant* columns show the candidate invariant as supplied to CLAM and the invariant resulting from the proof patching process. The candidates either correspond to the overall program post-condition or are potentially the result of user interaction. The *critic* columns refer to the *i) replacement of constants by variables* and *ii) tail-invariant* patches respectively. Non-definitional equations used in discovering invariants are indicated in the *lemma* column. The *branching rate* columns compare the applicability of wave-rules and rewrite rules with respect to the schematic conclusion generated by an invariant critic (see (18) and (20) in the case of **exp1**). Note that the branching rates relate to the first step in each proof attempt. The *MW* and *TW* columns give the number of wave-rule matches and the total number of wave-rules respectively. The *MR* and *TR* columns give the same information for rewrite rules. Note that in some cases the *MR* value is greater than the *TR* value. This occurs because some matches give rise to multiple unifiers. This is not the case with rippling since it provides greater constraints on the matching process.

Table 1: Results for Loop Invariant Critics

however, to recursive definitions. Properties such as associativity and distributivity are also a rich source of wave-rules. Consequently our rippling critics can be seen as a generalization of the tail-invariant heuristic described by Kaldewaij. To illustrate this point more precisely consider **exp2** given in figure 1. For this algorithm the definition of *exp*, which provides wave-rule (14), does not suggest an appropriate tail-invariant. The suggestion comes instead from (29), a property of *exp* which gives rise to the following hybrid wave-rule:

$$odd(Y) \rightarrow \boxed{X * \underline{exp(\boxed{\underline{X} * X}^{\uparrow}, \boxed{\underline{Y}\ div\ 2}^{\uparrow})}}^{\downarrow} \Rightarrow exp(X, Y)$$

Using this rule our technique discovers the required invariant automatically[7].

7 Future work

Building on the results presented here we are planning to undertake larger scale verification case studies. To achieve this goal we have begun to generalize our strategies to deal with nested loops and arrays.

As discussed in §4, the work described here is based upon the generalization critic developed for inductive proof. Other critics developed for inductive proof may also prove useful in guiding the discovery of loop invariants. For instance there exists a strong connection between the induction revision critic [21] and the *replacement of constants by terms* heuristic [16, 22] which we plan to investigate.

As hinted in §6 and demonstrated in [21] our heuristic approach provides the potential for automating the discovery of inductive lemmas such as (29). We aim to exploit this potential in future work.

Our long term aim is to develop new proof methods and critics for guiding the synthesis of imperative programs building upon the wealth of heuristic knowledge which exists within the literature.

8 Conclusion

A loop invariant represents a key eureka step in the verification of an imperative program. By generalizing the preconditions of rippling we have shown how it may be used to guide the verification of loop invariants.

Building upon critics developed for induction we have demonstrated how rippling can also be used to guide the discovery of loop invariants automatically. Second-order meta-variables play an important role in a critic's attempt to patch a proof. This new application has led to a generalization of the mechanism we use for constraining the instantiation of second-order meta-variables.

More generally, a major benefit of developing strategies within the proof planning framework is that our work will readily transfer to any formal development environment which uses a tactic or proof script based theorem prover.

[7]This example is based upon an exercise presented in [14] where the loop invariant is provided as a hint to the reader.

Acknowledgements

We would like to thank Lincoln Wallen for drawing our attention to the connection between tail-invariants and our generalization critic. Thanks also go to Alan Bundy for his encouragement with regard to this work.

Appendix: definitions

$$
\begin{array}{rcll}
0 * Y & = & 0 & \\
(X+1) * Y & = & (X * Y) + Y & (23) \\
X = 0 \rightarrow sum(X) & = & 0 & \\
X > 0 \rightarrow sum(X) & = & X + sum(X-1) & \\
X = 0 \rightarrow fac(X) & = & 0 & \\
X > 0 \rightarrow fac(X) & = & X * fac(X-1) & \\
Y = 0 \rightarrow exp(X,Y) & = & 1 & \\
Y > 0 \rightarrow exp(X,Y) & = & X * exp(X, Y-1) & (24) \\
mult(0, Y, Z) & = & Z & \\
mult(X+1, Y, Z) & = & mult(X, Y, Y+Z) & (25) \\
X < Y \rightarrow div(X,Y) & = & 0 & \\
X >= Y \rightarrow div(X,Y) & = & 1 + div(X-Y, Y) & \\
(X+Y)+Z & = & X + (Y+Z) & (26) \\
(X*Y)*Z & = & X * (Y*Z) & (27) \\
s(X)+Y & = & X + s(Y) & (28) \\
X = 0 \rightarrow sumd(X) & = & 0 & \\
X > 0 \rightarrow sumd(X) & = & (X\ mod\ 10) + sumd(X\ div\ 10) & \\
odd(B) \rightarrow exp(A,B) & = & A * exp(A*A, B\ div\ 2) & (29)
\end{array}
$$

References

[1] R.C. Backhouse. *Program Construction and Verification.* Prentice Hall, 1986.

[2] David Basin and Toby Walsh. A calculus for and termination of rippling. *Journal of Automated Reasoning*, 16(1-2):147–180, 1996.

[3] Alan Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988.

[4] Alan Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993.

[5] Alan Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449.

[6] Chadha and Plaisted. On the mechanical derivation of loop invariants. *JSL*, 15:705–744, 1993.

[7] R. L. Constable, S. F. Allen, H. M. Bromley, et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[8] E. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[9] E. W. Dijkstra. Invariance and non-determinacy. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, chapter 9, pages 157–165. Prentice-Hall, 1985.

[10] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, pages 19–32. American Mathematical Society, 1967.

[11] S. Garland, J. Stephen, and John V. Guttag. *A Guide to LP, The Larch Prover*, November 1991.

[12] D. I. Good. Mechanical proofs about computer programs. In C. A.R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, chapter 3, pages 55–75. Prentice-Hall, 1985.

[13] M. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis.* Kluwer, 1988.

[14] M. J. Gordon. *Programming Language Theory and its Implementation.* International Series in Computer Science. Prentice-Hall, 1988.

[15] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science.* Springer Verlag, 1979.

[16] David Gries. *The Science of Programming.* Springer-Verlag, New York, 1981.

[17] D. Guaspari, C. Marceau, and W. Polak. Formal verification of Ada programs. *IEEE Trans. on Software Engineering*, 16(9):1058–1075, September 1990.

[18] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.

[19] A. Ireland. The Use of Planning Critics in Mechanizing Inductive Proofs. In A. Voronkov, editor, *International Conference on Logic Programming and Automated Reasoning – LPAR 92, St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624, pages 178–189. Springer-Verlag, 1992.

[20] A. Ireland and A. Bundy. Extensions to a Generalization Critic for Inductive Proof. In M. A. McRobbie and J. K. Slaney, editors, *13th Conference on Automated Deduction*, pages 47–61. Springer-Verlag, 1996. Springer Lecture Notes in Artificial Intelligence No. 1104.

[21] A. Ireland and A. Bundy. Productive Use of Failure in Inductive Proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996.

[22] A. Kaldewaij. *Programming: The Derivation of Algorithms.* Prentice Hall, London, 1990.

[23] S.M. Katz and Z. Manna. A heuristic approach to program verification. In *Proceedings of IJCAI-73.*

[24] D. Miller and G. Nadathur. An overview of λProlog. In R. Bowen, K. & Kowalski, editor, *Proceedings of the Fifth International Logic Programming Conference/ Fifth Symposium on Logic Programming.* MIT Press, 1988.

[25] Wegbreit. Heuristic methods for mechanically deriving inductive assertions. In *Proceedings of IJCAI-73.*

PV: A Model-Checker for Verifying LTL-X Properties

Ratan Nalumasu Ganesh Gopalakrishnan

Department of Computer Science
University of Utah, Salt Lake City, UT 84112
{ratan,ganesh}@cs.utah.edu

Abstract

We present a verification tool *PV* (Protocol Verifier) that checks next-time free linear time temporal logic (LTL-X) properties using a new partial order reduction algorithm called *Two phase*. Two phase significantly reduces space and time requirements on many practically important protocols on which the partial order reduction algorithms implemented in previous tools [5,8,13] yield very little savings. In some cases, such as one version of the *invalidate* directory protocol of the Utah Avalanche multiprocessor, current tools did not finish their search while Two phase finished and discovered a bug that was missed by the unfinished runs of existing tools. Two phase's performance is largely due to the fact that it does not employ a *run-time proviso* such as used in other tools.

In [11], we motivated the problems caused by the proviso, presented Two phase, and proved that it preserves next-time free *safety* properties. In this paper, we provide a proof that the Two phase algorithm also preserves *all* next-time free LTL (LTL-X) properties. We also characterize the type of protocols that benefit from the Two phase search strategy and provide experimental evidence showing the advantages of its search strategy.

Keywords: Finite system verification, Explicit enumeration, Partial order reductions

1 Introduction

With the increasing scale of hardware systems and the corresponding increase in the number of concurrent protocols involved in their design, formal verification of concurrent protocols is an important practical need. Explicit state enumeration methods [3,4,7] have shown considerable promise in verification of real-world protocol verification problems and have been used with success on many industrial designs [15]. Using most explicit state enumeration tools, a concurrent system is modeled as a set of concurrent processes communicating via shared variables [4] and/or communication channels [7] executing under an interleaving model. An important run-time optimization called partial-order reductions [5, 13, 14] helps avoid having to examine all possible interleavings among processes, and is crucial to handling large models. For example, if two transitions x and y are "independent" (to be defined precisely in the Section 2), then from any state S it is sufficient to execute the two transitions in the order x, y or y, x. Partial order reductions attempt to explore only one of the above two orders.

In our research in system-level hardware design, specifically in the verification of cache coherence protocols used in the Utah Avalanche multiprocessor [1], we observed that existing tools that support partial-order reductions [5,8] failed to provide sufficient reductions. We traced this state explosion to their use of run-time *provisos* (explained later) in deciding which processes to run in a given state. This paper presents a new partial-order reduction algorithm called *Two phase* that, in most cases, outperforms all comparable algorithms, and is part of a new protocol verification tool called PV that finds routine application in our multiprocessor design project [1]. In some cases (*e.g.*, the *invalidate* protocol considered for use in the Avalanche processor), not only did PV's search finish when others' didn't, but it also found some bugs which the others missed in their incomplete search. In an earlier paper [11], we showed that Two phase preserves safety properties. In this paper, we prove that *Two phase* preserves all next-time free linear time temporal logic (LTL-X) properties. We also provide experimental results of running PV on a number of examples.

Supported in part by ARPA Order #B990 Under SPAWAR Contract #N0039-95-C-0018 (Avalanche), DARPA under contract #DABT6396C0094 (Utah Verifier), and NSF MIP MIP-9321836.

2 Background

Two phase and other partial order reduction algorithms depend on commutativity of transitions in the system to reduce the time and memory requirements of the verification job. Commutativity of transitions depends on the communication primitives provided by the modeling language. Typical communication primitives include shared memory (global variables), bounded buffers, unbounded buffers, and rendezvous communication. The PV tool supports communication through shared memory and bounded buffers. In this paper, for the sake of simplicity, we assume that there is exactly one shared variable (implicitly named g), and that there is no other means of communication. If there are multiple global variables in the model, logically they are grouped into a single global variable g. The Two phase algorithm and the proofs presented in this paper can also be easily modified to work when the processes communication using bounded buffers.

A process P_i is a tuple $(Q_i, \underset{l}{\overset{i}{\rightarrow}}, \underset{g}{\overset{i}{\rightarrow}}, q^i_{init}, \Sigma, \sigma_{init})$ where Q_i is a finite set of states that P_i can assume, $q^i_{init} \in Q_i$ is the initial state of P_i, Σ is the range of an implicit global variable g, and $\sigma_{init} \in \Sigma$ is the initial value of g. $\underset{l}{\overset{i}{\rightarrow}}$ is a relation from Q_i to Q_i indicating a set of local transitions of P_i, i.e., the set of transitions that do not depend on g in any way. The relation $\underset{g}{\overset{i}{\rightarrow}}$ from $Q_i \times \Sigma$ to $Q_i \times \Sigma$ is a set of global transitions, i.e., the set of transitions of P_i that read/write the global variable g. For convenience, we write $(q, \sigma)\overset{i}{\rightarrow}(q', \sigma')$ to indicate that (a) $q \underset{l}{\overset{i}{\rightarrow}} q' \wedge \sigma = \sigma'$ or (b) $(q, \sigma) \underset{g}{\overset{i}{\rightarrow}} (q', \sigma')$. We also assume that the domains of $\underset{l}{\overset{i}{\rightarrow}}$ and $\underset{g}{\overset{i}{\rightarrow}}$ are *disjoint* and that the union of the two domains is Q_i[1].

To find the full state space generated by interleaved execution of processes $P_1, P_2, \cdots, P_n$, referred to as the graph $G_f = (V_f, E_f)$, following graph traversal algorithm can be used:

- $(q^1_{init}, q^2_{init}, \cdots, q^n_{init}, \sigma_{init}) \in V_f$,
- if $Q = (q_1, q_2 \cdots, q_i, \cdots, q_n, \sigma) \in V_f$ and $(q_i, \sigma)\overset{i}{\rightarrow}(q_i', \sigma')$ then $Q' = (q_1, q_2, \cdots, q_i', \cdots, q_n, \sigma') \in V_f$ and $(Q, Q') \in E_f$. (Q' is called a P_i *successor* of Q).

[1]Domains of $\underset{l}{\overset{i}{\rightarrow}}$and $\underset{g}{\overset{i}{\rightarrow}}$are disjoint means that if in a non-deterministic state some of the transitions depend on g and while others do not, then we treat as though *all* transitions depend on g.

NOTATION: To simplify presentation of the algorithm and proofs, we write "$Q \rightarrow Q'$" to indicate that $(Q, Q') \in E_f$ and "$Q \rightarrow$" to indicate the set $\{Q' | (Q, Q') \in E_f\}$. Similarly, for any graph $G = (V, E)$, we write $\ll Q_0, Q_1, \ldots, Q_n \gg \in E$ to denote $(Q_i, Q_{i+1}) \in E$ for $i \in \{0..n-1\}$. If $Q = (q_1, q_2, \ldots, q_i, \ldots, q_n, \sigma)$, $Q_i(Q) = q_i$ and $\Sigma(Q) = \sigma$.

As mentioned earlier, partial order reductions depend on the notion of commutativity of transitions to obtain a subgraph G_r of G_f while preserving the properties of interest. In particular, we note that all transitions in $\underset{l}{\overset{i}{\rightarrow}}$ commute with all the transitions in $\underset{l}{\overset{j}{\rightarrow}}$and $\underset{g}{\overset{j}{\rightarrow}}(j \neq i)$. If two transitions $l_i \in \underset{l}{\overset{i}{\rightarrow}}$ and $g_j \in \underset{g}{\overset{j}{\rightarrow}}$ are enabled from a given state S, then the partial order reduction algorithm may execute l_i from S and postpone the execution of g_j from S. Of course, special care must be taken to ensure that g_j is not perpetually postponed (referred to as "ignoring problem").

DEFINITION 1 (LOCAL) A state $q \in Q_i$ is said to be local if it is in the domain of $\underset{l}{\overset{i}{\rightarrow}}$, i.e., there is at least one q' such that $q \underset{l}{\overset{i}{\rightarrow}} q'$. When P_i is in a *local* state, all of its transitions commute with transitions of other processes.

DEFINITION 2 (DETERMINISTIC) A state $q \in Q_i$ is said to be deterministic if it is local and the image of q via $\underset{l}{\overset{i}{\rightarrow}}$is singleton, i.e., if $q \underset{l}{\overset{i}{\rightarrow}} q'$ and $q \underset{l}{\overset{i}{\rightarrow}} q''$ then $q' = q''$. In this case, refer to q' as **next**(q). While other partial order reduction algorithms depend on the notion of *local* to reduce the graph, the Two phase algorithm uses *deterministic*.

Definitions of *local* and *deterministic* are extended to a concurrent system as follows:

DEFINITION 3 (LOCAL) A process P_i is said to be local in state $(q_1, q_2, \ldots, q_i, \ldots, q_n, \sigma)$ iff q_i is local in P_i.

DEFINITION 4 (DETERMINISTIC) A process P_i is deterministic in $Q = (q_1, q_2, \ldots, q_i, \ldots, q_n, \sigma)$ iff q_i is deterministic in P_i. In this case, we write **deterministic**(i,Q). In this case, we use **next**(i,Q) to indicate the state generated by moving P_i, i.e., if **next**$(q_i) = q_i'$, **next**(i,Q)=$(q_1, q_2 \ldots, q_i', \ldots, q_n, \sigma)$. **next**(i,$Q$) is called *deterministic successor* of Q.

Note that if a process P_i is local in Q, then executing a process P_j $(j \neq i)$ does not affect P_i in anyway.

DEFINITION 5 (GLOBAL PROPERTY) A global property is a LTL-X formulae that involves only the global variable g. Example: $\Box\Diamond g = 1$. We are interested in preserving the truth value of all such LTL-X properties.

Related Work

To determine whether a global property of the above form is true or not, many times it is not necessary to construct the entire G_f, which can be extremely large. Partial order reductions attempt to generate a much smaller subgraph of G_f, called reduced graph G_r, that satisfies the property iff the property is satisfied by G_f. As mentioned earlier, these algorithms [5, 11, 13, 14] attempt to generate G_r by exploiting the fact that when a process P_i is in a *local* state, its transitions commute with the transitions of P_j ($j \neq i$). When P_i is local from a state S that is about to be expanded by the graph generating algorithm, the partial order algorithm may, for example, select to expand transitions of process P_i only, and postponing transitions of all other processes. Of course, special care must be taken to ensure that no enabled transition is indefinitely postponed.

Previous algorithms such as [5] and [13] use proviso[2] to ensure that no transition is indefinitely postponed ("ignoring problem"). Both these algorithms (as well as *Two Phase*) use a stack to maintain the list of states currently being expanded. At every step, the algorithm chooses the subset of the transitions that need to be expanded from the current top of the stack. [5] and [13] algorithms require that the subset of transitions selected to explore do not result in a state that is already on the stack[3]. This condition, referred to as proviso, in some cases, causes G_r to be quite large (approaching the size of G_f) even though a much smaller G_r can be computed [11]. The Two phase algorithm attempts to rectify this problem by avoiding the proviso and using a different search strategy.

A simple scenario highlighting the difference between the Two phase algorithm and other partial order reduction algorithms is shown in Figure 1. S_1 is the initial state of the system; and processes P1 and P2 engage in a loop between S_1 and S_n, and all these moves of P1 and P2 commute with moves of another process P3. P3 has one enabled transition at every state in the path $S_1 \ldots S_n$. Partial order reduction algorithms that employ proviso may postpone the transitions of P3 between S_1 and S_{n-1}, but not at S_n. In other words, proviso forces the algorithm to take the thick transition from S_n to R_n instead of resetting the state of P_1 by executing the dotted transition from S_n to S_1. In this example, the algorithm generated R_n instead of R_1, but in a more complex situation where the proviso gets invoked often, not only does algorithm generate both R_1 and R_n, *but it also generates many other states that are equivalent except that state of one or more processes is not reset*, causing the G_r to be quite large. Our two-phase algorithm avoids generating such equivalent states by *not* taking the thick transition, but by taking the dotted lines from S_n to S_1 and from S_1 to R_1. This strategy, in many examples, reduces the size of G_r.

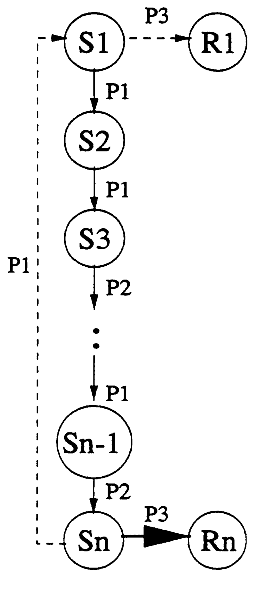


Figure 1: An algorithm that uses proviso generates the thick transition while the two-phase algorithm generates the dotted transitions.

3 Two Phase Algorithm

The two phase algorithm is shown in Figure 2. Unlike previous algorithms, this algorithm does not use proviso. Instead its execution is divided into two phases. In phase I, the algorithm executes transitions of *deterministic* processes. In this phase, the algorithm maintains a list of states visited in variable `list`. Without this variable, if a process is in a determinis-

[2] The provisos differ slightly depending on whether they preserve LTL-X or safety only.

[3] The proviso used in the two algorithms differ slightly because [5] preserves only next-time free safety properties while [13] preserves LTL-X. However, the effect of proviso is very similar on performance of both algorithms.

```
    init stack to contain initial state
    init $V_r$ to $\Phi$
    init $E_r$ to $\Phi$

    Two_phase()
    {
1       s := top(stack);
2       list := {s};
3       /* Phase I: partial order step */
4        for i := 1 to nprocesses {
5          while (deterministic(i,s)) {
6              /* Execute the only enabled
7                 transition of $P_i$ */
8              $E_r$ := $E_r$ + {(s, next(i,s))};
9              s := next(i, s);
10             if (s $\in$ list) goto NEXT_PROC;
11             list :=  list + {s};
12         }
13         NEXT_PROC: /* next i */
14      }
15      /* Phase II: classical DFS */
16      if (s $\notin$ $V_r$) then {
17         $V_r$ := $V_r$ + list;
18         $E_r$ := $E_r$ + {(s,t) | s$\rightarrow$t};
19         for each succ in s$\rightarrow$ {
20             if(succ$\notin V_r$) then {
21                 push(succ, stack);
22                 Two_phase();
23                 pop(stack);
24             }
25         }
26         assert(s$\rightarrow$ $\subseteq$ $V_r$);
27      } else {
28          $V_r$ := $V_r$ + list;
29      }
    }
```

Figure 2: Two_phase() avoids proviso using a different execution strategy.

tic loop, the while loop in the first phase would not terminate (see Lemma 4.) In the second phase, `list` is added to the V_r. In phase II, the algorithm first checks if `s` is already in V_r. If it is not in V_r, `s` is fully expanded (lines 19–25). If `s` is already in V_r, then either `s` has already been expanded or a deterministic successor of `s` has been expanded. Hence, it is not necessary to explore `s` (line 28) and the recursive call terminates.

