NASA Contractor Report 166008

Investigation, Development, and Evaluation of Performance Proving for Fault-Tolerant Computers

Karl N. Levitt
P. Michael Melliar-Smith
Richard Schwartz
Robert E. Shostak
Dwight Hare
Robert Boyer
J S Moore
Milton Green
W. David Elliott

SRI International
Menlo Park, California 94025

Contract NAS1-15528

August 1983

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

NASA Contractor Report 166008

Investigation, Development, and Evaluation of Performance Proving for Fault-Tolerant Computers

Karl N. Levitt
Richard Schwartz
Dwight Hare
J S Moore
P. Michael Melliar-Smith
Robert E. Shostak
Robert Boyer
Milton Green
W. David Elliott

SRI International
Menlo Park, California 94025

Contract NAS1-15528

August 1983

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

N84-23507#

TABLE OF CONTENTS

Chapter Page

CHAPTER 1

INTRODUCTION

INTRODUCTION

1. Project Objectives

This report is concerned with the Formal Verification of computer systems. In the course of carrying out the work reported herein we have developed a number of methodologies for verifying systems, developed computer-based tools that assist users in verifying their systems, and have applied these tools to verifying in part the SIFT ultrareliable aircraft computer.

By *formal verification* we mean showing by mathematical reasoning that a system satisfies its requirement. By a *system* we mean the computer hardware and the collection of programs that run on the hardware. A *requirement* is a description of the function to be carried out by the system. The requirement indicates the system's response to all sequences of inputs that could be applied to the system. If the verification is successfully carried out, the system is, in principle, guaranteed to be *correct*; no further validation (e.g., testing) should be required.* However, it should be noted that the system might still contain some errors that require conventional testing to uncover e.g. due to: errors in the requirement, ommisions in the requirement, errors in portions of the system that are not verified. Thus one should view formal verification as a systematic approach to analyzing a system that when combined with standard methods, is potentially capable of reducing significantly the number of errors in delivered systems.

How can it be assured that requirements are free of errors? The most obvious answer is to produce requirements that are short enough and simple enough to be carefully reviewed. We have found that even for very complex systems, requirements can be written that indicate only what is essential to understand *what* the system is supposed to be doing. Details of the system's implementation need not be part of the requirement. It is our conclusion, then, that short requirements statements can be produced. In order to be processible by the verification tools, the requirement must be expressed in a formal language. We will be using three different languages for stating requirements (STP, Boyer-Moore theory and SPECIAL). Those familiar with mathematical logic will have no trouble reading and understanding the requirements we present in this report. Those who have not been exposed to mathematical logic will be able to understand the requirements through the English comments that accompany the formal logic statements.

We wish to contrast our approach to system verification to the traditional approach, in particular stressing why our approach leads to simpler and more believable requirements, and why it makes the process of verifying large systems feasible. The *traditional* approach, which we call *code verification,* is concerned with verifying algorithms expressed in a programming language. For example a *sort* algorithm is shown to satisfy the requirement that the output is ordered and a permutation of the input. (The requirement for such algorithms is commonly called the *specification.*)

Code verification in itself is inadequate since it does not lead to requirements that are short and

*To understand the implications of verification, the reader might find it helpful to make analogy to what a proof in, say, plane geometry means. Reasoning similar to what we describe in this report is used to prove the Pythagorean Theorem. Once verified, a theorem can be freely used in any circumstance where it applies -- the assumptions underlying the theorem (there is a right triangle with sides a, b and hypotenuse c.) are satisfied.

simple. For example, an operating system consists of many subprograms. Each of these subprograms can be given a specification: The specification for the *scheduler* will indicate what it means to schedule; the specification for the *directory* manager will indicate what it means for directory entries to handled properly. However, these specifications in toto fail to clearly indicate the overall purpose of the operating system which, in the case of SIFT, is to assure that aircraft control programs are processed correctly. It is the interaction of these programs that determines if the overall goals of the system are to be satisfied.

What is needed, then, is a requirement statement that addresses higher-level issues. To achieve this, we suggest that a system requirement be expressed as a *model*, which we take to mean a collection of higher-level functions together with properties (expressed as axioms on the functions). This model must be shown to be consistent with the specifications for the subprograms that comprise the system; we call this process *design verification*. (Code verification is then employed to show the consistency of the subprograms' specifications with their implementation.)

Often, as was the case in SIFT, the jump from the model to the specifications is too large to be carried out in one step. Hence we introduce additional models, the collection of models forming a hierarchy. Design verification, then, consists of proving that each model is consistent with the one directly below it in the hierarchy. It should be noted that in a well-conceived hierarchy, each model will introduce a particular element of design; the SIFT hierarchy nicely illustrates this concept.

In support of the steps of design verification and code verification, we have developed a collection of interactive tools. The heart of the design verification tools is the STP theorem prover, the language which it supports being used to define the models. Besides the theorem prover, the tool set contains various support packages, including ones that manage the overall verification process and that output a final proof in a form that is reasonably readable.

The code verification system supports the verification of Pascal programs whose specifications are expressed in the SPECIAL specification language. The methodology underlying the tools, (called the Hierarchical Development Methodology [HDM]), allows the code itself to be decomposed in a hierarchical layering of levels. This process greatly simplifies the verification of large programs. Another significant feature of the code verification system is that it can be easily tailored to handle any particular programming language. This independence of programming language is achieved through a tool called the *meta- verification condition generator*, which accepts the syntax and (axiomatic) semantics of a programming to produce a code verification system unique to that language.

As indicated above, the scope of this project is aircraft electronics systems. Our primary accomplishment was the development of the verification systems (for design and code verification) and the application of the design verification system to SIFT. It should be noted that design and code verification, as carried out in this project, do not cover all parts of an aircraft electronics system. Missing from our verification (besides the parts of SIFT we did not verify -- see below) are the following: Assembly-level programs, Hardware logic, and Application programs

Consideration of these areas begins to complete a full hierarchy for an aircraft electronics system, the components of which are:

- Application programs -- in particular, flight control programs

- Design for a fault-tolerant aircraft computer -- SIFT

- Higher order code (Pascal) for SIFT software

- Assembly level code, which in the case of SIFT is in the Bendix BDX930 instruction set

- Hardware logic -- implementation of the BDX930 instruction set

In each of these areas, we developed techniques that give some promise of being suitable for verifying real systems.

2. Significant Accomplishments of Project

The significant accomplishments of the project are the following:

1. The development of experimental verification tools (for design and code verification). It should be noted that versions of these tools have been used to verify security-related properties of operating systems; this work was carried out on other SRI projects.

2. The application of the design verification tools tools to the verification (in part) of the SIFT operating system. As we indicate later, SIFT achieves reliability by having tasks execute on 3 or more processors, the results being voted on after completion of each task. When an error can be pinpointed to a processor, it is logically removed and replaced by another processor. The requirement for SIFT (informally stated) is that the probability of producing an incorrect result shall be less than 10^-10 per hour over a 10 hour mission. The major property addressed by the design verification exercise and expressed in a model is that all aircraft *tasks* managed by the SIFT system will yield *correct* results within their prescribed *deadlines*, as long as the system is in a *safe state*. Here *tasks* are programs that implement the various aircraft functions (flight control, navigation, etc.); *correct* means that the tasks will always get the right inputs and deliver the output as would be produced by a working processor; within the *deadline* means that the result is produced according to some preassigned schedule. *Safe state*, is not given a definition at the highest model; when defined in a lower model, it means that the number of good processors exceeds the number of failed processors in a configuration -- voting works. We believe that this model expresses exactly the significant functional properties of SIFT. Moreover, it should be noted that the design verification turned up a significant design error, that previously escaped our attention. Our verification of the SIFT design is currently incomplete, failing to prove the following

   - **Reconfiguration**: When a processor is found to be faulty, the reconfiguration design will logically remove from the configuration of processors. Current work is considering this verification.

   - **Quantitative reliability**, the failure probability for SIFT is 10^-10/hour for a 10 hour mission. Our design hierarchy does include a model, called the ***reliability*** *model* that expresses, in terms of a Markov model, the concept of system failure. However, we did not formally relate the reliability model to the other models; this connection might be carried out in current work. This connection by itself, however, will not lead immediately to a verification of quantitative reliability since the rates of processor failure and reconfiguration must be derived; these rates can only be derived from significant experimentation with SIFT, as currently being carried out by NASA-Langley.

3. The application of the code verification tools to the verification (in part) to *a* Pascal implementation of SIFT. As in the design verification exercise, our concern was just with the safety-related properties; we did not carry out the verification of the SIFT code concerned with *reconfiguration*. Among other current deficiencies of the code verification effort are:

    - Although the code we did verify would successfully run SIFT, it is not the code that in the SIFT system delivered to NASA-Langley. The delivered code is a combination of Pascal and BDX930 assembly code. Moreover, the Pascal portion is written in a version of Pascal tailored to an efficient real-time processing; for example, it permits the specification of absolute addresses, and accomplishes the transfer of data among processors by a special assignment statement. Our code verification system, support standard Pascal, does not handle such features.

    - The program that assures the clocks of the SIFT processors are in synchronization was not verified. The specifications for the clock synchronization program were, however, used in the design proof.

    - The program that handles *interactive consistency*, i.e., the transfer of single source data among processors.

4. An initial approach to verifying assembly language programs. This approach, if mechanized, would be used to verify that portion of a system not expressible in a high-level language. We attempted, but did not complete, the verification of a scheduler for a real time system. A byproduct of this effort was a formal definition of the BDX930 instruction set in the Boyer-Moore theory.

5. An approach to verifying the precision of numerical algorithms, e.g., navigation programs. This approach is suitable only for programs where the *correctness* property (ignoring precision, the algorithm computes a certain function), and the precision property (the error introduced through round-off and other error-introducing operations is bounded by a specified value) can be handled separately.

6. An approach to verifying control applications. Again, separating design verification from code verification proved to be extremely useful. The code verification exercise proves that the program correctly implements a particular filter function. The design verification shows that a configuration of filters achieves a particular control law. We used the Boyer-Moore theory to express the control law for a simple application -- the control of a vehicle subject to bounded disturbances in one dimension.

7. An approach to verifying hardware logic that demonstrates the consistency between a hardware logic circuit and a specification for the functional behavior of that circuit. Using our method, we successfully verified a hardware frequency comparator. A more ambitious undertaking, not yet attempted, would be to verify the implementation of the instruction set for a computer.

3. Organization of Report

This report, having been compiled largely from existing reports, papers, listings, etc., is large and not well structured. The following is a brief guide to the report.

The report is organized into the following areas:

1. STP Theorem Prover -- Chapters 2-3. Chapter 2 provides an easy to follow introduction to the STP logic. Chapter 3 describes the command interface of the STP theorem prover for those who want to consider using it.

2. Design Verification of SIFT -- Chapters 4-7. Chapter 4 presents an easy introduction to the technique of design verification, presents an informal verification of a very simple voting system, and presents a synopsis of the design verification of SIFT. Chapter 5 gives the listing of the mechanization (using STP) of the verification of the simple voting system. Chapter 6 presents more detail on the SIFT design verification, in particular describing the hierarchy of models. Chapter 7 presents the complete listing of the design verification of SIFT.

3. High Level Language Code Verification -- Chapters 8-15. Chapter 8 introduces, in a tutorial manner, the subject of code verification in general. Our approach to verifying a hierarchy of modules -- the Hierarchical Development Methodology (HDM) -- is introduced in Chapter 9. The discussion here is centered around the language (SPECIAL) used to specify these modules and an *ideal* language (called ILPL) used for module implementation. In later chapters we show how an existing language (Pascal) can be used for module implementation. The Meta-Verification Condition Generator, the approach to producing a language-independent code verification system is presented in Chapter 10. Included at the end of this chapter are rules for the context-independent semantics of a Pascal subset. Chapters 11-13 introduce the code verification system we developed to prove hierarchies of Pascal programs, each of which is specified in SPECIAL. Chapter 14 presents the portion of the SIFT code we verified and Chapter 15 gives sample listings of the proofs. Finally, Chapter 15 presents a formal definition of a subset of the HDM proof methodology; the Boyer-Moore logic is used to express the formal definition. Our motivation in producing such a formal definition is to reduce the chances for errors being introduced in the verification process itself; a careful reading of such a formal definition should uncover errors in the verification technique.

4. Assembly language-level Verification -- Chapters 17-18. Chapter 17 presents our (incomplete) attempt to verify a scheduler written in assembly language. The definition of the assembly language (BDX930) that would have been used is presented in Chapter 18.

5. Numerical Algorithm Verification -- Chapter 19

6. Verification of Flight Control Programs -- Chapter 20

7. Verification of Hardware Logic -- Chapter 21

8. Tutorial on the Boyer-Moore Logic and its application to Verification of Fortran Programs -- Chapter 22

9. Conclusions -- Chapter 23

4. Recommended Chapters

We expect that most readers will be interested in the verification of SIFT and the tools that made it possible. To this end, the chapters to read are: 2, 5, and 7; chapter 8, containing the complete listing of the verification, is recommended for those few who will want to carefully scrutinize the detailed steps in the verification.

5. Acknowledgments

Most of the authors have had a long-term involvement in the SIFT design, development and verification -- and in SRI's verification projects. Others who have also been involved are John Wensley (the SIFT concept and its detailed design), Jack Goldberg (leader of the final 2 years of the SIFT implementation project) and Chuck Weinstock (chief implementer of SIFT). Larry Robinson was the principal developer of the HDM proof methodology.

We are grateful to a number of individuals of the NASA-Langley Research Center for their assistance on many aspects of this project. Nick Murray was the initial monitor of the work. He provided constant encouragement, technical feedback on the aspects of this work related to SIFT, and helped guide our initial attempts to verify flight control programs. Rick Butler, the current monitor, has carefully read this report, and has provided us with valuable criticism and suggestions on a wide range of technical areas related to the project. In addition, he did much of the work on the mechanized verification of the simple voting system (Chapters 5 and 6). Rick has also helped out considerably in the creation of a plan for producing a demonstrable verification system that could be used for a wide variety of applications.

Billy Dove (at the start of this effort) and, currently, Milt Holt are responsible for NASA-Langley's program in reliable aircraft computers. We also wish to thank the various NASA-Langley professionals who participated in the oral reviews we had them suffer through. These individuals (Brian Lupton, Earl Migneault, Larry Spencer, Sal Bavuso, Dan Palumbo -- among others), although often skeptical of the verification approach, provided us with gentle and helpful criticism.

Earl Boebert (Honeywell -- Systems and Research Division) suggested the problem we used in Chapter 20 to illustrate our technique for flight control verification.

We are also grateful to the DoD Computer Security Center for their current support of SRI's verification effort and to particular individuals (Ann Marmor-Squires, Bret Hartman, Marv Schaefer, and Roger Schell) for their ongoing technical advice.

CHAPTER 2

STP THEOREM PROVER – OVERVIEW

STP: A Mechanized Logic for Specification and Verification

R. E. Shostak, Richard Schwartz, and P.M. Melliar-Smith

1 Introduction

This paper describes a logic and proof theory that has been mechanized and successfully applied to prove nontrivial properties of a fully distributed fault-tolerant system. We believe the system is close to achieving the critical balance in man-machine interaction necessary for successful application by users other than the system developers.

STP is an implemented system supporting specification and verification of theories expressed in an extension of multisorted first-order logic. The logic includes type parameterization and type hierarchies. STP support includes syntactic checking and proof components as part of an interactive environment for developing and managing theories in the logic. In formulating each new theory, the user begins with a certain *core theory* that comprises a set of primitive types and function symbols, and extends this theory by introducing new types and symbols, together with axioms that capture the intended semantics of the new concepts. The mechanical proof component of the system is predicated on a fast, complete decision procedure for a certain syntactically characterizable subtheory. By providing aid to this component in the form of the selection of appropriate instances of axioms and lemmas, the user raises the level of competence of the prover to encompass the extended theory in its entirety. As a result of a successful proof attempt using STP, one obtains the sequence of intermediate lemmas, together with the axioms, auxiliary lemmas, and their necessary instantiations, which lead to the theorem. The system automatically keeps track of which formulas have been proved and which have not, so that the user is not forced to prove lemmas in advance of their application. The system also monitors the incremental introduction and modification of specifications to maintain soundness.

This paper is organized as follows: Section 2 discusses motivation for the form of man-machine interaction embodied by STP. Section 3 contains a formal description of the logic and the proof theory, and illustrates the description with an example. Section 4 discusses the use of STP in a large-scale effort to prove nontrivial properties of SIFT, a distributed Software-Implemented Fault-Tolerant operating system for aircraft flight control. Finally, Section 5 describes directions for further work.

2 Issues in Mechanized Verification Support

STP's design was guided to a considerable extent by our experience in attempting to formulate and reason about properties of SIFT. The following concerns were strongly influential.

This research was supported in part by NASA contract NAS1-15528, NSF grant MCS-7904081, and by AFOSR contract F49620-79-C-0099.

2.1 Property-Theoretic Specification

It is often desirable to specify program or system characteristics abstractly by stating properties possessed without defining the method of attainment. High-level system specifications represent, in effect, system *requirements*, rather than a prescription of implementation characteristics. That the specification method allow such partial specification of system properties is important; without this capability, one is forced to *overspecify* system descriptions. As a consequence, one both overconstrains possible implementation solutions and introduces spurious detail into the design specification.

2.2 Credible Specifications and Proofs

The intent of formal specification and verification is to increase one's confidence that a system will function correctly. As such, the specification and proof of system conformance must be *believable*. To produce credible specifications, the specification language must be sufficiently close to the user's conceptual understanding of the task the system is to perform. Specifications that are as long as or longer than the actual code for the system are likely to be harder to understand than the code itself.

Credibility of a verification effort requires that the end product be a proof that can be independently scrutinized and subjected to the social process. It must be possible to separate the process by which the proof was obtained from the proof itself.

2.3 Form of Verification Support

Mechanical theorem provers can be characterized by the style and level of user direction necessary to complete the proof. The spectrum of possibilities ranges from completely automatic "out to lunch" verification, where no user interaction is necessary to direct the proof to completion, to a proof checker (e.g., the FOL [Wey 80] system) where all steps are provided by the user. Between these extremes are interactive semi-automatic systems (such as LCF [Mil 79]) in which proofs are generated by a symbiosis of mechanically derived and user-provided steps. This is necessary because, in practice, theories that are sufficiently rich to be useful are usually either undecidable or have combinatorics that preclude practical decision procedures. Research in theorem proving has thus focused on methods by which the user can direct machine inference.

Mechanical deduction in most systems has taken the form of heuristic algorithms for searching large state spaces to determine the sequence of intermediate steps necessary to form a proof. Because of the difficulty in determining the ultimate success or failure of a heuristically chosen proof strategy without exhaustive search, the user is charged with the responsibility for monitoring the proof attempt and aborting an unpromising path. Where the user can determine that an inappropriate proof strategy has been chosen, he then introduces additional lemmas in an attempt to induce the system to follow a more fruitful path.

A drawback to heuristic theorem-proving attempts is that successful proof depends upon intimate knowledge of the heuristics employed. One must understand how very subtle changes in specification structure, even those that preserve semantic equivalence, can affect the direction and final outcome of the proof attempt. Lemma form becomes as important as content. In many cases the user may be aware of the proof steps necessary to justify the lemma within the supported theory, but he may be unable to suggest the lemmas in the form appropriate to guide the verification system down the proper path. This difficulty may be attributed to the inability to provide a succinct, yet complete, characterization of the heuristics employed by the theorem prover. Without this characterization, effective use of the system will depend not only on the understanding of the underlying theory, but also upon intimate knowledge of the theorem prover implementation.

Our experience has led us to believe that effective symbiosis between man and machine depends upon

a. The *predictability* of machine-supplied deduction

b. the user's ability *directly* to provide proof strategies and steps beyond the automatic deductive capability

c. machine interaction with the user in the style and level of conceptualization natural to him

d. the machine's ability to provide *responsive* deductive support to maintain continuity of user interaction.

Our proof experience indicates that the predictability of machine aid is far more important than the occasional burst of insight. Successful interaction with a theorem prover depends upon the user's having a clear picture of how the formula is deducible within the theory and when user assistance is necessary. It seems unlikely in systems supporting extensive, but incomplete, deduction that the user would succeed without this insight.

In a system involving extensive user interaction, one should not underestimate the importance of the user interface. It is crucial that all interaction be presented at the level of user input and in a natural and succinct notation. Management of information becomes a major problem during proof construction. That the user retain a clear intuitive understanding of the specifications is paramount. Techniques for aggregation of information, such as theory parameterizaton, as suggested by Goguen and Burstall [BuG 77], are extremely important. Data base management aids for organizing and retrieving theories are critical.

For the man-machine relationship to be symbiotic, machine response must keep pace with the user. The size of conceptual steps comprehensible to the user must be well matched to the computational efficiency of the theorem prover. Our experience indicates that a delay of more than on the order of one minute in machine response tends to cause loss of concentration.

2.4 Related Work

Much progress has been made in the last ten years on techniques for formal specification and verification. Early contributions to formal specification include Milner's work [Mil 67] on weak simulation and Parnas' [Par 72] on hierarchical methodology. The concepts introduced in this early work were further developed and incorporated in the HDM methodology [RoL 77,LRS 79]. The more recent research of Goguen and Burstall on Clear [BuG 77] and of Nakajima on Iota [Nak 77] introduced the notion of higher-order theories and theory parameterization.

At the same time, a great deal of research has focused on systems for mechanical verification. The earliest such systems depended strongly on heuristic-based, theorem proving strategies. The systems of King [Kin 69] and Levitt and Waldinger [LeW 75] are among these. The Boyer-Moore theorem prover [BoM 79] is one of the most striking examples of the power possible using heuristic techniques. The deductive component of STP, however, is more akin to the theorem provers of Bledsoe [Ble 74], Nelson and Oppen [OpN 78], Shostak [Sho 77], and Suzuki [Suz 75], all of which are founded on the use of decision procedures. The GYPSY system [Goo 79], the Jovial Verification System [Els 79], the Stanford Verifier [Luc 79], and the SDC system [Sch 80] are recent examples of program verification systems.

By and large, specification research has been pursued independently of work on verification. Only in the last few years has emphasis been placed on the interaction between the specification medium and the verification component. The Affirm system [Mu 80], for example, utilizes a term rewriting system, both as an algebraic specification medium and as a vehicle for mechanical proof. The system described in the current paper continues the emphasis on maintaining a close balance between the level of conceptualization supported in the specification and the level at which machine-aided deduction occurs.

3 The Logic of STP

Before presenting a formal description of the logic supported by the system, we present a simple example to give an intuitive feeling for the specification style.

We define a parameterized theory of Pairs of objects of two arbitrary type domains. We then use this theory to derive a theory of integer Intervals, represented by pairs of beginning and end points.

Figure 1 shows the specification of these theories in STP. The user declares the parameterized type PAIR.OF(T1 T2) in line 3, having previously declared type variables T1 and T2 in lines 1 and 2. The accessor operation FIRST is defined by the DS (Declare function Symbol) command in line 4 to take a value of type PAIR.OF(T1 T2) and return a value of type T1. The SECOND component accessor is analogously defined in line 5. A pair constructor MAKE.PAIR(T1 T2) is declared in line 6. Variables X and Y are declared to be of schematic types T1 and T2 (respectively) in lines 7 and 8. These declarations introduce new function and variable symbols, but attach no semantics. Lines 9 and 11 introduce two axiom schemes to define the properties of Pairs. Axiom A1 defines the accessor functions FIRST and SECOND to retrieve the first and second components (respectively) of a pair constructed by MAKE.PAIR. Axiom A2 extends the equality operation by defining two Pairs to be equal exactly when the corresponding components are equal. Equality is predefined over all domains.