As an example, consider the protocol shown in Figure 3(a). On this protocol, the state space generated by an algorithm using the proviso [13] (implemented in SPIN [9]) is shown in Figure 3(c). As can be seen, when the search reaches the state `<S1, S0>`, due to the proviso, the algorithm selects process P2 and generates `<S1, S1>`. In fact, since state `<S0, S0>` is in the stack until the entire graph generation is completed, proviso is invoked many times, causing the algorithm to generate all states in the system (though not all edges).

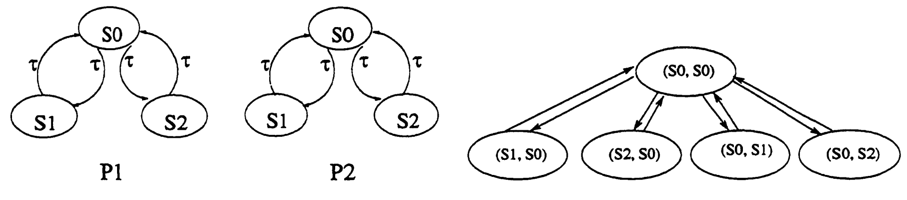
Two phase avoids this problem by completely avoiding the proviso. Instead, it depends on *deterministic* states such as S1 and S2 in P1 and P2 to bring the reductions. The reduced graph constructed by the Two phase algorithm on the same protocol is shown in Figure 3(b). In this protocol, the initial state `<S0, S0>` is not deterministic with respect to either of the processes. Hence no states are generated in Phase I of the algorithm. In phase II, `<S0, S0>` is completely expanded, resulting in four states `<S1, S0>`, `<S2, S0>`, `<S0, S1>`, and `<S0, S2>`. Then *Two_phase* is called on all these four states. Let us consider the case of expanding `<S1, S0>`. Expansion of other states is similar. Process P1 is deterministic in `<S1, S0>`, and hence P1 is executed in phase I, resulting in `<S0, S0>`. At this point because neither P1 nor P2 is deterministic in `<S0, S0>` phase I terminates, and in phase II, the algorithm would discover that `<S0, S0>` is already in V_r, and the recursive call terminates, resulting in the state graph shown in Figure 3(b). As can be seen, Two phase generates a much smaller graph as a direct result of avoiding the proviso.

In Figure 2 *all* intermediate states are added to `list` on line 11. Note that the only reason for maintaining `list` is to ensure that the `while` loop in Phase I terminates. Instead of adding all the intermediate states, we can add only a *subset* of the intermediate states to `list` and still guarantee that the `while` loop terminates. For example, one can add `s` to `list` only when the new value of `s` is bit-wise smaller than the value of old `s`. In this case, of course, line 8 also has to be modified so that E_r does not contain edges between states that are not present in V_r. This technique constitutes a simple form of selective caching, which is also supported by PV.

4 Proof of Correctness

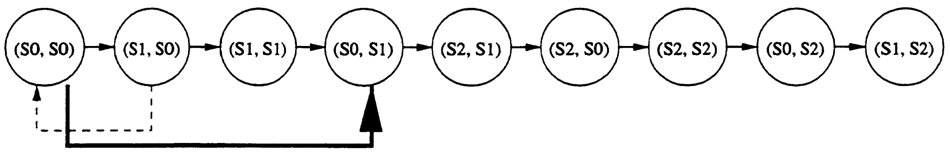
To show the Two phase algorithm preserves all global properties (Definition 5), we show that when the algorithm postpones execution of a transition x of P_i from a state S_0, there is a sequence $\ll S_0, S_1, \ldots, S_t, S' \gg \in E_r$ such that

- x is executed from S_t resulting in state S'
- each S_i is obtained as a deterministic successor of S_{i-1} ($i \in \{1 \ldots t\}$), and hence the value of the global variable g is same for all the states in this sequence



(a) Best case

(b) State space by 2 phase



(c) State space generated by SPIN tool

Figure 3: Best case protocol. (a) The protocol. (b) State space that generated by the *Two_phase* algorithm. (c) State space generated by SPIN. The dotted line in (c) show one of the transitions that were not attempted due to the proviso. The thick line in (c) shows one of the transitions that would be taken by the algorithm, and truncated since the state (S0, S1) has already been generated.

Note that, if executing the transition x from S_0 results in a state R, sequences $\ll S_0, \ldots, S_t, S' \gg$ and $\ll S_0, R \gg$ satisfy the same global properties.

LEMMA 1 The **assert** statement (line 26) in *Two_phase* holds. Due to the assignment on line 18, during the construction of the reduced graph, E_r may contain edges of the form (x, y) where y is not yet in V_r, at the time of termination y is added to V_r, and hence $E_r \subseteq V_r \times V_r$.

Proof: To prove that the **assert** statement holds, we note that in every iteration, **list** contains **top(stack)**, and **list** is added to V_r in every call to *Two_phase*. Thus, at the end of each call, **top(stack)** is in V_r. Since the recursive calls to *Two_phase* in the **for each** loop (line 19) make each member of $s \rightarrow$ that is not already in V_r to be the top of the stack, the **assert** statement holds.

LEMMA 2 Let $S = (q_1, \cdots, q_{i-1}, q_i, q_{i+1}, \cdots, q_n, \sigma)$ be a member of **list** that is added to V_r on line 17. If $x = (q_i, \sigma) \xrightarrow{i} (q_i', \sigma')$ is a transition of P_i then after executing line 18, $\exists \ll S_0, S_1, \ldots, S_t, S' \gg \in E_r$ such that (a) $S_0 = S$, (b) $Q_i(S_0) = Q_i(S_1) = \ldots = Q_i(S_t) = q_i$, (c) $\Sigma_i(S_0) = \Sigma_i(S_1) = \ldots = \Sigma_i(S_t) = \sigma$, (d) $Q_i(S') = q_i'$ and $\Sigma(S') = \sigma'$ (i.e., S' is a P_i successor of S_t, S' is generated by the transition x from S_t). In other words, every change in g along with the change of the state of every process P_i from q_i to q_i' in the full state space G_f is reflected in the reduced state space G_r if S is added in line 17.

This situation is shown in Figure 4.

Proof: In phase I, **list** is constructed starting from **top(stack)** by examining only deterministic processes. Let r_0 =**top(stack)**, and the order of the states added to **list** in the **while** loop of phase I be $r_1, r_2, \ldots, r_m$=**s**. In the **then** clause of phase II, all elements of **list**, viz. $r_0, r_1, \ldots, r_m$, are added to V_r. Let $S = r_j$ for some $j \in \{0 \cdots m\}$ (see Figure 4).

There are two cases to consider. (i) P_i is executed while generating the sequence $\ll r_j, r_{j+1}, \ldots, r_m \gg \in E_r$ (this corresponds to the case where the dashed transition from r_k of Figure 4 is added the reduced graph), and (ii) P_i is not executed while generating the sequence $\ll r_j, r_{j+1}, \ldots, r_m \gg \in E_r$ in phase I (this corresponds to the case where the thick transition from r_m is added to the reduced graph.) In the first case, since P_i is executed in phase I, we can conclude that P_i is deterministic in r_j. Let P_i be first executed at state $r_k \in \{r_j, \ldots, r_{m-1}\}$ resulting in state r_{k+1}. In this case, the lemma holds with $\ll S_0, S_1, \ldots, S_t, S' \gg = \ll r_j, r_{j+1}, \ldots, r_k, r_{k+1} \gg$.

If P_i is not executed while generating the sequence $\ll r_j, r_{j+1}, \ldots, r_m \gg$ we will show that the lemma holds with $\ll S_0, S_1, \ldots, S_t, S' \gg = \ll r_j, r_{j+1}, \ldots, r_m, S' \gg$ where S' is generated from r_m

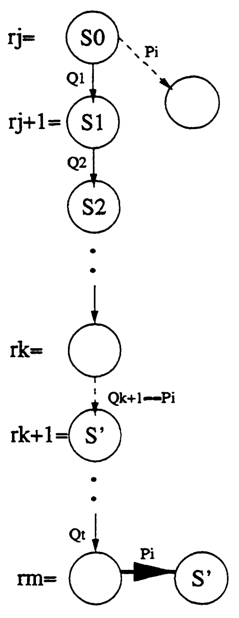


Figure 4: State S_0 is added to V_r on line 17. The dashed edge from S_0 is in the full graph. $\ll S_0, S_1 \dots r_m \gg$ is in E_r and all these states are in `list` when S_0 is added to V_r. Lemma 2 shows that after executing line 17, either the dashed edge from r_k to r_{k+1} or the thick edge from r_m is present in the reduced graph.

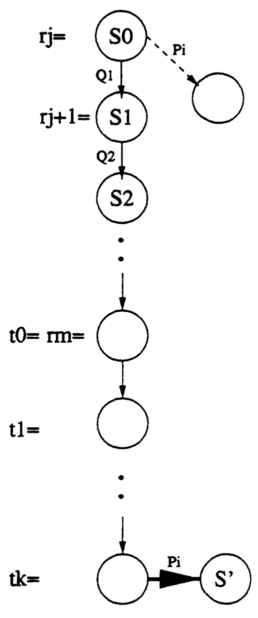


Figure 5: State S_0 is added to V_r on line 28. The dashed edge from S_0 is in the full graph. Case 2 of Lemma 3 shows when S_0 is added to V_r, the $\ll S_0, S_1 \dots r_m, t_1, \dots t_k \gg$ is in E_r and states $r_j \dots r_m$ are in `list`, and the thick edge from t_k to S' is present in the reduced graph.

by x. Since P_i is not executed while generating $r_{j+1}, \dots, r_m$, and g is not assigned in Phase I, we can conclude that r_j and r_m agree on the i^{th} component of the state (i.e., $Q_i(r_j) = Q_i(r_m) = q_i$), and on g (i.e., $\Sigma(r_j) = \Sigma(r_m) = \sigma$). Since $\mathtt{s} = r_m = S_t$ is fully expanded in phase II (lines 19–25), x is executed at `s`. S' is obtained by executing x from S_t. Due to the assignment to E_r on line 18, the edge in (S_t, S') are in E_r. Due to the assignment to E_r on line 8 Thus in this case the lemma holds with $\ll S_0, S_1, \dots, S_t, S' \gg = \ll r_j, r_{j+1}, \dots, r_m, S' \gg$.

LEMMA 3 Let $S = (q_1, \dots, q_{i-1}, q_i, q_{i+1}, \dots, q_n, \sigma)$ be a member of `list` that is added to V_r on line 28. If $x = (q_i, \sigma) \xrightarrow{i} (q_i', \sigma')$ is a transition of P_i then when S is added to V_r, (a) $\exists \ll S_0, S_1, \dots, S_t, S' \gg \in E_r$ (a) $S_0 = S$, (b) $Q_i(S_0) = Q_i(S_1) = \dots = Q_i(S_t) = q_i$, (c) $\Sigma_i(S_0) = \Sigma_i(S_1) = \dots = \Sigma_i(S_t) = \sigma$, (d) $Q_i(S') = q_i'$ and $\Sigma(S') = \sigma'$ (i.e., S' is a P_i successor of S_t, S' is generated by the transition x from S_t).

This is the similar to Lemma 2 except that it deals with the states added in the outermost else clause.

Proof: The proof is based on the induction on the call at which the S is added to V_r.

Induction basis: During the first call to *Two_phase*, the outer **then** clause of phase II is executed. Since line 28 is not executed, the lemma holds vacuously.

Induction hypothesis: Let the states added in the **else** clause during the first l calls of *Two_phase* satisfy the lemma.

Induction step: Let S be added during the $l+1^{th}$ call. If the **then** clause is executed in this call, the lemma holds vacuously. Otherwise, let `list` contain states `top(stack)`$=r_0, r_1, \dots, r_m=$`s` at the end of phase I of $l+1^{th}$ call. Let $S = r_j$ for some $j \in \{0 \cdots m\}$. There are two cases to consider.

Case(i): P_i is executed while generating the sequence $\ll r_j, r_{j+1}, \dots, r_m \gg \in E_r$ By reasoning as in the first case of Lemma 2, we can show that this lemma holds.

Case(ii) (Figure 5): P_i is not executed while generating the sequence $\ll r_j, r_{j+1}, \dots, r_m \gg \in E_r$ in phase I. Since the global variable g is not assigned in phase I, $\Sigma(r_m) = \Sigma(r_j = S) = \sigma$. Since the **else** clause is executed, $\mathtt{s} = r_m$ must have been added to

V_r either in the **then** clause of a previous call or in the **else** clause of a previous call. If it was added in the **then** clause, by Lemma 2 we can find a sequence $\ll t_0, t_1, \ldots, t_k, S' \gg$ where $t_0 = r_m$. On the other hand, if it was added in the else clause, by induction hypothesis, we can find sequence $\ll t_0, t_1, \ldots, t_k, S' \gg$ where $t_0 = r_m$. In both cases, the lemma holds with $\ll S_0, S_1, \ldots, S_t, S' \gg = \ll r_j, r_{j+1}, \ldots, r_m, t_1, \ldots, t_k, S' \gg$.

LEMMA 4 (TERMINATION) *Two_phase* terminates after a finite number of calls.

Proof: There are two parts to the proof of termination: (a) eventually no new calls to *Two_phase* are made, and (b) the `while` loop in the phase I terminates. To prove (a), note that new calls to *Two_phase* are made only in the body of the outer "if" statement in the second phase. Before these calls are made, all elements of `list` are added to V_r. The precondition to execute the "if" statement is that `s` is not in V_r. By construction of `list`, `s` is in `list`. Thus the number of states in V_r increases at least by one as a result of adding `list` to V_r. In other words, if number of states in V_r before the i^{th} level call of *Two_phase* is made is k, then the number of states in V_r before $i + 1^{th}$ level call of *Two_phase* is made is at least $k + 1$. Thus the maximum depth of calls to *Two_phase* cannot exceed the number of states in the protocol, which is finite. To prove (b), note that one new state is added to `list` in each iteration of `while` loop. Again, since the number of states in the protocol is finite, eventually no new states can be added to `list`, thus the `while` loop terminates.

DEFINITION 6 (EQUIVALENT TRACES) Two traces T and T' are said to be equivalent iff they satisfy the same set of global properties (Definition 5). In other words, T and T' are stutter equivalent when they are projected on to the global variable g.

Let $T = \ll S_0, \ldots, S_l \gg$ be a trace (of G_r or G_f depending on the context) where each S_i is obtained by executing a transition s_i from S_{i-1} ($i \in \{1..l\}$). The trace T may be equivalently represented by the transition sequence "$\ll s_1, \ldots, s_l \gg$ starting from S_0". Unless there is any confusion, we refer to both the state sequence and its transition sequence representation as simply "sequence".

Note that if two transitions x, y appear adjacent to each other in the transition sequence of T, x and y are independent of each other (i.e., at least one of the two transitions is *local* and they belong to different processes), and T' is identical to T except that the order of x and y is interchanged, then T and T' are equivalent traces. Similarly, if z is a deterministic transition that never appears in T, and T' is obtained by prepending x to T, then also T and T' are equivalent. We use these two properties to show that the reduced graph G_r satisfies a global property iff the full graph G_f does. We show this by demonstrating that for every trace T_f of G_f there is an equivalent trace T_r in G_r.

THEOREM 1 (LTL-X) The graph constructed by *Two_phase*, $G_r = (V_r, E_r)$, satisfies a LTL-X property P involving only the global variable g iff G_f satisfies P.

Proof: We show that for every sequence T_r of G_r, we can find an equivalent sequence T_f in G_f and vice versa. It is easy to see that G_r is a subgraph of G_f. Hence, every trace of G_r is also a trace of G_f.

Now, we will show that for every trace in G_f, there is a corresponding trace in G_r. First, we observe that at the time of termination, initial state is in V_r. From Lemmas 2 and 3 we can conclude that any sequence of changes in g that is present in G_f is also present in G_r. Together, they imply that every trace of G_f is also a trace of G_r.

A formal proof that for every trace T_f in G_f there is a corresponding trace T_r in G_r is a little more involved. For simplicity, we first show that for every fair path [10] of G_f there is an equivalent trace in G_r. Let T_f be a fair path in G_f. By Lemmas 2 and 3, we can conclude that *no* transition is indefinitely postponed. Hence, by reordering the transitions in T_f, we can obtain an equivalent sequence T_r in G_r. Hence, both G_r and G_f satisfy the same fair LTL-X formulae.

If T_f is not fair, we may not be able to construct T_r by simply reordering the transitions in T_f. This happens, for example, when a state S that appears in both T_f and T_r, x is a deterministic transition of P_i, x is enabled in S, that P_i is not executed at all after S in T_f, and x is executed at S in Phase I of Two phase. In this case all traces of G_r starting from S contain x. But since T_f does not contain x, we cannot construct T_r by simply reordering the transitions in T_f. T_r, instead, is obtained by prepending x to T_f and then reordering the adjacent independent transitions in T_f. Hence for every T_f we can construct an equivalent T_r by adding deterministic transitions to T_f and reordering independent transitions. Thus we can conclude that Two phase preserves all LTL-X properties involving only g.

5 Experimental Results

The Two phase algorithm outperforms the proviso algorithm [13] and a similar algorithm implemented in PO-PACKAGE[4] [5] when the proviso is invoked often. In most reactive systems, a transaction typically involves a subset of processes. For example, in a server/client model of computation, a server and a client may communicate without any interruption from other servers or clients to complete a transaction. After the transaction is completed, the state of the system is reset to the initial state. If the partial order reduction algorithm uses the proviso, state resetting cannot be done as the initial state will be in the stack until the entire reachability analysis is completed. Since at least one process is not reset, the algorithm generates unnecessary states, thus increasing the number of states visited. In other realistic systems also the number of extra states generated due to the proviso can be high. Two phase does not use the proviso, thus avoiding generating the extra states.

Table 1 shows results of running the [13] algorithm implemented in SPIN [9], Two phase with selective state caching disabled, and Two phase with selective caching enabled on various protocols. This table shows number of states in V_r and time taken in seconds to complete the graph construction on a Super Sparc 20. All verification runs are limited to 64MB so that the entire graph would fit in physical memory. Protocols B5–B7 are best case protocol in Figure 3 with N=5, 6, and 7. Protocol W5–W7 is example of a protocol that runs better with SPIN search algorithm. On this protocol, the proviso is never invoked, and this protocol has no deterministic states. As a result the Two phase degenerates to full state space, while SPIN reduces the number of states appreciably (from 3^n states to $2^{n+1} - 1$ states for where n=5, 6, or 7).

Mig and *inv* are two cache coherency protocols used in Avalanche [1]. On *inv*, SPIN fails to complete the graph construction in 64MB of memory. On the other hand, PV tool finishes comfortably generating 255,781 states (without selective caching) or 135,404 states (with selective caching). *SC* is a server/client protocol. This protocol consists of n servers and n clients. A client chooses a server and requests for a service. A service consists of a two round trip messages between server and client and some local computations. As can be seen, SPIN cannot handle 4 servers and 4 clients; it aborts search after generating 260,928 states, while PV finishes generating 59,436 states (without selective caching) or 14,173 states (with selective caching).

Pftp and *snoopy* protocols are provided as part of SPIN distribution. On *pftp*, SPIN generates fewer states than PV without state caching. The reason is that there is very little determinism in this protocol. Since Two phase depends on determinism to bring reductions, PV generates a larger state space. However, with state caching, the number of states in the graph goes down by a factor of 2.7. On *snoopy*, even though PV generates fewer states, the number of states generated SPIN tool and PV (without selective caching) is very close to obtain any meaningful conclusion. The reason for this is two-fold. First, this protocol contains some determinism, which helps PV. However, there are a number of deadlocks in this protocol. Because of this, apparently, the proviso is not invoked many times. Hence the number of states generated is very close.

[4]PO-PACKAGE uses a weaker form of proviso along with sleepsets [6], but it preserves only safety properties.

6 Conclusions

We presented a new partial order reduction algorithm called Two phase and showed that it preserves LTL-X properties. By avoiding the proviso and using *deterministic* transitions to bring the reductions, the algorithm can bring better reductions than other algorithms on a number of practical protocols. Two phase algorithm is implemented in PV. Source code for PV may be obtained by contacting the authors. Currently, we are planning to verify the proofs presented in this paper using PVS [12] similar to the work done by Chou and Peled [2].

References

[1] John B. Carter, Chen-Chi Kuo, and Ravindra Kuramkote. A comparison of software and hardware synchronization mechanisms for distributed shared memory multiprocessors. Technical Report UUCS-96-011, University of Utah, Salt Lake City, UT, USA, September 1996.

[2] C.-T. Chou and D. Peled. Formal verification of a partial-order reduction technique for model checking. *Lecture Notes in Computer Science*, 1055, 1996.

[3] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications.