INTERVAL is introduced as a subtype of PAIR.OF(INTEGER INTEGER) in line 13. Note that type variables T1 and T2 are thus both instantiated as ground type INTEGER. The subtype declaration declares Intervals to be an extension of the theory of Pairs of Integers. The type theory allows implicit type coercion from a subtype to a supertype (but not vice versa). Thus, all axioms defining Pairs of Integers are applicable to Intervals - in this case, instances of axiom schemes A1 and A2.

Lines 15 and 16 introduce derived Interval operations BEGINNING and END, defined as the selection of the first and second Pair values (respectively). The DD (Declare Definition) construct can be viewed as a means of conservative extension. Semantically, line 15 is equivalent to introducing the axiom (EQUAL (BEGINNING II) (FIRST II)). Operationally, the DD defining BEGINNING is automatically instantiated and applied as an axiom. Similarly, a MAKE.INTERVAL constructor is derived in line 20 in terms of the MAKE.PAIR operation of the supertype.

After introducing the signature for an Interval MEMBER operation in line 21, axiom A3 in line 22 begins to introduce Interval semantics. An Integer I is defined to be a Member of Interval II exactly when it lies between the beginning and ending points of the Interval. This completes our abbreviated definition of Integer Intervals.

```
1.   (DTV T1)
2.   (DTV T2)

     (QUOTE "The following is a partial theory of Pairs")
3.   (DT PAIR.OF (T1 T2))
4.   (DS T1 FIRST ((PAIR.OF T1 T2)))
5.   (DS T2 SECOND ((PAIR.OF T1 T2)))
6.   (DS (PAIR.OF T1 T2)  MAKE.PAIR(T1 T2))
7.   (DSV T1 X)
8.   (DSV T2 Y)
9.   (DA A1 (AND (EQUAL  X  (FIRST (MAKE.PAIR X Y)))
                 (EQUAL  Y  (SECOND (MAKE.PAIR X Y)))))
10.  (DSV (PAIR.OF T1 T2)  P)
11.  (DSV (PAIR.OF T1 T2)  P1)
12.  (DA A2 (IFF (EQUAL P P1)
                (AND (EQUAL  (FIRST P)  (FIRST P1))
                     (EQUAL  (SECOND P)  (SECOND P1)))))

     (QUOTE "The theory of Intervals is now derived as a subtheory of Pairs")
13.  (DST INTERVAL (PAIR.OF INTEGER INTEGER))
14.  (DSV INTERVAL II)
15.  (DD INTEGER  BEGINNING (II)  (FIRST II))
16.  (DD INTEGER  END (II)  (SECOND II))
17.  (DSV INTEGER I)
18.  (DSV INTEGER J)
19.  (DSV INTEGER K)
20.  (DD INTERVAL  MAKE.INTERVAL (I J)  (MAKE.PAIR I J))
21.  (DS BOOL MEMBER (INTEGER INTERVAL))
22.  (DA A3 (IFF (MEMBER I II)
                (AND (LESSEQP  (BEGINNING II)  I)
                     (LESSEQP  I  (END II)))))
```

Figure 1

3.1 Formal Description of the Language

The language of our logic is similar to that of conventional multisorted first-order logic, but provides for parameterized sorts and sort hierarchies. Before describing the structure of formulas in our logic, we need first to define the language of sort expressions (which, for reasons of conformance with the specification literature, we call *type* expressions).

3.1.1 Language of Type Expressions

The vocabulary of type expressions is very much like that of ordinary first-order terms. A theory in our logic has a countable set of *type variables*, and for each $n \geq 0$, a countable set of n-ary *type symbols*. Type symbols of degree 0 are said to be *elementary*, while those of nonzero degree are said to be *parameterized*. Every n-ary parameterized type symbol has associated with it a *parameterized type template* given by an n-tuple of (not necessarily distinct) type variables. The intended meaning of the templates will be clear shortly. A *legal type expression*, or simply *type expression* is a term recursively constructed from type variables and symbols in the following manner:

a. A type variable is a type expression.

b. An elementary type symbol is a type expression.

c. If t is an n-ary parameterized type symbol, $t_1, t_2, ..., t_n$ are type expressions such that $t_i = t_j$ whenever the ith and jth components of the type template of t are equal, then $t(t_1, t_2, ..., t_n)$ is a type expression.

Note that the template of a parameterized type symbol forces certain of the symbol arguments in a type expression to be identical. If, for example, INTEGER and REAL are elementary type symbols, U and V are type variables, and MIXEDTRIPLE is a trinary type symbol with template <U,U,V>, then MIXEDTRIPLE(INTEGER INTEGER REAL), MIXEDTRIPLE(REAL REAL REAL), and MIXEDTRIPLE(V V U) are all legal type expressions, but MIXEDTRIPLE(INTEGER REAL REAL) is not.

We refer to type expressions that contain type variables as *schematic types* and those that do not as *ground types*. By *type substitution*, we mean a substitution that replaces type variables by type expressions. By a *type instance* of a given type, we mean any type resulting from the application of a type substitution to the given type.

The *raison d'etre* for schematic types is to permit us to talk about many types of objects at once. For example, we may want to formulate and apply a certain property of SETs, in various contexts, to SETs of INTEGERs, SETs of FOOs, and so on. Rather than stating and proving the property separately for SET(INTEGER), SET(FOO), etc., we need only prove it about SET(U), where U is a type variable. As will be seen later, we will then be able to apply the property in the context of each specific instance of U.

In addition to a set of type variables and type symbols (and templates), each theory in our logic has associated with it a *subtype structure*, expressed as a binary relation over type expressions. The subtype relation is defined in the following way.

First, certain type symbols are designated as *subtype symbols* . Associated with each elementary subtype symbol is a ground type expression, said to be its *immediate supertype*. Associated with each parameterized symbol s is a schematic type expression t, said to be the *immediate supertype* of the type expression $s(t_1, t_2, ..., t_n)$, where $< t_1, t_2, ..., t_n >$ is the template of s. The type expression t is constrained to have exactly the same set of type variables as the set of type variables occurring in the template of s. As a further constraint, it must be possible to find a total ordering of all subtype symbols in such a way that each is junior in the ordering to every subtype symbol occurring in its associated immediate supertype. (This constraint is necessary to prevent circularity in the subtype structure, and is automatically satisfied in the mechanization by virtue of the chronological ordering of subtype declarations.) Now, the subtype relation is defined recursively as the coarsest binary relation over type expressions that:

i. contains $< s, t >$ for each elementary subtype symbol s with immediate supertype t.

ii. contains $< s(t_1, t_2, ..., t_n), t >$ for each parameterized subtype symbol s with immediate supertype t and template $< t_1, t_2, ..., t_n >$.

iii. is closed under reflexivity, transitivity, and type instantiation.

By "closed under type instantiation", we mean that if t is a subtype of t' (i.e., $< t, t' >$ is in the relation) and σ is a type substitution, then $\sigma(t)$ is a subtype of $\sigma(t')$.

Suppose, for example, that U and V are type variables, that SET and SETOFPAIRS are unary type symbols with template <U>, that HOMOGPAIR is a binary type symbol with template <V,V>, and that INTEGER, RATIONAL, and REAL are all elementary type symbols. Suppose also that INTEGER is a subtype symbol with immediate supertype RATIONAL, RATIONAL is a subtype symbol with immediate supertype REAL, and that SETOFPAIRS is a subtype symbol with immediate supertype SET(HOMOGPAIR(U U)). Then the following are true:

INTEGER is a subtype of RATIONAL and REAL

RATIONAL is a subtype of REAL

SET(INTEGER) is a subtype of SET(INTEGER)

SETOFPAIRS(V) is a subtype of SET(HOMOGPAIR(V V))

SETOFPAIRS(SET(INTEGER)) is a subtype of SET(HOMOGPAIR(SET(INTEGER),SET(INTEGER)))

SETOFPAIRS(SET(SET(U))) is a subtype of SET(HOMOGPAIR(SET(SET(U)),SET(SET(U))))

Note, however, that SET(INTEGER) is *not* a subtype of SET(REAL).

One can prove from the definitions that the subtype relation imposes a well-founded partial ordering on the type expressions of the theory. This partial ordering, moreover, is structured as a set of top-rooted trees (thinking of sons as subtypes of fathers). A type expression can have several sons, but no type expression can have two unrelated ancestors.

3.1.2 Primitive Types

Different theories in our logic can, of course, have quite different type vocabularies, supertype structures, or both. All, however, are considered to share certain primitive types. These include the elementary type BOOL and the elementary types INTEGER, RATIONAL, REAL, and NUMBER. INTEGER is a subtype symbol with immediated supertype RATIONAL, RATIONAL is a subtype symbol with immediate supertype REAL, and REAL is a subtype symbol with immediate supertype NUMBER. Neither BOOL nor NUMBER is a subtype symbol. As we will see later, these symbols are all interpreted, i.e., have *a priori* semantics in interpretations. We will see that the semantics are as one would expect, except that NUMBER, REAL, and RATIONAL are considered to have identical semantics. In addition, each theory is considered to have the type variable *T*. The inclusion of at least one type variable is necessary for defining certain primitive function symbols, such as EQUAL.

As a theoretical aside, it might be noted that BOOL is the only primitive type that is truly necessary to provide the bootstrapping power needed to define interesting theories. For once BOOL is provided, one has all the power of conventional first-order logic, and can axiomatize other concepts (such as INTEGERs). We have included the other primitive types as an important convenience. As the conventional semantics of other useful types (such as SETs, SEQUENCES, and so on) are mechanized, these types will also be considered as primitives.

3.1.3 The Language of Formulas

In conventional predicate calculus, formulas are constructed from atomic formulas and the familiar propositional and first-order connectives. The atomic formulas, in turn, are constructed from predicate letters and term expressions. All of the structure at or above the level of predicates in a first-order formula is of course Boolean, whereas all of the function symbols occurring beneath the predicate symbols are interpreted over an arbitrary nonempty set (said to be the *domain* of the interpretation).

Formulas in our logic are constructed similarly, except that the symbols occurring in terms can have arbitrary types, including type BOOL. There is therefore no reason to distinguish between "predicates" and "terms". In recognition of this point, it will be convenient simply to speak of *symbolic expressions*; formulas, in particular, will merely be symbolic expressions of type BOOL. As an abbreviation, we will sometimes simply say "expression" rather than "symbolic expression" when there is no possible confusion with type expressions. It is important to note, in this connection, that we want to draw a firm distinction between symbolic expressions and type expressions; in particular, "TYPE" is not itself a type, contrary to the viewpoint expressed in some programming languages.

These remarks having been made, we return to formal description. Beyond the vocabulary of type expressions described earlier, a theory in our logic has a countable set of *symbolic variables* (or just *variables*) $v_1, v_2, \ldots$, and for each integer $n \geq 0$, a countable set of *n-ary function symbols* $f_1^n, f_2^n, \ldots$

Associated with each variable and function symbol is a *signature*. The signature of a variable is an arbitrary type expression, said to give the *type of that variable*. The type signature of an n-ary function symbol is an $n+1$-tuple of type expressions. The first component gives the *return type* of the function symbol, and the remaining n components the *formal argument types*. The only restriction placed on these type expressions is that for function symbols other than constants (i.e., of degree ≥ 1), *each type variable that occurs in the return type must occur in at least one of the argument types*. For example, if F is a unary function symbol with return type SET(U), then the formal argument type of F could be HOMOGPAIR(U U) but could not be HOMOGPAIR(INTEGER INTEGER). The intent here is that any ground binding of the type variables in the formal argument types should uniquely determine a ground instance of the return type. We will say that a variable or function symbol whose signature has at least one schematic type is *schematic*. The intuitive meaning is that a schematic symbol is a kind of abbreviation for an entire class of symbols, the signature of each member of which is a ground instance of the signature of the schematic symbol.

The *legal symbolic expressions*, or just *expressions* of a given theory are defined recursively as follows. We say that t is an *expression* if

i. t is a term, i.e., either a variable or of the form $f(t_1, t_2, \ldots, t_n)$, where f is an n-ary function symbol ($n \geq 0$) and each t_i is (recursively) a term, and

ii. *t typechecks.*

The only departure from predicate calculus is thus the type-checking restriction. Roughly speaking, the meaning of "typechecks" is what one would expect: that the arguments to a function symbol are of the appropriate type. Because of the presence of type variables and subtype structure, however, the exact meaning of "appropriate" needs some explanation.

Since the subtlety owes primarily to the type variables, let us first consider terms none of whose symbols is schematic. We will say that the *return type* of such a term is just the return type of the outermost function symbol (or, if the term is just a variable, the return type of the variable.) We will say that such a term typechecks if and only if the return type of each actual argument to a function symbol occurring in the term is a subtype of the corresponding formal argument type in the signature of the function symbol. For example, if QPLUS takes two RATIONALS and returns a RATIONAL, and if INTEGER is a subtype of RATIONAL is a subtype of REAL, then QPLUS(X X) typechecks if the type of X is INTEGER or RATIONAL, but not if the type of X is REAL.

The situation becomes more interesting if any of the symbols occurring in the term *t* is schematic. In this case, *t* typechecks if there is a way of instantiating the signature of each such symbol that causes *t* to typecheck in the sense just given for terms with no schematic symbols. For example, suppose that F is a schematic symbol with signature <SET(U),U,U> (i.e, F takes two arguments of type U, U a type variable, and returns a SET(U)). Suppose also that I and X are variables with types INTEGER and RATIONAL, respectively. Then F(I X) typechecks, since applying the substitution {U / RATIONAL }to the signature of F (meaning that F now takes two RATIONALS and returns a SET(RATIONAL)) causes the type of each actual to be a subtype of the corresponding expected argument type.

In the case where a schematic symbol occurs more than once in a term, we permit a separate instantiation of its signature for each occurrence. For example, if G is a binary function symbol that takes a SET(BOOL) and a SET(REAL) as arguments, the term G(F(TRUE, TRUE), F(I, I)) typechecks, using the substitution {U / BOOL}for the left occurrence and {U / REAL}for the right occurrence of F.

Schematic variables and constants are treated specially. The substitutions associated with occurrences of variables and constants in an expression must agree on all type variables their signatures have in common. For example, if variables A and B are both of type SET(U), the term G(A,B) does not typecheck, since the substitutions associated with A and B must give U the same value.

Each assignment of substitutions to the symbol occurrences of a term that causes the typechecking requirement to be satisfied will be called a *syntactic interpretation* of the term. We can say, then, that a term is a legal expression if and only if it has at least one syntactic interpretation. Just as schematic symbols can be viewed as representing a class of symbols, terms involving schematic symbols can be considered as representing classes of terms, one for each syntactic interpretation.

It should be noted that the definition of "typechecks" just presented is completely nonconstructive; we have not said how to find the needed signature instances; indeed, we have not even explained how or whether it is possible to tell whether one type expression is a subtype of another. Clearly, without an effective way of testing for syntactic well-formedness, a logic is hardly of practical interest. Fortunately, both the subtype relation and the typecheck predicate are effectively (and easily) computable. We show in a forthcoming paper that the subtype relation can actually be computed using a finitely terminating canonical term rewrite system. A type expression *t* is a subtype of another type expression *t'* iff *t* (eventually) rewrites to *t'*; the canonical forms of the system are those types that are subtypes only of themselves.

We also give an algorithm for type checking that depends on the notion of *least general syntactic interpretation*. We say that one syntactic interpretation is *less general* than another if the type substituted for each type variable in the substitution corresponding to a given symbol occurrence in the first syntactic interpretation is a subtype of the corresponding substituted type in the second. (For example, the syntactic interpretation of F(I I) in which the substitution {U / INTEGER }is used for the occurrence of F is less general than that in which {U / RATIONAL }is used.) One can show that each expression has a unique least general syntactic interpretation modulo the assignment of types to the type variables in the signatures of variable and constant symbols.

3.1.4 Primitive Function Symbols

Each theory is considered to include the following function symbols as primitives. For each symbol, the return type is given first, then the function symbol followed by a list of argument types.

BOOL TRUE()

BOOL FALSE()

BOOL AND(BOOL BOOL)

BOOL OR(BOOL BOOL)

BOOL IMPLIES(BOOL BOOL)

BOOL NOT(BOOL)

BOOL IFF(BOOL BOOL)

BOOL FORALL(*T* BOOL)

BOOL EXISTS(*T* BOOL)

BOOL EQUAL(*T* *T*)

BOOL LESSEQP(NUMBER NUMBER)

BOOL LESSP(NUMBER NUMBER)

BOOL GREATEREQP(NUMBER NUMBER)

BOOL GREATERP(NUMBER NUMBER)

T IF(BOOL *T* *T*)

NUMBER PLUS(NUMBER NUMBER)

NUMBER DIFFERENCE(NUMBER NUMBER)

NUMBER MINUS(NUMBER)

NUMBER TIMES(NUMBER NUMBER)

The IFF construct shown above is interpreted as Boolean equivalence. The IF construct is the McCarthy three-placed if statement that is interpreted to return the value of the second or third argument, depending on whether the predicate in the first argument place is true or false, respectively.

In addition to those in the list above, there is a constant symbol for each integer and rational number. Note that arithmetic functions, such as PLUS, all take NUMBERs as arguments. Thus, if I is an INTEGER variable, and F is a monadic function symbol with argument type INTEGER, then PLUS(I I) is a legal expression, but F(PLUS(I I)) is *not*, even though the semantics of INTEGERs guarantee that the sum of two integers is always an integer. The desired effect can be obtained in practical applications by introducing a function symbol IPLUS with INTEGER arguments and return type, and endowing that symbol (using a definition) with the semantics of PLUS, restricted in integers.

The signatures of EQUAL and IF in the list above both use the primitive type variable *T*. It follows from the typecheck requirement that the types of the two actual arguments to EQUAL in a legal expresssion need not be the same; they must, however, have a common supertype. Similarly, the types of the two branch expressions of a three-placed IF construct must have a common supertype.

For the two quantifiers (FORALL and EXISTS), an additional syntactic restriction is imposed beyond the usual typecheck rule: the first argument to each of these in a legal expression must be a variable.

3.2 Semantics of Theories in the Logic

As one would hope, the meaning of formulas in the language reflects intuition. Once again, the only real departure from conventional quantification theory with equality stems from the presence of sorts.

Expressions have meaning only with respect to *interpretations*. An interpretation consists of the assignment of sets, called *domains*, to the type expressions of the theory, and of functions to the function symbols of the theory.

More specifically, an interpretation assigns to each ground type expression a nonempty set in such a way that for any two ground type expressions t and t'

i. The set associated with t is a subset of that associated with t' if and only if t is a subtype of t'.

ii. The set associated with t has a non-empty intersection with that associated with t' if and only if t and t' have a common supertype.

iii. The set of rational numbers is associated with types NUMBER, RATIONAL, and REAL, the set of integers is associated with type INTEGER, and the set of truth values (*true* and *false*) is associated with type BOOL.

Clause (i) says that the partial ordering of assigned domains under set inclusion must be isomorphic to the subtype partial ordering. Clause (iii) gives RATIONALS, INTEGERS, and BOOLS their standard interpretation, but identifies NUMBERS and REALS with rationals. REALs thus acquire the same interpretation they are given in most programming languages in recognition of the limitations of machine representation.

An intepretation also assigns to each n-ary nonschematic function symbol an n-ary function whose signature is obtained from that of the function symbol by replacing each type expression with its assigned domain. Constant symbols, in particular, are assigned elements from the domain associated with their return types. Primitive nonschematic function symbols are given their standard meanings.

Every schematic function symbol is assigned a multiplicity of functions – one for each ground instance of its signature. (Intuitively, one can think of schematic function symbols as abbreviations for a class of symbols.) The signature of the assigned function corresponding to a given signature instance is that obtained by replacing each type expression in the instance by its assigned domain. Once again, primitive symbols, including the quantifiers, are given their usual semantics. Note that each quantified variable ranges over a single domain.

Now, each closed expression (closed in the sense of having no unbound variables) takes a meaning, or *valuation*, with respect to a given interpretation I and a given assignment A of ground type expressions to the type variables occurring in the signatures of variable and constant symbols. For expressions involving only nonschematic symbols, the valuation is defined recursively in the way one would expect: the valuation of a constant is just the domain element assigned to it by I, and the valuation of a nonconstant is obtained by recursively applying the function assigned to its outermost symbol to the valuations of its arguments. For expressions involving schematic symbols, the valuation is defined as that of the least general syntactic interpretation (as defined earlier) modulo the assignment A. (Note that the assignment A is analogous to the assignment of domain values to free variables in interpretations of predicate calculus.)

The notions of validity and satisfiability are now defined in the usual way. A formula (i.e., an expression of type BOOL) is said to be *valid* if and only if its universal closure has valuation *true* in all interpretations; it is *satisfiable* if its existential closure is *true* in at least one interpretation.

3.3 The Proof Theory and its Mechanization

The connection between the model-theoretic semantics described in the previous section and the proof mechanism used in our system is made by a generalized form of the Skolem-Herbrand-Gödel theorem. Herbrandian semi-decision procedures, of course, have long been the most popular means of mechanizing quantified logics. Because of the extensions to ordinary quantification theory provided by our language, a generalized form of the theorem must be used and a mechanical proof procedure formulated on the basis of this generalization.

Before outlining the theorem and the derived procedure, it will be helpful to describe the deductive mechanism that lies at the heart of the prover. This mechanism consists of an efficient implementation of a decision procedure for unquantified formulas in an extension of Presburger arithmetic. The theory includes the usual propositional connectives: equality, rational and integer variables and constants, the arithmetic relations $<$, $\leq$, $>$, $\geq$, and addition and multiplication. Uninterpreted predicate and function symbols of type integer and rational are also permitted. The decision procedure is complete for the subtheory of this theory having no integer variables or function symbols, and containing no nonlinear use of multiplication. The procedure is sound, of course, for the entire theory, and is able to prove the vast majority of formulas involving integer constructs that are actually encountered in practice. The speed of proof, moreover, is fairly impressive: theorems occupying several pages in the SIFT effort were usually proved in well under a minute. (Indeed, as we noted elsewhere, such speed has proven to be essential to our methodology.) It should be noted, however, that this decision theory does not include *quantified* formulas, nor does it support function symbols of user-defined types.

The proof procedure derived from our generalization of the S.-H.-G. theorem can be viewed as a means of reducing the proof of formulas in the typed first-order theory that the user has formulated to the automatic proof of formulas in the underlying unquantified decision theory we have just described. Informally speaking, the S.-H.-G. theorem states that a formula of predicate calculus is valid if and only if some disjunction of instances of its validity Skolem form is tautological. The instances must replace variables in the Skolem form with terms in the Herbrand Universe of the formula. The generalization of the theorem to deal with formulas in our typed language is stated similarly, but requires that

i. Each variable be instantiated with an expression whose type is a subtype of that of the variable, and

ii. Each instance typechecks.

The generalization states that the given formula is valid according to the semantics defined in the last section if and only if some disjunction of instances satisfying (i) and (ii) above is *true, considered as a formula in the underlying decision theory,* where it is understood that symbols of a subtype of BOOL are considered to be of type BOOL, symbols of a subtype of INTEGER are considered to be of type INTEGER, and all other symbols are considered to be of type RATIONAL. In effect, the generalization holds that once the instances have been typechecked, all of the type information other than that distinguishing RATIONALs, INTEGERs, and BOOLs can be stripped away.