Protocol	SPIN	PV	
		all	selective
B5	243/0.34	11/0.33	1/0.3
B6	729/0.38	13/0.33	1/0.3
B7	2187/0.50	15/0.33	1/0.3
W5	63/0.33	243/0.39	243/0.3
W6	127/0.39	729/0.49	729/0.4
W7	255/0.43	2187/0.76	2187/0.6
Mig	113628/13.6	22805/2.6	9185/1.7
Inv	over 620446/—	255781/39.2	135404/21.2
SC2	290/0.4	123/0.3	47/0.3
SC3	17741/4.6	2687/1.6	733/1.4
SC4	over 260928/—	59436/13.4	14173/11.9
Pftp	95241/11.0	187614/29.9	70653/19.2
Snoopy	16279/4.4	14305/2.7	8611/2.4

Table 1: Number of states explored and time taken for reachability analysis in seconds by the SPIN algorithm and the PV algorithm on various protocols. "PV all" column is the result of running PV with selective caching disabled. "PV selective" column indicates the results of running PV with selective caching enabled. When SPIN couldn't finish the search, time is shown as —.

ACM Transactions on Programming Languages and Systems, 8(2):244–263, 1986.

[4] David Dill. The stanford murphi verifier. In *Computer Aided Verification*, pages 390–393, New Brunswick, New Jersey, July 1996. Tool demo.

[5] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An approach to the State-Explosion Problem*. PhD thesis, Univerite De Liege, 1994–95.

[6] Patrice Godefroid, Gerard Holzmann, and Didier Pirottin. State-space caching revisited. In *Computer Aided Verification*, pages 178–191, Montreal, Canada, June 1992.

[7] Gerard Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[8] Gerard Holzmann and Doron Peled. An improvement in formal verification. In *FORTE*, Bern, Switzerland, October 1994.

[9] Gerard J. Holzmann and Doron Peled. The state of SPIN. In *Computer Aided Verification*, pages 385–389, New Brunswick, New Jersey, July 1996. Tool demo.

[10] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrency Systems*. Springer-Verlag, 1992.

[11] Ratan Nalumasu and Ganesh Gopalakrishnan. A new partial order reduction algorithm for concurrent system verification. In *CHDL*, pages 305 - 314, Toledo, Spain, April 1997. Chapman Hall, ISBN 0 412 78810 1.

[12] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking and model checking. In *CAV*, pages 411–414, New Brunswick, NJ, USA, 1996.

[13] Doron Peled. Combining partial order reductions with on-the-fly model-checking. *Journal of Formal Methods in Systems Design*, 8 (1):39–64, 1996. also in CAV, 1994.

[14] Antti Valmari. On-the-fly verification with stubborn sets. In *Computer Aided Verification*, pages 397–408, Elounda, Greece, June 1993.

[15] Lawrence Yang, David Gao, Jamshid Mostoufi, Raju Joshi, and Paul Loewenstein. System design methodology of UltraSPARC-I. In *Proc. ACM/IEEE 24th Design Automation Conference*, pages 7–12, June 1995.

Automated Deductive Verification of Parallel Systems

Hassen Saïdi

VERIMAG
Centre Equation, 2, Avenue de la Vignate, 38610 Gières
France
Email: saidi@imag.fr
URL: http://www.imag.fr/VERIMAG/PEOPLE/Hassen.Saidi/
Phone: (+33) 4.76.63.48.44, Fax: (+33) 4.76.63.48.50

Abstract

This paper describes the use of deductive methods for the verification of invariance properties of parallel systems. We show how a combination of proof rules, invariant generation techniques and abstraction techniques integrated to a theorem prover can be used effectively to prove invariants of systems given as a parallel composition of sequential processes with infinite data types. We present an implementation of these various techniques in our tool the Invariant-Checker. The Invariant-Checker is build as a front-end for the PVS theorem prover. The tool is an extension of the PVS specification language to handle the notion of transition systems describing the parallel composition of sequential processes. The PVS prover is also extended with a new proof scheme corresponding to the combination of our proof techniques.

keywords: *Computer-Aided verification, Combination of formal methods, theorem proving,* PVS, *invariants proofs, automatic generation of invariants, abstraction techniques.*

1 Introduction

Formal verification methods form a set of more or less automatic verification techniques for hardware and software computing. Since the use of formal methods is in constant growing, tools for the computer-aided verification supporting formal methods are needed and encouraged. A large variety of formal methods techniques is available, such as Model-checking, theorem proving, programs and circuits synthesis, refinement, abstraction techniques, simulation, testing, etc. Some of these techniques like model checking are used with a lot of success as they are fully automatic but are restricted to finite state systems, whereas others like theorem proving are generally considered labor intensive, which may sow doubts about their use in real life situations. The aim is then to combine several methods and techniques in order to complements them and to take advantage of the benefits of each of them.

Theorem proving has been applied to program verification with relatives success[1], specially systems like [5], [16], [9] and [6]. The use of theorem proving as a computer-aided verification technique consists in the validation of a set of logical formulas expressing the facts that the specification satisfies some properties. Higher order logic which is the specification language of most of the known theorem provers is a rich specification language allowing the definition of usual mathematical objects such as types, sets, functions and propositions. Mathematical objects can be constructed as set of previously declared objects or

[1]see [7] for many examples of the use of PVS.

as functions mapping previously declared objects to previously declared objects.

Other verification methods like model checking are used with success due to complete automatization. Model-checking consist in the exhaustive enumeration of all the states of a system and to check whether all theses states satisfy the desired property. While model-checking is restricted to the verification of finite state systems, theorem proving can deal with both finite and infinite state systems, but with loss of complete automatization and with permanent user guidance. Combination of Model-Checking and Theorem proving is one of the most promising combination of verification techniques since it took benefits from the automatization of model-checking and the powerful of theorem proving.

Many works have been achieved to integrate theorem proving to other verification techniques. In [12] Model checking is used to prove abstract models of specifications, while theorem proving is used to prove that the abstraction relation used is a "good" abstraction which preserve some desired properties. In [13] Theorem proving is used in order to prove the correctness of systems given as a parallel composition of processes by means of compositional rules embedded as inference rules in a theorem prover, while the correctness of each components is established using model checking. In [7] the PVS specification language is extended to support duration calculus syntax, and a set of proof rules dedicated to duration calculus logic are implemented. In [1], the PVS theorem prover is integrated to an environment for the verification of hardware specifications. It is used discharge verification conditions in order to test whether two specifications are equivalents modulo some equivalence relation.

In this work we are interested in the verification of invariance properties of parallel systems given as a parallel composition of sequential process with infinite data types. Such systems can not be verified by model checking. The verification techniques we consider are deductive model checking and abstraction techniques. Deductive model checking [10] consist in computing *inductive* invariants of a system by fix point calculation using a deductive proof rule. This proof rule reduces the problem of finding an appropriate invariant to the validity of first order formulas. Techniques for the automatic generation of invariants [2] are used to accelerate convergence. In the case where the proof rule fails to find an appropriate inductive invariant, abstraction techniques [11] are used to construct automatically an abstract state graph of the original system. The construction of the abstract state graph requires also to discharge first order formulas. An abstract state graph can be seen either as a finite state system on which model checking techniques can be applied, or as a more precise control structure of the original system.

The goal of this work is to show how to design in a fashion and an effective way tools for the computer-aided verification by combining techniques such as theorem proving, model-checking and abstraction in order to verify invariance properties of parallel system. We also show how theorem prover as a general framework can be a bootstrap for a combination of formal verification techniques. We choose for this purpose the PVS theorem prover developed at SRI to discharge the generated formulas.

This paper is organized as follow: in section 2, we present the principles of the verification techniques used such as deductive proof rules and abstraction techniques. In section 3 we present our approach to implement these methods and how they can be automated in an efficient way. In section 4, we describe our tool the Invariant-Checker implementing these methods. Finally in section 5, we sketch about our use of the Invariant-Checker and some examples we have studied.

2 Deductive Methods

We are interested in using deductive methods to verify invariance properties of parallel systems. This methods consist in applying a set of proof rules to establish that a property P expressed as a first order formulae is an invariant of a system S, that is, P remains true in every state of the system S, starting from its initial state.

2.1 Preliminary definitions

We consider systems which are parallel compositions of processes, where we consider parallel composition by interleaving and synchronization by shared variables as in Unity [3].

As a model of parallel systems we use transition systems. This notion is one of the simplest one used to describe states of a system and elementary steps (transitions) modifying its states. Also, every sequential process including classical algorithmic construction such as loops, tests and assignments can be translated into a transition system. Transitions are guarded commands where each guard is a boolean expression indicating the condition under which the transition is enabled.

Definition 1 (transition system)
A transition system S is a tuple $S = < \mathcal{V}, \mathcal{T} = \{\tau_1, \cdots, \tau_n\}, \mathcal{L}, Init >$, where $\mathcal{V}$ is a set of system variables, $\mathcal{T}$ a set of transitions or actions, $\mathcal{L}$ a set of control locations and $Init$ an initial state.

Each transition τ_i is a guarded command

$$l_i : \ guard_i \longrightarrow v_1 ::= e_1, \cdots, v_k ::= e_k \ \text{goto} \ l_j$$

where $\{v_1, \cdots, v_k\} \subseteq \mathcal{V}$ and $\{l_i, l_j\} \subseteq \mathcal{L}$.
Each guard $guard_i$ is the guard of the transition τ_i. Each variable v_i is assigned with an expression e_i of a compatible type. Locations l_i and l_j are respectively the source and target location of the transition τ_i.

We also give the following informal definition of the weakest precondition and the strongest postcondition predicate transformers.

Definition 2 (predicate transformer)

1. *$\widetilde{pre}\{\tau_i\}(P)$ is the weakest precondition of the predicate P with respect the transition τ_i and is defined as the set of states from which it is not possible to reach via a transition τ_i a state satisfying $\neg P$.*
   $\widetilde{pre}\{\tau_i\}(P)$ is defined as the predicate

   $$guard_i \Rightarrow P[e_1/v_1, \cdots, e_k/v_k]$$

2. *$post\{\tau_i\}(P)$ is the strongest postcondition of the predicate P with respect the transition τ_i and is defined as the set of states which can not be reached from a state satisfying $\neg P$ via a transition τ_i.*

2.2 Invariance proof rule

We use the simple invariance proof rule (Figure 1) which states that a predicate P is an invariant of a system S, if it exists a predicate P' stronger than P, such that P' holds at the initial state and is preserved by all the transitions of the system.

$$\frac{\begin{array}{l} \models \ Init \Rightarrow P' \\ \models \ P' \Rightarrow P \\ \models \ \forall \tau_i.\ P'\{\tau_i\}P' \end{array}}{Prog \ \models \ \Box P}$$

Figure 1: Invariance proof rule

the predicate $\forall \tau_i.\ P\{\tau_i\}P$ can be expressed using the predicate transformer $\widetilde{pre}$ and $post$ as follow:

- $\bigwedge_{i=1}^{n}(P \Rightarrow \widetilde{pre}\{\tau_i\}(P))$
- $\bigwedge_{i=1}^{n}(post\{\tau_i\}(P) \Rightarrow P)$

We propose to use the predicate transformer $\widetilde{pre}$, since its application does not introduce existential quantifications which are not easy to eliminate.

2.3 Strengthening techniques

This rule does not tell how to find an appropriate P', but many techniques are used to find a P' by refining the predicate P. The most used one is the strengthening method which consists in finding the greatest inductive invariant (implied by all the other inductive invariants of S) implying the predicate P.

That is to find in a model checking like manner the greatest solution of the equation:

$$\forall \tau_i.\ P'\{\tau_i\}P'$$

The strengthening method consists in instantiating P' successively by a sequence $P_0, \cdots, P_n$ of predicates such that:

- $P_0 = P$
- $P_{k+1} = P_k \bigwedge_{i=1}^{n} \widetilde{pre}\{\tau_i\}(P_k)$

At each k^{th} step the second premise of the proof rule is trivially true, and only the two others premises are checked, that is the predicates:

$$k^{th} \text{ step} \left\{ \begin{array}{l} Init \Rightarrow P_k \\ P_k \Rightarrow \bigwedge_{i=1}^{n} \widetilde{pre}\{\tau_i\}(P_k) \end{array} \right.$$

In fact it is not necessary to consider at each new instantiation the predicate $P_k \bigwedge_{i=1}^{n} \widetilde{pre}\{\tau_i\}(P_k)$, but only the predicate $P_k \bigwedge_{j=1}^{m} \widetilde{pre}\{\tau_j\}(P_k)$ where the transitions τ_j are those for which the verification condition $P_k\{\tau_j\}P_k$ is not proved valid.

Figure 2 illustrates this fix point calculation known as the backward analysis using the predicate transformer $\widetilde{pre}$. The dual analysis method is the Forward analysis which consist in the computation of the reachable states from the initial state of the system S iteratively as a fix point using the predicate transformer *post*, and than to check whether the reachable predicate *Reach* implies the predicate P.

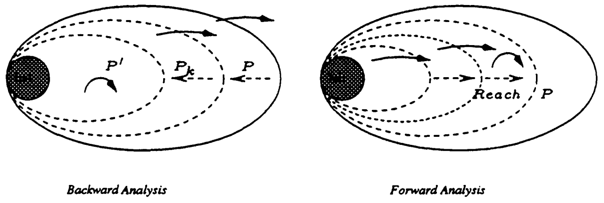


Figure 2: Fixpoint calculation

These two dual techniques may not terminate when the state space of the analyzed system is infinite. Moreover, when one may terminate the other may not. Also, in the forward analysis the generated VCs have no more quantifiers than the predicate P, however in the forward analysis, the generated VCs are existentially quantified formulas. This makes the use of forward analysis technique more difficult than the backward technique due to the quantifier elimination problem.

2.4 Refined strengthening techniques

The drawback of the proposed strengthening techniques is the growing size of the predicates P_k computed at each iteration. To avoid this problem, other strengthening rules have been proposed. In [12] it is proposed to instantiating P' successively by a sequence $P_0, \cdots, P_n$ of predicates such that:

- $P_1 = P$
- $P_{k+1} = \bigwedge_{j=1}^{m}(P_k \Rightarrow \widetilde{pre}\{\tau_j\}(P_k))$

The idea behind the use of this rule is the following: If the proof of the validity of the verification condition $P_k \Rightarrow \widetilde{pre}\{\tau_i\}(P_k)$ fails, it is tried to prove the validity of the simplified form of this VC in the program and not in every model. However, this rule does not accelerate convergence any more. This is due to the fact that at each iteration not all the state from which it is possible to reach a state satisfying $\neg P$ are eliminated.

In [2] a refined strengthening technique is proposed. It consist in using the following rules:
Instantiate P' successively by

- $P_1 = P$
- $P_{k+1} = P_k \wedge Q$

where Q is the "simplified" form of $P_k \Rightarrow \widetilde{pre}\{\tau_i\}(P_k)$. This rule is equivalent to the first one since the predicate P_{k+1} are equivalents in both techniques:
$P_k \wedge (P_k \Rightarrow \widetilde{pre}\{\tau_i\}(P_k)) \Leftrightarrow P_k \wedge \widetilde{pre}\{\tau_i\}(P_k)$

This method is used to avoid the size increasing of expressions due to the multiple applications of the strengthening techniques. However in practice it does not give interesting results. This is essentially due to the proof strategy used to have a "simplified" form of the considered VC. In fact this strategy should not create additional subgoals. Furthermore since the result of the application of this strategy will be considered in the next step, this strategy should not change significantly the current goal to prove by increasing its size. Thus, proof strategies such as rewriting, induction may not be used. Also, if a non appropriate proof strategy is used such as rewriting a bad instance of a function symbol or an induction on a non relevant variable may create non relevant subgoals which are propagated along the iteration process.

2.5 Automatic generation of invariants

The application of the proof rule in the case of infinite state systems may lead to an infinite computation. Also, when the computation terminates, convergence is often slow. Convergence can be accelerated using some already proved invariants of the system. Let $\mathcal{I}$ be a conjunction of such invariants. The verification conditions generated by the proof rule can be weakened by $\mathcal{I}$, that is by generating at each k^{th} step the following VCs:

$$k^{th} \text{ step} \left\{ \begin{array}{l} \mathcal{I} \wedge \mathit{Init} \Rightarrow P_k \\ \mathcal{I} \wedge P_k \Rightarrow \bigwedge_{i=1}^{n} \widetilde{pre}\{\tau_i\}(P_k) \end{array} \right.$$

In [2] we present techniques for the automatic generation of inductive invariants by static analysis. These techniques allows us to generate invariants of a sequential process as well as for a system given as a parallel composition of sequential processes. The invariant $\mathcal{I}$ represents an upper approximation of the set *Reach* of reachable states. The proof rule allows us to combine backward and forward analysis, since the backward analysis considers only states satisfying $\mathcal{I}$.

2.6 Abstraction techniques

In the case where the combination of the proof rule and the invariant generation techniques fails to prove the property, it is useful to construct a corresponding finite state system called an abstraction where the property we want to verify can be easily checked by model checking techniques. Most abstraction techniques used in program verification are based on abstract interpretation of programs [4]. The main concern of abstract interpretation is the construction of descriptions of concrete systems by "mimicking" the effect of concrete operations with suitable operations defined over descriptions. The construction of an abstract system requires the definition of an abstract state space, where each abstract state represent a possibly infinite set of concrete states.

In [11] we define a particular abstraction scheme, where an abstract state is defined as a valuation of a set of predicates $\varphi_1, ..., \varphi_\ell$ on program variables and the valuation of the concrete control variables. The construction process starts by defining the initial abstract state of the system. This can be done by checking the possible valuations of each predicate φ_i in the concrete initial state. The successors of an abstract state are computed as follows:

1. Select all the possible transitions τ_i from the abstract state.
2. For each such transition compute the valuation of each predicate φ_i after the execution of τ_i , that is whether φ_i or $\neg\varphi_i$ is a postcondition. The successor s' of a valuation s is then equal to:

$$\begin{array}{ll} \mathit{false} & \text{if} \quad s \Rightarrow \neg \mathit{guard}_i \\ \forall i. \varphi_i = \left\{ \begin{array}{ll} \mathtt{t} & \text{if } s \Rightarrow \widetilde{pre}\{\tau_i\}(\varphi_i) \\ \mathtt{f} & \text{if } s \Rightarrow \widetilde{pre}\{\tau_i\}(\neg\varphi_i) \\ \mathtt{T} & \text{otherwise} \end{array} \right\} & \text{otherwise} \end{array}$$

In the case where T is a postcondition of a predicate φ_i, two successors are created corresponding to the case where φ_i or $\neg\varphi_i$ is a postcondition. t and f are the boolean values true and false. The successor *false* is interpreted as the empty set, that is no successor exists for the considered abstract state via the transition τ_i.

Using these rule an abstract state graph can be constructed. This graph is an upper approximation of the global reachable state of a system and can be used to prove any property involving the predicate

$\varphi_1, ..., \varphi_\ell$ using model-checking techniques. Also this state graph gives a more precise idea about the behavior of a system given as a parallel composition of processes by eliminating non reachable global control configuration. Thus from a practical point of view it is useful to include in the set $\varphi_1, ..., \varphi_\ell$ of predicates the literals appearing in the guards of each transition of the system and the literals appearing in the property we want to verify. The presented proof rules of the previous sections can then be applied on the system defined by this abstract state graph which is a system with only one component. Notice that our abstraction technique is also based on the proof of a set of first order implications, which can also be weakened by the invariant $\mathcal{I}$. This allows us to combine proof rules, invariant generation and abstraction techniques in a same framework based on discharging verification conditions.

3 Automated deductive methods

We propose to implement the proof rule of figure 1, the techniques for the automatic generation of invariants and the abstraction techniques we have defined. A tool implementing such techniques must have two essential features: a powerful specification language allowing the describtion of transition systems with infinite data types, a verification condition generator for each proposed techniques and finally a proof engine to discharge all the generated VCs. A theorem prover seems to be an appropriate tool for our verification techniques. We choose the PVS theorem prover for this purpose.

3.1 PVS

PVS is an environment for writing specifications and developing proofs. It consists of a specification language integrated with a powerful and highly interactive theorem prover. PVS uses higher order logic as a specification language, the type system of PVS includes uninterpreted types, sub-typing and recursively defined data types. Four sorts characterize this language: **Theory**, **Type**, **Expression**(*term*), **Formula** (*proposition*). Any PVS specification is structured into parameterized theories. A **Theory** is a set of **Type**, variable, constant, function and **Formula** declarations. The theorem prover or proof checker implements a set of powerful tactics with a mechanism for composing them into proof tacticals. Some of these tactics such as *assert* and and *bddsimp* invoke efficient decision procedures for arithmetic and boolean expressions. PVS has emacs as user interface.

3.2 A first approach

We investigated two approaches to implement the described techniques using the PVS theorem prover. A first approach consists in encoding the proof rule in the specification language of the prover. This approach is the "classical" one adopted when a theorem prover is used in formal verification.

Using the PVS specification language, a system S can be described in a PVS theory. Given a predicate P, we express the invariance property S$\models$ □P as a theorem:

```
th : THEOREM
    is_invariant?(S,P)
```

where **is_invariant?** is a recursive function defined on the structure of the system S.

The strengthening techniques can also be defined using a recursive functions parameterized by the maximal number of iteration allowed.

We found this way of proving invariants using a theorem prover not satisfactory for many reasons:

1. The syntax is not adapted to describe systems easily.
2. The proofs are too long, which is due to many unnecessary rewriting steps.
3. A lot of user interaction is required and few automatization is provided.

3.3 Our approach

To deal with the weakness reported previously, we choose to built a verification system using the PVS theorem prover as a back-end where the verification conditions are generated automatically and submitted to the Pvs prover in a fully automatic way. The objectives of this approach are the following:

- Full automatization whenever it is possible.
- Use all the power of the Pvs specification language and the Pvs prover.
- Extend the Pvs verification system with new features, including our particular proof scheme.

Following these objectives, we have designed our tool the Invariant Checker [10, 19]. The system is designed as a front-end for the Pvs [7] theorem prover. The Invariant Checker can be seen as an extension of the Pvs verification system to handle the notion of transition systems and invariants as well as the usual mathematical objects. These extensions appears at two different levels: the Pvs specification language is extended with the notion of a `system`, or a parallel composition of transition systems. The Pvs prover is also extended with the deductive proof rule we use.

This type of invariant verification makes a different use of theorem proving from the "classical" one where program's semantics are encoded in the prover's specification language (usually higher order logic or set theory). In this "classical" approach the proof process is complicated due to the heavy encoding of semantics and the unnecessary rewriting of semantics definitions, while usually the most important and difficult part of the verification process is the reasoning about the program variables and their values. Also, it requires too much user interaction. The objective of our tool is to provide more automatization and less user intervention using a set of features. The principals of our tool are illustrated in Figure 3. To extend the prover with our particular proof schemes and techniques, a front end is required, where VCs are only submitted to the prover without user interaction. Notice that this extension scheme is general and can be applied to any verification scheme where discharging first order assertions is required. Of course the verification process can be completely automatized under the condition that the generated VCs are decidable predicates.

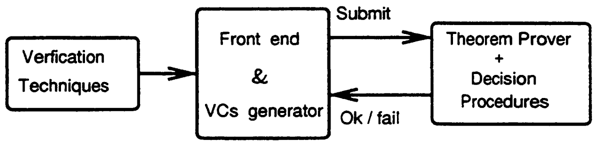

Verfication Techniques
Front end & VCs generator
Submit
Ok / fail
Theorem Prover + Decision Procedures

Figure 3: Design principal

4 A verification tool: The Invariant-Checker

4.1 Design Philosophy

As reported previously, the objective of the design of such tool is the complete automatization of the verification process whenever it is possible, and also to take advantage of the Pvs specification language and its rich type system and the Pvs prover and its powerful proof strategies.

In order to be efficient it is only necessary to interact automatically with Pvs, and then to generate VCs in a format acceptable by the prover. Systems can then be described in any other specification language, but in order to have an efficient integration with Pvs we decided to use the Pvs specification language to describe systems. However it was necessary to extend this language to support the definition of transition systems in a syntax close to the one defined in section 2.2. Thus, the manipulated variables can be defined as Pvs variables and the manipulated expressions as Pvs expressions. The typechecker is also extended in order to parse and to typecheck specifications written in this syntax. This allow us to use all the power of the Pvs specification language with a suitable syntax. It is also necessary to consider a new notion of proof status of a system, that is which are the assumed invariants (axioms) of the considered system and which are the already proved invariants and the non yet proved ones. The following correspondence scheme gives the design philosophy of the tool where PVS is extended with the notion of **system**, **invariant** and **invariant proof** :

PVS		*Invariant – Checker*
Theory	$\longleftrightarrow$	**System**
Theorem	$\longleftrightarrow$	**Invariant**
Proof	$\longleftrightarrow$	**Invariant proof**

4.2 Features

Figure 4 describes the general scheme of the Invariant-Checker. In this section we explain the role of each component of the tool and how the proof process is organized.

Syntax: Systems can be described in a Simple Programming Language (SPL), close to the one used in [17], but with the rich data types and expressions definition mechanism available in PVS. Our SPL language includes common algorithmic constructions such as single and multiple assignments statements, conditionals If-Then-Else and loop statements. We also allow parallel composition by interleaving and synchronization by shared variables as in Unity [3]. Systems described in SPL are translated automatically into guarded commands with explicit control. Program variables can be of any type definable in PVS, and can be assigned by any definable PVS expression of a compatible type. Also, it is possible to import any defined PVS theory. Figure 5 shows a simple example of a system given as a parallel composition of two processes Process_1 and Process_2. this system is knows as the bakery system. It guarantees the access to the critical section 3 in a mutual exclusion between the two processes. The example is given in the concrete syntax allowed for transition systems[2]. **Req_i**, **In_i**, **Out_i** are labels of the transitions of Process_i.

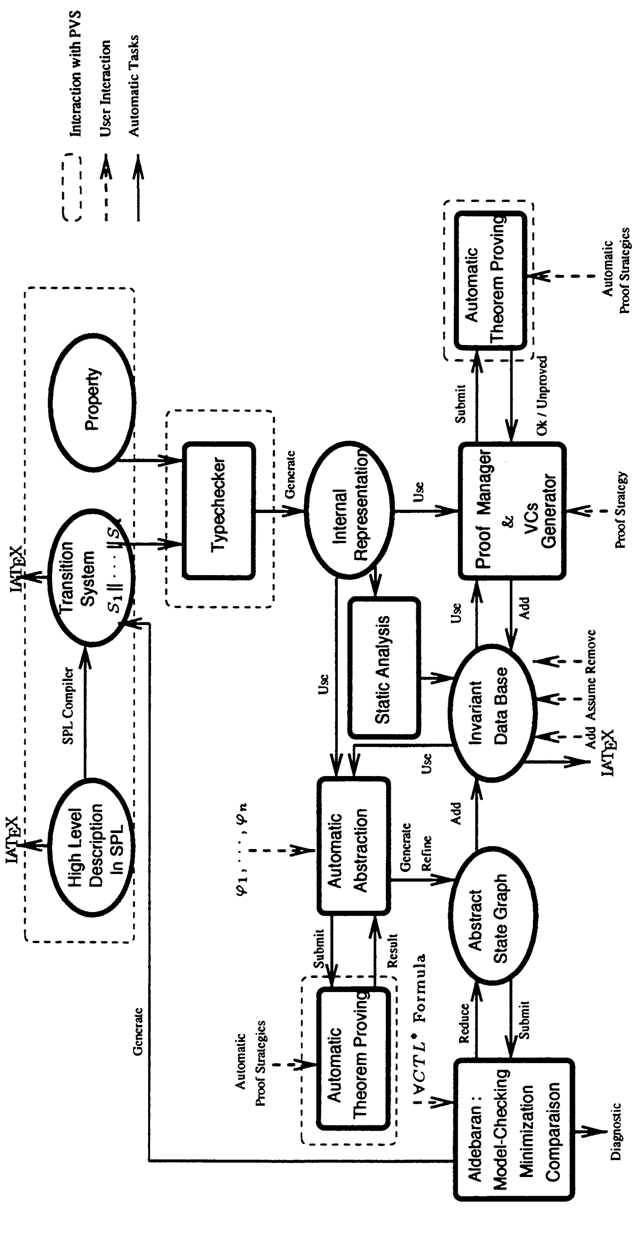


Figure 4: The Invariant Checker Architecture

[2]The example is presented in the generated LaTeX format for guarded commands. Variables $pc1$ and $pc2$ are respectively the control variable of Process_1 and Process_2.

```
bakery : SYSTEM
 BEGIN

  comp1 : PROGRAM
    y1 : VAR nat

   BEGIN
    Req_1
    1: TRUE                  → y1 := y2+1    GOTO 2
    In_1
    2: y2 = 0 ∨ y1 ≤ y2      →   SKIP        GOTO 3
    Out_1
    3: TRUE                  → y1 := 0       GOTO 1
   END comp1

 ||

  comp2 : PROGRAM
    y2 : VAR nat

   BEGIN
    Req_2
    1: TRUE                  → y2 := y1+1    GOTO 2
    In_2
    2: y1 = 0 ∨ y2 < y1      →   SKIP        GOTO 3
    Out_2
    3: TRUE                  → y2 := 0       GOTO 1
   END comp2

 INITIALLY  : y1 = 0 ∧ y2 = 0 ∧
              pc2 = 1 ∧ pc1 = 1

END bakery
```

Figure 5: Bakery transition system

Typechecking: Typechecking concerns only systems described as a parallel composition of guarded commands since the translation of an SPL system into a transition system is only a syntactical transformation. Typechecking a system consists in checking that every guarded command is well typed according to a typing context. This typing context consists of all variable declarations and may be some imported PVS **theory**. Since the manipulated expressions are PVS expressions, typechecking a system may lead to the generation of type correctness conditions (TCCs). In PVS a generated TCC must be proved before proving any assertions of the corresponding theory. This is due to the fact that a theory is a logical specification and a generated TCC indicates some correctness criterion of the specification.

Our description of systems can be seen as a description at a hight (specification) or at a low (implementation) level. They are not logical specification. Therefore the generated TCCs are predicates that guarantee "absence of run-time errors"(division by zero, application of the tail function to the empty list...). Moreover, these predicates have to be proved as invariants and not as valid formulas as it is the case of TCCs generated for a PVS theory. It is just required that they are valid in the model generated by the system and not in every model. In the case where they cannot be proved, absence of run-time errors is not guaranteed but this does not affect the proof of any invariant.

Consider the following transition where x is declared as a positive integer:

$$i : \mathrm{x} > 0 \quad \longrightarrow \quad x := x - 1 \ \text{GOTO}\ j$$

The expression $x-1$ is assigned to x and should be of a compatible type, that is $x - 1 \geq 0$. This constraint is not required to be true at any moment of the execution of the program but only when a transition is activated, that is only when the program counter equals i and the guard $\mathrm{x} > 0$ is true. Thus the following TCC is generated:

Tcc_i : OBLIGATION
$$pc = i \wedge x > 0 \Rightarrow x - 1 \geq 0$$

this TCC is trivially valid and can be discharged using the PVS arithmetics decision procedures.

Of course, one can encode the semantics of guarded commands within the PVS specification language, and then use the "classical" PVS typechecker to generate such TCCs. However they must be proved as invariants. The generated TCCs can be used in this case to prove other invariants of the considered system, and can also be proved using some other invariants, which is not the case of TCCs generated for a theory, which must be proved independently of any other assertions.

Internal representation: PVS is implemented in LISP. every object manipulated in PVS such as a theory, a theorem or a proof is represented as an instance of a predefined object class. We have defined for all the abject manipulated in the Invariant-Checker an internal representation which is also a class such as: **`system`**, **`process`**, **`transition`**, **`invariant`** and **`proof`**. An important aspect in these structures is that they are independent from the PVS internal structures and make our implementation independent from the possible changes in the PVS internal representation. However the expression manipulated and the verification condition generated are represented as PVS expressions and PVS obligations. This is necessary for the automatic interaction with the prover we want to achieve.

Static analysis As it is shown, any known auxiliary invariant $\mathcal{I}$ of a system can be used to accelerate convergence. Static analysis consists in a set of techniques for the automatic generation of such invariants. Theses techniques are based on propagation of guards and assignments through program control points. two kinds of invariants are generated: local invariants for each process and global invariants of the parallel composition. The techniques we used are described in [2]. These invariants are added automatically to the invariant database. For example, the following global invariants are automatically generated for the bakery system:

$$\mathrm{pc1} = 1 \Rightarrow y_1 = 0$$

$$\mathrm{pc1} = 2 \Rightarrow \exists\,(y_2):\; y_1 = y_2 + 1$$

$$\mathrm{pc1} = 3 \Rightarrow \exists\,(y_2):\; y_1 = y_2 + 1$$

$$\mathrm{pc2} = 1 \Rightarrow y_2 = 0$$

$$\mathrm{pc2} = 2 \Rightarrow \exists\,(y_1):\; y_2 = y_1 + 1$$

$$\mathrm{pc2} = 3 \Rightarrow \exists\,(y_1):\; y_2 = y_1 + 1$$

$$\mathrm{pc1} = 2 \wedge \mathrm{pc2} = 3 \Rightarrow y_1 = y_2 + 1 \vee y_1 = 0 \vee y_2 < y_1$$

$$\mathrm{pc1} = 1 \wedge \mathrm{pc2} = 3 \Rightarrow y_1 = 0 \vee y_2 < y_1$$

$$\mathrm{pc1} = 3 \wedge \mathrm{pc2} = 2 \Rightarrow y_2 = 0 \vee y_1 \leq y_2 \vee y_2 = y_1 + 1$$

$$\mathrm{pc1} = 2 \wedge \mathrm{pc2} = 2 \Rightarrow y_1 = y_2 + 1 \vee y_2 = y_1 + 1$$

$$\mathrm{pc1} = 1 \wedge \mathrm{pc2} = 2 \Rightarrow y_1 = 0 \vee y_2 = y_1 + 1$$

$$\mathrm{pc1} = 3 \wedge \mathrm{pc2} = 1 \Rightarrow y_2 = 0 \vee y_1 \leq y_2$$

$$\mathrm{pc1} = 2 \wedge \mathrm{pc2} = 1 \Rightarrow y_1 = y_2 + 1 \vee y_2 = 0$$

Proof manager: The proof manager is the module which coordinates the proof process by applying the proof commands invoked by the used. This module controls the number of iterations, controls the validity of each VCs and invokes the proof rule whenever it is necessary.

VC generator: Each application of a proof rule leads to the generation of a set of first order formulas to discharge. For a predicate P and a system S only non trivial VCs are generated. For example no VC is generated when a transition does not affect the free variables of P. Also, for every generated VC, a set of relevant generated invariants is automatically selected to achieve the proof.

Proof session: A proof session starts with type-checking the system and the property we want to verify. The system is then translated into an internal representation which will be used by all components of the tool. One can then apply static analysis to the system in order to extract invariants during the static analysis . The user can then start the proof by indicating to the proof manager, which strategy he wants to invoke. That is for example the maximal number of strengthening steps, the automatic use or not of some of the invariants in the data base. At each call of the proof rule, a set of VCs is generated and submitted automatically to the prover. If the applied proof strategy fails to prove some of them, the user can either prove them interactively or automatically strengthen the invariant and apply the proof rule again. Non provable assertions are considered as non valid.

Automatic theorem proving: The generated VCs are submitted automatically to the PVS prover, where automatic proof strategies combining automatic induction, automatic rewriting, boolean simplification using Bdds and decision procedures can be applied. The user can defined such strategies, by combining predefined PVS strategy and user defined ones. We have defined a set of powerful proof strategies which allows us to discharge non trivial goals, especially using heuristics for the automatic induction.

Invariants data base The invariant data base contains the invariants generated during the static analysis. The user can always enrich the invariant data base with some invariants. With each invariant a status is associated which may change during the proof. The status has three possible values:

- "assumed": assumed invariants are user defined and no proof is required for them. They play the role of axioms, and can therefore lead to inconsistent proofs.
- "unproved".
- "proved": with each proved invariant its proof is associated. It consists of the applied proof strategy and the invariants used during the proof. If some invariant is removed from the data base, all the already proved invariants whose associated proof depends on the removed invariant, become "unproved".

Automatic abstraction The abstraction technique we use in our tool are those described in [11] and explained in section 2.6. Given a set of predicates $\varphi_1, ..., \varphi_\ell$ on the variables of a system, an abstract state graph (where states are valuations of $\varphi_1, ..., \varphi_\ell$) is constructed in an automatic way using user defined proof strategy.

An abstract state graph can be used in many ways:

- It can be used as a global control graph from which stronger invariants can be generated and added to the invariants data base.
- It can be minimized modulo strong equivalence using the ALDÉBARAN tool [8]. The reduced graph defines a new system with a single component, on which we can prove the property we want to verify using the implemented proof rule.
- It is possible to prove a temporal formula involving $\varphi_1, ..., \varphi_\ell$ using the model checker of ALDÉBARAN.

The abstract state graph of figure 6 is obtained using the predicates appearing in the guards of the bakery transition system, that is the predicates $y_1 = 0$, $y_2 = 0$, $y_1 \leq y_2$ and $y_2 < y_1$. The control configuration $pc1 = 3 \wedge pc2 = 3$ indicating violation of the mutual exclusion property is not reachable.

User interface: PVS has emacs as user interface. We found convenient to use the same user interface for our prototype. All the functions of the tool can be invoked by some emacs command.

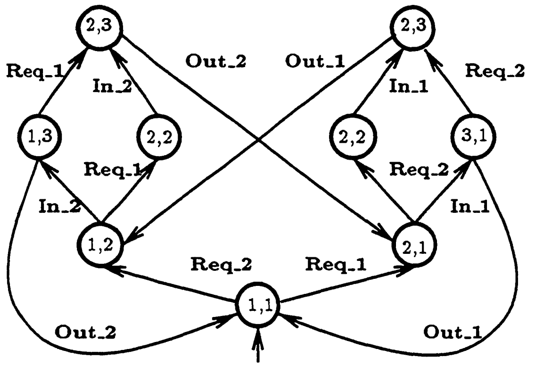


Figure 6: Abstract state graph for the bakery system

5 Experiments with the Invariant-Checker

Using the Invariant Checker we verified various classical mutual exclusion algorithms [17], a read and write buffer using complex data types [10]. The mutual exclusion property of the bakery system has been proved using the proof rule and using abstraction techniques in a fully automatic way. The use of abstraction techniques allow us to prove in a fully automatic way an alternating bit and a bounded retransmission protocol [11] developed by Philips and used in one of its commercial products. This example has been already proved using theorem proving techniques, but a big amount of user interaction has been necessary to provide powerful enough auxiliary invariants. We have also used successfully the Invariant Checker tool for the verification of parameterized networks following the techniques described in [15]. Additional information can be found on our experiences using the tool in the Invariant Checker home page [14].

6 Conclusion

This paper describes a tool supporting the automated formal verification of invariance properties of parallel processes. It shows how the use of theorem proving techniques can be done in an effective and efficient way with a combination of other verification techniques such as model checking and abstraction. It shows that automatization of deductive methods can be effective and leads to effective proof tools. This work can be extended to other verification techniques which can be combined with theorem proving. The first step of the combination process is to allow different specification languages in the same tool. The second one is to allow different verification techniques. The verification process can be than a combination of such techniques and can be decomposed into different steps. Our design philosophy is a consequence of the following remarks: Effective verification systems requires generally an enough powerful specification language and adequate and powerful verification techniques, but only the combination of formal verification techniques and tools can lead to effective use of formal methods. We found that a theorem prover can play a crucial role, by bootstrapping the combination process. The different specification languages can be formally defined using the PVS specification language. The PVS prover can then be used partially or totally in the verification process. It can also be used to prove that the combination process is correct.

References

[1] H. Barringer, G. Gough, B. Monahan, and A. Williams. The ELLA Verification Environment: A Tutorial Introduction. Technical Report UMCS CS-94-12-2. University of Manchester.

[2] S. Bensalem, Y. Lakhnech, and H. Saïdi. Powerful Techniques for the Automatic Generation of Invariants. In *Conference on Computer Aided Verification (CAV'96)*, Rutgers University, New Jersey, July 1996. LNCS 1102.

[3] K. M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, 1988.

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis

of programs by construction or approximation of fixpoints. In *4th POPL*, January 1977.

[5] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[6] C. Cornes, J. Courant, J. C. Filliâtre, G. Huet, P. Manoury, C. Paulin-Mohring, C. Muñoz, C. Murthy, C. Parent, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual. Version 5.10. Technical Report, I.N.R.I.A, February 1995.

[7] D. Cyrluk, P. Lincoln, P. Narendran, S. Owre, S. Ragan, J. Rushby, N. Shankar, J. U. Skekkebæk, M. Srivas, and F. von Henke. Seven Papers on Mechanized Formal Verification. Technical Report SRI-CSL-95-3, Computer Science Laboratory, SRI International, 1995.

[8] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier and M. Sighireanu. CADP (Cæsar/Aldébaran Development Package): A protocol validation and verification toolbox. In *CAV'96*. LNCS 1102, 1996.

[9] M. J. C. Gordon and T. F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.

[10] S. Graf and H. Saïdi. Verifying invariants using theorem proving. In *Conference on Computer Aided Verification (CAV'96)*, Rutgers University, New Jersey, July 1996. LNCS 1102.

[11] S. Graf and H. Saïdi. Construction of abstract state graphs with Pvs. Accepted for publication. In *CAV'97*.

[12] K. Havelund, and N. Shankar. Experiments In Theorem Proving and Model Checking for Protocol Verification. In *the proceedings of Formal Methods Europe 96*. 1996.

[13] H. Hungar. Combining Model checking and Theorem proving to Verify Parallel Processes. In *Computer-Aided Verification, CAV'93*, LNCS 697, Elounda, June 1993.

[14] The Invariant-Checker home page. `http://www.imag.fr/VERIMAG/PEOPLE /Hassen.Saidi/INVARIANT-CHECKER/`

[15] D.Lesens and H. Saïdi. Automatic Verification of Parameterized Networks of Processes by Abstraction. In *Proceedings of the 2nd International Workshop on the Verification of Infinite State Systems (INFINITY'97)*, Bologna, Italy, July 1997.

[16] S. Owre, N. Shankar, and J. M. Rushby. The PVS Specification Language. Computer Science Laboratory, SRI International, February 1993.

[17] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[18] J. M. Rushby. Model Checking and Other Ways of Automating Formal Methods. *TPosition paper for panel on Model Checking for Concurrent Programs Software Quality Week*, San Francisco, May/June 1995.

[19] H. Saïdi. A Tool for Proving Invariance Properties of Concurrent Systems Automatically. In *Proceedings of 2nd International Workshop, Tools and Algorithms for the Construction and Analysis of Systems*, Passau, Germany, March 1996. LNCS 1056, Springer-Verlag.

[20] H. Saïdi. . The Invariant-Checker: Deductive Verification of Reactive Systems. Accepted for publication. In *CAV'97*.

Robust Computer System Proofs in PVS

Matthew M. Wilding

Advanced Technology Center
Rockwell Collins, Inc.
Cedar Rapids, IA 52498 USA
mmwildin@collins.rockwell.com

Abstract

Practical formal verification of complex computer systems requires proof robustness and efficiency to protect against inevitable mistakes and system specification and design changes. PVS is a theorem-proving system based on higher-order logic with which we demonstrate the kind of robust code proofs needed for verification of realistic-sized computing systems.