The theorem immediately gives rise to a proof procedure: with assistance from the user, appropriate instances of the Skolem form of the theorem to be proved (together with instances of any axioms or lemmas needed to prove it) are formulated, disjoined, and submitted to the underlying decision procedure. A detailed description of the decision procedure is given in [Sho 82].

Operationally , the user is saved from any details of this process other than selecting appropriate instances. In particular, the process of Skolemization is completely transparent to him, as are the disjunction of instances and the submission of this disjunction to the underlying decision mechanism. The user is thus free to reason exclusively at the level of his first-order typed theory.

As an illustration, let us return to the example from the theory of Intervals presented in Section 2.1.

We prove the simple theorem that every interval with an end point greater than or equal to the beginning point contains at least one number as a member. Figure 2 shows the expression and proof of this theorem. Line 23 encodes the theorem within the theory of Intervals previously defined. As expected, one proves a formula of the form $\exists x\ P(x)$ by demonstrating some value v such that $P(v)$. In this case, the user must observe that, from (GREATEREQP J I) in the antecedent, axiom A3 defining Interval membership, and the definition given in A1 of the Interval constructor MAKE.INTERVAL, one can determine that the beginning interval value I satisfies the formula (MEMBER I (MAKE.INTERVAL I J)). In line 24, the user invokes the PR command to construct this proof. The system then prompts the user for the needed substitutions.

```
23. (DF THEOREM (IMPLIES (GREATEREQP J I)
                         (EXISTS K (MEMBER K (MAKE.INTERVAL I J)))))

24. (PR THEOREM
        A1
        A3)

    Want instance for THEOREM? Y
      K/ I
    Want instance for A1? Y
      Y/ J
      X/ I
    Want instance for A3? Y
      II/ (MAKE.INTERVAL I J)
      I/

    ———-Proving———-
    160 conses
    .2 seconds
    Proved
```

Figure 2

4 The Proof of SIFT

SIFT (Software-Implemented Fault Tolerance)[Wen 78] is a reliable aircraft flight control computer system. SIFT uses five to eight Bendix BDX930 computers, each equipped with its own private storage. A broadcast communication mechanism allows each processor to transmit its results to a buffer area in each processor. The design is fully distributed, with no common buses, no common clocks, no common interrupt or synchronization registers, no common storage, and no physical means of isolating a faulty processor. The SIFT processors (physically) share only the ability to communicate with one another.

In SIFT, fault masking, detection, and reconfiguration are all managed by software. Safety-critical tasks are replicated on three or more processors, with all processors voting on the results of each redundant computation. A Global Executive task, which is itself replicated, is responsible for fault diagnosis on the basis of error reports from the voting software, and for selecting a new configuration excluding the processors deemed to be faulty. The result of this reconfiguration is that tasks are shifted from faulty processors to those still working. Every processor votes on the results of the Global Executive and adjusts its task schedule accordingly.

SIFT's processors run asynchronously; each contains its own private clock. The software must maintain a loose synchronization to within approximately 50 microseconds, and each processor runs a task periodically to resynchronize its clock with those of the other processors in the configuration. Care was taken in the design to ensure that, even under fault conditions, all working processors retain synchronization and remain consistent in their schedule and configuration.

4.1 The Design Hierarchy of SIFT

The problem of specification credibility in the proof of SIFT is addressed through the use of hierarchical design specification and verification. This approach allows incremental introduction and verification of design aspects – making a step-by-step connection between the high-level, abstract view of the system to the detailed control and data structures employed in the implementation. Figure 3 illustrates the SIFT design hierarchy. At present, the STP system does not provide specific mechanical support for the hierarchical specification structure; we discuss future work in this direction in Section 4.

The IO Specification, the most abstract functional description of the system, asserts that, *in a safe configuration*, the result of a task computation will be the effect of applying its mathematical function to the results of its designated set of input tasks, and that this result will be obtained within a real-time constraint. Each task of the system is defined to have been performed correctly, with no specification of how this is achieved. The model has no concept of processor (thus no representation of replication of tasks or voting on results), and of course no representation of asynchrony among processors. The specification of this model contains only 8 axioms.

The Replication Specification elaborates upon the IO Specification by introducing the concept of processor, and can therefore describe the replication of tasks and their allocation to processors, voting on the results of these replicated tasks, and reconfiguring to accommodate faulty processors. The specification defines the results of a task instance on a *working* processor based on voted inputs, without defining any schedule of execution or processor communication. This model is expressed in terms of a global system state and system time.

The Broadcast Specification develops the design into a fully distributed system in which each processor has access only to local information. Each processor has a local clock and a broadcast communication interface and buffers. The asynchrony among processors and its effect upon communication is modeled. The specification explicitly defines each processor's independent information about the configuration and the appropriate schedule of activities. The schedule of activities defines the sequence of task executions and votes necessary to generate task results within the required computation window. The Broadcast Specification is the lowest level description of the complete multiprocessor SIFT *system*.

The PrePost Specification consists of specifications for the operating system for a single processor. The specification, in terms of pre-condition/post-condition pairs, facilitates the use of sequential proof techniques to prove properties of the Pascal-based operating system as a sequential program. These specifications are very close to the Pascal programs, and essentially require the programs to "do what they do".

The Reliability Analysis is a conventional discrete semi-Markov analysis that calculates the probability that the system reaches an unsafe configuration from the rates of solid and transient faults and from the reconfiguration rates. Neither this Reliability Analysis nor the other Fault and Error Models will be described here.

A more detailed presentation of the SIFT specifications and their verification can be found in [MeS 82].

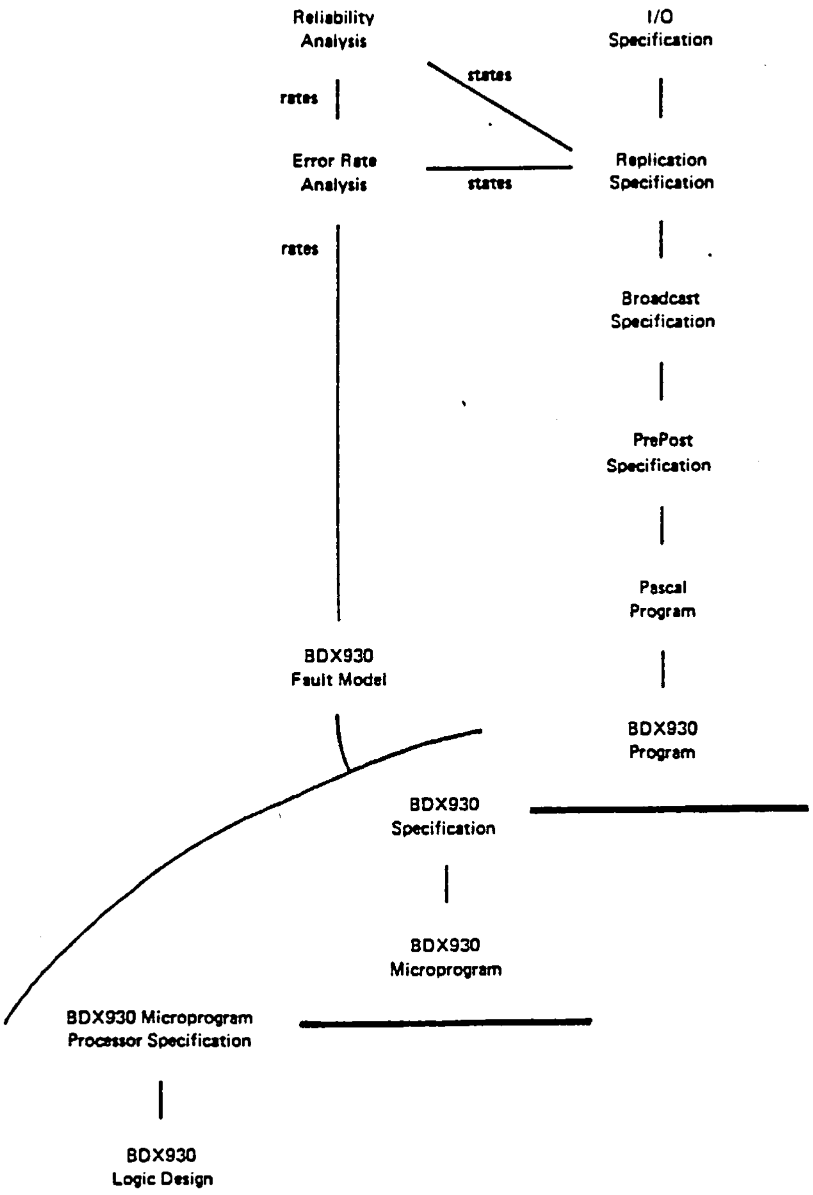

Reliability Analysis
I/O Specification
rates
states
Error Rate Analysis
states
Replication Specification
rates
Broadcast Specification
PrePost Specification
Pascal Program
BDX930 Fault Model
BDX930 Program
BDX930 Specification
BDX930 Microprogram
BDX930 Microprogram Processor Specification
BDX930 Logic Design

FIGURE 3

4.2 The Proof of the SIFT Design

Hierarchical proof of design involves axiomatically defining a mapping from functions and predicate symbols of a level of the design to terms of the level below. One then proves each axiom of the higher level as a theorem in the theory of the level below.

The most substantial portion of the verification of the IO Specification involved proof of the axioms that define the results of tasks in a safe configuration. To derive these from the Replication level involved a demonstration that task replication and majority voting suffice to mask errors in a safe configuration. To do so required approximately 22 proofs, with an average of 5 premises necessary per proof, and 106 instantiations of axioms and lemmas used overall.

The proof of the relationship between the Replication Specification and the Broadcast Specification was more challenging. This proof required demonstrating the consistency of information present in each working processor (the set of processors still in the configuration, in particular), of the correct schedule to be used, and of the results of past task iterations. Furthermore, the proof required showing that the various processors, operating independently and asynchronously with only local information, can communicate with each other without mutual interference, and can cooperate to produce a single global system defined by the Replication Specification. It was also necessary to show that the task schedules were such that the task results are always available in other processors when required. The derivation of the Replication axioms involved 56 proofs, with an average of 7 premises each, and 410 instantiations of axioms and lemmas overall.

The proof of the relationship between the Broadcast Specification and the PrePost Specification was easier. Most of the interest centered on the mapping between the properties of the changing values of variables in the Pascal system and the properties of the Broadcast model's more abstract state functions which are explicitly parameterized by time and processor. A futher complication concerned mapping the functional representation of data structures in the Broadcast model to the (finite) Pascal data structures. Derivation of the necessary Broadcast axioms involved 17 proofs, with an average of 9 premises each, and 148 instantiations overall. The proof of the Pascal programs from the PrePost Model specifications used conventional verification condition generation.

5 Future Work

As a result of experience with both design and code proof in the SIFT effort, several improvements and changes to STP are contemplated. We intend to implement decision procedures for an expanded set of primitive theories – all syntactically characterizable. We envision doing this at least for a large fragment of set theory (following the work of Ferro and Schwartz [FeS 80]), sequences, and tuples. Mechanical support for induction schemes is also needed. The user currently can introduce "induction" axioms, but with no implied semantics. We also anticipate providing direct mechanical support for hierarchical development.

Improvements to the user interface will take several forms. In the current system, the user is forced to specify theories in the abstract syntax of Lisp. For all but the heartiest of users, pure prefix syntax is clearly unsatisfactory. We are currently defining an external, enriched specification language and a corresponding reduction to the current type theory. The language will contain explicit support for hierarchical specification, theory encapsulation, scoping, and implicit state; it will support SPECIAL-like state-machine specifications [LRS 79] as a sublanguage.

A graphics interface is contemplated to assist in constructing and manipulating theories and libraries of theories. In addition, our experience with code proof indicates that forcing the user to reason at the level of formulas mechanically generated by a VCG is doomed to failure. Reasoning at this level is antithetical to our philosophy of man-machine discourse at the level of user conceptualization – in this case, the level of the program structure. We intend over the next year to experiment with user interaction at the level of path (and subpath) assertions within the program structure, hoping that this will achieve in the program domain what was gained in the present system by hiding the internal first-order/ground formula mapping from the user in present design proof.

Finally, we intend to supplant after-the-fact mechanical support for complete proof construction with incremental proof construction. Our goal is to replace, insofar as possible, explicit user instantiation of symbols with a formal language in which the user can incrementally construct and apply a high-level proof strategy. With this approach, we believe we have a more efficient proof strategy while requiring less effort on the user's part. In this way, we hope to preserve the man-machine balance as we increase the user's capability to formulate and reason about larger conceptual steps within the theory.

6 Acknowledgments

We want to acknowledge the contributions of Dwight Hare to the development of STP and the proof of SIFT, and of Leslie Lamport to the early formulations of the SIFT specifications.

References

[Ble 74] Bledsoe,W.W., "The Sup-inf method in Presburger arithmetic", Memo ATP-18, Mathematics Dept., Univ. of Texas, Austin, Texas, Dec. 1974.

[BoM 79] Boyer, R., J Moore, "A computational logic", Academic Press, 1979.

[BuG 77] Burstall, R., J. Goguen, "Putting theories together to make specifications", Proc. IJCAI, August 1977.

[Els 79] Elspas, B., et al. "A Jovial Verifier", SRI International, June, 1979.

[FeS 80] Ferro, A., E. Omodeo, J. Schwartz, "Decision procedures for some fragments of set theory", 5th Conf on Automated Deduction, July 1980.

[Goo 79] Good, D., R. M. Cohen, J. Keeton-Williams, "Principles of proving concurrent programs in Gypsy", *Proc 6th POPL*, 1979.

[Kin 69] King, J., "A program verifier", Ph.D. thesis, CMU, 1969.

[LeW 75] Levitt, K., R. Waldinger, "Reasoning about programs", *AI Journal* 5, 1974.

[LRS 79] Levitt, K., L. Robinson, B. Silverberg, "The HDM handbook", SRI International, 1979.

[Luc 79] Luckham, D., N. Suzuki, "Verification of array, record, and pointer operations in Pascal", *TOPLAS*, Oct, 1979.

[MeS 82] Melliar-Smith, P. M., R. L. Schwartz, "Formal specification and mechanical verification of SIFT: a fault-tolerant flight control system", *IEEE Transactions on Computers*, July 1982.

[Mil 71] Milner, R., "An algebraic definition of simulation between programs, CS 205, Stanford, 1971.

[Mil 79] Milner, R., "LCF: a way of doing proofs with a machine", *Proc 8th MFCS Symp*, 1979.

[Mus 80] Musser, D., "Abstract data type specification in the Affirm system", *IEEE TSE*, Jan. 1980.

[Nak 77] Nakajima, R., M. Honda, H. Nakahara, "Describing and verifying programs with abstract data types", *Formal Description of Programming Concepts*, North Holland, 1977.

[OpN 78] Oppen, D., G. Nelson, "A simplifier based on efficient decision algorithms", Proceedings of Fifth POPL, Tucson, Arizona, Jan. 1978.

[Par 72] Parnas, D., "A technique for software module specification with examples", *CACM*, May 1972.

[RoL 77] Robinson, L., K. Levitt, "Proof techniques for hierarchically structured programs", *CACM*, April 1977.

[Sch 80] Schorre, V., J. Stein, "The interactive theorem prover (ITP) user manual", TM-6889/000-/01, SDC, 1980.

[Sho 77] Shostak, R., "A practical decision procedure for arithmetic with function symbols", *JACM*, April 1979.

[Sho 82] Shostak, R., "Deciding Combinations of Theories", *Proceedings of the Sixth Conference on Automated Deduction*, June 1982.

[Suz 75] Suzuki, N., "Verifying programs by algebraic and logical reduction", Proc. Int. Conf. on Reliable Software, Los Angeles, 1975.

[Wen 78] Wensley, J., et al., "SIFT: Design and Analysis of a Fault-tolerant Computer for Aircraft Control", *Proc IEEE*, Vol. 66, No. 10, Oct. 1978.

[Wey 80] Weyhrauch, R., "Prolegamena to a theory of mechanized formal reasoning", *AI Journal*, 1980.

CHAPTER 3

STP THEOREM PROVER – COMMAND INTERFACE

STP Theorem Prover--Command Interface

Command Reference List for STP

Declaration Commands

DTV(symbol) (Declare Type Symbol)

Declares symbol to be a type variable. Symbol must be atomic and not presently declared as a type or type variable.

DT(symbol [(typevar1, typevar2 ... typevarn)]) (Declare Type)

Declares symbol to be a new type. Symbol must be atomic and not presently declared as a type or type variable. If the optional list (typevar1 typevar2 ... typevarn) is provided, symbol is declared as a parameterized type. In this case, n must be at least 1 and each typevari must be presently declared as a type variable. The typevari's need not be distinct.

DST(symbol [(typevar1 typevar2 ... typevarn)] typexp) (Declare SubType)

Declares symbol to be a subtype of the type given by the type expression typexp. Symbol must be atomic and not presently declared as a type or typevariable. If the optional list (typevar1 typevar2 ... typevarn) is provided, symbol is declared as a parameterized subtype of typexp. In this case, n must be at least 1 and each typevari must be presently declared as a type variable. Typexp must be a legal type expression. The set of parameters of the declared subtype must be identical to the set of type variables occurring in typexp. (In particular, if no parameter list is given, typexp must contain no occurrences of type variables).

DSV(typexp symbol) (Declare Symbol Variable)

Declares symbol to be a variable of type typexp. Symbol must be atomic and not presently declared as a variable or function symbol. Typexp must be a legal type expression.

DS(typexp symbol (typexp1 typexp2 ... typexpn)) (Declare Symbol)

Declares symbol to be function symbol of return type typexp and argument types typexp1, typexp2, ... typexpn. Symbol must be atomic and not presently declared as a variable or function symbol. Typexp, typexp1, ... typexpn must be legal type expressions. n may be zero, in which case symbol is declared as a constant. In this case, the third argument to DS need not be provided. Note that this is not the same as declaring a symbol variable. Unless n is 0, each type variable occuring in the return type typexp must occur as a type variable of at least one of typexp1 typexp2 ... typexpn.

DD(typexp symbol (var1 var2 ... varn) exp) (Declare Definition)

Declares symbol as a defined function symbol of type typexp with formal parameter list (var1 var2 ... varn) and body exp. Symbol must be atomic and not presently declared as a variable or function symbol. typexp must be a legal type expression. Each vari must be presently declared as a variable, and exp must be legal expression containing at least one occurrence of each vari. N may be zero. The declaration causes symbol to be declared per

DS(typexp symbol (type1 type2 ... typen))

where type1, type2, ... typen, respectively, are the types of var1, var2, ... varn. Note that since exp must be a (presently) legal expression, symbol cannot be defined recursively.

The body exp must be a legal symbolic expression not containing any quantifiers. The type of the body need not be the same as typexp. However, the expression (EQUAL symbol exp) must be a legal symbolic expression (where symbol is assumed to be declared as above.)

DF(name formula) (Declare Formula)

Associates name with formula. Name must be atomic, and not currently associated with a formula. Formula must be a legal expression of type BOOL. The status of name is initialized to UNPROVED.

DA(name formula) (Declare Axiom)

Associates name with formula, and changes its status to AXIOM. Equivalent to: DF(name formula) CS(name AXIOM)

DC(newname oldname) (Declare Copy)

Associates newname with the formula with which oldname is currently associated. Newname must be an atomic symbol not presently associated with any formula, and oldname must be presently associated with a formula. Equivalent to:

DF(newname form)

where form is (FORMULA oldname).

UDT(symbol) (UnDeclare Type)

Undeclares type or type variable symbol. Symbol must currently be declared as a type or type variable, but must not be referred to in the declaration of some other presently declared type or symbol.

UDS(symbol) (UnDeclare Symbol)

Undeclares function symbol or variable symbol. Symbol must currently be declared as a function symbol or variable, but must not be referred to in the declaration of any presently declared formula. If symbol was declared by a DD, the associated definition is lost.

UDF(symbol) (Undeclare Formula)

Undeclares the formula associated with symbol. Symbol must currently be associated with a formula. The status of all formulas whose proof depends upon (either directly or indirectly) the undeclared formula is changed to UNPROVED.

Commands Associated with Proving

PR(form form1 form2 ... formn) (PRove)

Attempts to prove the formula form using form1, form2, ..., formn as hypotheses. N may be zero. Form must be either the name associated with a presently declared formula, or a list of the form

(name sub)

where name is associated with the formula to be proved, and sub is a substitution to be applied to the free variables in the validity skolem form of that formula before the proof is attempted. Sub must be of the form ((var1 term1)(var2 term2) ... (varn termn)) where each vari is one of the free variables, and termi is a legal expression. The vari's must be distinct, but need not include all of the free variables. The expression obtained by substituting each termi for vari in the validity skolem form of the formula associated with name must be a legal expression (i.e., must typecheck.) In addition, the type of each termi must be a subtype of some instance of the type of vari. Further restrictions and conventions regarding the termi's are discussed below.

In the case where form is an atom rather than of the form (name sub)), if the named formula has free variables in its validity skolem form, the user is asked whether he desires an instance. If so, he is prompted for a substitution variable by variable. If no substitution is desired for a given variable, the user can so indicated by pressing carriage return.

Similarly, each of the hypotheses form1, form2, formn may be either the name of a declared formula, or a list of the form (name sub). Here, however, sub must be a substitution for the free variables of the satisfiability skolem form of the named formula, as opposed to the validity skolem form. Once again, if no list is supplied, the user is asked whether he desires an instance.

The formula submitted to the theorem prover consists of the implication of the instantiated formula to be proved by the conjunction of the instantiated hypotheses. (Upon completion of the PR command, this formula is the value of the atom IMPLICATION. A version of IMPLICATION with all defined symbols replaced by their definitions is given by EXP.IMPLICATION). If the proof is successful, the status of form (or the first element of form, if form is a list) is changed to PROVED. Otherwise, the status is unchanged.

Proof Debugging Commands

CHECK MONITOR

Enters the CHECK MONITOR subsystem. This subsystem provides some help in discovering why a proof in STP has failed. After a proof in STP has failed, calling the function (CHECK.MONITOR) will place the user in a subsystem which provides information about the failed proof. While in the CHECK.MONITOR, the user is prompted for a command by a "*". The subsystem commands are E, PP.PR, PP.PR.ATTEMPT, QI, QF, PR, and PC.

E

The E command exits the CHECK.MONITOR.

PP.PR

The PP.PR command prints the conclusion and each premise along with a formula number (the conclusion being numbered zero) and the substitutions for each formula.

PP.PR.ATTEMPT

The PP.PR.ATTEMPT command prints the information in PP.PR along with the instantiated versions of each formula.

(QI number)

The QI command is typed in parentheses along with a number as in (QI 3) and prints the substitutions and instantiated version of the numbered formula, as in PP.PR.ATTEMPT.

(QF name)

The QF command is parenthesized along with a formula name as in (QF PREMISE1) and prints the uninstantiated version of the named formula.

(PR number) or (PR numberlist)

The PR command is parenthesized and takes either a single number as an argument as in (PR 3), a list of numbers as in (PR (1 2 3)), or (PR ALL). This command attempts to check the applicability of each (specified) numbered premise in the proof by assuming the proof proceeds through the use of Modus Ponens.

(PC number) or (PC numberlist)

The PC command is similar to the PR command and takes the same type of arguments. It checks the applicability of premises assuming the contrapositive direction (i.e., attempts to show that the consequent is false and thus the antecedent is also false).