1 Introduction

Computer system correctness can be difficult to establish. Formal proofs about formal models of computer systems have the potential to improve the reliability of computer system designs, but they have several drawbacks. Formal proofs about computer systems are often very complex and hard to get right, and the social process that is usually counted on to certify mathematical proofs is ineffective because particular computer system designs are often proprietary and in any case not of general interest. Mechanical theorem provers can help overcome both of these problems with formal proof: proofs generated with computer programs can be easier to produce and more reliable.

PVS is a verification system for "specifying and verifying digital systems" [12, 13, 16]. It supports a specification language that is based on a simply typed higher-order logic, and provides a large number of prover commands that allow machine-checked reasoning about expressions in the logic. There is support for automating reasoning in PVS, namely a simple rewriting system and a facility for constructing new proof commands, although the emphasis in PVS is on building clear specifications and supporting user proof with domain-specific decision procedures.

The Rockwell AAMP5 and AAMP-FV are processor designs with microcoded instruction sets. Partial microcode correctness of these processors has been established using PVS [9, 10]. The hardware that executes microcode has been formalized in the PVS logic, and proofs that the microcode correctly implements some of the processor instruction sets have been constructed. While the application of PVS to realistic-sized processors in the AAMP5 and AAMP-FV projects led to a partial verification of their microcode, the experience of building these proofs led the developers to the pragmatic realization that practical computer systems proofs must be robust [9]. That is, computer system proofs must be able to demonstrate correctness with minimal human assistance despite modest system or specification changes.

Mistakes in proof development and changes to system design and specification are inevitable for realistic-sized verifications. For example, during the AAMP-FV verification effort a change was made in the formal model related to memory address decoding [9]. This change caused every previously-constructed instruction correctness proof to fail even though the change had little to do with the substance of most of the proofs. Large programming projects use software engineering techniques to make software robust despite inevitable changes. So too must large machine-checked proof projects use techniques to develop robust proofs.

Various projects besides the AAMP5 and AAMP-FV verifications have established computer system correctness using mechanical proof. A Piton [11] program that plays the puzzle-game Nim is proved to play optimally [17]. Compiled routines from the C string library and elsewhere targeted to the Motorola 68020 are proved to meet their specifications [5]. Microcode for the Motorola CAP processor is proved to implement several algorithms useful for digital signal processing [6]. Others verifications involve a stack of verified systems [2], an operating system kernel [1], code for simple real-time systems [18], and floating-point microcode [6, 15]. Each of these projects employed the theorem proving system Nqthm [3] or its successor ACL2 [8].

The logics supported by Nqthm and ACL2 are weaker than that supported by PVS: they do not conveniently support higher-order functions and quantification. The style of proof encouraged by the theorem proving system is also quite different: Nqthm and ACL2 provide several automatic proof techniques that are programmed by the user by proving theorems and adding them to the theorem prover database. A considerable amount of strategic planning is required to coopt the Nqthm and ACL2 proof heuristics to prove interesting theorems. However, the style of proof of these efforts has an important benefit: proof robustness. Since the proofs are "automatic" – at least in the shallow sense that the same proof heuristics are applied for every proof albeit with different rules databases – even dramatic changes in the system or the specification typically do not render old proofs obsolete. For example, when the verified processor FM8501 was redesigned to increase its wordsize the Nqthm proof of the modified processor correctness theorem worked with minimal human assistance [11]. Theorem provers based on first-order quantifier-free logic have been successful on larger system correctness problems in part because their mostly-automatic approach to guiding the theorem prover.

This paper explores how to use PVS to reason about computer systems in a robust style. We do this by adapting the computational specification style of the Nqthm/ACL2 verifications and by developing a specification and proof methodology that allows relatively automatic PVS proofs about code execution. The proofs employ some of the techniques used in the Nqthm/ACL2 proofs plus some PVS-specific techniques. The use of interpreters to define languages and the automation to improve proof resilience transcend particular theorem provers. However, this approach does not require that we forego the use of the full PVS language and prover in other proofs: we can use our theorems about code execution to prove whatever we wish using the full PVS logic and prover.

In the next section we present a formalization of a simple computing system in order to aid the exposition of this paper. Section 2 outlines our approach for proving code in PVS, using a simple computing system to illustrate our technique. Section 3 gives an example of how the full PVS language can be used for specification in concert with our robust proofs. Section 4 presents some brief conclusions.

2 Reasoning about Program Execution

We describe in this section how to specify and reason about code in a robust way. We introduce a simple machine formalized in the PVS logic with which we illustrate our approach. Two example programs for this machine are presented for which we construct code execution correctness statements. The proofs of these correctness statements are very simple owing to the creation of some simple reasoning support we have built into PVS and some simple conventions we follow in the expression of code correctness. The style of proof is similar in some respects to other verification projects, particularly [5, 11, 17]. These proofs are less sensitive to changes and therefore more robust.

2.1 A Simple Machine Interpreter

In order to make the ideas of this paper concrete we introduce a PVS computing machine formalization that supports examples in later sections. We present **sm**, a slightly modified version of John Rushby's formalization of Bob Boyer's and J Moore's simple machine-level language [4, 14].

An **sm** state is composed of five elements: a program counter, a stack containing subroutine call return addresses, a data memory that maps natural number addresses to natural number values, a flag whose boolean value indicates whether the processor is halted, and a program memory that maps natural number addresses to instructions. We fix both instruction and data memory size at 100 elements which limits the valid addresses for the memories to values less than 100, and represent an **sm** instruction as a record containing one of 13 opcodes and two addresses. The instructions are described informally in Figure 2.1.

The PVS function `step` defines precisely the effect of executing the instruction pointed to by the pc, thereby providing a formal version of the instruction descriptions of Figure 2.1 with which we can reason

move a b store value at location b in location a
movei a n move value n in location a
movewind a b store value at location b in location stored at location a
moverind a b store value at the location stored at location b in location a
add a b store sum of values at locations a and b in location a
sub a b store in location a the greater of 0 and the difference of a and b
incr a increment value at location a
decr a decrement value at location a
jump n store value n in pc
jumpz a n store value n in pc if value at location a is 0.
call n store (incremented) pc on the stack and store value n in pc
ret store a value popped from the stack in pc
halt set the halt flag

Figure 1: The **sm** Instructions

about programs. We define a function `sm` that returns the state resulting from running `n` instructions starting in state `s`.

```
sm(s: state, n: nat): RECURSIVE state =
  IF n = 0 THEN s ELSE sm(step(s), n - 1) ENDIF
 MEASURE n
```

This computing machine is considerably simpler than formalizations of actual machines but it provides enough complexity for sufficiently interesting examples. The specification style of **sm** is similar to many Nqthm and ACL2 efforts, but one difference that exists between **sm** and those other models illustrates an important difference between the styles encouraged by Nqthm/ACL2 and PVS. While **sm** memory is represented by a function, memory in the Nqthm and ACL2 code proof interpreters is represented by a particular datastructure implementation. For example, memory in the formal model of the FM9001 is represented by a binary tree of memory elements [7]. This style difference stems from a difference in proof system functionality. Nqthm/ACL2 provides execution of definitions and encourages concrete, efficient models. (An ACL2 interpreter for a commercial processor executes microcode programs faster than the executable processor model being used for microcode development [6].) PVS cannot conveniently simulate machine execution but provides higher-order logic and encourages specification unburdened by irrelevant detail.

```
address    code
   0       move 2 0
   1       move 3 0
   2       move 4 1
   3       sub 4 2
   4       jumpz 4 12
   5       incr 2
   6       moverind 4 2
   7       moverind 5 3
   8       sub 5 4
   9       jumpz 5 2
  10       move 3 2
  11       jump 2
  12       ret
```

Figure 2: **sm** min subroutine

2.2 An sm Program and Specification

We mainly use two features of PVS to prove program properties: automatic rewrite rules and strategies. We use example **sm** code to illustrate our approach to code correctness proofs. Figure 2 presents a "min" program that returns in register 3 the location of a least element of the array whose bounds are contained in registers 0 and 1.

We specify the behavior of this program using a PVS function that calculates the result using the same algorithm as the **sm** program.

```
least(max, cur, low, mem): RECURSIVE nat =
 IF (cur < max)
  THEN least(max,cur+1,
           IF mem(cur+1)<mem(low)
            THEN cur+1 ELSE low ENDIF,mem)
  ELSE low ENDIF
 MEASURE max(0,max-cur)
```

For convenience and readability we define functions to return the value of registers, so for example `R0(s)` for **sm** state `s` returns the value of `mem(s)(0)`. Also for convenience we define functions `write`, `goto`, and `update_stk` which update respectively the memory, program counter, and call stack of an **sm** state.

We write a function that calculates the number of instructions that are processed during execution of the subroutine. For the min subroutine example, this function is min_clock. The structure of the "clock" functions parallels the structure of the blocks of code in the program and is used to guide the proof. The function clock_plus is equivalent to natural number plus and is used in clock function definitions to keep the PVS prover from simplifying the expressions. The constant N is the size of **sm** data memory.

```
min_loop_once_clock(s):nat =
 if R2(s)+1<N AND R3(s)<N AND
    mem(s)(R2(s)+1)<mem(s)(R3(s))
  THEN 10 ELSE 8 ENDIF

min_loop_clock(s): RECURSIVE nat =
 if pc(s) = 2 AND defs(s) = program
  AND NOT halted(s)
  AND 5<R0(s) AND R0(s)<=R3(s) AND R3(s)<=R2(s)
  AND R2(s)<R1(s) AND R1(s)<N
 THEN
  clock_plus (min_loop_once_clock(s),
   min_loop_clock(sm(s,min_loop_once_clock(s))))
 ELSE 3 ENDIF
 MEASURE max(0,R1(s)-R2(s))

min_clock(s): nat =
 clock_plus (3,
  clock_plus (min_loop_clock(sm(s,3)), 1))
```

We have chosen a specification style that relies on specifying the complete result of a computation because it simplifies the task of automating proofs involving code. A drawback of this philosophy is that unimportant but hard-to-describe elements must be specified too. We specify the value of these irrelevant state elements using functions defined with the interpreter function in a manner that allows us to specify conveniently the entire state resulting from a computation.

An example of this kind of state element occurs in the min subroutine loop. After each iteration register 5 contains the difference between the least element so far encountered and the current value being checked. Although we could of course specify the final value of the loop for register 5, we would prefer to ignore it since the ultimate value of this temporary register is unimportant. We define a function that calculates the final value of the register using the interpreter:

```
min_loop_unspecified_R5(s):nat =
 R5(sm(s,min_loop_clock(s)))

min_correct_unspecified_R5(s):nat =
 min_loop_unspecified_R5(sm(s,3))
```

Using this function to specify the final value of register 5 eases the proof of the correctness theorem, since the final value of that register is specified to be whatever the interpreter produces. This is of course not very helpful, but it allows us to follow the convention that we specify the entire resulting state while not bothering very much with irrelevant state elements.

We are now ready to state a theorem about the effect of executing the min subroutine on an **sm** state. The PVS terms defs(s) is the program memory and halted(s) is the halted flag of state s. The PVS terms op(i) and arg1(i) are the opcode and first argument of an instruction i. We use the constant program to represent the programs we wish to execute – it is an array of instructions that contains the min program listed in Figure 2.

```
min_correct: LEMMA
 op(defs(s)(pc(s))) = call
 AND arg1(defs(s)(pc(s)))= 0
 AND defs(s) = program AND NOT halted(s)
 AND 5 < R0(s) AND R0(s) <= R1(s) AND R1(s) < N
 =>
 sm(s, min_clock(s)) =
 goto(inc(pc(s)),
 write(2,R1(s),
 write(3,least(R1(s), R0(s), R0(s), mem(s)),
 write(4,0,
 write(5, min_correct_unspecified_R5(s),s)))))
```

2.3 Correctness Theorem Proof

PVS proofs of code correctness theorems like min_correct are relatively straightforward. We build the proof by proving lemmas about the constituent blocks. As suggested in the previous section, the structure of the clock functions guides the proof.

Straightline code is proved using a PVS strategy. The strategy expands to a PVS "grind" command that uses a standard set of lemmas applied as auto-rewrite rules to execute the code symbolically. Loops are proved using a second PVS strategy that expands into a sequence of PVS commands that set up the appropriate inductive argument and simplify as needed. Some lemmas about specification functions like least are typically needed for the proof to complete successfully. In particular, theorems pertaining to the type of the specification functions and how the specification functions relate to each other must be proved.

The use of PVS rewrite rules and PVS strategies aids the development of proofs about code execu-

```
address   code
  30      move 6 0
  31      move 5 1
  32      sub 5 6
  33      jumpz 5 42
  34      move 0 6
  35      call 0
  36      moverind 5 3
  37      moverind 4 6
  38      movewind 3 4
  39      movewind 6 5
  40      incr 6
  41      jump 31
  42      ret
```

Figure 3: **sm** sort subroutine

tion. By following our restrictive conventions about how to express code correctness lemmas these simple PVS strategies work quite well. However, even more importantly they provide a kind of proof resilience. Since the proofs are mostly automatic, modest changes to the code or specifications do not require that a wholly new proof be constructed.

2.4 A Second Example

We present a second example of an **sm** subroutine to emphasize that our approach is indeed largely automatic. Figure 3 presents a subroutine that sorts the array whose bounds are contained in registers 0 and 1. It is implemented using the min program described previously, and its proof is an example of how to build on other subroutine correctness theorems. We automate reasoning about code that calls subroutines as much as possible, and a subroutine correctness theorem of the form we have described does just that. By applying min_correct we can reason about **sm** code that calls min just as we reason about code that employs builtin **sm** instructions.

In order to specify the behavior of the sort subroutine, we define a function `sort` that sorts an array in the manner of the subroutine.

```
sort(cur, max, s): RECURSIVE state =
 if (cur < max)
 THEN let least=least(max, cur, cur, mem(s)) IN
       sort(cur+1,max,write(cur,mem(s)(least),
            write(least,mem(s)(cur),s)))
 ELSE s ENDIF
 MEASURE max(0,max-cur)
```

We prove a theorem about the effect of executing a sort subroutine call.

```
sort_correct: LEMMA
 op(defs(s)(pc(s))) = call
    AND arg1(defs(s)(pc(s)))= 30
    AND defs(s)=program AND NOT halted(s)
    AND 6<R0(s) AND R0(s)<=R1(s) AND R1(s)<N
 =>
 sm(s, sort_clock(s)) =
 goto(inc(pc(s)),
  write(0,if R0(s)<R1(s) THEN R1(s)-1
                         ELSE R0(s) ENDIF,
  write(2,if R0(s)<R1(s) THEN R1(s)
                         ELSE R2(s) ENDIF,
  write(3,sort_unspecified_R3(s),
  write(4,sort_unspecified_R4(s),
  write(5,0,
  write(6,if R0(s)<R1(s) THEN R1(s)
                         ELSE R0(s) ENDIF,
  sort(R0(s),R1(s),s))))))))
```

The proof of sort_correct has the same structure as the proof of min_correct. Each constituent block of code is specified and proved, and the PVS strategies for straightline and loop code are employed. The min_correct theorem is used to reason about the call to the min program, just as with any builtin **sm** opcode.

3 Reasoning About Specifications

Largely automatic proofs of programs as described in the previous section are robust in the sense we need them to be. A change to a program or even to the language semantics defined by the interpreter would require minimal changes in the proofs. However, these kinds of specifications are unsatisfactorily unclear and complex. The specifications reflect the algorithm used by the code and do not effectively convey the needed program functionality.

To use our mostly automatic approach in code proofs we limit ourselves by avoiding existential quantifiers and using a (primarily) first-order, recursive style. But our specification need not be so constrained. For example, what we really want to know about the sorting program is that it produces a sorted permutation of the original array without disturbing irrelevant memory elements. A good specification of the sorting program is:

```
sort_works: LEMMA
 op(defs(s)(pc(s))) = call
  AND arg1(defs(s)(pc(s)))= 30
  AND defs(s)=program AND NOT halted(s)
  AND 6<R0(s) AND R0(s)<=R1(s) AND R1(s)<N
  =>
 let s2= sm(s, sort_clock(s)) IN

% array is sorted
 (FORALL (i,j:subrange(R0(s),R1(s))):
   i<j => mem(s2)(i)<=mem(s2)(j)) AND

% array is permutation
 (EXISTS (f: (bijective?[subrange(R0(s),R1(s)),
                         subrange(R0(s),R1(s))])):
  (FORALL (i:subrange(R0(s),R1(s))):
    mem(s2)(f(i)) = mem(s)(i))) AND

% irrelevant memory is unchanged
 (FORALL (i: address): i>6 AND (i<R0(s) OR i>R1(s)) =>
   mem(s)(i)=mem(s2)(i))
```

The proof of this lemma is relatively straightforward, although like most PVS proofs involving the higher-order language capability of PVS the proofs are less automatic than the proofs in the last section. By applying the lemma proved in the previous section sort_correct, we reduce this theorem to one that does not involve our program or even – except hidden in the "unspecified" functions – the **sm** interpreter. We satisfy the remaining proof obligation involving desired properties and the specification function in conventional PVS proof style.

4 Conclusion

The automation of proofs about computer systems increases robustness and aids formal verification of realistic-sized computer systems. Recursively-defined interpreters can be used to define computer system behavior in a clean and simple way. The usefulness of these techniques transcends the particularities of different theorem proving systems.

Realistic proofs require robustness and PVS is capable of a proof style that fosters resilience in proofs about computer systems. The approach relies in part on restricting the manner in which we describe code execution, but the full PVS logic and prover may be used in concert with our automatic approach.

References

[1] William R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1368–81, November 1989.

[2] William R. Bevier, Warren A. Hunt Jr., J Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

[3] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, Boston, 1988.

[4] Robert S. Boyer and J Strother Moore. Mechanized formal reasoning about programs and computing machines. In R. Veroff, editor, *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*. MIT Press, 1996.

[5] Robert S. Boyer and Yuan Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43(1):166–192, January 1996.

[6] Bishop Brock, Matt Kaufmann, and J Strother Moore. ACL2 theorems about commercial microprocessors. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design - FMCAD*, volume 1166 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[7] Warren A. Hunt, Jr. and Bishop C. Brock. A formal HDL and its use in the FM9001 verification. In C. A. R. Hoare and M. J. C. Gordon, editors, *Mechanized Reasoning and Hardware Design*, pages 35–47, Hemel Hempstead, UK, 1992. Prentice Hall International Series in Computer Science.

[8] Matthew Kaufmann and J S. Moore. ACL2: An industrial strength version of Nqthm. In *Proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS-96)*, pages 23–31, June 1996.

[9] Steven P. Miller, David A. Greve, Matthew M. Wilding, and Mandayam Srivas. Formal verification of the AAMP-FV microcode. Technical report, Rockwell Collins, Inc., Cedar Rapids, IA, 1997. DRAFT.

[10] Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT'95: Workshop on Industrial-Strength Formal specification Techniques*, Boca Raton, FL, 1995. IEEE Computer Society.

[11] J Strother Moore. *Piton - A Mechanically Verified Assembly-Level Language.* Kluwer Academic Publishers, 1996.

[12] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language (Beta Release).* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[13] S. Owre, N. Shankar, and J. M. Rushby. *User Guide for the PVS Specification and Verification System (Beta Release).* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[14] John Rushby. Personal Communication, November 1996.

[15] David M. Russinoff. A mechanically checked proof of IEEE compliance of the AMD K5 floating-point square root microcode. Available as www.onr.com/user/russ/david/fsqrt.html, August 1996.

[16] N. Shankar, S. Owre, and J. M. Rushby. *The PVS Proof Checker: A Reference Manual (Beta Release).* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[17] Matthew Wilding. A mechanically verified application for a mechanically verified environment. In Costas Courcoubetis, editor, *Computer-Aided Verification - CAV '93*, volume 697 of *Lecture Notes in Computer Science.* Springer-Verlag, 1993.

[18] Matthew Wilding. *Machine-Checked Real-Time System Verification.* PhD thesis, University of Texas at Austin, May 1996. Also available as ftp.cs.utexas.edu/pub/boyer/wilding-diss.ps.gz.

Acknowledgements I thank David Greve and David Hardin of the Advanced Technology Center for many suggestions that improved this paper considerably. This work was accomplished under the Rockwell Collins IR&D program.

Domain Checking Z Specifications

Mark Saaltink *
ORA Canada
267 Richmond Road,
Ottawa, ON K1Z 6X3
Canada
mark@ora.on.ca

Abstract

We describe how *guards* can be used to ensure that formulas in a partial logic are meaningful, and how guards and guarded formulas can be proved using classical logic. In addition to this theoretical utility, guards are useful in practice as a simple means of exposing flaws in specifications. We illustrate this use of guards with several examples in the Z specification language, using Z/EVES.

1 Introduction

Z [11, 16, 20, 22] is a formal specification language based on typed set theory, originally developed at Oxford University in the early 1980s. Z has been used in a variety of situations [1, 8] and has become fairly popular.

The Z/EVES system [19, 13] is a formal methods tool that can be used for analysing Z specifications in several different ways, including syntax and type checking; domain checking; schema expansion; precondition calculations; and general theorem proving. Z/EVES is built on top of the EVES system [5], which provides a theorem prover for classical first-order logic (and, in particular, for ZF set theory).

There is a technical hurdle to overcome in using a prover for classical logic to deal with Z: the Z language allows for "undefined" terms while classical logic does not. In this paper, we show how *guard* conditions can be generated for parts of Z specifications; how these conditions guarantee that the use of classical logic is sound for Z; and how these conditions are not just a technical necessity but in fact help uncover flaws in a specification.

*This work was sponsored by the United States Department of Defense under contract MDA904-95-C-2031.