CHAPTER 4

OVERVIEW OF DESIGN VERIFICATION

Specifying and Verifying Ultra-Reliability and Fault-Tolerance Properties

Richard L. Schwartz and P.M. Melliar-Smith

Abstract

A methodology to rigorously verify ultra-reliability and fault-tolerance system properties is described. The methodology utilizes a hierarchy of formal mathematical specifications of system design and incremental design proof to prove the system has the desired properties. A small example of the approach is given, and the application of the methodology to the large-scale proof of SIFT, a fault-tolerant flight control operating system, is discussed.

1 Introduction

How does one begin to substantiate a claimed Mean Time Between Failures (MTBF) of a million years? This was the problem facing the designers of SIFT [1]. Clearly, rates of failure this small are beyond the point where testing and fault injection can suffice.* Validation by fault injection, while necessary, is unlikely to convince one that the reliability requirements have been met.

Substantiation of such an ultra-reliability requirement must be based on some form of analytic reliability analysis. Discrete Markov analysis is frequently used to analyze system failure and recovery transition rates. Because of the normally quite large number of actual system states and failure modes, one typically uses an extrapolation from fault rates and system states that are easier to measure.

The validity of this extrapolation depends on a number of assumptions, and, at the desired level of reliability, even "minor" violations of the assumptions can have

This work was supported by the NASA Langley Research Center under Contract NAS1-15528

*Indeed, this has been referred to as the "Smithsonian Experiment".

significant effects on the reliability achieved. Thus the assumptions must themselves be quite rigorously substantiated if the claimed reliability is to be believed. For instance, one important assumption of the Markov analysis is that the occurrence of faults is well described by a Poisson model with complete independence between processors.

The validity of the Markov analysis depends also on the assumptions (1) that the states and the transitions of the Markov model correspond accurately to the actual system, and (2) that the states in which system failure is possible are correctly identified. But this correspondence is far from obvious: actual systems have very many states with many complex transitions between them. Without some means to reconcile the assumed states and transitions of the Markov analysis with those of the real system, one can produce highly optimistic reliability estimates.

In attempting to substantiate the ultra-reliability requirement of SIFT, we employed a three-part methodology. The first of these is a demonstration that, so long as a **system safe** predicate remains true, the system performs the desired flight control function, even though one or more processors may be faulty. This is a correctness property for the *function performed* by the system.

The second is a demonstration that the Markov analysis computes an upper bound on the probability that **system safe** becomes false. This is a correctness property for the *probabilistic reliability analysis* of the system.

The third and last step in the methodology is to prove that each state and transition of the Markov model reflects a valid abstraction of the states and transitions of the functional specification.

By showing that **system safe** remains true over the desired period and that its being true implies the system will perform as desired, one can establish correct and reliable system operation.

Because even a very small defect in the demonstrations could allow failures at an unacceptable rate, these demonstrations must be performed with the rigor of mathematical proof. Our experience has been that it is simply too easy either to overlook or abstract details of system operation inappropriately. *A formal, unambiguous, specification and a formal system of mathematical deduction* is necessary to attain the degree of confidence expected for critical system components. Run-time validation, (i.e., testing) techniques simply cannot be used to ensure that software, operating on a working processor, will perform as specified. *Exhaustive* software testing being impractical, a rigorous methodology for specifying and proving properties of all possible program behavior becomes necessary.

The need for formal mathematical proof to ensure the desired functional and reliability requirements presents two major issues:

- ▸ How does one define the criteria sufficient to ensure the correct functioning of the system?

- How does one prove that the criteria are satisfied by the actual system?

The first issue is crucial if the formal verification effort is to have any practical significance. Even as a noncomputer scientist, one must have confidence, that the formal specifications, stating what is meant by the correct functioning of the system, in fact reflect the *intended behavior.* That a formal specification expresses what the system designer intuitively means must, in the end, be determined by inspection. A formal specification must therefore be *believable* if rigorous mathematical correspondence to the specification is to ensure the desired effect.

The larger and more complex the system, the more acute the problem becomes. Specifications reflecting the detailed behavior of the system allow the most straightforward formal verification effort, but it is difficult to ensure that low-level specifications embody what is meant by the proper functioning of the system. Very high-level specifications, abstracting from the details of the system, are necessary if we are to state the overall functional and fault-tolerance properties of the system in a way that can be understood and believed. The problem then becomes one of reconciling the very high-level specifications with the detailed transformations performed by the programs of the actual system.

In order to state high-level system specifications that can be shown to be consistent with the actual program, one must formulate not just a single specification of the system, but a *hierarchy* of specifications. Our approach is to state a tiered set of system specifications, as illustrated in Figure 1.

Each level L_i in the hierarchy specifies an *abstract view of the system design* in terms of a set of primitive predicates P_i and functions F_i. The specification for the model is given by a set of axioms, characterizing those properties of the model appropriate for that level of system abstraction. At each level in the hierarchy, a specification L_i can be seen as an *abstraction* of the previous level L_{i+1}. Correspondence between successive levels is maintained by expressing each primitive function and predicate of higher-level L_i in terms of the functions and predicates of the lower-level L_{i+1}.

With this mapping, one must then prove that each property derivable from the higher-level specification can be proved from the lower-level specification. The mapping between levels need not be complete; the mapping itself may be given as a set of axioms, saying only enough about the correspondence to derive the necessary axioms of the higher level as theorems from the axioms of the lower-level. It is required only that the mapping axioms be *consistent,* i.e., that there exist a complete functional mapping between levels which satisifies the mapping axioms. By demonstrating the correspondence between successive levels L_i and L_{i+1}, one can conclude by induction that any property provable from the highest-level specification is also provable from the lowest-level specification. Thus, any analysis of the system based on a higher-level specification in the hierarchy is valid and could have been performed on the lowest-level system specification.

To completely couple the analysis with the system, the lowest-level specification of the system should be the actual program executed by the hardware, while the highest-level specification should reflect the intended overall function performed by the fault-tolerant system. The higher-level specifications represent, in effect, *system requirements*, stating properties to be possessed without defining method of attainment. As one moves down the hierarchy, each lower-level specification successively introduces additional mechanism in the design specification to achieve the fault-tolerance and expresses a more detailed and operational view of system transformation.

The fundamental idea of the paradigm is to gradually introduce aspects and complexities of design and algorithm as one moves down the hierarchy. Between successive specification levels one can perform *incremental design verification*, proving that the more detailed design specification at the lower level supports the abstracted view at the higher level. By gradually introducing the algorithms used to achieve fault-tolerance, one can verify each aspect of the design at the highest level of abstraction containing the necessary concepts. This produces a tractable specification and verification exercise and serves to highlight and isolate design decisions and fault-tolerance paradigms.

It is this paradigm that will be discussed first in the context of the SIFT experience and then illustrated later in the paper by a small, almost trivial, example.

2 An Outline of the Design of SIFT

The SIFT aircraft control computer system is designed to achieve high reliability from standard computers by replication of the hardware and adaptive majority voting. The use of majority voting, rather than a hot standby, is necessary to avoid even minor perturbations to high performance real-time tasks during error recovery. In contrast to other majority-voted systems, for instance FTMP[2], in SIFT the voting mechanism that detects and masks hardware faults, is implemented entirely in software. This allows the construction of SIFT from conventional computer components and allows greater flexibility. Hardware detected to be faulty is reconfigured out of the system, again by software, with its workload being transferred to other processors. Thus several successive faults can be survived if there is sufficient time between them to permit the reconfiguration.

SIFT is a *fully distributed* system with no global, shared components. The system is constructed from up to eight identical computer units, each containing a Bendix BDX930 processor, a 32K main store, a broadcast interface, and a 1553 interface, as shown in Figure 2. Each processor has its own 32k word main store and internal clock, neither of which can be accessed by any other processor. The 1553 interface provides a serial bus connecting the processor to the various aircraft sensors and actuators. The mean time between failures of one of these units, containing processor, store, and interfaces, is something less than one thousand hours.

Processors communicate with each other through a broadcast interface. In SIFT there is, conceptually, a single instance of each logical task, but, for reliability, that task is actually replicated and executed on three or five processors. Each instance of the task, upon completion of the computation, invokes an executive function which broadcasts its results through the datafile to each processor. In each of the processors, the three (or more) versions of the task results are extracted from the data file by voting software, and the majority result is placed in the input buffer, from which it can be obtained by any task that needs to use the results.

The voting software notes any discrepancies among the values on which it votes. A task *errorreporter*, run periodically on every processor, generates a synopsis of the errors detected on that processor and broadcasts the synopsis. The *globalexecutive* task, which is replicated like other critical tasks, receives the error synopses broadcast from the various processors and decides from them which processors are faulty. The global executive is responsible for the reconfiguration of the system, generating the configuration of processors to be used, excluding the processors deemed faulty and distributing the execution of application tasks appropriate to the current phase of the flight among the configured processors. In each processor, the results from the various replications of the global executive are voted and then used by the *localexecutive* task to select a task schedule for its scheduler, and to set up the sets of processors executing each task for use by the voting software. Note that, while the global executive task is a replicated and voted task common to the whole system, the error reporter and the local executive are tasks specific to each processor individually, and their results cannot be voted. Even though they are run on every processor, the results they generate relate only to their own processor.

The validity of the majority-voting approach depends on all task replications on *working* processors generating identical results, which in turn depends on these replications performing identical calculations on identical inputs. Provided that the system remains safe, majority voting of the results of replicated tasks suffices to ensure that all working processors obtain the same values for the results of those tasks. Where an input is obtained from an unreplicated source, no such assurance applies. Not only may the result obtained from an unreplicated source be erroneous, which the other tasks might accommodate, but the faulty source could broadcast different values to different processors. This would cause replicated tasks on those processors to obtain different results, destroying the utility of majority voting. In SIFT, a mechanism called *interactive consistency* [3] is used to ensure that all working processors obtain the same value for any input derived from an unreplicated source, whether that be an unreplicated application task, a sensor, or an error-reporting task.

3 An Outline of the SIFT Design Hierarchy

Figure 3 shows an outline of the various specifications and analyses that are used in the justification of the reliability of SIFT. On the right of the figure is a hierarchy of specifications of the correct functional behavior of SIFT, while on the left are a set of analyses that yield the probability of that correct behavior. The models at the bottom of the figure describe the hardware of SIFT, upon which the more abstract analysis is based.

The Hardware Fault and Error Rate Models describe the nature of possible hardware faults and assign error rates to aggregate failure modes. The Reliability Analysis then provides a Markov model of safe and unsafe system states and the probability of staying within safe states through the process of fault detection and reconfiguration.

The IO Specification, the most abstract functional description of the system, asserts that, *in a safe configuration*, the result of a task computation will be the effect of applying its designated mathematical function to the results of its designated set of input tasks, and that this result will be obtained within a real-time constraint. Each task of the system is defined to be correctly performed, with no indication of how this is attained. No *mechanism* used to achieve this is specified; the model includes only the assertion that each conceptual task will be correctly performed. The model has no concept of processor (thus no representation of replication of tasks or voting on results), and, of course, no representation of asynchrony among processors. The specification of this model contains only eight axioms and is intended to be understandable by an informed aircraft flight control engineer. This specification, together with the Reliabililty Analysis (and faith in the overall methodology), are intended to provide enough information to show that the system is capable of satisfying the project requirements.

The Replication Specification elaborates upon the IO Specification by introducing the concept of processor, and can therefore describe the replication of tasks and their allocation to processors, voting on the results of these replicated tasks and reconfiguring to accommodate faulty processors. The specification defines the results of a task instance on a *working* processor based on voted inputs, without defining any schedule of execution or processor communication. A distinction is made between the value (or set of values, for a fault processor) computed by a task instance, the value received by each other processor, and the value determined by a majority vote on each processor. This model is expressed in terms of a global system state and sychronous system time. The specification does not include lower level concepts of resource requirements, schedules, or broadcast buffer communication.

At this level, sufficient design detail about replicated task computation on multiple processors and voted input values is present to prove that majority voting suffices to mask errors (provided enough working processors are included in the configuration).

The Activity Specification develops the design into a fully distributed system in which each processor has access only to local information. Each processor has a local clock, a broadcast communication interface and buffers. The asynchrony among processors and its effect upon communication is modeled. The specification explicitly defines each processor's independent information about the configuration and the appropriate schedule of activities. The schedule of activities defines the sequence of task executions and votes necessary to generate task results within the required computation window. The Activity Specification is the lowest level description of the complete multiprocessor SIFT *system*.

In proving that this level design specification supports the Replication level, one must prove that the asynchronous distributed system remains consistent, even in the presence of errors. This involves showing that, despite the purely local information used by each processor to determine current time, current schedule and current configuration, all working processor will reach the same conclusions at roughly the same time. One must show that enough processors remain in approximate synchronization and that no effective use is made of the slight clock skew.

This is a key design decision in SIFT: distributed computation is used to achieve only fault-tolerance, *not* an increase in performance. All task instances execute at roughly the same time, and all instances are intended to be completed prior to any possible use of the results. The synchronous, global design abstraction of the system can thus be validated.

The PrePost Specification consists of specifications for the operating system for a single processor. The specification, in terms of pre-condition/post-condition pairs, facilitates the use of sequential program techniques to prove properties of the Pascal-based operating system as a sequential program. These specifications are very close to the Pascal programs, and essentially require the programs to "do what they do".

The various programs that form the SIFT executive are written in Pascal and form the *Pascal Implementation*, from which is derived by compilation the *BDX930 Implementation*. This is the lowest level specification of the SIFT software.

Each of the design specifications is defined by a set of axioms, written in a strongly typed variant of first-order (quantified) predicate calculus. The specification language is part of the STP specification and verification system [4] developed at SRI as part of the SIFT effort.

The hierarchy of specifications, from the I/O level down to the Pascal level, has been mechanically verified for the major system functionality. The proof demonstrates that a SIFT system in a "safe" state operates correctly despite the presence of arbitrary faults. Not yet completed is the proof that the SIFT executive performs an appropriate, safe, and timely reconfiguration in the presence of faults. A more extensive discussion of the SIFT hierarchy and design proof can be found in [5].

In the following section, a simple example of a two level specification and design proof illustrates the basic concepts of the methodology.

4 A Simple Example of Design Proof

Consider a simple, almost trivial, example of a system implementing fault masking through majority voting. We assume some set of tasks to be performed, each with an associated mathematical function it should perform. For a task k, we use function(k) to denote the associated mathematical function. For simplicity, we assume only constant functions of no arguments. The result of applying the function is thus function(k)().

Let's assume that the system is to guarantee reliable computation of each task by replicating the task and executing it on a number of processors. We wish to say that, provided enough of the processors are actually working, the majority of values produced by each executing processor will in fact be the correct value. In this case, task replication is sufficient to mask faults in a minority of processors. For simplicity of illustration, let us assume that each task is executed by each processor. In our simple example, we will also assume a single iteration of each task, with its results placed in dedicated memory locations for later inspection.

As a top-level design specification, we introduce a function produce(k), which tells us,for a task k, what value was conceptually produced. The predicate task-safe(k) will indicate that task k is "safe". In terms of these functions and predicates, we can now give our specification, comprised of a single axiom.

Axiom L1.A1

L1.A1 $\quad \text{task-safe}(k) \quad \supset \quad \text{produce}(k) = \text{function}(k)()$

The meaning of task-safe will not be given by this level specification. Like the system-safe predicate in the SIFT specification, a reliability analysis will assign a probability that task-safe(k), for each task k, will remain true for the appropriate period.

This completes our level 1 specification. Conceptually, this and the associated reliability analysis would be given to the flight control engineer. Having guaranteed that each task would produce the appropriate value, no further knowledge of how this is achieved would be theoretically required (discounting natural curiosity as to how the effect was achieved).

At the next level of specification, we introduce the concept of a set of processors, *procset*, and task computations on the processors. The function produce.on(k,p) denotes the value that processor p produces for task k. A predicate working(p) indicates that processor p is physically working during the relevant period. At this level, working(p) is left uninterpreted. Again, our specification consists of only one axiom:

Axiom L2.A1

L2.A1 $\quad \text{working}(p) \quad \supset \quad \text{produce.on}(k,p) = \text{function}(k)()$

This states that a working processor p will produce the correct value for task k.

We now wish to demonstrate that axiom L1.A1, expressing our high level design specification, can be proved from L2.A1. In doing so, we need to define each primitive function of the higher level specification, namely task.safe and produce, in terms of our lower level functions.* In order to exhibit these mappings, we must first define some auxiliary set abstractions.

We define:

$$workset = \{p \mid \mathrm{working}(p) \wedge p \in procset\}$$

$$prodset(k) = \{< v,p > \mid v = \mathrm{produce.on}(k,p)\}$$

$$workans(k) =$$

$$\{< v,p > \mid v = \mathrm{produce.on}(k,p) = \mathrm{function}(k)() \wedge p \in \mathrm{workset}\}$$

The set *workset* is the set of working processors, while *prodset*(k), pairs, for task k, each processor with its value produced for k. The set *workans*(k) is the subset of *prodset*(k) corresponding to right answers produced by working processors (after all, even malfunctioning processors *could* produce correct results).

Given these definitions, we can now define our mappings as follows:

Map.1 $\mathrm{task.safe}(k) \quad \equiv \quad 2 \cdot |workset| > |procset|$

Map.2 $\mathrm{task.safe}(k) \quad \supset \quad \mathrm{produce}(k) = majority(prodset(k))$

Map.1 defines our task.safe(k) as having at least half the processors working. Map.2 defines that, in the case where task.safe(k), the Level 1 produce(k) can be mapped as the majority of values produced by all processors. We assume a standard axiomatization for such a majority function, and for cardinality of sets, denoted | |.

Our basic argument in establishing that L1.A1 follows from L2.A1 and the mappings is as follows: We assume the antecedent of L2.A1 and show that the consequent must therefore be true. We know that every processor contributes a value to *prodset*, and that each working processor contributes the correct value to *prodset*. Our assumption that task-safe(k) tells us at least half the processors are working, and therefore that at least half the values in *prodset* will be the correct value. The majority value in prodset must therefore be the correct value and thus produce(k) must be the correct value. The above informal argument can be (and has been) easily formalized and mechanically verified.

*Technically, we will also use the identity mapping for the function(k) function present in both levels.

5 Conclusions

In this paper we have described a methodology to verify reliability and fault-tolerance properties of systems. We have advocated using a hierarchy of design specifications and incremental design verification. This task is neither easy, nor in all cases, practical. In the SIFT effort, the majority of the overall effort was spent producing the *simplest possible* design. It was only because of the simple, clean design that one could abstract succinct, *high level* views of the system and its function. Without this, the verification effort would not have been possible. Indeed, **a strong case can be made that, verification issues aside, the best chance of producing an ultra-reliable system design is to use the simplest, most straightforward techniques practical.** Complex error recovery and fault-tolerance mechanisms are likely to introduce as much unreliability as they remove.

The mechanical verification effort using the STP verification system was an expensive *tour de force.* The system is currently suitable for the use only of its designers. We are currently designing and implementing a specification and verification system, targeted for completion about mid-1984, that should be useable by enlightened engineers or computer scientists with at least an undergraduate background in mathematical logic. Of course, such verification exercises will continue to be expensive, appropriate only for applications with a very high reliability requirement.

One should also bear in mind that the application of mathematical design verification techniques *increases confidence* in "correctness" of a system design, but does not provide an iron clad guarantee that the system will perform as expected. Mathematical reasoning, even when aided by computer program, should be carefully reviewed and subjected to some form of social process. In addition, still left for inspection outside the formal system is (1) whether the highest-level specification reflects the users' requirements and expected performance, (2) whether the hardware properly supports the virtual machine assumed at the lowest-level of proof, and (3) whether the assumptions of the hardware fault model, and of the probabilistic Markov analysis are valid.

Acknowledgments

Also involved in the SIFT design effort were J. Wensley, J. Goldberg, K. Levitt, L. Lamport, M. Pease, M. Green, W. Kautz, R. Shostak, and C. Weinstock. The proof was carried out with the assistance of D. Hare. The STP verification system was developed in close collaboration with R. Shostak.

References

[1] J. Wensley, et al., "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control", *Proceedings of the IEEE,* 60(10):1240-1254, October 1978.

[2] A. Hopkins, T.B. Smith, J. Lala, "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft", *Proceedings of the IEEE,* 66(10): 1221-1239, Oct 1978.

[3] M. Pease, R. Shostak, L. Lamport, "Reaching Agreement in the Presence of Faults", *JACM,* 27(2):228-234, April 1980.

[4] R. Shostak, R. Schwartz, P.M. Melliar-Smith, "STP: A Mechanized Logic for Specification and Verification", 6th Conference on Automated Deduction, Springer Verlag, June 1982.

[5] P.M. Melliar-Smith, R. Schwartz, "Formal Specification and Mechanical Verification of SIFT: A Fault-Tolerant Flight Control System", *IEEE Transactions on Computers,* Vol. C-31, No. 7, July 1982.

CHAPTER 5

A SIMPLE EXAMPLE OF DESIGN VERIFICATION

Formal Verification of the Simple SIFT Voting System

```
/*
The set of right answers will be at least as large as the set of
results from working processors
*/

LEM1: formula
        CARD(WORKANS(K)) ≤ CARD(RIGHTANS(K))

prove LEM1
using CARD.SUBSET [S1 ← WORKANS(K),
                   S2 ← RIGHTANS(K)]
      SUBSET [S1 ← WORKANS(K),
              S2 ← RIGHTANS(K),
              X ← *X:2]
      WORKABS [I_P ← *X:2]

var SP: SET.OF(PAIR.OF(TYPE1, TYPE2))

var XP: TYPE2

APPLY1: (FUNS, TYPE2, TYPE1) TYPE1

var PAIR12: PAIR.OF(TYPE1, TYPE2)

var SQ: SET.OF(TYPE2)

var F: FUNS

/*  card(sq) = card( { < f(xp),xp > | xp ∈ sq } )  */

PAIRCARDEQUALITY: formula
        (∀ PAIR12:
            PAIR12 ∈ SP
                 ≡
            (∃ XP: PAIR12 = MAKE.PAIR(APPLY1(F, XP, X1), XP) ∧ XP ∈ SQ))
             ⊃
        CARD(SP) = CARD(SQ)

/*
 We introduce a functional value, prodfun, such that
prodfun(k)(p) = produce.on( k,p )
*/

PRODFUN: (TASKS) FUNS

PRODAX: axiom
        APPLY1(PRODFUN(K), P, 2) = PRODUCE.ON(K, P)

/*
 Lemmas 6 and 7 establish the two directions of the equivalence forming
the antecedent of Paircardequality, where prodfun is used for f
*/

LEM6: formula
        I_P ∈ WORKANS(K)
             ⊃
        (∃ P: I_P = MAKE.PAIR(APPLY1(PRODFUN(K), P, 2), P) ∧ P ∈ WORKSET())

prove LEM6 [P ← SOURCE(I_P)]
```

```
using WORKABS
      RIGHTABS
      PRODABS
      PAIR.EQUALITY [PAIR ← I_P,
                     PAIR1 ← MAKE.PAIR(PRODUCE.ON(K, SOURCE(I_P)), SOURCE(I_P))]
      PAIR.AXIOM.2 [X1 ← PRODUCE.ON(K, SOURCE(I_P)),
                    X2 ← SOURCE(I_P)]
      PRODAX [P ← SOURCE(I_P)]

LEM7: formula
        (∃ P: I_P = MAKE.PAIR(APPLY1(PRODFUN(K), P, 2), P) ∧ P ∈ WORKSET())
             ⊃
        I_P ∈ WORKANS(K)

prove LEM7
using PRODAX [P ← *P:C]
      WORKP [P ← *P:C]
      L2.A1 [P ← *P:C]
      PAIR.AXIOM.2 [X1 ← PRODUCE.ON(K, *P:C),
                    X2 ← *P:C]
      PRODABS [I_P ← MAKE.PAIR(PRODUCE.ON(K, *P:C), *P:C)]
      RIGHTABS [I_P ← MAKE.PAIR(PRODUCE.ON(K, *P:C), *P:C)]
      WORKABS [I_P ← MAKE.PAIR(PRODUCE.ON(K, *P:C), *P:C)]

/*
 Using lemmas 6 and 7 in Paircardequality, we prove that the number
of correct answers from working processors is equal the number of working
processors
*/

LEM3: formula
        CARD(WORKSET()) = CARD(WORKANS(K))

prove LEM3
using PAIRCARDEQUALITY [SP ← WORKANS(K),
                        SQ ← WORKSET(),
                        X1 ← 2,
                        XP ← *P:2,
                        F ← PRODFUN(K)]
      LEM6 [I_P ← *PAIR12:1]
      LEM7 [I_P ← *PAIR12:1,
            P ← *XP:1]

/*  Similarly lemmas 8 and 9 will be used in Paircardequality to prove
lemma 4
*/

LEM8: formula
        I_P ∈ PRODSET(K)  ⊃  (∃ P: I_P = MAKE.PAIR(APPLY1(PRODFUN(K), P, 2), P))

prove LEM8 [P ← SOURCE(I_P)]
using PRODABS
      PAIR.EQUALITY [PAIR ← I_P,
                     PAIR1 ← MAKE.PAIR(PRODUCE.ON(K, SOURCE(I_P)), SOURCE(I_P))]
      PAIR.AXIOM.2 [X1 ← PRODUCE.ON(K, SOURCE(I_P)),
                    X2 ← SOURCE(I_P)]
      PRODAX [P ← SOURCE(I_P)]

LEM9: formula
```

```
        (∃ P: I_P = MAKE.PAIR(APPLY1(PRODFUN(K), P, 2), P))  ⊃  I_P ∈ PRODSET(K)

prove LEM9
using PRODAX [P ← *P:C]
      PAIR.AXIOM.2 [X1 ← PRODUCE.ON(K, *P:C),
                    X2 ← *P:C]
      PRODABS [I_P ← MAKE.PAIR(PRODUCE.ON(K, *P:C), *P:C)]

/*  The number of produced values is equal to the number of processors */

LEM4: formula
        CARD(PROCSET()) = CARD(PRODSET(K))

prove LEM4
using PAIRCARDEQUALITY [SP ← PRODSET(K),
                        SQ ← PROCSET(),
                        X1 ← 2,
                        XP ← *P:2,
                        F ← PRODFUN(K)]
      LEM8 [I_P ← *PAIR12:1]
      LEM9 [I_P ← *PAIR12:1,
            P ← *XP:1]
      PROCABS [P ← *P:2]

/*  We now prove the main axiom of Level 1 as a theorem of Level 2  */

prove L1.A1
using MAJ.1 [M.BAG.1 ← RIGHTANS(K),
             M.BAG ← PRODSET(K),
             T1.V ← APPLY(FUNCTION(K))]
      RIGHTABS [I_P ← *V1.V2:1]
      MAPPING1
      MAPPING2
      LEM1
      LEM3
      LEM4

/*
 We now derive the Paircardequality lemma from a fundamental axiom
of set theory
*/

var ST1: SET.OF(TYPE1)

var ST2: SET.OF(TYPE2)

var T2.V1: TYPE2

var T2.V2: TYPE2

var T2.V3: TYPE2

var T2.V4: TYPE2

var T2.V5: TYPE2

var T1.V1: TYPE1

var T1.V2: TYPE1
var T1.V3: TYPE1

var T1.V4: TYPE1

/*
 The cardinality of two sets are equal if and only if there exists
a bijective mapping between the two sets
*/

CARDEQUALITY: axiom
        CARD(ST1) = CARD(ST2)
             ≡
        (∃ F, T2.V4, T1.V4:
            (∀ T2.V1, T2.V2:
                T2.V1 ∈ ST2
                ∧ T2.V2 ∈ ST2
                ∧ ¬(APPLY1(F, T2.V1, T1.V2) = APPLY1(F, T2.V2, T1.V2))
                     ⊃
                ¬(T2.V1 = T2.V2))
            ∧ (∀ T1.V1, T2.V5:
                  T1.V1 ∈ ST1
                       ⊃
                  (∃ T2.V3: T2.V3 ∈ ST2 ∧ APPLY1(F, T2.V3, T1.V3) = T1.V1)))

/*
 Because this axiom expresses a second order axiom scheme, the type
system of our current STP is strained.  Spurious variables had to be introduced
in order to coerce the types.  Future system will not suffer from this.
*/

/*  Genpair is our bijective mapping function  */

GENPAIR: FUNS

GENPAIRAX: axiom
        APPLY1(GENPAIR(), XP, PAIR12) = MAKE.PAIR(APPLY1(F, XP, X1), XP)

AWFULTYPEHACK: axiom
        XP = XP ∧ PAIR12 = PAIR12 ∧ X1 = X1

prove PAIRCARDEQUALITY [PAIR12 ← *T1.V1:1,
                        XP ← *XP:C]
using CARDEQUALITY [ST1 ← SP,
                    ST2 ← SQ,
                    F ← GENPAIR(),
                    T2.V3 ← *XP:C,
                    T2.V4 ← XP:6,
                    T2.V1 ← XP:6,
                    T2.V2 ← XP:6,
                    T2.V5 ← XP:6,
                    T1.V1 ← PAIR12:6,
                    T1.V2 ← PAIR12:6,
                    T1.V3 ← PAIR12:6,
                    T1.V4 ← PAIR12:6]
      GENPAIRAX [F ← F:C,
                 XP ← *T2.V1:1,
                 PAIR12 ← PAIR12:6]
      GENPAIRAX [F ← F:C,
           XP ← *T2.V2:1,
           PAIR12 ← PAIR12:6]
GENPAIRAX [XP ← *XP:C,
           F ← F:C,
           PAIR12 ← PAIR12:6]
PAIR.EQUALITY [PAIR ← MAKE.PAIR(APPLY1(F:C, *T2.V1:1, *X1:C), *T2.V1:1),
               PAIR1 ← MAKE.PAIR(APPLY1(F:C, *T2.V2:1, *X1:C), *T2.V2:1)]
AWFULTYPEHACK [XP ← XP:6,
               X1 ← X1:6,
               PAIR12 ← PAIR12:6]
PAIR.AXIOM.2 [X1 ← APPLY1(F:C, *T2.V1:1, *X1:C),
              X2 ← *T2.V1:1]
PAIR.AXIOM.2 [X1 ← APPLY1(F:C, *T2.V2:1, *X1:C),
              X2 ← *T2.V2:1]
```

CHAPTER 6

DESIGN VERIFICATION OF SIFT – OVERVIEW

The Hierarchical Specification and Mechanical Verification of the SIFT Design

This section describes the formal specification and proof methodology employed to demonstrate that the SIFT computer system meets its requirements. The hierarchy of design specifications is shown, from very abstract descriptions of system function down to the implementation. The most abstract design specifications are simple and easy to understand, almost all details of the realization having been abstracted out, and can be used to ensure that the system functions reliably and as intended. A succession of lower-level specifications refines these specifications into more detailed, and more complex, views of the system design, culminating in the Pascal implementation. The section describes the rigorous mechanical proof that the abstract specifications are satisfied by the actual implementation.

1. The Role of Formal Proof

The extreme reliability requirement on SIFT imposes a very severe problem in substantiating the achievement of that level of reliability. At the required reliability, a mere observation, even of a large number of systems, will be ineffective. Further, a SIFT system must be able to recover successfully from several million faults for every allowable system failure, and must, therefore, be able to recover from quite improbable and unforeseen faults and even combinations of faults. Thus validation by fault injection, while necessary, is unlikely to convince us that SIFT meets its reliability requirements.

The justification that SIFT meets the reliability requirement must be based on an extrapolation from fault rates that are easier to measure, such as those for an individual processor. For SIFT, this extrapolation takes the form of a discrete Markov analysis, with the numbers of working and faulty processors defining the states and the fault and reconfiguration rates defining the transitions. The validity of this extrapolation depends on a number of assumptions, and, at the desired level of reliability, even "minor" violations of the assumptions can have significant effects on the reliability achieved. Thus the assumptions must, themselves, be quite rigorously substantiated if the claimed reliability is to be believed. For instance, one important assumption of the Markov analysis is that the occurrence of faults is well described by a Poisson model with complete independence between processors. Much of the electronic and mechanical design of SIFT is intended to maintain this independence.

The validity of the Markov analysis depends also on the assumption that the states and the transitions of the Markov model correspond accurately to the actual system, and that the states in which system failure is possible are correctly identified. But this correspondence is far from obvious, for the actual system has very many states with many complex transitions between them, and the correspondence must be maintained even when one or more of the processors has suffered a fault. In SIFT, this correspondence is based on a predicate **system safe** indicating that the replication of each of the tasks is sufficient so that the voting can mask the effects of the faults present in the system. The validation of SIFT now consists of two parts. The first of these is a demonstration that, so long as **system safe** is true, the system performs the desired flight-control function, even though one or more processors may be faulty. This is a correctness property for the function performed by the system. The second is a demonstration that the Markov analysis computes an upper bound on the probability that **system safe** becomes false. This is a correctness property for the probabilistic reliability analysis of the system. Because even a very small defect in the demonstrations could allow failures at an unacceptable rate, these demonstrations must be performed with the rigor of mathematical proof. In this paper we consider only the first of these parts. An outline of the probabilistic reliability analysis is given in Wensley [1].

The necessity for formal mathematical proof to ensure that SIFT meets the desired functional and reliability requirements presents two major issues:

- How does one define the criteria sufficient to ensure the correct functioning of the system?

- How does one prove that the criteria are satisfied by the actual system?

The first issue is crucial if the formal verification effort is to have any practical significance. One must have confidence, even as a noncomputer scientist, that the formal specifications stating what is meant by the correct functioning of the system in fact reflect the *intended behavior*. That a formal specification expresses what the system designer intuitively means must, in the end, be determined by inspection. A formal specification must therefore be *believable* if rigorous mathematical correspondence to the specification is to ensure the desired effect. The larger and more complex the system, the more acute the problem becomes. Specifications reflecting the detailed behavior of the system allow the most straightforward formal verification effort, but it is difficult to ensure that low-level specifications embody what is meant by the proper functioning of the system. Very high-level specifications, abstracting from the details of the system, are necessary if we are to state the overall functional and fault-tolerance properties of the system in a way that can be understood and believed. The problem then becomes one of reconciling the very high-level specifications with the detailed transformations performed by the programs of the actual system.

In order to state high-level system specifications that can be shown to be consistent with the actual program, one must formulate not just a single specification of the system, but a *hierarchy* of specifications. Our approach is to state a tiered set of system specifications, as illustrated in Figure IV-11.

Each level L_i in the hierarchy specifies an abstract view of the system in terms of a set of primitive predicates P_i and functions F_i. The specification for the model is given by a set of axioms, characterizing those properties of the model appropriate for that level of system abstraction. At each level in the hierarchy, a specification L_i can be seen as an *abstraction* of the previous level L_{i+1}. Correspondence between successive levels is done by expressing each primitive function and predicate of higher-level L_i in terms of the functions and predicates of the lower-level L_{i+1} . With this mapping, one must then prove that each property derivable from the higher-level specification can be proved from the lower-level specification. The mapping between levels need not be complete; the mapping itself may be given as a set of axioms, saying only enough about the correspondence to derive the necessary axioms of the higher level as theorems from the axioms of the lower-level. It is required only that the mapping axioms be *consistent,* i.e., that there exist a complete functional mapping between levels that satisifies the mapping axioms. By demonstrating the correspondence between successive levels L_i and L_{i+1}, one can conclude by induction that any property provable from the highest-level specification is also provable from the lowest-level specification. Thus, any analysis of the system based on a higher level specification in the hierarchy is valid and could have been performed on the lowest-level system specification.

Within the hierarchy, the lowest-level specification of the system is the actual SIFT system executed by the hardware, while the highest-level specification reflects the intended overall function performed by the fault-tolerant system. The higher-level specifications represent, in effect, *system requirements,* stating properties to be possessed without defining method of attainment. As one moves down the hierarchy, each lower-level specification successively introduces additional mechanism in the design specification to achieve the fault-tolerance and expresses a more detailed and operational view of system transformation. Between successive specification levels, one can perform *incremental design verification,* proving that the more detailed design specification at the lower level supports the abstracted view at the higher level. By gradually introducing the algorithms used to achieve fault-tolerance, one can verify each aspect of the design at the highest level of abstraction containing the necessary concepts.

As an example, one can prove that replication and majority voting serve to mask faults, using a specification of the system as a single (and therefore synchronous) global object. Having proven this paradigm with respect to that specification level, one can then define a lower-level specification of the system as a distributed asynchronous system with a broadcast communication interface. It is then required to exhibit a mapping from the distributed system view to the global system view at the higher specification level. Demonstration that each axiom of the global-state specification is provable from the axioms defining the distributed-state specification will ensure that any theorems about fault masking in the global-system view are valid for the distributed-system view as well. Thus the paradigm of fault masking through task replication is introduced and validated prior to introducing techniques for fault isolation through distribution of resources.

2. Mechanized Specification and Verification

Attempting to formally characterize and justify the design of any real system is complex and tedious. Without mechanical aids for constructing formal specifications and rigorously enforcing sound proofs, this task would be completely impractical and would not produce a credible result. Our early experience in formulating formal "paper" specifications and giving informal mathematical arguments of correctness was fraught with specification ambiguity and oversights in the informal correctness proofs. In response to this, and our desire to mechanize the style of specification and verification employed in our previous "paper" attempts, a new mechanical verification system was designed and implemented.

STP [8] is an implemented system supporting specification and verification of theories. As implemented, STP did not contain a parser for SPECIAL, and thus, for this verification the specifications were expressed in the LISP-like internal representation of STP. The logic of STP is an extension of a multisorted (strongly-typed) first-order logic. The logic includes type parameterization and type hierarchies. STP support includes syntactic type checking and proof components as part of an interactive environment for developing and managing theories in the logic. At the core of the system is a fast, complete decision procedure [9] for a quantifier-free theory of (Presburger-like) arithmetic. The user of the system can introduce new types and function symbols, with the semantics specified through a set of first-order axioms. By providing aid to the theorem prover in the form of selection of appropriate instances of axioms and lemmas, the user raises the level of competence of the prover to the full first-order theory specified. A fundamental

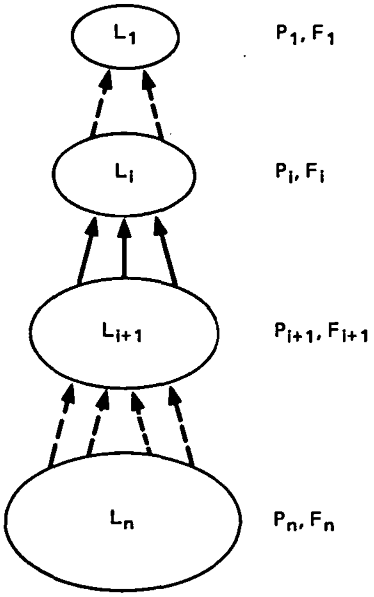


FIGURE IV-11 A HIERARCHY OF SPECIFICATIONS

characteristic of the system is that the user need know no details of the theorem prover itself; the system forms a complete mechanization of a simply-characterized theory. As a result of a successful proof attempt using STP, one obtains the sequence of axioms and intermediate lemmas, together with their necessary instantiations, which lead to the theorem. The system automatically keeps track of which formulas have been proved and which have not, so that the user is not forced to prove lemmas in advance of use. The system also monitors the incremental introduction and modification of specifications to monitor soundness.

3. An Outline of the Specification Hierarchy

Figure IV-12 shows an outline of the various specifications and analyses that are used in the justification of the reliability of SIFT. Before the individual specifications are described in detail, we give a description of their intent and interaction. On the right of the figure is a hierarchy of specifications of the correct functional behavior of SIFT, while on the left is a set of analyses that yield the probability of that correct behavior. The models at the bottom of the figure describe the hardware of SIFT, upon which the more abstract analysis is based.

The IO Specification, the most abstract functional description of the system, asserts that, *in a safe configuration*, the result of a task computation will be the effect of applying its designated mathematical function to the results of its designated set of input tasks, and that this result will be obtained within a real-time constraint. Each task of the system is defined to have been performed correctly, with no specification of how this is achieved. The model has no concept of processor (thus no representation of replication of tasks or voting on results), and of course no representation of asynchrony among processors. The specification of this model contains only 8 axioms and is intended to be understandable by an informed aircraft flight control-engineer.

The Replication Specification elaborates upon the IO Specification by introducing the concept of processor, and can therefore describe the replication of tasks and their allocation to processors, voting on the results of these replicated tasks, and reconfiguring to accommodate faulty processors. The specification defines the results of a task instance on a *working* processor based on voted inputs, without defining any schedule of execution or processor communication. This model is expressed in terms of a global system state and system time.

The Activity Specification develops the design into a fully distributed system in which each processor has access only to local information. Each processor has a local clock and a broadcast communication interface and buffers. The asynchrony among processors and its effect upon communication is modeled. The specification explicitly defines each processor's independent information about the configuration and the appropriate schedule of activities. The schedule of activities defines the sequence of task executions and votes necessary to generate task results within the required computation window. The Activity Specification is the lowest level description of the complete multiprocessor SIFT *system.*

The PrePost Specification consists of specifications for the operating system for a single processor. The specification, in terms of pre-condition/post-condition pairs, facilitates the use of sequential proof techniques to prove properties of the Pascal-based operating system as a sequential program. These specifications are very close to the Pascal programs, and essentially require the programs to "do what they do".

The various programs that form the SIFT executive are written in Pascal and form the *Pascal Implementation,* from which is derived by compilation the *BDX930 Implementation.* This is the lowest level specification of the SIFT software.

The functional behavior described by the *I/O Model* is assured only so long as the predicate **system safe** remains true. The analyses shown on the left of Figure IV-12 provide the probability that **system safe** will remain true and hence that the desired functional behavior will continue.

In the remainder of the paper, we present details of the specifications comprising the SIFT design hierarchy. *Unless otherwise noted, all specifications and mappings are taken from actual system specifications and completed proofs.* For pedagogical purposes, we have used a syntactic transliteration of the actual form of the specifications. The STP system forced all user interaction to use a LISP-like prefix notation; we have transformed this into more common mathematical notation.

The mechanical proof of consistency between the various levels of specification and further details of its derivation are contained in Chapters 6 and 7.

4. Input/Output Specification

The Input/Output Specification of SIFT, the highest level specifying functional behavior, defines the input/output characteristics of tasks performed by SIFT. The specification defines the configuration of system tasks and expresses the flow of information between tasks. Based on an abstract notion of time, which may be interpreted as subframe time, we refer to iterations of a task taking place during various time intervals. The time interval

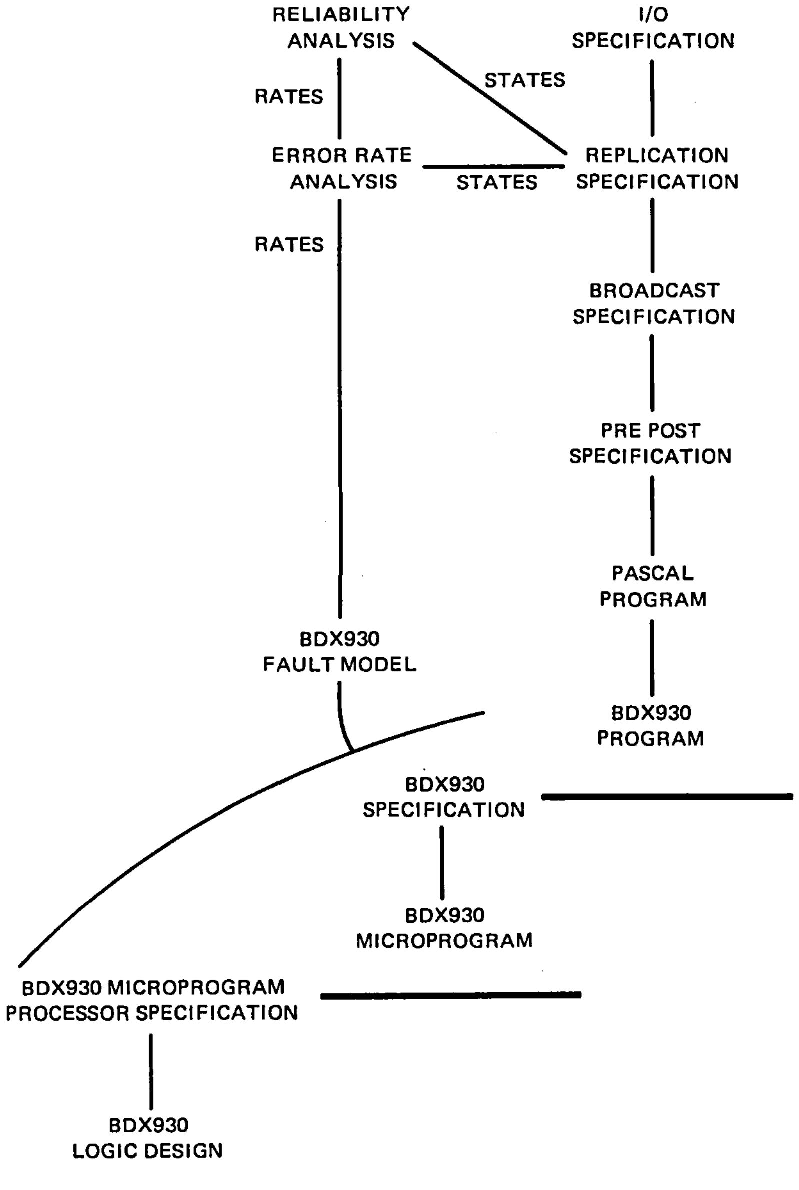


FIGURE IV-12 THE HIERARCHY OF SPECIFICATIONS AND ANALYSES USED TO SUBSTANTIATE THE RELIABILITY OF SIFT

for a particular iteration of a task is referred to as its *execution window*, having a begining time and an ending time. Each task is defined to use as inputs the values produced by its input tasks and produces one or more outputs during its execution window. Based on a high-level predicate specifying whether a task is *safe* during a particular iteration of a task, the specification defines that a task which is safe during an iteration will produce exactly one output value, computed as a function of its input values. Provided that the entire system is safe throughout some interval (i.e., that all tasks are safe for that interval), we can prove by induction that all tasks will compute correct functions of their intended inputs. This defines at a high level what it means for SIFT to function correctly.

Conspicuously absent from this model is any notion that a task is replicated and computed on a set of processors. At a lower level, we shall explain that the value the I/O specification defines as resulting from a given task iteration will actually be the outcome of a majority vote of processors assigned to compute the task. The *task safety* predicate taken as primitive in the I/O specification, specifying when a task can be relied upon to produce correct results, will be defined at a lower level to be a function of the amount of task replications and the number of working processors.