2 The Z Notation

The Z specification notation is based on standard mathematical notions and notations for discrete mathematics. Z specifications are intended to be readable and understandable: standard notations are used (e.g., the normal logical symbols $\wedge$, $\Rightarrow$, and so on; the standard set notations $\in$, $\subseteq$, $\cup$, and so on); a "mathematical toolkit" defines a solid basis of familiar notions of discrete mathematics (such as relations, functions, and sequences). As is common in the development of discrete mathematics, relations are sets of ordered pairs, and functions are relations satisfying a certain property.

Z introduced a special construct, the *schema*, for use in models. Schemas collect together a set of variables used in the model and a set of predicates describing the possible observations of the modelled system. There are several aspects of the schema notation and "schema calculus" that allow quite complex systems to be described in a compact and structured way. We will not, however, delve into the details of schemas here; interested readers can consult the Z books (e.g., [16, 20, 22]).

Z, while based in standard ZF set theory, imposes a system of types, and specifications are rejected if they violate the type checking rules. Each basic "given" set in a specification has an associated type, and types are built from these primitive types by the formation of power types, cross-product types, and binding types (which are analogous to record types). Variables are declared to range over *sets*, rather than types, and the type of the variable is then the element type of the set. Thus, declarations $x : \mathbb{N}$ and $x : \mathbb{Z}$ both determine that x is of integer type (but the first declaration also constrains x to lie in the set of non-negative integers).

3 Undefined Terms in Z

Z breaks from mathematical tradition in allowing for undefined terms (although, to be honest, the mathematical tradition is not very well codified). Undefined terms arise in two possible ways: first, by the application of a function to a value outside its domain. Recall that functions in Z are simply sets of pairs. The type system cannot guarantee that the applied value is even a function, or that the argument is in its domain. Examples of undefined terms are $1 \operatorname{div} 0$ and $\emptyset(1)$, which have arguments outside the domain of the function, and $(\mathbb{Z} \times \mathbb{Z})(0)$, where the applied value is not even a function. Second, an undefined term can arise from the use of a definite description where there are no possible values (as in $\mu\, x : \mathbb{Z} \mid 0 < x < 1$) or where there are two or more possible values (as in $\mu\, x : \mathbb{Z} \mid x > 0$). (The notation "$\mu\, x : S \mid P$" denotes the unique element of S satisfying property P.)

Philosophers and logicians have struggled with undefined terms, without attaining any general agreement on the most suitable treatment of such terms. Many possible approaches to the issue have been formulated [2, 3, 4, 6, 14, 17, 21]. Different Z authors have adopted different positions on undefinedness in Z:

- Spivey [20] states that any atomic predicate containing an undefined term is *indeterminate*, meaning that it has a truth value (so classical logic applies) but he has chosen not to tell us whether it is true or false.

- The draft ISO standard for Z [9] specifies that an atomic predicate containing an undefined term is false. However, the draft is not definitive on which terms are undefined; the semantics leaves open the possibility that, say, $\mu\, x : \mathbb{Z} \mid x > 0$ may or may not have a value. Furthermore, undefinedness is not inherited in all expression contexts, e.g., if $1 \operatorname{div} 0$ is undefined, the set $\{x : -1 \mathinner{..} 1 \bullet 1 \operatorname{div} x\}$ is defined and has the value $\{-1, 1\}$, as any undefined value is simply ignored in a set comprehension.

- Some authors (e.g., [22]) use "strong equality", which is true if both arguments are undefined or if both arguments are defined and equal.

So, the proper treatment of undefinedness in Z is not clear. In any case, it is clear that normal mathematical reasoning might not be applicable. Consider, for example, the formula

$$a \in \{a, b\}.$$

This is always true in classical ZF set theory; it can fail in Standard Z if, say, a is defined and b is undefined (as then $\{a, b\}$ is also undefined).

4 Guards

Our treatment of undefined terms in Z/EVES is based on the notion of *guards*. A guard formula for a given formula has two properties: first, it is never itself undefined if all its free variables have values; and second, if the guard is true (and all free variables have values) the given formula is defined. (A guard thus represents the *presupposition* [3] of a formula, that is, whatever must be true in order for the formula to have meaning.) As described elsewhere [18], provided the partial logic meets certain criteria, if a formula and its guard can be proven in classical logic, then the formula is a theorem of the partial logic. As it turns out, these criteria do not apply to the "semi-classical" semantics proposed by the draft Z Standard. Fortunately, however, the criteria apply to a variety of more stringent ways of handling undefinedness, as described below. So, we apply guards for a stringent handling of undefinedness. Any theorems of the more stringent logic are theorems of the "semi-classical" logic, so we retain soundness. Of course, we lose completeness in doing this, but are not terribly bothered for two reasons: first, in a mechanical theorem prover (like Z/EVES), theoretical completeness is not as interesting a property as the practical ability to prove theorems (even provers using complete techniques may fail to find a proof due to time or space constraints). Second, we are not convinced that the "semi-classical" handling of undefinedness is the most appropriate. It was chosen (we believe) to preserve as much classical reasoning as possible, rather than for the appeal of its semantical interpretation. There is some evidence that the "Kleene" semantics (described below) is the most satisfactory from a theoretical standpoint.

We have identified (among others) two different systems of partial logics [18]: the *strict* system, where any undefined subterm or subformula makes its containing term or formula undefined; the *Kleene* system, which does not insist on strictness but requires *monotonicity* (so that making any part of a term or formula more defined makes the overall meaning more defined), and has its connectives as defined as possible subject to the monotonicity requirement. So, in the Kleene system, the disjunction of a true predicate with an undefined predicate is true, and a quantification like $\exists\, x : \mathbb{Z} \bullet 1 \operatorname{div} x = x$ is true because there is a satisfying value for x (namely 1)—the fact that the body of the quantification is undefined for $x = 0$ then does not matter. However, $\forall x, y : \mathbb{N} \bullet y*(x \mathrm{div} y) \leq x$ is undefined.

Guard formulas are defined with respect to a particular system of logic. For example, guards for the strict logic differ from guards in Kleene's logic. In the strict logic, the guard for a disjunction requires that both disjuncts are defined. For Kleene's logic, one of the disjuncts may be undefined if the other is true (as then the disjunction as a whole is defined and true).

The strict system of partial logic is unsuitable; its guards are too stringent and it is difficult to write meaningful assertions. For example, the formula $\forall x, y : \mathbb{Z} \bullet y > 0 \Rightarrow y * (x/y) = x$ is undefined. To write some defined formula expressing that property, it is necessary to use some awkward formulation like $\forall x, y : \mathbb{Z} \bullet y > 0 \Rightarrow y * (x/max(1, y)) = x$.

The Kleene system is quite usable, as terms and formulas are defined whenever possible. However, we found that the guards are, in general, too complex. Suppose P and Q are predicates, with guards G_P and G_Q respectively. Then the guard for $P \vee Q$ is

$$(G_P \wedge P) \vee (G_Q \wedge Q) \vee (G_P \wedge G_Q).$$

This quickly becomes awkward; for example, the guard for $(P \vee Q) \Rightarrow R$ is (after some simplification!)

$$\begin{aligned} & G_P \wedge P \wedge G_R \\ \vee\ & G_Q \wedge Q \wedge G_R \\ \vee\ & G_P \wedge G_Q \wedge (P \vee Q \Rightarrow G_R) \\ \vee\ & G_R \wedge R. \end{aligned}$$

The guard for quantification is also somewhat awkward; the guard for $\forall x : T \bullet P$ is the formula

$$(\forall x : T \bullet G_P \wedge P) \vee (\exists x : T \bullet G_P \wedge \neg P).$$

In our application of this approach in Z/EVES, we therefore looked for a system intermediate between the strict and Kleene systems. We therefore use a "left-to-right" system of interpretation for the Boolean connectives (e.g., the left argument must be defined; the right argument need not be if the value of the left argument is enough to determine the overall value), and a strict interpretation of the quantifiers. Thus, for $P \vee Q$ we have the guard

$$G_P \wedge (P \vee G_Q)$$

and for $(P \vee Q) \Rightarrow R$ we have the guard

$$G_P \wedge (P \vee G_Q) \wedge (P \vee Q \Rightarrow G_R).$$

For the quantification $\forall x : T \bullet P$, we have the guard $\forall x : T \bullet G_P$. These guards are not too complex, and scale up well for complex formulas. Furthermore, our experience has been that specifiers write formulas that are meaningful in the left-to-right reading (if they are meaningful at all).

People sometimes object that the left-to-right interpretation of the connectives is flawed because of the assymetry, and that not all expected propositional manipulations are valid. This is only partly true in our approach. When guards are generated, the order matters; the guard for $P \vee Q$ differs from the guard for $Q \vee P$. All our proofs, however, are done in classical logic, where the connectives are commutative (and indeed have all their familiar properties).

5 Examples

We claimed above that guards provided a useful error screen. In this section, we present a few examples that we encountered in using Z/EVES to analyse various specifications.

5.1 A POSIX specification

Z is being used to specify the facilities defined in the POSIX 1003.21 specification [7]. Early in that specification we have the definitions of time

$$Time == \mathbb{N},$$

and of priority

$$Priority ::= iprio\langle\langle \mathbb{N}_1 \rangle\rangle \mid dprio\langle\langle Time \rangle\rangle.$$

(This is a "free type" definition, which specifies that the set *Priority* is composed of values of the form $iprio(i)$ for $i \in \mathbb{N}_1$ or of the form $dprio(t)$ for $t \in Time$, that these values are distinct, and that there are no other values in the set. $\mathbb{N}_1$ is the set of positive integers $\{1, 2, 3, \ldots\}$.)

Comparisons between priority levels are then defined:

$$\begin{array}{|l} \_ \succ \_ : Priority \leftrightarrow Priority \\ \_ \succeq \_ : Priority \leftrightarrow Priority \\ \hline \forall p, q : Priority \bullet \\ \quad (\exists i, j : \mathbb{N} \mid p = iprio(i) \wedge q = iprio(j) \\ \quad\quad \bullet p \succ q \Leftrightarrow i > j \wedge p \succeq q \Leftrightarrow i \geq j) \vee \\ \quad (\exists s, t : Time \mid p = dprio(s) \wedge q = dprio(t) \\ \quad\quad \bullet p \succ q \Leftrightarrow s < t \wedge p \succeq q \Leftrightarrow s \leq t) \end{array}$$

This is a Z "axiomatic definition" that declares two functions (above the line) and states their properties (below the line).

Z/EVES accepts the definitions of *Time* and *Priority* without comment. For the specification of $\succ$ and $\succeq$, it generates the formidable guard

$$\begin{aligned}
&(\_ \succ \_) \in \mathit{Priority} \leftrightarrow \mathit{Priority}\\
&\land (\_ \succeq \_) \in \mathit{Priority} \leftrightarrow \mathit{Priority}\\
&\land p \in \mathit{Priority}\\
&\land q \in \mathit{Priority}\\
\Rightarrow\ &(\forall i : \mathbb{N};\ j : \mathbb{N} \bullet\\
&\quad i \in \mathrm{dom}\,\mathit{iprio}\\
&\quad \land (p = \mathit{iprio}(i) \Rightarrow j \in \mathrm{dom}\,\mathit{iprio}))\\
&\land (\ (\exists i\_\_0 : \mathbb{N};\ j\_\_0 : \mathbb{N} \bullet\\
&\quad p = \mathit{iprio}(i\_\_0)\\
&\quad \land q = \mathit{iprio}(j\_\_0)\\
&\quad \land p \succ q \Leftrightarrow i\_\_0 > j\_\_0\\
&\quad \land p \succeq q \Leftrightarrow i\_\_0 \geq j\_\_0)\\
&\quad \lor (\forall s : \mathit{Time};\ t : \mathit{Time} \bullet\\
&\quad s \in \mathrm{dom}\,\mathit{dprio}\\
&\quad \land (p = \mathit{dprio}(s)\\
&\qquad \Rightarrow t \in \mathrm{dom}\,\mathit{dprio}))).
\end{aligned}$$

An application of the Z/EVES "rewrite" command reduces this to the simpler

$$\begin{aligned}
&(\_ \succ \_) \in \mathit{Priority} \leftrightarrow \mathit{Priority}\\
&\land (\_ \succeq \_) \in \mathit{Priority} \leftrightarrow \mathit{Priority}\\
&\land p \in \mathit{Priority}\\
&\land q \in \mathit{Priority}\\
\Rightarrow\ &(\forall i : \mathbb{N};\ j : \mathbb{N} \bullet\\
&\quad i \geq 1 \land (p = \mathit{iprio}\ i \Rightarrow j \geq 1))
\end{aligned}$$

This is obviously not a fact; i and j might be zero. The declaration should have constrained i and j to range over $\mathbb{N}_1$ (the set $\{1, 2, \ldots\}$) instead of $\mathbb{N}$ (which includes 0). With this change, the "rewrite" command reduces the new guard to *true*.

The POSIX specification has been revised since this analysis; this and several other guard failures have been eliminated.

5.2 Jackson's "views" specification

Jackson's paper [10] describes a useful structuring technique for Z specifications. In it, as part of a specification of an editor, we have the following paragraphs. (In fact, to save space Jackson omits these preliminary definitions, but they are easily inferred by the reader.)

The edited data is composed of atomic characters:

$[\mathit{Char}]$

(This is a Z "given set" declaration that states that *Char* is some set.)

Carriage return and space characters exist:

$| \ \mathit{cr}, \mathit{spc} : \mathit{Char}$

as well as a maximum line length:

$| \ \mathit{maxlen} : \mathbb{N}_1$

The "grid view" assembles the edited data into a sequence of lines, each a sequence of characters, and gives row and column coordinates for the cursor:

$$\begin{array}{|l}
\textbf{— } \mathit{Grid} \text{ ———————————}\\
\mathit{lines} : \mathrm{seq}(\mathrm{seq}\ \mathit{Char})\\
x, y : \mathbb{N}_1\\
\hline
y \in \mathrm{dom}\,\mathit{lines}\\
x \in \mathrm{dom}(\mathit{lines}(y))\\
\forall l : \mathrm{ran}\,\mathit{lines} \bullet\\
\quad \#l \leq \mathit{maxlen}\\
\quad \land \mathit{last}(l) \in \{\mathit{spc}, \mathit{cr}\}\\
\quad \land \mathit{cr} \notin \mathrm{ran}(\mathit{front}(l))\\
\hline
\end{array}$$

The set $\mathrm{seq}\,X$ is the set of sequences whose elements lie in the set X; in Z, these are partial functions from $1 \mathinner{..} n$ to X, for some n. Functions dom and ran give the domain and range of a function, respectively. $\#S$ is the cardinality of finite set S.

The guard condition for this paragraph concerns the term $\#l$, which requires l to be finite, and the terms $\mathit{last}(l)$ and $\mathit{front}(l)$, which give the last element of l and the sequence of all but the last elements, respectively, and which are defined for nonempty sequences only. After rewriting, the guard condition is reduced to

$$\begin{aligned}
&\mathit{lines} \in \mathrm{seq}(\mathrm{seq}\ \mathit{Char})\\
&\land x \in \mathbb{Z}\\
&\land x \geq 1\\
&\land y \in \mathbb{Z}\\
&\land y \geq 1\\
&\land y \leq \#\mathit{lines}\\
&\land x \in \mathrm{dom}(\mathit{lines}(y))\\
\Rightarrow\ &(\forall l : \mathrm{ran}\,\mathit{lines} \bullet\\
&\quad (\#l \leq \mathit{maxlen} \Rightarrow \neg\, l = \langle\rangle)\\
&\quad \land \quad (\#l \leq \mathit{maxlen}\\
&\qquad\quad \land (\mathit{last}(l) = \mathit{cr} \lor \mathit{last}(l) = \mathit{spc})\\
&\qquad \Rightarrow \neg\, l = \langle\rangle))
\end{aligned}$$

There is clearly a problem here, as nothing prevents $\mathit{lines} = \langle\langle\rangle\rangle$ (that is, a single line containing no characters), which would make the guard fail. Lines must have nonzero length according to this specification. This fact might be of some significance to the specifiers, implementors, and users of the system! Indeed, we might now wonder how an empty file would be represented, or whether it can be distinguished from a file containing a single carriage return.

The "delete line" operation is specified as

$$\begin{array}{|l}
\textit{Grid\_delLine} \\
\Delta \textit{Grid} \\
\hline
\textit{lines}' = (1 \mathrel{..} y - 1) \lhd \textit{lines} \\
\qquad\qquad {}^\frown\ (y + 1 \mathrel{..} \#\textit{lines}) \lhd \textit{lines} \\
x' = 1 \land y' = y \\
\hline
\end{array}$$

(When f is a function, $X \lhd f$ is the "domain restriction" of f to X; function ${}^\frown$ appends two sequences.)

The intention in this definition is to form a new sequence of lines by concatenating the lines before the current line and the lines after the current line. (The most recent version of the Mathematical Toolkit as described by Spivey [20] has additional sequence functions that make this easier.) Unfortunately, this plan fails. The guard is

$$\begin{array}{ll}
& \Delta \textit{Grid} \\
\Rightarrow & \textit{lines} \in \mathrm{dom}\,\# \\
& \land\ ((1 \mathrel{..} y - 1) \lhd \textit{lines}, \\
& \quad (y + 1 \mathrel{..} \#\textit{lines}) \lhd \textit{lines}) \in \mathrm{dom}(\_ {}^\frown \_)
\end{array}$$

which the "rewrite" command reduces to

$$\begin{array}{l}
\quad \Delta \textit{Grid} \\
\quad \land \neg\ 1 + y \leq 1 \\
\quad \land \neg\ \#\textit{lines} < 1 + y \\
\Rightarrow 1 + y > \#\textit{lines},
\end{array}$$

which, while somewhat torturous, shows that there is a problem if y is in the range $1 \mathrel{..} \#\textit{lines}$. Indeed, the formula that takes the part of the sequence after the yth element, $(y+1 \mathrel{..} \#\textit{lines}) \lhd \textit{lines}$, returns a partial function whose domain runs from $y + 1$ to $\#\textit{lines}$. The argument of concatenation, however, must be a sequence (i.e., a partial function whose domain runs from 1 to n for some n).

This error is one that would not likely disturb a reader; the author's intention was certainly clear enough. Correcting the error is, however, an important prelude to formal manipulation of the specification. In formal manipulation, the incorrect term could be copied to a context where the intended interpretation would no longer be clear, and incorrect conclusions might be drawn.

5.3 Signalling System No. 7

There are many examples of Z specifications in Woodcock and Davies' book *Using Z* [22]. We have analysed only a few. Here is an example from Chapter 20, which describes a signalling system. In this specification, it is necessary to define a "pairwise concatenation" operation, which takes two sequences of sequences and returns a new sequence, each element of which is the concatenation of the corresponding elements of the original sequences. The definition given is

$$\begin{array}{|l}
[X] \\
\_ \mathbin{\hat{\frown}} \_ : \mathrm{seq}(\mathrm{seq}\,X) \times \mathrm{seq}(\mathrm{seq}\,X) \\
\qquad \to \mathrm{seq}(\mathrm{seq}\,X) \\
\hline
\forall s, t : \mathrm{seq}(\mathrm{seq}\,X) \mid \#s = \#t \bullet \\
\quad \forall i : \mathrm{dom}\,s \bullet \\
\qquad (s \mathbin{\hat{\frown}} t)(i) = (s\ i) {}^\frown (t\ i) \\
\hline
\end{array}$$

The guard condition can be rewritten to

$$\begin{array}{l}
\quad (\_ \mathbin{\hat{\frown}} \_) \in \mathrm{seq}(\mathrm{seq}\,X) \times \mathrm{seq}(\mathrm{seq}\,X) \\
\qquad\qquad \to \mathrm{seq}(\mathrm{seq}\,X) \\
\quad \land\ s \in \mathrm{seq}(\mathrm{seq}\,X) \\
\quad \land\ t \in \mathrm{seq}(\mathrm{seq}\,X) \\
\quad \land\ \#s = \#t \\
\Rightarrow (\forall i : 1 \mathrel{..} \#s \bullet \\
\quad i \in \mathrm{dom}(s \mathbin{\hat{\frown}} t) \\
\quad \land\ s(i) \in \mathrm{seq}\,X \\
\quad \land\ t(i) \in \mathrm{seq}\,X)
\end{array}$$

The last two conjuncts of the conclusion are in fact true, (and can be shown with further proof steps). The first conjunct is not necessarily true; the specifiers have forgotten to tell us (formally) what the length of $s \mathbin{\hat{\frown}} t$ is. It would thus be useful to add the conjunct

$$\forall s, t : \mathrm{seq}(\mathrm{seq}\,X) \mid \#s = \#t \bullet \#(s \mathbin{\hat{\frown}} t) = \#s$$

to the definition.

6 Relationship to Other Work

To our knowledge, no other tool for Z has adopted a similar approach to undefinedness. However, other theorem provers and formal methods tools have analogous mechanisms.

6.1 ACL2

The ACL2 prover [12] implements a quantifier-free first-order logic for a subset of Common Lisp. The logic of ACL2 is classical and all functions are total. Common Lisp functions, in contrast, are partial. The semantics of ACL2 functions are totalizations of the corresponding Common Lisp functions. ACL2 defines a notion of guards; if the arguments to a function call satisfy its guards then the call can be evaluated in Common Lisp without error, and the result of the evaluation agrees with the ACL2 logical value of the function application.

The significance of guards in ACL2 is quite different than their significance in Z/EVES; guards allow the partial Common Lisp functions instead of their (total) ACL2 counterparts to be used in executing specification expressions, and allow some run-time checks to be omitted. Thus, ACL2 guards concern execution and evaluation only, and play no logical role.