Briefly, the model is organized as follows. Each task a in $Tasks$ the set of all executive and application tasks, computes a (mathematical) function, denoted by **function** (a), of its input values. The function **apply**$(f, \bar{v})$ takes as parameters a functional value and an argument list and produces the result of applying the function to the argument list.[1] **Inputs**(a) denotes the set of tasks providing inputs to a. For task $b \in$ **Inputs** (a), the input to an iteration of a is provided by the most recently completed iteration of b prior to the execution window of that iteration of a. A derived function b **to** i **of** a denotes the iteration of b providing input to the i-th iteration of a. Because all tasks iterate once per frame, one can prove (as indeed we do) that b **to** i **of** a is equal to i or $i-1$, that is, that the input task is either "executed" in the same frame as the task or in the previous frame. During each iteration i of a task a, **Result**(a, i) denotes the set of output values which are produced. In order to map task iterations to subframe time, the function i **of** a is used to denote the time interval $[t_1, t_2]$ comprising the execution window of the i-th iteration of a. The functions **beg**(i **of** a) and **end**(i **of** a) are used to denote the begining and end of the execution window, respectively.

The overall structure of task configurations within the I/O model is illustrated in Figure IV-13. For a task such that the predicate **task** a **safe during** i is true, a will produce exactly one output value during its execution window. The output(s) of a task which is not

[1] This is in fact a trick to use a first-order encoding of functional value domains.

safe during its iteration is unspecified. Because the configuration of tasks is different for different phases of the flight, not all tasks necessarily compute each iteration. A predicate a **on during** i determines whether **Result**(a, i) is expected to compute a function of its inputs or to return a special $\perp$ element as its value.

Within the I/O specification, the interactive consistency algorithm is defined as a special form of task. For such a task a, satisfying the predicate **i/c**(a), its associated mathematical function **function**(a) is defined to be the identity function. Recall from our discussion in Section 4 the interactive consistency algorithm is used in order for multiple processors reading unreplicated (and possibly unstable) input to reach agreement on an input value. As we explain below, a safe interactive consistency task will *always* produce a single output value.

Based on these primitive functions and predicates, the I/O specification contains eight axioms, expressing constraints on when task iterations are to take place and that safe tasks compute functions of their designated inputs. We do not illustrate the entire set of axioms here. The axioms related to the scheduling of task iterations are straightforward. They express basic requirements that successive iterations of a task are properly ordered in time and that the execution window of a task b must precede the execution window of a task a to which it provides input.

The major axiom defining the Input/Output behavior of a task is the following:

$$\begin{aligned}
&a\ \textbf{on during}\ i \quad \land\\
&\textbf{task}\ a\ \textbf{safe during}\ i \quad \land\\
&\forall b \in \textbf{Inputs}(a)\\
&\qquad |\textbf{Result}(b, b\ \textbf{to}\ i\ \textbf{of}\ a)| = 1\\
&\ \supset\\
&\textbf{Result}(a, i) =\\
&\qquad \left\{ \textbf{apply}\left(\textbf{function}(a), \left\{ < v, t > \middle| \begin{array}{l} t \in \textbf{Inputs}(a) \quad \land \\ v \in \textbf{Result}(t, t\ \textbf{to}\ i\ \textbf{of}\ a) \end{array} \right\} \right) \right\}
\end{aligned}$$

This axiom defines that any iteration of a task a, such that (1) a is both **on** and **safe** and (2) each task b providing input to the i-th iteration of a returns exactly one output value during its corresponding iteration (the notation $|s|$ denotes the cardinality of set s), will return exactly one output during its iteration (i.e., that **Result** (a, i) will be a singleton set). The value produced will be that resulting from applying its designated function **function**(a) to the set of (tagged) values produced by its input tasks. The set of input values is specified as a set of pairs $< v, t >$, where, for each task t in the input set, v is the value in the (singleton) set **Result**$(t, t$ **to** i **of** $a)$. Thus, provided a is safe and its input is stable, it will correctly compute an output value. This is the main statement of functional correctness of the system that is demonstrated by the proof effort.

In the case of interactive consistency tasks, one additional axiom governs its input/-output characteristics:

$(\mathbf{i/c}(a) \quad \wedge \quad \mathbf{i/c\ task}\ a\ \mathbf{safe\ during}\ i) \quad \supset \quad |\mathbf{Result}(a,i)| = 1$

This defines that an interactive consistency task which is safe during its iteration will always produce a single value as output. By the previous axiom, if its input task is **safe** and thus provides a single output, the interactive consistency task will perform its associated function (in this case the identity function) on the input. Even if the input task is not **safe**, however, the current axiom defines that *some* single output value will be produced. This is the main correctness criterion for the interactive consistency algorithm. We did not carry out a mechanical proof of this axiom – a hand proof can be found in [6].

These are the major axioms of the I/O specification. In the next section, we present the next lower-level specification and show how the primitives and stated axioms of the I/O specification are supported at the next level.

5. The Replication Specification

The Replication Specification, at the next lower level, introduces the notion that tasks are replicated and executed by some number of processors. Based on a high-level concept of each processor communicating its results to all other processors, a specification of the majority voting performed by each processor is given. Also defined (but not proven) is the information flow through which error reports from individual processors are provided to the global executive. This information is used by the global executive in order to diagnose processor faults and remove, from the configuration, processors deemed to have solid faults.

The concept of task scheduling has been refined to define not only the execution window for task execution but also the set of processors assigned to execute the task. The function **poll for** i **of** a denotes the set of processors assigned to compute the i-th iteration of task a. The I/O model primitive predicate a **on during** i is derived within the Replication model as:

$a\ \mathbf{on\ during}\ i \quad \equiv \quad \exists p \in \mathbf{poll\ for}\ i\ \mathbf{of}\ a$

With the concept of processor computation occuring in the Replication model, the **task safe** predicate appearing as primitive within the I/O model can be derived within the Replication model in terms of working processors. The Replication model includes a variable S, which denotes the set of "safe" processors at any given time. $S^{[t_1,t_2]}$ denotes the set of processors safe during the interval $[t_1 ,t_2]$. At the Activity model level, we will define a processor being "safe" as a rather complex function of having correctly functioning hardware, being in the correct configuration, and having a clock within some skew of other processor clocks. Of course this set will not have an implementation counterpart, since the implementation will never have perfect information concerning the set of correctly functioning processors.

A derived concept at this level is that of a task iteration's *data window*.
The **DWindow for** b **to** i **of** a is defined to be the time interval
[**beg** (b **to** i **of** a) **of** b, **end** (i **of** a)].
Based on this function, we define **DWindow for** i **of** a to be the interval extending from the begining of the execution window of the earliest input task to a and extending to the end of the execution of i **of** a.

Using these concepts of data window and the set of working processors, we can now derive the **task safe** predicate of the I/O model as follows:
task a **safe during** i

$$\equiv$$

$$2 \times \left|\textbf{poll for } i \textbf{ of } a \cap \textbf{S}^{\textbf{DWindow for } i \textbf{ of } a}\right| > |\textbf{poll for } i \textbf{ of } a|$$
$$\vee \quad \sim a \textbf{ on during } i$$

The definition states that a task a is safe either if a majority of the processors assigned to compute the task are working for the data window of the task or if the task is not **on during** i. It is necessary that the processors are in the working set S for the entire data window of the task in order that we can be assured (in mapping to the next lower-level specification) that the processor will not corrupt its input data prior to its use. We omit discussion of the conditions necessary to define the safety of interactive consistency tasks.

With the concept that a processor computes an iteration of a task comes the function **Result**(a, i) **on** p which denotes the set of outputs produced by processor p for the i-th iteration of task a. In a manner left unspecified by this level, processor p communicates its results to all other system processors. The function **Result** (a, i) **on** p **in** q denotes the value that processor q has reportedly received from processor p for the i-th iteration of a. The relationship between **Result on** and **Result on in** is defined by the following axiom:

$$q \in \textbf{poll for } j \textbf{ of } b \bigcap \mathrm{S}^{\textbf{DWindow for } j \textbf{ of } b}$$
$$\supset$$
$$\textbf{Result}(b,j) \textbf{ on } q =$$
$$\left\{ v \;\middle|\; \begin{array}{l} \exists p \;\; p \in \mathrm{S}^{j \textbf{ of } b} \quad \wedge \\ \qquad v = \textbf{Result}(b,j) \textbf{ on } q \textbf{ in } p \end{array} \right\}$$

This defines that, for a processor q in the **poll** set that is **safe** for the **DWindow**, the **Result** set on q is equal to the set of values that processors **safe** for the execution window have reportedly received from q. More intuitively, this states that the output of a working processor in the poll is the set of values reportedly received by working processors.

The function $\textbf{Result}(a,i)$ **in** q is used to define the result of processor q voting on the output of the i-th iteration of a based on the results communicated to it.

The overall structure of the Replication model is illustrated in Figure IV-14. The task structure shown is a refinement of the task configuration illustrated in Figure IV-13.

As we shall show shortly, the I/O primitive $\textbf{Result}(a,i)$ for a safe task iteration will be derived as the value a majority of assigned processors obtained by their voting. All processors are required to report the results of each task computation to all processors, and all processors are required to vote on all received values. Rather than a task producing a set of output values as in the I/O model, in the Replication model, a task produces a set of *sequences* of values. This reflects the fact that conceptual values in the system actually consist of a sequence of "machine words". Processor voting is scheduled (as specified at the next level of specification) on a word by word basis. We define voting via the following axiom:

$$p \in \mathrm{S}^{j \textbf{ of } b} \quad \wedge$$
$$1 \leq y \leq \textbf{ result size}(b)$$
$$\supset$$
$$(\textbf{Result}(b,j) \textbf{ in } p)[y] =$$
$$\textbf{majority}\left(\left\{ < v,q > \;\middle|\; \begin{array}{l} q \in \textbf{poll for } j \textbf{ of } b \quad \wedge \\ v = (\textbf{Result}(b,j) \textbf{ on } q \textbf{ in } p)[y] \end{array} \right\}\right)$$

For a safe processor p, a vote on a defined value position y, the y-th element in $\textbf{Result}(b,j)$ **in** p is defined to be equal to the **majority** of first components in the set of value-processor pairs $< v,q >$, where q is in the **poll** set and v is the y-th component of the **result on in** value in processor p. This represents an encoding of majority value in the bag of all values in p reportedly received from processors in the **poll** set for task b.

The main execution axiom of the Replication Specification is now given as follows:

$$p \in \textbf{poll for } i \textbf{ of } a \bigcap \mathbf{S}^{\textbf{DWindow for } i \textbf{ of } a}$$

$$\supset$$

$$\textbf{Result}(a,i) \textbf{ on } p =$$

$$\left\{ \textbf{apply}\left(\textbf{function}(a), \left\{ < v,t > \middle| \begin{array}{l} t \in \textbf{Inputs}(a) \quad \wedge \\ v \in \textbf{Result}(t, t \textbf{ to } i \textbf{ of } a) \textbf{ in } p \end{array} \right\}\right)\right\}$$

This axiom, quite similar to its counterpart in the I/O model, defines that a working processor p in the **poll** set for the i-th iteration of task a, will compute the proper function of its locally-voted input values. Note that, unlike its I/O axiom counterpart, this is purely a local specification of the actions of a single, working processor operating on locally-computed information – still with respect to a synchronous system.

We are now in a position to define the mapping up to the I/O concept of **Result** (a,i). This is given by the following axiom:

$$\textbf{Result}(a,i) =$$

$$\left\{ v \middle| \begin{array}{l} \exists p \;\; p \in \mathbf{S}^{i \textbf{ of } a} \quad \wedge \\ \qquad v = \textbf{Result}(a,i) \textbf{ in } p \end{array} \right\}$$

This expresses the set **Result**(a,i) as consisting of the set of values that safe processors obtained as a result of voting.

We omit discussion of the other axioms of the Replication Specification. In order to show that the I/O Specification is a valid abstraction of the Replication Specification, we must prove that the I/O axioms follow as theorems from the Replication axioms and the mappings.

The proof of the main Execute axiom of the I/O Specification required that each safe processor voting be shown to obtain the same voted value, assuming from the antecedent of the I/O Execute axiom that the task is safe and that there is only one value of the **Result** of each input task. This implies that each safe processor applies the correct mathematical function to the same set of input values and thus every safe processor produces the same correct output value. But our I/O assumption of **task safe** asserts that a majority of the processors computing the task are safe; therefore, the majority of computed values must be the correct value.

The proof of the main I/O Execute axiom from the Replication axioms required approximately 22 proofs, with an average of 5 premises necessary per proof, and 106 instantiations of axioms and lemmas overall.

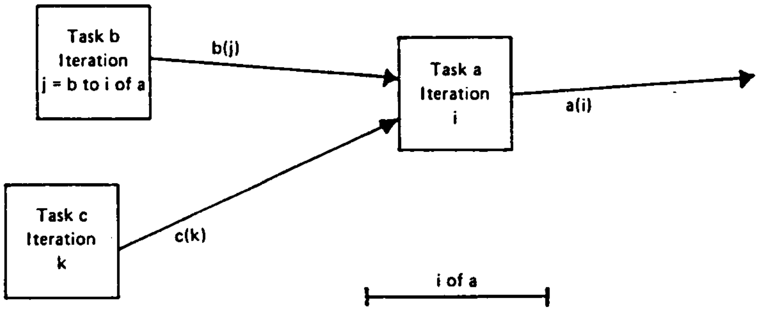


FIGURE IV–13 THREE TASKS IN THE I/O MODEL

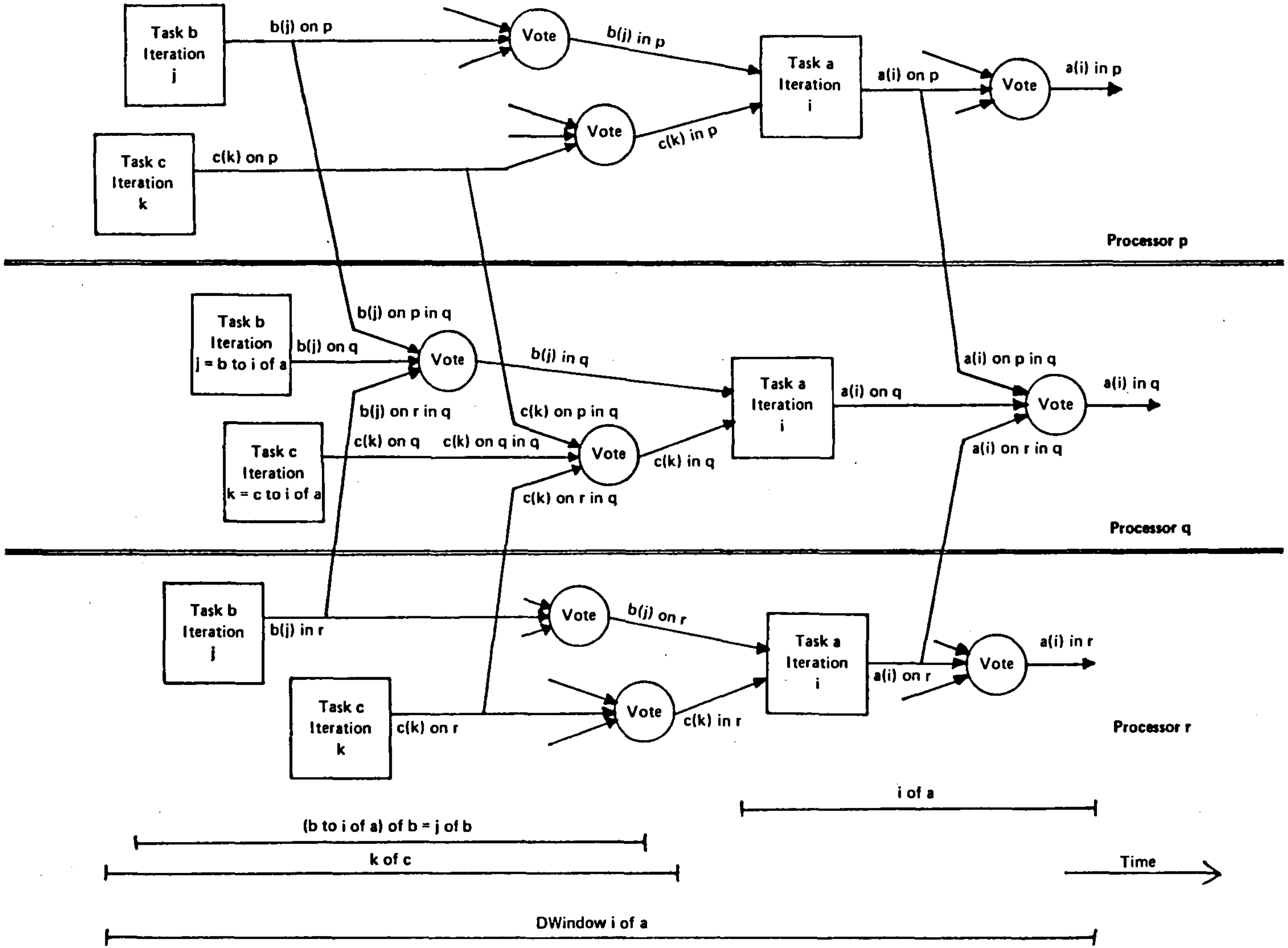


FIGURE IV–14 THREE TASKS IN THE REPLICATION MODEL

1 The Activity Specification

This level of specification defines a completely local view of the behavior of a single processor in the SIFT system. The *fully distributed* nature of the SIFT system is specified at this level: each processor has an independent concept of time, configuration, and schedule. Also at this level is a more explicit model of the activities and data structures carry out the transformations specified at the Replication level. Whereas the Replication level defines the executed and voted values for each execution window of a task, the Activity level defines a schedule of *execute* and *vote* activities to realize this within the execution window.

Within the Activity model is the first indication that the SIFT system is not synchronous; the subframes on the various processors start and finish at slightly different real times. Two functions, **start**(t,p) and **finish**(t,p) map subframe time on processor p to real times at which the subframe starts and finishes, as shown in Figure IV-15."Real-time" is represented in the specification as a discrete domain, which can be thought of as "clock ticks," to allow induction. A short **overhead** interval occurs between the finish of one subframe and the start of the next. Because of clock skew and transport delay within SIFT, the processors will not be exactly synchronized, but, for the system to function correctly, it is necessary that the clocks remain within a specified tolerance, **max skew** , of each other. This is the responsibility of the clock synchronization task, a part of each processor's Local Executive, using an algorithm whose proof is given in [10]. The required synchronization is expressed by:

$$\begin{aligned}&\textbf{clock safe}(p,t) \quad \wedge \quad \textbf{clock safe}(q,t)\\&\quad\supset\\&\textbf{finish}(t,p) + \textbf{broadcast delay} \leq \textbf{start}(t+1,q) \quad \vee\\&\textbf{finish}(t,q) + \textbf{broadcast delay} \leq \textbf{start}(t+1,p)\end{aligned}$$

As we discussed earlier, SIFT is carefully designed so that the distributed system is *effectively synchronous.* Within the limits given above, asynchronism caused by processor clock skew has no external effect. In the case of the broadcasting of the results of a task, for example, our specifications define the value at the destination only after the latest time at which the broadcast could have been completed, given the maximum processor skew. It is necessary to prove that no access to these data is attempted before that time, in order to map this asynchronous system up to the higher-level, synchronous Replication and I/O models.

The state of each processor is specified using two state-selector functions, corresponding to two data structures of the SIFT operating system: a data file connected via a broadcast interface to all system processors, and an input file into which voted values are placed and from which a task retrieves its input values. In the Activity specification, the function **datafile in** p **for** a **on** q **at** rt denotes the value in the datafile in processor p at real-time rt for the result of task a on processor q. The function **input in** p **for** a **at** rt denotes the value in the input file in processor p at real-time rt for the voted result of task a.

As we mentioned earlier, each processor has an independent opinion of the configuration it is expected to use in scheduling activities. At the start of a subframetime t, processor q uses as the appropriate configuration **config** (t, q) the value in a configuration subfield of **input in** q **for** GE() **at start**(t, q), where GE() denotes the replicated Global Executive task. For configuration c, the function **sched**(c, t, q) denotes the sequence of activities scheduled for subframetime t on processor q. An activity is either $< execute, a >$, specifying the execution of task a or $< vote, a, y >$ specifying a vote on element y of the output of task a. Figure IV-16illustrates the interaction between the data structures and scheduled activities.

The effect of an *execute* is specified by the following axiom:

$$
\begin{array}{l}
p, q \in \mathbf{W}^t \quad \wedge \\
< execute, a > \in \mathbf{sched}(\ \mathbf{config}(t, q), t, q) \\
\quad \supset \\
\mathbf{datafile\ in}\ p\ \mathbf{for}\ a\ \mathbf{on}\ q\ \mathbf{at}\ (\ \mathbf{finish}(t, q) + \ \mathbf{broadcast\ delay}()) = \\
\quad \mathbf{apply}\left(\mathbf{function}(a), \left\{ < v, b > \left| \begin{array}{l} b \in \mathbf{Inputs}(a) \quad \wedge \\ v = \mathbf{input\ in}\ q\ \mathbf{for}\ b\ \mathbf{at\ start}(t+1, q) \end{array} \right. \right\}\right)
\end{array}
$$

The set W^t denotes the set of *correctly functioning* (working) processors during subframetime t. The antecedent of the axiom defines that processors p and q are working during subframe t and that an execute activity for a is among the activities scheduled for processor q, according to its perceived configuration. The consequent specifies that the datafile in each working processor p for a on q at the finish of that subframe plus the broadcast delay, ***according to q's clock,*** is equal to the correct function applied to the set of input values present in the input file at the ***start of the next subframe.*** Several explanations are in order. The hardware broadcast interface connecting processor q's datafile to all processor datafiles is asynchronous and can be initiated at any time during the subframe, with respect to q's clock. In the event of an execute and a broadcast by processor q sometime during subframe t, the earliest moment at which the entry for a on q can be guaranteed is the finish of the subframe plus the maximum broadcast delay. Thus the value is only defined at this moment in time, and with respect to the broadcasting processor's clock. It was necesary to demonstrate that, with respect to

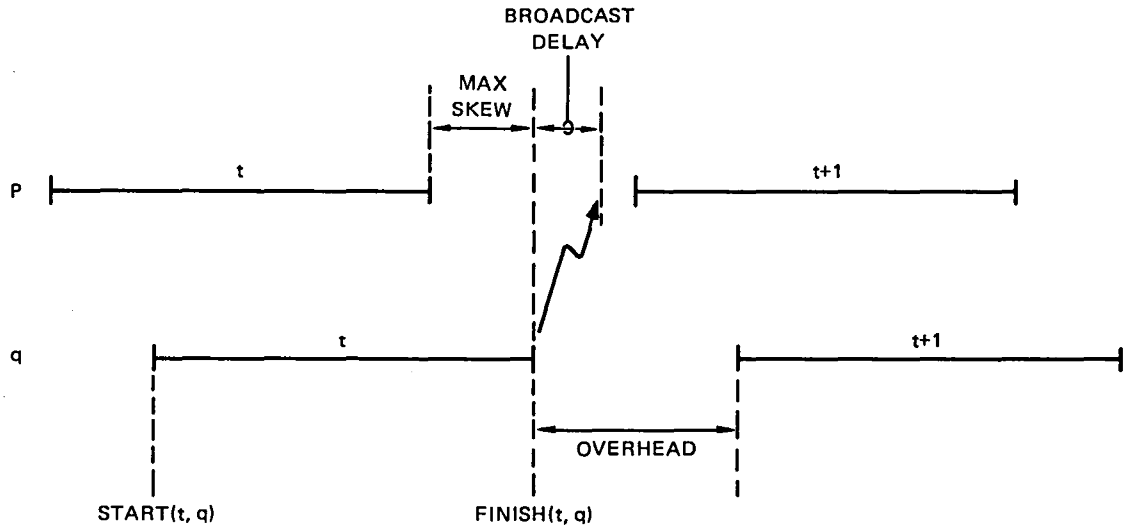


FIGURE IV-15 THE TIMING RELATIONSHIPS BETWEEN SUBFRAMES ON ASYNCHRONOUS PROCESSORS

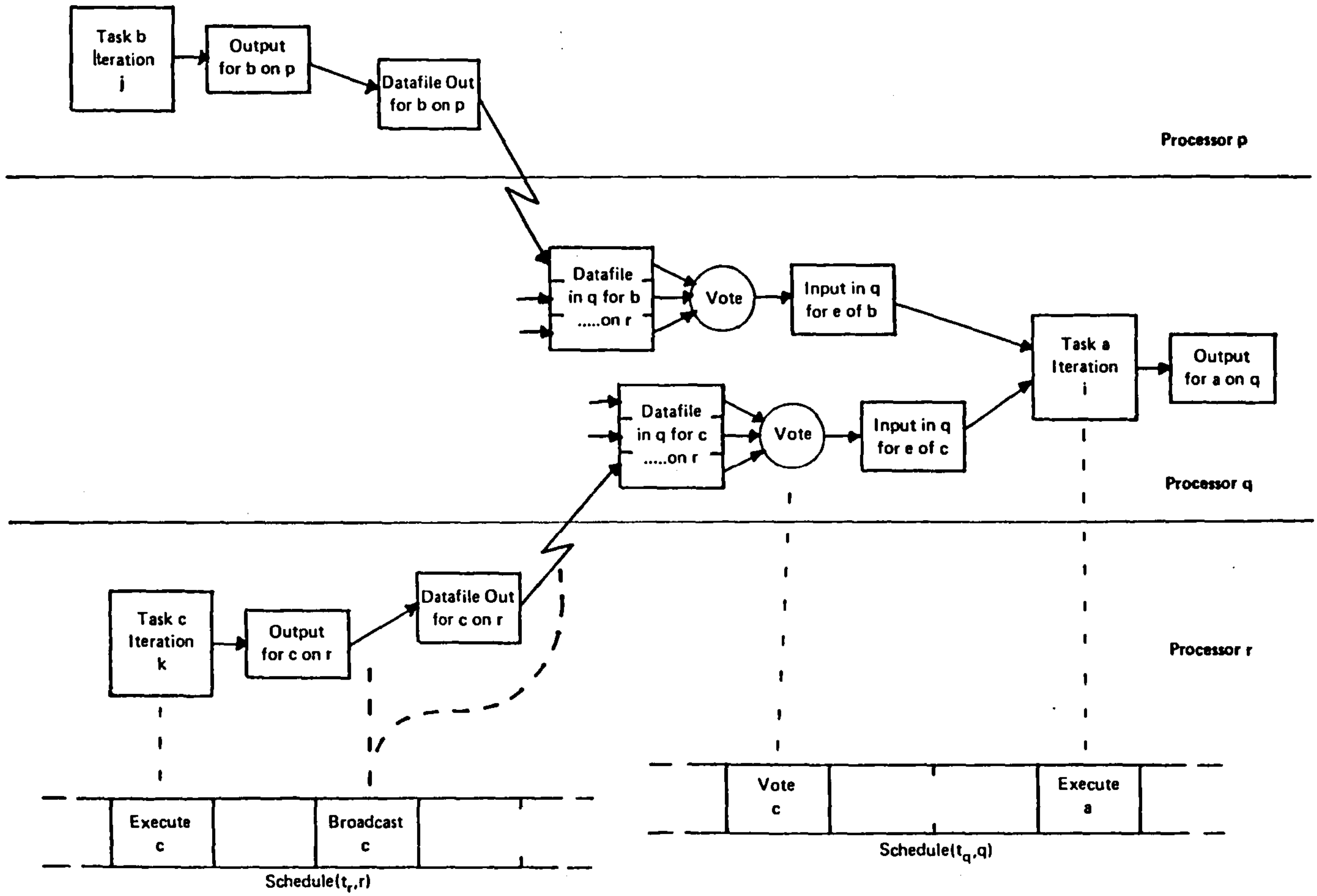


FIGURE IV–16 A PARTIAL VIEW OF THREE TASKS IN THE BROADCAST MODEL

receiving processor p's clock, the information is present by **start** $(t+1,p)$. Given the set of specified schedule constraints, it was shown that the information is present in all loosely synchronized processors prior to the first moment at which access can occur.

One might notice that an execute activity scheduled during subframe time t causes the datafile at the start of time $t+1$ to contain the result of applying the appropriate function to the arguments present at the start of time $t+1$. This rather noncomputational definition is due to the possibility of one subframe containing a vote on an input value and subsequent use in an execute. The effect of this sequence can be characterized by stating that the execution uses as inputs the values defined after the end of the subframe. In mapping this to the computation performed by the implementation, it was necessary to prove that schedule constraints allow this to be achieved by sequentially performing the activity sequence scheduled for the subframe.

The axiom defining a vote activity scheduled for the subframe is the following:

$$
\begin{array}{l}
p \in \mathbf{W}^t \quad \wedge \\
<vote, a, y> \in \ \mathbf{sched}(\ \mathbf{config}(t,p), t, p) \\
\quad \supset \\
(\mathbf{input\ in}\ p\ \mathbf{for}\ a\ \mathbf{at\ start}(t+1,p))[y] = \\
\quad \mathbf{majority}\left(\left\{ < d, q > \;\middle|\; \begin{array}{l} q \in \mathbf{poll\ by}\ p\ \mathbf{for}\ a\ \mathbf{at}\ t \quad \wedge \\ d = (\mathbf{datafile\ in}\ p\ \mathbf{for}\ a\ \mathbf{on}\ q\ \mathbf{at\ start}(t,p))[y] \end{array} \right\}\right)
\end{array}
$$

Given a working processor p scheduled to perform a vote on the y-th component of a during subframe t, the input file in p at the start of the following subframe is defined to be the majority of datafile values present in the datafile at the start of subframe t. The function **poll by**p **for** a **at** t denotes the set of processors determined by p at the time of the vote to have executed the last iteration of task a. This is defined as a rather complex function of p's view of the system configuration at the start of the subframe and of the schedule table. We do not give the definition here.

These axioms constitute the primary axioms defining the Activity specification. There are in all approximately 40 axioms defining the introduced functions and predicates of the model and constraining the composition of the schedule table.

In terms of the functions of the Activity model, we can now define the mappings to the function symbols of the Replication model. The function **Result on in** of the Replication level is derived with the following axiom:

$$
\begin{aligned}
&v = \mathbf{Result}(a,i)\ \mathbf{on}\ p\ \mathbf{in}\ q \\
&\quad \equiv \\
&\forall y,t\ \mathbf{beg}(i\ \mathbf{of}\ a) \leq t < \mathbf{end}(i\ \mathbf{of}\ a) \quad \wedge \\
&\qquad 1 \leq y \leq \mathbf{result\ size}(a) \quad \wedge \\
&\qquad < vote, k, y > \in \mathbf{sched}(\ \mathbf{config}(t,q), t, q) \\
&\qquad\quad \supset \\
&\qquad v[y] = \big(\mathbf{datafile\ in}\ q\ \mathbf{for}\ a\ \mathbf{on}\ p\ \mathbf{at\ start}(t,s)\big)[y]
\end{aligned}
$$

Briefly, the mapping axiom defines each component y of the **Result on in** value to be the value present in the datafile at the time during the execution window when a vote activity is scheduled for element y. Thus, the concept of value reportedly received by processor q from processor p is defined as the value used at the time of a scheduled vote on q.

In an analogous manner, the mapping up to the Replication **Result in** voted value is defined by the following axiom:

$$
\begin{aligned}
&\mathbf{start\ frame}(\ \mathbf{frame}(t)) = i \times \mathbf{frame\ size}() \quad \wedge \\
&1 \leq y \leq \mathbf{result\ size}(a) \quad \wedge \\
&< vote, k, y > \in \mathbf{sched}(\ \mathbf{config}(t,p), t, p) \\
&\quad \supset \\
&(\mathbf{Result}(a,i)\ \mathbf{in}\ \ p)[y] = \big(\mathbf{input\ in}\ p\ \mathbf{for}\ a\ \mathbf{at\ start}(t+1,p)\big)[y]
\end{aligned}
$$

Briefly stated once again, each y-th component of **Result in** for processor p is defined to be the value in the input file in p at the start of a subframe following a vote scheduled on element y during a subframe corresponding to the i-th iteration of task a. Intuitively, the voted value is the value in the input file following a scheduled vote. Schedule constraints allow only one vote to be scheduled on a given element during an execution window.

The **poll for of** concept of a global poll set found in the Replication level is mapped up from the Activity level with the following axiom.

$$
\mathbf{pollfor}\ i\ \mathbf{of}\ a = \left\{ \begin{aligned}
&q \mid \exists p \exists t \exists y \\
&\qquad \mathbf{start\ frame}(\ \mathbf{frame}(t)) = i \times \mathbf{framesize}() \quad \wedge \\
&\qquad 1 \leq y \leq \mathbf{result\ size}(a) \quad \wedge \\
&\qquad < vote, a, y > \in \mathbf{sched}(\ \mathbf{config}(t,p), t, p) \quad \wedge \\
&\qquad p \in \mathrm{S}^{i\ \mathbf{of}\ a} \quad \wedge \\
&\qquad q \in \mathbf{poll\ by}\ p\ \mathbf{for}\ a\ \mathbf{at}\ t
\end{aligned} \right\}
$$

The global concept of **poll for** i **of** a is derived as the set of all processors included in **poll by** p **for** a **at** t at the time of a scheduled vote (of any element) on a processor p safe for the execution window.

Finally, the last mapping to be illustrated is the derivation of the set of safe processors, as used in the Replication model. This is defined by the following mapping axiom:

$$
\mathbf{S}^t = \left\{ \begin{array}{l}
p \mid p \in W^t \quad \wedge \\
\qquad \mathbf{clock\ safe}(p,t) \quad \wedge \\
\qquad \mathbf{task}\ GE()\ \mathbf{safe\ during\ last}(t, GE()) \quad \wedge \\
\qquad \mathbf{Result}(GE(), \mathbf{last}(t, GE()))= \\
\qquad\qquad \mathbf{input\ in}\ p\ \mathbf{for}\ GE()\ \mathbf{at\ start}(t,p)
\end{array} \right\}
$$

The above definition represents a precise statement of a processor that is correctly functioning, has a view of the last Global Executive output reflecting the consensus, and whose clock is close enough to other safe processors to properly communicate. The interaction between processor safety and the output of the Global Executive is worthy of further explanation. The definition does not require the processor to have been safe during previous subframes; this allows transient faults to have affected the processor in the past. The only requirements expressed are (1) that the Global Executive task have had sufficient replication to remain safe (effectively since system start-up), (2) that the configuration (contained within the output of the Global Executive) for the current subframe be unaffected, and (3) that clock safety be recovered despite any transients affecting the clock in the past.

The proof of the relationship between the Replication Specification and the Broadcast Specification was quite challenging. The proof involved showing that the distributed system has, as a valid abstraction, the synchronous, global characterization expressed in the Replication Specification. This required that the axioms and schedule constraints imply consistency of configuration and schedule within a single processor and between processors during an execution window. It was necessary to show that vote and execute activities, replicated on different processors and running during different subframes within the frame, use the same information for input. Furthermore, the proof required that the various processors, operating independently and asynchronously with only local information, communicate with each other without mutual interference; that the task schedules guarantee that results are always available in other processors when required, and are never accessed during broadcast. The derivation of the Replication axioms involved 56 proofs, with an average of 7 premises each, and 410 instantiations of axioms and lemmas overall.

2 PrePost and Imperative Levels

The PrePost specification intended to form a bridge between the Activity Specification and the Pascal programs of the operating system. It is expressed in terms of preconditions and postconditions for operating system operations and the specifications are very close to the Pascal programs, essntially requiring the programs to "do what they do".

The Activity level represents a specification for each processor in the distributed, multiprocessor system. In contrast, the PrePost level, very similar in abstraction to the Activity level, defines the behavior of a single, independent processor. The model employs the data structure abstractions present in the actual Pascal operating system implementation and is intended to facilitate a connnection between the multiprocessor system specification and the proof of the Pascal operating system executing on a single processor.

At the program level of abstraction, even conceptually simple properties require very complex specification and tedious verification. Because of the difficulty inherent in mapping between design specifications and an imperative implementation model, we deliberately limited the conceptual jump between the two levels. Having proved all considered aspects of the design correct at higher levels in the hierarchy, the only conceptual jump between the lowest level design specification and the implementation was the change in specification medium; the PrePost specification expresses that the "code does what it does." A traditional verification condition generation paradigm [11] was employed to prove precondition/postcondition procedure characterizations from the Pascal procedures, each treated as a sequential program. We explain only enough of the model and its specification for the reader to glean an overall understanding of the nature of the specification.

Within the PrePost model, the state of a processor is specified as a pair $< p,t >$, where p is a processor id and t is a subframe time. Accessor functions **proc** and **time** map states into component processor and time components (respectively). For a state pair $< p,t >$, the function **next**$(< p,t >) =< p,t+1 >$. Within the PrePost model, each data structure of the Pascal program is declared as an explicit function of the state. At the program-level, the datafile is implemented as a two-dimensional array of type array [proc,task] of array[Integer] of Integer, mapping a processor id and task name into the array of Integer values currently in the datafile. The input file is a program structure declared of type array [task,Integer] of Integer, task name and element number into an Integer value. Similarly, the schedule table is implemented as an array of type array [proc, config, subframe, activity-index] of activity, defining for each processor, configuration, subframe, and activity index, which activity is to be performed. The schedule table is a constant data structure present in each processor and thus not a function of the state.

The following PrePost axiom defines the semantics of the Execute activity:

$$
\begin{array}{l}
\quad \mathbf{proc}(\mathit{siftstate}) \in \mathbf{W}^{\mathbf{time}(\mathit{siftstate})} \qquad \wedge \\
\quad \exists j \; 1 \le j \le \; \mathbf{max\ activities}() \quad \wedge \\
\qquad < \mathit{execute}, a >= \mathbf{sched\ table}()[\, \mathbf{real\ to\ virt}(\mathit{siftstate})[\, \mathbf{proc}(\mathit{siftstate})], \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{pconfig}(\mathit{siftstate}), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{subframe}(\mathit{siftstate}), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad j\,] \qquad \wedge \\
\quad \forall b \forall j \forall y \; 1 \le y \le \mathbf{result\ size}()[b] \quad \wedge \\
\qquad\qquad \mathbf{p.inputs}[a,j] = b \ne \mathbf{null\ task}() \\
\qquad\qquad\quad \supset \quad \mathbf{inp}[j,y] = \mathbf{input}(\, \mathbf{next}(\mathit{siftstate}))[b,y] \\
\supset \\
\mathbf{datafile}(\, \mathbf{next}(\mathit{siftstate}))[\, \mathbf{proc}(\mathit{siftstate}), a] = \mathbf{task\ results}(a, \mathbf{inp})
\end{array}
$$

The antecedent of the axiom defines the case where the processor component of the state is correctly functioning for the current subframe, some activity of the schedule for the current configuration and subframe specifies an Execute for task a, and the auxiliary array variable **inp** contains the value in the input data structure in the state **next** ($\mathit{siftstate}$), for each input task b indicated by the array **p.inputs**. The array **real to virt**, shown here as an explicit function of the state, maps a real processor id into a logical processor id, in terms of which the schedule table is defined. Assuming the antecedent holds, the axiom then defines the datafile in *the executing processor* in state **next**($\mathit{siftstate}$) to contain the results of applying the appropriate mathematical function to the input array **inp** . As we discussed in the previous section, it is required to prove during code verification that sequential execution of the schedule activities will satisfy this noncomputational specification of effect. A mapping axiom defines that the value corresponding to the processor's own entry in the datafile of a safe processor will be in all other datafiles by the start of the next subframe.

In order to apply sequential verification techniques to the Pascal program executing on the processor, it is necessary to make the state $< p,t >$ of the processor and the dependence upon a correctly functioning processor implicit. The sequential proof, in effect, considers execution on a properly functioning Pascal machine satisfying the axiomatic specification of Pascal. Furthermore, the **next** ($\mathit{siftstate}$) transition is taken to be one iteration of the Pascal **dispatcher** procedure, called once per subframe by a clock interrupt to execute the scheduled activity sequence. This "metatheoretic" jump is the only departure from our formal notion of hierarchy and is made as a concession to allow traditional code verification tools to form the last link in the proof. The validity of this jump is dependent upon a proof that the dispatcher in fact is allowed to execute as a sequential program, with no clock interrupts before completion and with no interference between internal and external data structure access. The former assumption was demonstrated by a timing analysis of the actual Bendix 930 code and the latter by the non-interference proof at the Activity Specification level.

The following precondition/postcondition characterization of the dispatcher is produced and verified for the actual dispatcher procedure:

$$\begin{array}{l}
\exists j\ 1 \leq j \leq \textbf{max activities}() \quad \wedge \\
\quad < execute, a > = \textbf{sched table}()[\ \textbf{real to virt}(myproc), \\
\qquad\qquad\qquad\qquad\qquad\qquad pconfig, \\
\qquad\qquad\qquad\qquad\qquad\qquad subframe, \\
\qquad\qquad\qquad\qquad\qquad\qquad j\,] \qquad \wedge \\
\forall b \forall j \forall y\ 1 \leq y \leq \textbf{result size}()[b] \quad \wedge \\
\qquad \textbf{pinputs}[a, j] = b \neq \textbf{null task}() \\
\qquad\quad \supset \quad \textbf{inp}[j, y] = \textbf{input}[b, y] \\
\left\{ \textbf{dispatcher} \right\} \\
\textbf{datafile}[myproc, a] = \textbf{task results}(a, \textbf{inp})
\end{array}$$

Hoare sentences like the above, asserting properties of a sequential dispatch procedure and its effect on the Pascal data structures, were proven consistent with the actual implementation.

The code proof required demonstration of approximately 40 verification conditions and was carried out by Dwight Hare and Karl Levitt using a version of the SPECIAL code verification system. The design proof between the Activity and PrePost specifications required 17 proofs, with an average of 9 premises each, and 148 instantiations overall.

8. Conclusions and Further Work

Our proof has demonstrated that the Pascal implementation of the SIFT distributed system satisfies the execution axioms of the I/O Specification. That the axioms of the I/O Specification characterize "correct" system operation remains a subjective judgement. The soundness of the axiomatic specifications is demonstrated by the existence of an imperative model at the lowest level of the hierarchy, relative to interpretations for all unimplemented function and predicate symbols (such as **W**, the set of working processors). Also assumed is the correct implementation of the Pascal machine, realized by the Pascal compiler and the Bendix BDX930 hardware.

The proof of the fault tolerant clock syncronization algrithm was performed independently, without mechanical support, and is given in Appendix B. The mechanical proof given here, the proof of correspondence to the I/O Specification, encompasses scheduling, rating, and interprocessor communication. Yet to be performed is the proof of orrespondence to the probabilistic reliability analysis, encompassing error diagnosis and reconfiguration. We expect to perform this remaining proof over the next year.

The process of formal specification and verification of SIFT resulted in the discovery of four design errors – errors that would have been difficult or impossible to detect by testing. Early specification efforts uncovered the insufficiency of three clocks for fault-tolerant clock synchronization (see Appendix B). The formal proof revealed that tasks not scheduled to execute did not regenerate their default result value every iteration, thus exposing that result to the accumulation of errors from transient faults.

A conclusion of our work is the importance of *design verification* prior to implementation verification. The highest-level design specifications for the SIFT system could not have been expressed in terms of specifications of individual Pascal programs.

The STP system used for specification and mechanical proof was developed concurrently with the proof effort, with its approach heavily influenced by our ongoing experience in attempting the proofs. The success of the man-machine symbiosis depended upon the user being able to express naturally his understanding of the proof in guiding the proof. Under other sponsorship, SRI is currently developing a new specification language for HDM, including parameterized theories, specification of state-modifying operations, and Hoare sentences, and are constructing an enhanced STP verification system.

References

[1] J. Wensley, et al., "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control", *Proceedings of the IEEE,* 60(10):1240-1254, October 1978.

[2] J. Goldberg, "SIFT: A Provable Fault-Tolerant Computer for Aircraft Flight Control", *IFIP Congress 80,* 1980.

[3] C. Weinstock, "SIFT: System Design and Implementation", *Tenth International Symposium on Fault-Tolerant Computing,* Oct 1980.

[4] J. Goldberg, "Development and Evaluation of a Software-Implemented Fault-Tolerant Computer: SIFT Hardware", Interim Technical Report, SRI International, Nov. 1979.

[5] K. Mosses, "SIFT – A Ultra-Reliable Avionic Computing System", *NATO AGARD Conference on Tactical Airborne Distributed Computing and Networks,* June 1981.

[6] M. Pease, R. Shostak, L. Lamport, "Reaching Agreement in the Presence of Faults", *JACM,* 27(2):228-234, April 1980.

[7] A. Hopkins, T.B. Smith, J. Lala, "FTMP – A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft", *Proceedings of the IEEE,* 66(10): 1221-1239, Oct 1978.

[8] R. Shostak, R. Schwartz, P.M. Melliar-Smith, "STP: A Mechanized Logic for Specification and Verification", 6th Conference on Automated Deduction, June 1982.

[9] R. Shostak, "Deciding Combinations of Theories", 6th Conference on Automated Deduction, June 1982.

[10] L. Lamport, P.M. Melliar-Smith, "Synchronizing Clocks in the Presence of Faults", in preparation, SRI International, Feb 1982.

[11] S. Igarashi, R. London, D. Luckham, "Automatic Program Verification I: A Logical Basis and its Implementation", *Acta Informatica,* 4:145-182, 1975.

CHAPTER 7

DESIGN VERIFICATION OF SIFT – LISTINGS

The Listing of the Design Proof for SIFT

This listing has been produced by a *prettyprinting* program from the original proof listings used in the proof. Manual intervention has been used only for pagination. The font used to print the listing was specially developed to improve the legibility of the proof for this report.

Reading a proof such as that below is a major undertaking, and complete familiarity with the STP system (described in Chapter 3), with the commands to STP (described in Chapter 4), and with the overall structure of the hierarchical specifications and design proof of SIFT (described in Chapter 5) are essential. To assist the reader to understand the actual proofs, we first include a sample proof in which the prove command has been expanded and annotated to show the substitutions made in the lemmas and axioms cited. Particular attention should be paid to the manner in which reference is made to free and bound quantified variables in the proof.

The proof commences with the definition of Integers (an augment to the built-in theory), Natural Numbers, Pairs, and Sets. Next the concept of a subframe is defined followed by the axioms of the IO Specification, the most abstract specification of the system. This is followed by a definition of Majority and the axioms of the Replication Specification. Next comes the proof of the relationship between the IO Specification and the Replication Specification, arranged as a series of lemmas, each with its proof, followed by the proofs for axioms of the IO Specification.