6.2 PVS

The PVS system [15] uses an undecidable type system, with all functions total over their types. The type checker generates "type correctness conditions" (TCCs) that appear to be similar to guards. These TCCs need to be proved to ensure that all arguments of function applications lie within their types. While the logical content of TCCs are similar to those of guards, the significance of TCCs is different, as they are deeply interwoven into the semantics of the PVS language. Guards, in contrast, are defined independently of the various semantical systems of partial and total first-order logic. Thus, we can (and do) relate guards to different systems of semantics of partial logic. For PVS, such a discussion seems impossible.

7 Conclusions

Guard conditions let us use classical logic to work on proofs in a partial logic, and, in particular, let us use the EVES prover to reason soundly about Z specifications.

Guard conditions have another benefit: they let us check that a specification is meaningful. This does not guarantee that the meaning is correct, but saying something is surely better than saying nothing!

We have found unprovable guards for the majority of Z specifications we have examined. As the examples show, the errors exposed by guard conditions range from the trivial oversights, such as the application of # (determining the number of elements) to a set that might not be finite, to more serious omissions (as in Jackson's specification above).

These results show that having a formal specification is no guarantee of perfection; formal specifications can have errors. Furthermore, these errors may not be easily detected in casual reviews. Domain errors are just one of a class of semantic errors that can be found in specifications; others include inconsistency, omitted cases, and unintentional underspecification. Tools like Z/EVES, which support formal manipulations of specifications, can be helpful in analysing specifications to ensure that these types of errors do not occur.

References

[1] R. Barden, S. Stepney, and D. Cooper. *Z in Practice.* BCS Practitioner Series. Prentice Hall, 1994.

[2] Michael J. Beeson. *Foundations of Constructive Mathematics.* Ergebnisse der Mathematik und ihrer Grenzgebiete; 3. Folge · Band 6. Springer-Verlag, 1985.

[3] S. Blamey. Partial logic. In D. Gabbay and A. Blikle, editors, *Handbook of Philosophical Logic.* D. Reidel Publishing Company, 1986.

[4] J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. In C. Morgan and J. Woodcock, editors, *Proceedings of the Third Refinement Workshop.* Springer-Verlag, 1990.

[5] Dan Craigen, Sentot Kromodimoeljo, Irwin Meisels, Bill Pase, and Mark Saaltink. EVES: An Overview. In *Proceedings of VDM '91, Noordwijkerhout, The Netherlands (October 1991).* Springer-Verlag, 1991.

[6] W. M. Farmer. A partial functions version of Church's simple theory of types. *Journal of Symbolic Logic*, 55(3):1269–91, 1990.

[7] IEEE P1003.21 Working Group. POSIX 1003.21 realtime distributed systems communication application program interface (API) [Language Independent], 1996. Draft LIS/V1.0.

[8] I. J. Hayes, editor. *Specification Case Studies.* Prentice Hall International Series in Computer Science, 2nd edition, 1993.

[9] ISO. Z notation. Technical report, ISO/IEC JTC1/SC22 N1970, 1995. ISO CD 13568; Committee Draft of the proposed Z standard.

[10] Daniel Jackson. Structuring Z specifications with views. *ACM Transactions on Software Engineering and Methodology*, 4(4):365–398, October 1995.

[11] Jonathan Jacky. *The way of Z: Practical programming with formal methods.* Cambridge University Press, 1997.

[12] Matt Kaufmann and J Strother Moore. ACL2: An industrial strength version of Nqthm. In *COMPASS '96 (Proceedings of the Eleventh Annual Conference on Computer Assurance)*, pages 23–34, Gaithersburg, MD, June 1996. IEEE Washington Section.

[13] Irwin Meisels and Mark Saaltink. The Z/EVES reference manual. Technical Report TR-96-5493-03b, ORA Canada, November 1996.

[14] Olaf Owe. Partial logics reconsidered: A conservative approach. *Formal Aspects of Computing*, 5(3):208–223, 1993.

[15] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[16] B. F. Potter, J. E. Sinclair, and D. Till. *An Introduction to Formal Specification and Z.* Prentice Hall International Series in Computer Science, 2nd edition, 1996.

[17] Bertrand Russell. On denoting. *Mind*, 14:479–493, 1905.

[18] Mark Saaltink. Dealing with the undefined in classical logic. Technical Report WP-97-5493-242, ORA Canada, March 1997.

[19] Mark Saaltink. The Z/EVES system. In *Proceedings of the 10th International Conference of Z Users (ZUM'97)*. Springer-Verlag, 1997.

[20] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[21] Patrick Suppes. *Introduction to Logic*, chapter 8.7: Five Approaches to Division by Zero, pages 166–168. van Nostrand, 1957.

[22] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement.* Prentice Hall International Series in Computer Science, 1996.

Fundamental Hardware Design in PVS

James F. Leathrum, Jr.*
Electrical and Computer Engineering
Old Dominion University
Norfolk, VA 23529
leathrum@ee.odu.edu

Abstract

The development of programmable logic devices (PLDs) has introduced programming as a primary tool in the development of digital circuits. This work attempts to create a generic verification environment in which designs can be specified and verified using the Prototype Verification System (PVS). This is accomplished by providing library support for general hardware constructs. The environment is intended for use with any PLD and any PLD programming language. The goal of the environment is to allow the easy translation of digital designs to PVS and provide sufficient support to make verification possible without a great deal of effort.

1 Introduction

The design of digital hardware has become increasingly a programming exercise. As such, it is unsettling to see the design practice become a repetitive effort of code development followed by compilation and simulation to determine the soundness of the implementation. Programming is extremely difficult to develop without bugs which must then be removed to produce a final product as bug-free as possible. However, this approach is not acceptable at the hardware level where future patches cannot be readily released to fix bugs found by users. Therefore, a process is proposed in this paper to augment hardware development to verify certain design aspects prior to simulation, and to handle situations which are difficult or impractical to test by simulation.

Programmable logic devices (PLD) are generally programmed by use of a hardware description language such as VHDL or PLDasm. A description of the hardware activity is described in the language which is then compiled for simulation and to program the appropriate device. The designs are characterized by a set of equations to define combinational logic, a set of modular devices to be used in the design (counters, registers, memory, etc.), and a description of a state machine to drive the design.

Other work in the field includes [1], [2] and [7]. However, general work has focused on either demonstrating how basic digital logic can be specified in a formal logic [1], or how to formally verify a hardware description language [2]. This work focuses on the specification of digital systems which are defined in an arbitrary hardware description language and then mapped into an appropriate formal logic. Thus the focus is not so much on low level logic or languages, but rather on the general specification of a digital system and the support to assist in the verification of a design. The design is studied, not the actual implementation (translation errors are not considered).

This paper focuses on the use of the Prototype Verification System (PVS) [3] to formally define a design and to verify certain characteristics about a design. The process can verify that a design meets certain specifications and that it satisfies some basic requirements of digital design. The work is divided into three parts:

1. the development of a support environment for the definition of digital circuits by introducing constructs in PVS for management of hardware devices such as buses, tristate devices, and open collector devices,
2. the development of MSI building blocks for digital design such as counters, multiplexers, and decoders,
3. the development of a process for verifying state machine design.

The end result is the initial environment for the future development of programmable hardware. The process is not proposed to replace simulation as it does not address issues such as timing, but rather to augment

*Work completed under NASA Grant NAS-1-19858 Task #84.

the design process to eliminate as many errors as possible as early in the design cycle as possible. The process is language independent, which unfortunately requires the user to translate a design from the implementation language to th PVS language.

2 PVS

PVS is a verification system that provides support for general purpose theorem proving. The specification language is a higher order logic augmented with dependent types. Theories can be parameterized, and the dependent type mechanism allows for stating arbitrary constraints on theory parameters. The type system of PVS includes predicate subtypes and is therefore undecidable. PVS frequently generates proof obligations to ensure that expressions are well typed. PVS has powerful decision procedures, so many proofs involving simple arithmetic expressions can be discharged automatically. In addition, PVS provides a collection of pre-proven results in the `prelude`. Also included with PVS are libraries providing support for bit-vectors and finite sets. A more complete description of PVS can be found in [3].

3 Digital Constructs

3.1 Bit Vectors

Bit vectors [4] were selected as the underlying logic representation. Since library support already exists, the majority of work is therefore complete. The library includes theories to handles bits as well as bit vectors. All the necessary logic functions are included, as well as useful mappings to integers and naturals. The theories map easily to the necessary line and bus logic for digital devices (with a *bit* representing a *line* and a *bvec*, or bit vector, representing a *bus*). They do not, however, handle tristate devices and open collector devices as easily, but theories for those devices are presented below.

3.2 Tristate Devices

Tristate devices are a class of devices whose outputs can be connected together provided that only one device is asserting an output at a time while all other devices are in a high impedance state. This is a convenient mechanism for bus management.

The theory *tristate* defines a tristate line. The theory is parameterized by the natural number *N* to define the number of devices whose outputs are connected to the tristate line. This is crucial since a tristate line is actually handled like any other digital device as will be demonstrated. The key benefit to the theory is that the user of the theory is forced to verify that only one device may drive a tristate line at a time.

A tristate line is defined as a digital device which has a set of values applied to the line and a set of control signals indicating which driving devices are active. The tristate line then takes on a value based on the set of control signals and the output for the appropriate driving device. The set of outpus which can drive the line have the type:

```
tristate_outputs : type = bvec[N]
  tristate_control : type =
    {c : bvec[N] |
      count[N,bit](c,1) <= 1}
```

The line is then defined by:

```
tristate_line : type =
  {tsl : [tristate_control,
    tristate_outputs -> bit] |
      forall (o : tristate_outputs,
              c : tristate_control,
              i : below(N)) :
        c(i)=1 => tsl(c,o)=o(i)}
```

such that if a control line c(i) is 1, the tristate line takes on the value of the corresponding device driving the line. Note that the case where no control line is set, the tristate line is undefined. This corresponds with the high impedence state where the output is undefined.

A tristate bus is then defined to be a set of tristate lines, all with common control. A bus with *N* output devices driving the bus and *M* lines on the bus is defined as:

```
tristate_bus_out : type =
  [below(M) -> tristate_outputs]

tristate_bus : type =
  {tsb : [tristate_control,
          tristate_bus_out ->
          bvec[M]] |
     forall (c : tristate_control,
             o : [below(M) ->
                   tristate_outputs],
             i : below(N)) :
       c(i)=1 =>
         forall (j : below(M)) :
           tsb(c,o)(j) = o(j)(i)}
```

3.3 Open-Collector Devices

Open-collector devices are conceptually similar to tristate devices. A line driven by open-collector devices is viewed as a device itself which takes the set of outputs placed on the line and derives the appropriate value for the line. An open collector line is defined by:

```
open_collector_output : type = bit

open_collector_line : type =
  {ocl : [bvec[M] -> bit] |
    forall (bv : bvec[M]) :
      ocl(bv) = and_rec( bv)}
```

The function `and_rec` computes the and of all the bits in the bit vector *bv*. Also, a type has been defined for an *open_collector_output* to differentiate it from a standard TTL device with an output *bit*. Then an example of an open collector device is:

```
and_oc (i1, i2 : bit) :
 open_collector_output = i1 and i2
```

and all other open-collector devices are defined similarly.

3.4 Sequential Components

Sequential components are defined based on the inputs to the component and the current state of the component to compute outputs and next states. This is demonstrated with simple JK flip flops.

A function is designed for a type of sequential component. However, each component of a given type in a circuit is distinctly different from others of the same type as differentiated by its current state information. For a flip flop, its current state is maintained in a variable of type

```
FF_state : type = bit
```

A different variable is defined for each instance of the component in the circuit. The function defining a JK flip flop is:

```
JK_FF (J, K : bit, CLR, SET : bit,
       CS : FF_state) :
  [# Q, Q_inv : bit,
     NS : FF_state #] =
  let
    JK_cond =
      if (J = 0 & K = 0) then CS
      elsif (J = 0 & K = 1) then 0
      elsif (J = 1 & K = 0) then 1
      else not(CS)  % (J=1 & K=1)
  in
(#
  Q     := CLR and (SET or CS),
  Q_inv := not(CLR and (SET or
           CS)),
  NS    := CLR and (SET or JK_cond)
#)
```

Note that the outputs are a function of the current state and the asynchronous inputs *CLR* and *SET*. The next state determines the state of the flip flop after the next clock transition.

The axiomatic definition of the JK flip flop in Figure 1 might appear as:

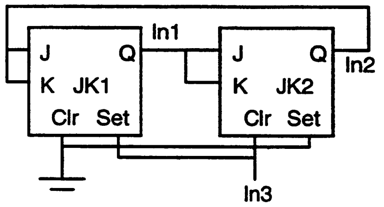


Figure 1.

```
Jin, Kin, CLRin : var line
SETin, Qout, Qinvout : var line
JK1 : var signal[FF_state]

JK1_connect : axiom
Qout(t) =
  Q( JK_FF( Jin(t), Kin(t),
    CLRin(t), SETin(t), JK1(t)) &
Qinvout(t) =
  Q_inv( JK_FF( Jin(t), Kin(t),
    CLRin(t), SETin(t), JK1(t)) &
JK1(t+1) =
  NS( JK_FF( Jin(t), Kin(t),
    CLRin(t), SETin(t), JK1(t))
```

where *t* is defined to have type `time` (a natural number) and line defines the type `signal[bit]` where `signal` allows a type to vary with time (a strategy adopted from [7]). The time relationship between *Qout*, *Qinvout*, and *JK1* reflects the current outputs versus the next state information.

4 MSI Specification in PVS

Given the definition of basic digital devices, it is standard practice in digital design to use "building blocks," or standard components which are pieced together in a design. This is true both in classical digital design and with programmable logic devices. Several basic building blocks are presented here to demonstrate how they are specified in PVS as behavioral entities where the inner specifications are not of concern, but rather an implementation is developed and then verified against the behavioral system. Two sets of examples are presented:

encoders and decoders to demonstrate combinational circuits, and registers and counters for sequential circuits.

4.1 Combinational Logic: Encoders and Decoders

The theory encoder_decoder is defined for multiplexers, demultiplexers, encoders, and decoders.

```
encoder_decoder
[N : {p:posnat |exists (n:posnat)
                    : 2^n = p}] :
THEORY
BEGIN

bv_lib : library = "bitvectors"
importing bv_lib@bv_top
n : var nat
M : {p : posnat | exp2(p) = N}
bv : var bvec[N] % input or output
                 % lines
s : var bvec[M]  % select lines
b : var bit  % input or output bit
v : var bvec[M]

mux (bv,s) : bit = bv( bv2nat( s))

demux (b, s) : bvec[N] =
  (lambda (n : below(N)) :
    if n = bv2nat( s) then b
    else 0 endif)

decode (s) : bvec[N] =
  (lambda (n : below(N)) :
    if n = bv2nat( s) then 1
    else 0 endif)

mux_demux : lemma
  let n : below(N) = bv2nat( s) in
  demux( mux( bv,s), s)(n) = bv(n)

demux_mux : lemma
  mux( demux( b, s), s) = b

decode_mux : lemma
  (v = s =>
    mux( decode( s), v) = 1) &
  (v /= s =>
    mux( decode( s), v) = 0)

decode_count : lemma
  count(decode(s),1) = 1

END encoder_decoder
```

It is assumed that all encoders or decoders will have a power of 2 number of inputs and outputs (due to the binary encoding of the select lines). This theory demonstrates the real benefit of using the bit vector library with the ability to convert bit vectors to natural numbers and vice versa.

The multiplexer function mux is an excellent example where the select lines *s* are converted to a natural number which is then used to index the inputs to select the appropriate output value. The demultiplexer function demux demonstrates how each output is specified using a lambda function where an element of the output bit vector is equal to the input b if the natural number representation of the select lines equals the index of the element in the output vector, else it is 0. A decoder is defined similarly. Then several well know properties of encoders and decoders are verified. The lemma decode_count is particularly useful when dealing with control of tristate devices where the selected device is encoded as a binary value which is then passed through a decoder to select the appropriate device. Since at most one device can drive the line at a time as encapsulated in the type tristate_control, then when a decoder is applied as the selection device for tristate lines, this theory easily discharges the type condition on tristate_control.

4.2 Sequential Logic

4.2.1 Registers

Registers are presented as a sequential circuit building block. A register can be easily defined given state information:

```
reg_state : type = bvec[N]
```

and the function:

```
ts_bus : type =
  [below(N) -> tristate]

register (invec : bvec[N],
          load, clr, OE : bit,
          CS : reg_state) :
 [# outvec : ts_bus, OE_out : bit,
    NS : reg_state #] =
 (#
   outvec := CS,
   OE_out := OE,
   NS := if clr then bv0
         elsif load then invec
         else CS endif
 #)
```

Note that this register has a synchronous clear as opposed to the JK flip flop which had an asynchronous clear and set. Therefore, the output vector is simply a function of the current state and the clear signal only has an affect on the next state.

Variations of the register and other registers such as shift registers can be similarly specified.

4.2.2 Counters

A binary counter is introduced. The counter handles N-bit quantities with a carry in generating a carry out and a next count value. The counter also has parallel load capabilities:

```
counter
  (count, par_load : bvec[N],
   carry_in, load : bit) :
[# carry_out : bit,
   next_count : bvec #] =
let
  n : below(exp2(N)) =
    bv2nat(count),
  max : bool = (n = exp2(N)-1) in
(#
  carry_out :=
    if (load = 0) and max and
       (carry_in = 1)
      then 1 else 0 endif,
  next_count :=
    if load = 1 then par_load
    elsif max and (carry_in = 1)
      then bvec0
    elsif carry_in = 1 then
      nat2bv( n+1)
    else count endif
#)
```

The value of count is the current state information, and is not part of the external circuit. The other inputs are actual inputs to the counter. The counter is then be verified to operate as a natural number counter modulo 2^N:

```
counter_binary_lemma : lemma
   bv2nat( next_count(
     counter_binary( count,
       par_load, 1, 0))) =
   mod( bv2nat( count) + 1,
        exp2(N))
```

This is now a general purpose tool to use in a digital design. An example use (using an axiomatic definition of the circuit) would be:

```
% current count state
current_state : var bvec[N]

% circuit description:
line1, line2, line3 : var bit
bus1, bus2 : var bvec[N]

counter1 : axiom
  bus2 = next_count( counter(
    current_state, bus1, line1,
    line2)) &
  line3 = carry_out( counter(
    current_state, bus1, line1,
    line2))
```

4.4 Example: Registers with outputs on a common bus

An example is given which demonstrates the use of sequential logic, combinational logic, and tristate devices. The circuit is a simple four register system connected to a common bus for data transfer shown in Figure 2. Control of the registers is provided by three 2-bit control buses (*bus_s1, bus_s2, bus_3*) and two control enable lines (*ln1, ln2*). *Bus_s1* selects which register will load its value from the common bus *cbus* and the control is enabled by *ln1*. *Bus_s2* selects which register will be cleared and *ln*2 enables the signal. *Bus_s3* selects which register will drive the common bus *cbus*.

```
t : var time
% register states
R0, R1, R2, R3 : var reg_state
% decoder outputs
bus_b1, bus_b2, bus_b3 : var bus[4]
% decoder control
bus_s1, bus_s2, bus_s3 : var bus[2]
cbus : var bus[16] % common bus
ln1, l2 : var line % demux inputs
% tristate control
tsc : var signal[
      tristate_control[4]]
% outputs to tsb
tsb_out : var signal[
      tristate_bus_out[4,16]]
% tristate bus
tsb : var tristate_bus[4,16]

% control for load
demux1_connect : axiom
  bus_b1(t) = demux( ln1(t),
                     bus_s1(t))

% control for clr
demux2_connect : axiom
  bus_b2(t) = demux( ln2(t),
                     bus_s2(t))
```

```
% control for output enable
decode_connect : axiom
  bus_b3(t) = decode( bus_s3(t))

R0_connect : axiom
tsb_out(t)(0) = outvec(
  register( cbus(t), bus_b1(t)(0),
    bus_b2(t)(0), bus_b3(t)(0),
    R0(t))) &
tsc(t)(0) = OE_out( register(
  cbus(t), bus_b1(t)(0),
  bus_b2(t)(0), bus_b3(t)(0),
  R0(t))) &
R0(t+1) = NS( register( cbus(t),
  bus_b1(t)(0), bus_b2(t)(0),
  bus_b3(t)(0), R0(t)))

R1_connect : axiom
tsb_out(t)(1) = outvec(
  register( cbus(t), bus_b1(t)(1),
  bus_b2(t)(1), bus_b3(t)(1),
  R1(t))) &
tsc(t)(1) = OE_out( register(
  cbus(t), bus_b1(t)(1),
  bus_b2(t)(1), bus_b3(t)(1),
  R1(t))) &
R1(t+1) = NS( register( cbus(t),
  bus_b1(t)(1), bus_b2(t)(1),
  bus_b3(t)(1), R1(t)))

R2_connect : axiom
tsb_out(t)(2) = outvec(
  register( cbus(t), bus_b1(t)(2),
  bus_b2(t)(2), bus_b3(t)(2),
  R2(t))) &
tsc(t)(2) = OE_out( register(
  cbus(t), bus_b1(t)(2),
  bus_b2(t)(2), bus_b3(t)(2),
  R2(t))) &
R2(t+1) = NS( register( cbus(t),
  bus_b1(t)(2), bus_b2(t)(2),
  bus_b3(t)(2), R2(t)))

R3_connect : axiom
tsb_out(t)(3) = outvec( register(
  cbus(t), bus_b1(t)(3),
  bus_b2(t)(3), bus_b3(t)(3),
  R3(t))) &
tsc(t)(3) = OE_out( register(
  cbus(t), bus_b1(t)(3),
  bus_b2(t)(3),
  bus_b3(t)(3), R3(t))) &
R3(t+1) = NS( register( cbus(t),
  bus_b1(t)(3), bus_b2(t)(3),
  bus_b3(t)(3), R3(t)))

tristate_connect : axiom
  cbus(t) = tsb(tsc(t),tsb_out(t))
```

Due to the type of the tristate bus, PVS typechecking requires the verification that only one of the output enable lines will be active at a time. This is accomplished using the lemma `decode_count`.