The listing continues with the axioms that describe constraints on the schedules for SIFT, followed by the axioms of the Activity Specification. Next comes the mapping of the Replication Specification onto the Activity Specification, and the lemmas and proofs that demonstrate the consistency of those two levels of the specifications. This is followed by the specifications of the PrePost level of abstraction of SIFT, and the mappings and proofs between the Activity and PrePost Specifications. Lastly, the proof status of a number of axioms is exhibited, showing what other axioms and unproven formulas the proof of each of these axioms depends on.

```
/*
A Primary Lemma.  If a task executes and is Safe, and if
        all its inputs are will behaved, a Safe processor voting on the
        broadcast results will obtain the correct result value for
        that task
*/

RP.L14: formula
        TASK.SAFE(K, I)
        ∧ ON.DURING(K, I)
        ∧ QQ ∈ SAFE.FOR(OF(I, K))
        ∧ (∀ L: L ∈ INPUTS(K)  ⊃  1 = CARD(RESULT(L, TO.OF(L, I, K))))
        ∧ 1 ≤ Y
        ∧ Y ≤ RESULT.SIZE(K)
              ⊃
        SEQ.ELEM(APPLY(FUNCTION(K), V.INPUTS.A2(I, K)), Y)
           = MAJORITY(D.BAG.D4(K, I, QQ, Y))

prove RP.L14 [L ← *L:1]
using RP.L13
      RP.D9A
      CARD.D.BAG.D4
      CARD.D.BAG.L10
      CARD.SUBSET [S2 ← D.BAG.L12(K, I, QQ, Y),
                   S1 ← D.BAG.L10(K, I, QQ, Y)]
      MAJ.1 [T1.V ← SEQ.ELEM(APPLY(FUNCTION(K), V.INPUTS.A2(I, K)), Y),
             M.BAG.1 ← D.BAG.L12(K, I, QQ, Y),
             M.BAG ← D.BAG.D4(K, I, QQ, Y)]
      RP.L12A [D.P.1 ← *V1.V2:6]
```

```
46_PP.PR.ATTEMPT)
PREMISES FOR PROOF OF RP.L14 ARE:

----------------
1. RP.L13

QQ ∈ SAFE.FOR(OF(I, K))
∧ (∀ L: L ∈ INPUTS(K)  ⊃  1 = CARD(RESULT(L, TO.OF(L, I, K))))
     ⊃
D.BAG.L10(K, I, QQ, Y) ⊆ D.BAG.L12(K, I, QQ, Y)

/* Note that, to satisfy the antecedent, we must show for an arbitary L
   in the input set, the cardinality of the result is 1.  The arbitary
   L can be referred to as *L:1 in composing substitutions. */
----------------

----------------
2. RP.D9A

TASK.SAFE(K, I:D)
     ≡
¬ON.DURING(K, I)
∨ CARD(POLL.FOR.OF(I, K))
     < 2*CARD(POLL.FOR.OF(I, K) ∩ SAFE.FOR(DW.OF(I, K)))

----------------

----------------
3. CARD.D.BAG.D4

CARD(D.BAG.D4(K, I, QQ, Y)) = CARD(POLL.FOR.OF(I, K))

----------------

----------------
4. CARD.D.BAG.L10

CARD(D.BAG.L10(K, I, QQ, Y))
   = CARD(POLL.FOR.OF(I, K) ∩ SAFE.FOR(DW.OF(I, K)))

----------------

----------------
5.       CARD.SUBSET [S2 ← D.BAG.L12(K, I, QQ, Y),
                      S1 ← D.BAG.L10(K, I, QQ, Y)]


D.BAG.L10(K, I, QQ, Y) ⊆ D.BAG.L12(K, I, QQ, Y)
     ⊃
CARD(D.BAG.L12(K, I, QQ, Y)) ≥ CARD(D.BAG.L10(K, I, QQ, Y))

----------------
```

```
6.     MAJ.1 [T1.V ← SEQ.ELEM(APPLY(FUNCTION(K), V.INPUTS.A2(I, K)), Y),
              M.BAG.1 ← D.BAG.L12(K, I, QQ, Y),
              M.BAG ← D.BAG.D4(K, I, QQ, Y)]


(∀ V1.V2:
    V1.V2 ∈ D.BAG.L12(K, I, QQ, Y)
          ≡
    SEQ.ELEM(APPLY(FUNCTION(K), V.INPUTS.A2(I, K)), Y) = VALUE(V1.V2)
    ∧ V1.V2 ∈ D.BAG.D4(K, I, QQ, Y))
      ⊃
(CARD(D.BAG.D4(K, I, QQ, Y)) < 2*CARD(D.BAG.L12(K, I, QQ, Y))
      ⊃
 SEQ.ELEM(APPLY(FUNCTION(K), V.INPUTS.A2(I, K)), Y)
    = MAJORITY(D.BAG.D4(K, I, QQ, Y)))

/* Again, the antecedent requires showing a property for an arbitary
   v1.v2 value pair.  *v1.v2:6 will refer within the substitutions
   to this value. */
----------------


----------------
7.  RP.L12A [D.P.1 ← *V1.V2:6]


*V1.V2:6 ∈ D.BAG.L12(K, I, QQ, Y)
     ≡
SEQ.ELEM(APPLY(FUNCTION(K), V.INPUTS.A2(I, K)), Y)
   = VALUE(*V1.V2:6)
∧ *V1.V2:6 ∈ D.BAG.D4(K, I, QQ, Y)

/* Here we substitute *v1.v2:6 for the free d.p.1 variable to
   discharge the hypothesis of premise 6. */
----------------

CONCLUSION IS

TASK.SAFE(K, I)
∧ ON.DURING(K, I)
∧ QQ ∈ SAFE.FOR(OF(I, K))
∧ (∀ *L:1:
      *L:1 ∈ INPUTS(K)  ⊃  1 = CARD(RESULT(*L:1, TO.OF(*L:1, I, K))))
∧ 1 ≤ Y
∧ Y ≤ RESULT.SIZE(K)
     ⊃
SEQ.ELEM(APPLY(FUNCTION(K), V.INPUTS.A2(I, K)), Y)
   = MAJORITY(D.BAG.D4(K, I, QQ, Y))

/* The antecedent of the lemma to be proven asserted that, for
   all input tasks L, the cardinality of the result set was one.
   We therefore substitute *L:1 for L, discharging the hypothesis of
   premise 1. */
```

SUBSECTION 7.1

INTEGER STP

```
var INT1: INTEGER

var INT2: INTEGER

var INT3: INTEGER

UNINTERPRETED.TIMES: (INTEGER, INTEGER) → INTEGER

IPLUS: (INT1, INT2) → INTEGER = INT1+INT2

IDIFFERENCE: (INT1, INT2) → INTEGER = INT1-INT2

ITIMES: (INT1, INT2) → INTEGER = UNINTERPRETED.TIMES(INT1, INT2)

TIMES.AXIOM.1: axiom
        ¬(INT1 = INT2) ∧ ¬(INT3 = 0)  ⊃  ¬(INT1*INT3 = INT2*INT3)

/* Induction Scheme over Integers with Upper Bound: */

/*
  (FORALL INT1 (FORALL INT3
   (IMPLIES
     (AND
       (LESSP INT3 INT1)
       (p INT1)
       (FORALL INT2
           (IMPLIES
            (AND
             (LESSEQP INT2 INT1)
             (p INT2))
            (p (IDIFFERENCE INT2 1)))))
        (p INT3)))))
*/

/* Induction Scheme over Integers with Lower Bound */

/*

  (FORALL INT1 (FORALL INT3
   (IMPLIES
     (AND
       (GREATERP INT3 INT1)
       (p INT1)
       (FORALL INT2
           (IMPLIES
            (AND
             (GREATEREQP INT2 INT1)
             (p INT2))
            (p (IPLUS INT2 1)))))
        (p INT3))))
*/
```

```
/* Induction Scheme over Integer Intervals: */

/*
  (FORALL INT1 (FORALL INT3
   (IMPLIES
       (FORALL INT2
         (AND
           (p INT1)
           (IMPLIES
             (AND
               (LESSEQP INT1 INT2)
               (LESSP INT2 INT3)
               (p INT2))
             (p (IPLUS INT2 1)))))
        (p INT3))))
*/

TIMES.AXIOM.2: axiom
        INT1*INT2+INT2 = (INT1+1)*INT2

TIMES.AXIOM.3: axiom
        INT1 > INT2 ∧ INT3 > 0  ⊃  INT1*INT3 > INT2*INT3

TIMES.AXIOM.4: axiom
        INT1 = 0  ⊃  INT2*INT1 = 0

TIMES.AXIOM.5: axiom
        INT1*INT2 = INT2*INT1
```

SUBSECTION 7.2

NAT STP

```
/* SPECIFICATION FOR NATURAL NUMBERS AS A SUBTYPE OF INTEGERS */

NAT: type is INTEGER

var Y: NAT

var Z: NAT

var W: NAT

var W1: NAT

NAT.NONNEGATIVE: axiom
        Y ≥ 0

NATBOT1: (NAT, NAT) → NAT

NATBOT2: (INTEGER) → NAT

NPLUS: (Y, Z) → NAT = Y+Z

var INT1: INTEGER

INT.NAT: (INT1) → NAT = if INT1 ≥ 0 then INT1 else NATBOT2(INT1) end if

NDIFFERENCE: (Y, Z) → NAT = if Y ≥ Z then Y-Z else NATBOT1(Y, Z) end if

NTIMES: (Z, W) → NAT = Z*W

MOD: (INTEGER, NAT) → NAT

MOD.AXIOM: axiom
        Y = MOD(INT1, W)  ≡  (∃ INT2: Y < W ∧ INT1 = W*INT2+Y)
```

SUBSECTION 7.3

PAIR OF STP

```
/* ********** Theory of Pairs ********** */

var TYPE1: type

var TYPE2: type

PAIR.OF: type(TYPE1, TYPE2)

FIRST: (PAIR.OF(TYPE1, TYPE2)) → TYPE1

SECOND: (PAIR.OF(TYPE1, TYPE2)) → TYPE2

MAKE.PAIR: (TYPE1, TYPE2) → PAIR.OF(TYPE1, TYPE2)

var PAIR: PAIR.OF(TYPE1, TYPE2)

var PAIR1: PAIR.OF(TYPE1, TYPE2)

var X1: TYPE1

var X2: TYPE2

PAIR.AXIOM.1: axiom
        ∃ PAIR: FIRST(PAIR) = X1 ∧ SECOND(PAIR) = X2

PAIR.AXIOM.2: axiom
        X1 = FIRST(MAKE.PAIR(X1, X2)) ∧ X2 = SECOND(MAKE.PAIR(X1, X2))

PAIR.EQUALITY: axiom
        PAIR = PAIR1
             ≡
        FIRST(PAIR) = FIRST(PAIR1) ∧ SECOND(PAIR) = SECOND(PAIR1)
```

SUBSECTION 7.4

SETS AXIOMS

```
/* ********** Set Theory Lemmas Assumed ********** */

using NAT.STP

var TYPE1: type

SET.OF: type(TYPE1)

var TYPE2: type

MEMBER: (TYPE1, SET.OF(TYPE1)) → BOOL

UNION: (SET.OF(TYPE1), SET.OF(TYPE1)) → SET.OF(TYPE1)

INTERSECTION: (SET.OF(TYPE1), SET.OF(TYPE1)) → SET.OF(TYPE1)

SUBSET: (SET.OF(TYPE1), SET.OF(TYPE1)) → BOOL

CARD: (SET.OF(TYPE1)) → INTEGER

var S: SET.OF(TYPE1)

var X1: TYPE1

/* (DTV TYPE1 X) */

var S1: SET.OF(TYPE1)

var V.CARD.1: TYPE1

var V.CARD.3: TYPE1

var S2: SET.OF(TYPE1)

SINGLETON: (S, X) → BOOL = X ∈ S ∧ CARD(S) = 1

SELECT: (SET.OF(TYPE1)) → TYPE1
```

SET.SELECTION: axiom

$$\text{SELECT}(S) \in S$$

INTERSECT: formula

$$X \in S \cap S1 \;\equiv\; X \in S \wedge X \in S1$$

INTERSECTION.COMMUTES: formula

$$S \cap S1 = S1 \cap S$$

CARD.INTERSECTION: formula

$$\text{CARD}(S \cap S1) \leq \text{CARD}(S1)$$

CARD.SUBSET: formula

$$S1 \subseteq S2 \;\supset\; \text{CARD}(S2) \geq \text{CARD}(S1)$$

SUBSET: formula

$$S1 \subseteq S2 \;\equiv\; (\forall X\text{: } X \in S1 \;\supset\; X \in S2)$$

CARD.1: formula

$$\text{CARD}(S) = 1 \;\supset\; (\exists \text{ V.CARD.1: V.CARD.1} \in S)$$

CARD.2: formula

$$\text{CARD}(S) = 1 \;\supset\; (X \in S \wedge X1 \in S \;\supset\; X = X1)$$

CARD.5: formula

$$S \subseteq S1 \wedge \text{CARD}(S) > 0 \;\supset\; \text{CARD}(S1) > 0$$

CARD.6: formula

$$\text{CARD}(S) \geq 0$$

SETEQUALITY: formula

$$S1 = S2 \;\equiv\; (\forall X\text{: } X \in S1 \;\equiv\; X \in S2)$$

CARD.4: formula

$$\text{CARD}(S) > 0 \;\supset\; (\exists X\text{: } X \in S)$$

CARD.3: formula

$$\begin{aligned}
&(\exists \text{ V.CARD.3: V.CARD.3} \in S \wedge (\forall X\text{: } X \in S \;\supset\; \text{V.CARD.3} = X))\\
&\quad\supset\\
&\text{CARD}(S) = 1
\end{aligned}$$

SUBSECTION 7.5

SUBFRAME AXIOMS

```
/* ********** Theory of Subframe Time ********** */

/* Declaration of Subframetime */

SUBFRAMETIME: type is INTEGER

var T.SUB: SUBFRAMETIME

var T1.SUB: SUBFRAMETIME

var T2.SUB: SUBFRAMETIME

SUB.INCR: (T.SUB) → SUBFRAMETIME = T.SUB+1

SUB.DECR: (T.SUB) → SUBFRAMETIME = T.SUB-1

/*
Induction Scheme over Subframetime:
  (FORALL T.SUB (FORALL T1.SUB
   (IMPLIES
       (FORALL T2.SUB
         (AND
           (p T.SUB)
           (IMPLIES
             (AND
               (LESSEQP T.SUB T2.SUB)
               (LESSP T2.SUB T1.SUB)
               (p T2.SUB))
             (p (SUB.INCR T2.SUB)))))
        (p T1.SUB))))
*/
```

SUBSECTION 7.6

I O AXIOMS

```
/* ********** Input/Output Axioms --- The Highest Level ********** */

using INTEGER.STP

using SEQ.STP

using SETS.AXIOMS

using PAIROF.STP

var TYPE1: type

var TYPE2: type

/* WAS (DTV TYPE1) */

REALTIME: type is INTEGER

SUBFRAMETIME: type is INTEGER

INTERVAL: type is PAIR.OF(SUBFRAMETIME, SUBFRAMETIME)

var INTERVAL1: INTERVAL

BEGIN: (INTERVAL1) → SUBFRAMETIME = FIRST(INTERVAL1)

END: (INTERVAL1) → SUBFRAMETIME = SECOND(INTERVAL1)

VALUE: (PAIR1) → TYPE1 = FIRST(PAIR1)

SOURCE: (PAIR1) → TYPE2 = SECOND(PAIR1)

FUNCTION.TYPE: type

SET.OF: type(TYPE1)

ITERATION: type is INTEGER

var I: ITERATION

INCR: (I) → ITERATION = 1+I

DATAVAL: type

DATA: type is SEQ(DATAVAL)

/* WAS (DT DATA) */

PROC: type

TASK: type

var K: TASK

var L: TASK

GLOBAL.EXEC: → TASK

CLOCK: → TASK
```

```
BOTTOM1: (TASK) → DATA
var J: ITERATION
var T: SUBFRAMETIME
var TT: SUBFRAMETIME
var II: INTERVAL
var P: PROC
var QQ: PROC
var V: DATA
var V.T: PAIR.OF(DATA, TASK)
var V.INPUTS: SET.OF(PAIR.OF(DATA, TASK))
var V.BAG: SET.OF(PAIR.OF(DATA, PROC))
EPSILON: → REALTIME
LAMBDA: → REALTIME
var T1: SUBFRAMETIME
var T2: SUBFRAMETIME
*X: (SET.OF(TYPE1)) → TYPE1
OF: (ITERATION, TASK) → INTERVAL
DW.OF: (ITERATION, TASK) → INTERVAL
DW.FOR.TO.OF: (TASK, ITERATION, TASK) → INTERVAL
TO.OF: (TASK, ITERATION, TASK) → ITERATION
ERROR.REPORTER: (PROC) → TASK
var T.SUB: SUBFRAMETIME
SUB.INCR: (T.SUB) → SUBFRAMETIME = T.SUB+1
SUB.DECR: (T.SUB) → SUBFRAMETIME = T.SUB-1
IC.ERROR.REPORTER: (PROC) → TASK
SAFE: (SUBFRAMETIME) → SET.OF(PROC)
SAFE.FOR: (INTERVAL) → SET.OF(PROC)
CONFIGURATION: (DATA) → SET.OF(PROC)
TASK.SAFE: (TASK, ITERATION) → BOOL
POLL.FOR.OF: (ITERATION, TASK) → SET.OF(PROC)
```

```
ON: (TASK, ITERATION, PROC) → SET.OF(DATA)

ON.IN: (TASK, ITERATION, PROC, PROC) → DATA

IN: (TASK, ITERATION, PROC) → DATA

RESULT: (TASK, ITERATION) → SET.OF(DATA)

IC: (TASK) → BOOL

ON.DURING: (TASK, ITERATION) → BOOL

SSF: (TASK, TASK) → BOOL

INPUTS: (TASK) → SET.OF(TASK)

APPLY: (FUNCTION.TYPE, SET.OF(PAIR.OF(DATA, TASK))) → DATA

FUNCTION: (TASK) → FUNCTION.TYPE

REAL.TIME: (SUBFRAMETIME) → REALTIME

REPORTS: (PROC, PROC, ITERATION, TASK) → BOOL

REPORTVAL: (PROC, PROC, ITERATION, TASK) → DATA

ON.DURING: (TASK, ITERATION) → BOOL

TO.OF: (TASK, ITERATION, TASK) → ITERATION

TASK.SAFE: (TASK, ITERATION) → BOOL

RESULT: (TASK, ITERATION) → SET.OF(DATA)

var V.P: PAIR.OF(DATA, PROC)

IC.TASK.SAFE: (TASK, ITERATION) → BOOL

IC.TASK.SAFE: (TASK, ITERATION) → BOOL

SELECT: (SET.OF(TYPE1)) → TYPE1
```

```
/*
The End of a task iteration follows the Beginning
        by at least one subframe
*/

IO.A1.1: axiom
        SUB.INCR(BEGIN(OF(I, K))) < END(OF(I, K))

/*
The End of one iteration of a task precedes the following
        iteration of the task
*/

IO.A1.2: axiom
        END(OF(I, K)) ≤ BEGIN(OF(INCR(I), K))

/*
SSF(L K) means that, for each iteration of task K,
        the corresponding iteration of input task L will
        have a one subframe overlap with K
*/

IO.A1.3: axiom
        SSF(L, K)  ⊃  SUB.INCR(BEGIN(OF(I, K))) = END(OF(TO.OF(L, I, K), L))

/*
Axiom defining correct behavior of an Interactive
        Consistency task K: If K is safe, it will always
        produce a single output value -- regardless of
        the stability of inputs to the task
*/

IO.A3: axiom
        IC(K) ∧ IC.TASK.SAFE(K, I)  ⊃  CARD(RESULT(K, I)) = 1

/*
An interactive consistency task K can only have
        one replication of one input task.  An interactive
        consistency task has as its associated mathematical
        function the identity function.
*/

IO.A4: axiom
        IC(K) ∧ SOURCE(V.T) ∈ INPUTS(K) ∧ SINGLETON(V.INPUTS, V.T)
            ⊃
        CARD(INPUTS(K)) = 1
        ∧ (L ∈ INPUTS(K)  ⊃  1 = CARD(POLL.FOR.OF(TO.OF(L, I, K), L)))
        ∧ VALUE(V.T) = APPLY(FUNCTION(K), V.INPUTS)
```

```
/*
A safe task K which is 'on' will use 'bottom' as an input
         value for any input task L which was not on for the
         corresponding iteration.  Further, task L is 'safe' in
         this case.
*/

IO.A5: axiom
        L ∈ INPUTS(K)
        ∧ ON.DURING(K, I)
        ∧ TASK.SAFE(K, I)
        ∧ ¬ON.DURING(L, TO.OF(L, I, K))
             ⊃
        SINGLETON(RESULT(L, TO.OF(L, I, K)), BOTTOM1(L))
        ∧ TASK.SAFE(L, TO.OF(L, I, K))

/*
If every previous iteration of the clock task
         prior to T1 was safe, the skew between
         any prior time T2 and now will be less than
         (T1-T2)*(1-Lambda)
*/

IO.A6: axiom
        T2 < T1 ∧ (∀ I: END(OF(I, CLOCK())) ≤ T1  ⊃  TASK.SAFE(CLOCK(), I))
             ⊃
        (T1-T2)*(1-LAMBDA())-EPSILON() < REAL.TIME(T1)-REAL.TIME(T2)
        ∧ REAL.TIME(T1)-REAL.TIME(T2) < (T1-T2)*(1+LAMBDA())+EPSILON()

V.INPUTS.A2: (ITERATION, TASK) → SET.OF(PAIR.OF(DATA, TASK))

/*
Set Abstraction for set V.INPUTS.A2: the
         set of <L,Result(L,L to I of K)> pairs for
         each input task L to task K
*/

IO.A2A: axiom
        SOURCE(V.T) ∈ INPUTS(K)
        ∧ VALUE(V.T) ∈ RESULT(SOURCE(V.T), TO.OF(SOURCE(V.T), I, K))
             ≡
        V.T ∈ V.INPUTS.A2(I, K)

/*
MAIN EXECUTE AXIOM: If a task K is 'on' and 'safe' during
         iteration I, then output set Result(K,I) will be the set
         resulting from applying the proper mathematical function
         to the input set V.INPUTS.A2, composed according to the
         previous set abstraction axiom
*/

IO.A2: axiom
        ON.DURING(K, I)
        ∧ TASK.SAFE(K, I)
        ∧ (∀ L: L ∈ INPUTS(K)  ⊃  CARD(RESULT(L, TO.OF(L, I, K))) = 1)
             ⊃
        SINGLETON(RESULT(K, I), APPLY(FUNCTION(K), V.INPUTS.A2(I, K)))
```

SUBSECTION 7.7

MAJORITY STP

```
/* ********** Theory of Majority over Bags ********** */

using SETS.AXIOMS

using PAIROF.STP

var TYPE1: type

var TYPE2: type

MAJORITY: (SET.OF(PAIR.OF(TYPE1, TYPE2))) → TYPE1

var M.BAG: SET.OF(PAIR.OF(TYPE1, TYPE2))

var M.BAG.1: SET.OF(PAIR.OF(TYPE1, TYPE2))

var V1.V2: PAIR.OF(TYPE1, TYPE2)

var T1.V: TYPE1

var T2.V: TYPE1

var T3.V: TYPE2

BOTTOM: (TYPE1) → TYPE1

MAJ.1: axiom
        (∀ V1.V2: V1.V2 ∈ M.BAG.1  ≡  T1.V = VALUE(V1.V2) ∧ V1.V2 ∈ M.BAG)
            ⊃
        (CARD(