5 State Machine Specification

State machine design is the core of digital design. A classic digital design is composed of an architecture

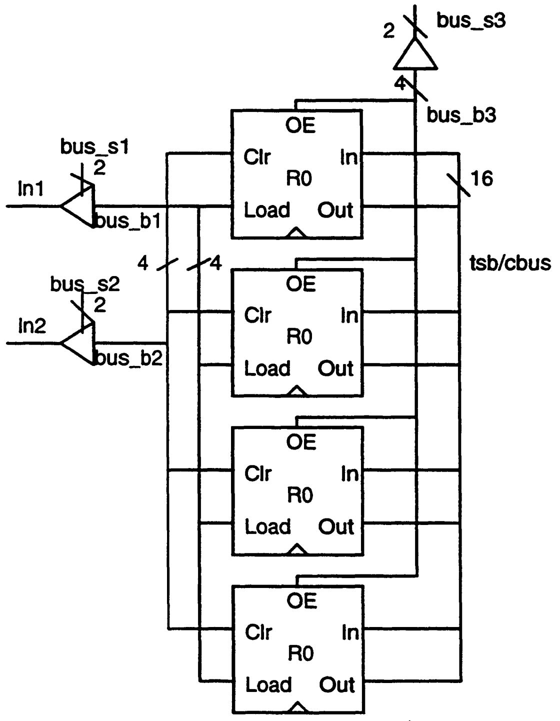


Figure 2.

and a controller with the controller providing commands to the architecture and the architecture providing status information to the controller. Each of these is frequently defined as a state machine. The process of development of a state machine is comprised of two parts, the state machine definition and the state machine implementation. The first component must be performed, while the second may be automatic depending on whether the hardware programming system can generate the appropriate design directly from the state machine definition.

Certain components of a sequential design should be verified prior to implementation. In particular, the design should account for all possible input combintations to generate appropriate state transitions and state transitions should be to valid states. Also, all valid states should be reachable given a reset condition and a set of input conditions. Several theories are presented which force the designer to verify these properties. The theories have been designed such that they are independent of the implementation logic. The example given is based on using bit vectors as previously used, but the theories are not specified using bit vectors, but rather the implementation type is passed as a parameter to the theory. There are four theories:

Theory `State_Verification_1`:

Given a set of states, a set of valid states, a reset function, a state transition function, and input conditions, verify that:
a) the reset function maps from a state to a valid state.
b) given a valid state, the state transition function maps to a valid state.
c) all states are reachable after a reset.

Theory `State_Implementation_1`:

Given a mapping function to map from the state space to state variables and a state equation function to define the state transitions in terms of state variables, verify that the mapping function is unique and that the state equations behave in the same manner as the state transition function from *State_Verification_1*.

Theory `State_Verification_2`:

Theory State_Verification_2 is the same as `State_Verification_1` except that the state transition function must map all states to a valid state. This is particularly useful in the development of systems for use in unpredictable environments where a power surge could result in the circuit transitioning undesirably into an undefined state.

Theory `State_Implementation_2`:

Theory State_Implemenation_2 is the same as `State_Implementation_1` except that the state machine must satisfy `State_Verification_2`.

`State_Verification_1` and `State_Implementation_1` are demonstrated here and the others are just minor modifications of these two.

5.1 Abstract State Machine Verification

The verification of the properties of the state transition function is handled by:

```
State_Verification_1
% given: set of states, set of
%   valid states, reset function,
%   a state transition function,
% verify that:
%   a) reset function maps from a
%      state to a valid state
%   b) given a valid state, the
%      state transition function
%      maps to a valid state
[state : type,
 valid_state : setof[state],
 logic_type : type,
 Nic : nat, %# of input conditions
 input_condition_type : type,
 state_transition :
   [(valid_state),
    input_condition_type ->
    (valid_state)],
 reset : (valid_state),
 reset_value : logic_type] :
Theory
Begin
% This theory is simply a
% mechanism to make sure that a
% state machine designer maintains
% certain necessary conditions for
% the state machine.  Importing
% this theory will verify that the
% functions have the appropriate
% structure.

Assuming

Importing signal
states : var signal[state]
reset_signal : var
  signal[logic_type]
input_conditions : var
  signal[input_condition_type]
t : var time
S : var state

state_transition_assumption :
  assumption
  valid_state( states(t)) =>
  states(t+1) =
    if (reset_signal(t) =
        reset_value) then reset
    else state_transition(
      states(t),
      input_conditions(t)) endif

reachable : assumption
  exists (ti : time) :
    reset_signal(ti) = reset_value
    =>
    exists (input_conditions, t) :
      states(t) = S
```

```
EndAssuming

End State_Verification_1
```

State_transition_assumption states how the system should behave: given a valid state, the system will either reset or will transition to a new state. Then reachable verifies that all valid states are reachable, i.e. there exists a set of input conditions which will transition the current state to a given state.

5.2 State Machine Implementation Verification

The implementation of the state machine is verified to satisfy the state machine by the theory State_Implementation_1. The theory accepts the same parameters as State_Verification_1 with the additional parameters:

```
state_impl : type,
mapping : [state -> state_impl],
state_equations : [state_impl,
  input_condition_type ->
  state_impl]
```

The basis of the theory are then an assumption that the mapping from states to state variables must be unique which is handled for states *S* and *T*:

```
mapping_unique : assumption
  T /= S <=>
  mapping(T) /= mapping(S)
```

and the function state_equations properly implements state_transition:

```
state_equations_correct :
  assumption
  state_equations( mapping(V),I) =
  mapping( state_transition(V,I))
```

These assumptions assist in the development of correct implementations of state machines.

5.3 State Machine Example: Local Bus Controller

An example of a bus controller as taken from *PLDshell Plus/PLDasm User's Guide V4.0* [5] is provided. The problem is specified in PLDasm as (the PLDasm code is shown to demonstrate the ease of mapping the definition into PVS):

```
STATE MOORE_MACHINE

;define defaults
OUTPUT_HOLD LOCAL MEMORY INTACK
DEFAULT_BRANCH S0

;state assignments
S0 = /Q3 * /Q2 * /Q1 * /Q0
S1 = /Q3 * /Q2 * /Q1 *  Q0
S2 = /Q3 * /Q2 *  Q1 * /Q0
S3 = /Q3 * /Q2 *  Q1 *  Q0
S4 = /Q3 *  Q2 * /Q1 * /Q0
S5 = /Q3 *  Q2 * /Q1 *  Q0
S6 = /Q3 *  Q2 *  Q1 * /Q0
S7 = /Q3 *  Q2 *  Q1 *  Q0
S8 =  Q3 * /Q2 * /Q1 * /Q0

;state transitions
S0 := MEM_REQ -> S1 ;S1 on local
                    ;memory request
   + INT_REQ -> S3  ;S3 on inter-
                    ;rupt request
   ; no-op (S0) if no request

;memory cycles
S1 := VCC -> S2 ;one fixed wait
                ;state
S2 := BUS_CONT -> S7 ;S7 if ready
   + BUS_WAIT -> S2  ;stay if not
                     ;ready

;interrupt cycles
S3 := BUS_CONT -> S4 ;S4 if ready
   + BUS_WAIT -> S3  ;stay if not
                     ;ready
S4 := VCC -> S5 ;jump to S5 - fixed
                ;wait
S5 := VCC -> S6 ;jump to S6 - fixed
                ;wait
S6 := BUS_CONT -> S7 ;jump to S7 if
                     ;interrupt done
   + BUS_WAIT -> S6 ;stay if not
                    ;done

;cleanup and idle cycles
S7 := VCC -> S8 ;cleanup, then
                ;jump to S8
S8 := VCC -> S0 ;jump to S0,
                ;ready to start

;transition outputs
S0.OUTF := LOCAL*/MEMORY*/INTACK
S1.OUTF := /LOCAL*MEMORY*/INTACK
S2.OUTF := /LOCAL*MEMORY*/INTACK
S3.OUTF := /LOCAL*/MEMORY*INTACK
S4.OUTF := /LOCAL*/MEMORY*/INTACK
S5.OUTF := /LOCAL*/MEMORY*/INTACK
S6.OUTF := /LOCAL*/MEMORY*INTACK
S7.OUTF := /LOCAL*/MEMORY*/INTACK
S8.OUTF := LOCAL*/MEMORY*/INTACK
```

```
CONDITIONS
MEM_REQ =  M_IO * /INT_CYC * /A20
INT_REQ = /M_IO *  INT_CYC * /A20
BUS_CONT = /READY
BUS_WAIT =  READY
```

The PVS specification of the state machine (theory bus_controller) starts with the assumption that *MEM_REQ* and *INT_REQ* cannot both be true at the same time:

```
ASSUMING
bv_lib : library = "bitvectors"
importing time, signal, bv_lib@bv,
          registers

t : var time
MEM_REQ (M_IO, INT_CYC, A20 : bit)
  : bit =
    M_IO AND NOT(INT_CYC) AND
    NOT(A20);

INT_REQ (M_IO, INT_CYC, A20 : bit)
  : bit  =
    NOT(M_IO) AND INT_CYC AND
    NOT(A20);

MEM_INT_REQ_lemma : assumption
  INT_REQ( M_IO(t), INT_CYC(t),
           A20(t)) /= 1 OR
  MEM_REQ( M_IO(t), INT_CYC(t),
      -    A20(t)) /= 1

ENDASSUMING
```

If this were not the case, then the state transition from *S0* is not well defined and the following PVS specification would not be accepted.

The following types are then defined:

```
states : type = {S0, S1, S2, S3,
  S4, S5, S6, S7, S8}
input_conditions : type =
  {ic : signal[[below(3) -> bit] |
  forall (t : time) :
   (ic(t)(0) = MEM_REQ( M_IO(t),
     INT_CYC(t), A20(t))) &
   (ic(t)(1) = INT_REQ( M_IO(t),
     INT_CYC(t), A20(t))) &
   (ic(t)(2) = not(READY(t)))
```

The type *states* defines all of the states in the system. As the example does not worry about invalid states for the state machine, neither does the PVS specification. Thus the set of valid states equals *states*. The type *inputs_conditions* encapsulates the three conditions *MEM_REQ*, *INT_REQ*, and *BUS_CONT* from the PLDasm definition (*BUS_WAIT* is just the inverse of *BUS_CONT*).

Then the variables

```
LOCAL, MEMORY, INTACK :
  var signal[bit]
```

define the system outputs and the variable

```
input_cond : var input_conditions
```

defines the set of input conditions or state qualifiers. The state *S* is a variable of type *states*. Then the *initial_state* is defined as *S0* and the reset input and reset state are defined as:

```
reset : var signal[bit]
reset_state : states =
  initial_state
```

From these, the state machine can now be defined.

The state transition function is then defined for state *S* and input conditions *input_cond* as:

```
transition (S,input_cond): states =
  let MEM_REQ = input_cond(0) = 1,
      INT_REQ = input_cond(1) = 1,
      BUS_CONT = input_cond(2) = 1
  in
cases s of
  S0: if MEM_REQ then S1
      elsif INT_REQ then S3
      else S0 endif,
  S1: S2,
  S2: if BUS_CONT then S7
                  else S2 endif,
  S3: if BUS_CONT then S4
                  else S3 endif,
  S4: S5,
  S5: S6,
  S6: if BUS_CONT then S7
                  else S6 endif,
  S7: S8,
  S8: S0
endcases
```

This is practically a direct translation of the PLDasm specification. The state transitions are then managed by the axiom

```
state_transition_axiom : axiom
  state(t+1) =
    if (reset(t) = 1) then
      initial_state
    else transition( state(t),
      input_cond(t)) endif
```

Note that the reset condition could have been encapsulated in the state transition function, but as digital designers rarely include it since it simply complicates the function and is generally managed separately, it is kept separate in this example.

Now the state machine is ready for verification with relationship to the theory *State_Verification_1*. This is done by first defining a predicate *valid_state?* which accepts a state and always returns true since the complete state space is encapsulated by *states*. Then *State_Verification_1* is imported by

```
importing
  State_Verification_1[ states,
    valid_state?, bit, 3,
    input_condition, transition,
    reset_state, 1]
```

This results in several TCCs, with the ones of interest being:

```
IMPORTING2_TCC1: OBLIGATION
(FORALL
  (x1: [states, input_condition],
   y1: [(valid_state?),
         {ic: [below(3) -> bit] |
           ic(0) /= 1 OR
           ic(1) /= 1}]):
(TRUE AND
 valid_state?(PROJ_1(x1))) AND
 valid_state?(transition(x1)));

IMPORTING2_TCC2: OBLIGATION
  valid_state?(reset_state);

IMPORTING2_TCC4: OBLIGATION
(FORALL
  (reset_signal: signal[bit],
   states: signal[states]):
EXISTS (ti: time):
  reset_signal(ti) = 1
  => FORALL (S: states):
    EXISTS (input_conditions, t):
      states(t) = S);
```

The first two obligations simply assert that the reset and transition functions must result in a valid state. Each of these is straightforward to prove. The last TCC is the primary concern. It stipulates that all states in *S* must be reachable. This requires a somewhat tedious proof which is discharged by use of a case analysis to handle the various states.

Given the state transition definition, the implementation is then defined. First the concept of state variables is introduced as a 4-bit vector to represent the four state variables.

```
state_variables : type = bvec[4]
```

Then two mapping functions are defined to map between states and state variables. The first, `state_var_of`, maps from a state to a set of state variables.

```
state_var_of (S : states) :
  state_variables =
Cases S of
  S0 : lambda (i : below(4)) : 0,
  S1 : lambda (i : below(4)) :
    if i = 0 then 1 else 0 endif,
  S2 : lambda (i : below(4)) :
    if i = 1 then 1 else 0 endif,
  S3 : lambda (i : below(4)) :
    if (i = 0) or (i = 1) then 1
      else 0 endif,
  S4 : lambda (i : below(4)) :
    if i = 2 then 1 else 0 endif,
  S5 : lambda (i : below(4)) :
    if (i = 0) or (i = 2) then 1
      else 0 endif,
  S6 : lambda (i : below(4)) :
    if (i = 1) or (i = 2) then 1
      else 0 endif,
  S7 : lambda (i : below(4)) :
    if i = 3 then 0 else 1 endif,
  S8 : lambda (i : below(4)) :
    if i = 3 then 1 else 0 endif
EndCases
```

Then the set of valid state variables, `valid_state_var?`,

```
valid_state_var?
  (CS : state_variables) : bool =
    exists (S : states) : CS =
      state_var_of( S)
```

is mapped to *states* by

```
state_of (CS:(valid_state_var?)) :
  states =
  if    CS = 0 then S0
  elsif CS = 1 then S1
  elsif CS = 2 then S2
  elsif CS = 3 then S3
  elsif CS = 4 then S4
  elsif CS = 5 then S5
  elsif CS = 6 then S6
  elsif CS = 7 then S7
  else S8 endif
```

The syntax for `state_of` has been modified to be more readable.

Given the mappings, the state equations are defined by:

```
state_equations
  (CS : (valid_state_var?),
   ic : input_condition) :
  (valid_state_var?) =
lambda (i : below(4)) :
  let S8 = S8?(state_of(CS)),
      S7 = S7?(state_of(CS)),
      S6 = S6?(state_of(CS)),
      S5 = S5?(state_of(CS)),
      S4 = S4?(state_of(CS)),
      S3 = S3?(state_of(CS)),
      S2 = S2?(state_of(CS)),
      S1 = S1?(state_of(CS)),
      S0 = S0?(state_of(CS)),
      MEM_REQ = ic(0) = 1,
      INT_REQ = ic(1) = 1,
      BUS_CONT = ic(2) = 1 in
  if i = 3 then
     if S7 then 1 else 0 endif
  elsif i = 2 then
    if ((S3 or S2 or S6) &
        BUS_CONT) or S4 or S5 or
        (S6 & not BUS_CONT) then 1
    else 0 endif
  elsif i = 1 then
    if (S3 & not BUS_CONT) or
       (S0 & INT_REQ) or  S1 or S2
       or S5 or S6 then 1
    else 0 endif
  else
    if (S0 & (MEM_REQ or INT_REQ))
       or (S2 & BUS_CONT) or
       (S3 & not BUS_CONT) or S4
       or (S6 & BUS_CONT) then 1
    else 0 endif endif
```

The set of state equations must perform the same function as `transition`. This is verified by importing `State_Implementation_1` with the parameters:

```
importing
  State_Implementation_1[
    states, valid_state?, bit, 3,
    input_condition, transition,
    reset_state, 1,
    (valid_state_var?),
    state_var_of, state_equations]
```

This results in the set of TCCs:

```
IMPORTING3_TCC1: OBLIGATION
(FORALL (x1: states, y1: states):
  (valid_state_var?)
  (state_var_of(x1)));

IMPORTING3_TCC2: OBLIGATION
(FORALL (S: states, T: states):
  T /= S <=> state_var_of(T) /=
             state_var_of(S));

IMPORTING3_TCC3: OBLIGATION
(FORALL (I: input_condition,
         V: (valid_state?)):
state_equations(state_var_of(V),I)=
state_var_of(transition(V, I)));
```

The first TCC ensures that the mapping function `state_var_of` results in a valid state variable. The second TCC requires the proof that the mapping of states to state variables is unique. The last TCC states that the state equations must implement the state transition function. Each of these is a straightforward proof.

6 Future Work

Future work is focused in three areas: advanced clocking structures, support for VHDL, and application of the theory to meaningful problems. The advanced clocking structures is necessary to permit gated clocks and multiphase clocks (this work is currently funded). The support for VHDL will be in the form of libraries to support IEEE standard libraries. Finally, the system will be tested against a real design. Current plans are to verify the design of a divider, thus testing both the combinational and sequential components.

7 Conclusions

The bus controller example demonstrates the ease with which a system specified for a PLD in an arbitrary hardware description language can be easily defined in PVS. While the transation process may introduce errors, the purpose of this work is to verify properties of the design, not of the actual implementation. Then the supporting environment assists the designer in verifying certain properties of the design.

What was not demonstrated in this paper (but has been done) is that the behavior of the system can be described as a set of conjectures which are then verified. This is done at the state level and not the implementation level, but then as the implementation is verified to meet the state machine definition, this is sufficient. This process revealed an error in the original description of the bus controller example [5] where the memory cycles were not properly managed. The example presented in this paper has been corrected based on those results.

As this process is applied to larger problems and the kinks are worked out, it is believed that this process can become a valuable tool to the developers of digital circuits, not just PLDs, but all digital circuits (a benefit of the fact that there is no dependence on one hardware description language). In fact, it is envisioned that this process could be incorporated into a computer engineering curriculum as a means to teach students how to specify problems, even if the verification process is not performed, or can be fully automated through use of strategies.

8 References

[1] Dodge, D., P. Undrill, and P. Ross, Application of Z in digital hardware design, *IEE Proceedings on Computers and Digital Technology*, 143(1): 79-87, January 1996.

[2] Reetz, R., and Kropf, T., A Flowgraph Semantics of VHDL: Toward a VHDL Verification Workbench in HOL, *Formal methods in system design*, 7(1/2) : 73-86, August 1995.

[3] Owre, S., J. Rushby, N. Shankar, and F. von Henke, "Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS," *IEEE Transactions on Software Engineering*, 21(2):107-125, February 1995.

[4] Miner, P. and R. Butler, "A Bitvector Library for PVS 2," Assessment Technology Branch, NASA Langley Research Center, Hampton, Va., January 1995.

[5] *PLDshell Plus/PLDasm User's guide V4.0*, Intel Corporations, 1994.

[6] Chin, S.-K., "White Paper on High Confidence Technology," Forum on Federal Information and Communications R & D, July 6-7, 1995, Bethesda, MD, URL: http://www.cat.syr.edu/chin/papers/cic.ps.

[7] Owre, S., J. Rushby, N. Shankar, and M. Srivas, "A Tutorial on Using PVS for Hardware Verification," *2nd International Conference on Theorem Provers in Circuit Design*, Sept. 1994, Springer Verlag LNCS 901, pp. 258-279.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 1997	**3. REPORT TYPE AND DATES COVERED** Conference Publication

4. TITLE AND SUBTITLE Lfm97 Fourth NASA Langley Formal Methods Workshop	**5. FUNDING NUMBERS** WU 522-33-11-03
6. AUTHOR(S) C. Michael Holloway and Kelly J. Hayhurst, Compilers	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) NASA Langley Research Center Hampton, VA 23681-2199	**8. PERFORMING ORGANIZATION REPORT NUMBER** L-17649
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER** NASA CP-3356

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified–Unlimited Subject Category 59 Availability: NASA CASI (301) 621-0390	**12b. DISTRIBUTION CODE**

13. ABSTRACT *(Maximum 200 words)*

This publication consists of papers presented at NASA Langley Research Center's fourth workshop on the application of formal methods to the design and verification of life-critical systems. Much of this material and material not available herein is available on the World-Wide Web via the following URL: http://atb-www.larc.nasa.gov/Lfm97/

14. SUBJECT TERMS Formal methods; Requirements; Mathematical modeling; Safety; Mathematical logic	**15. NUMBER OF PAGES** 214
	16. PRICE CODE A10

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT**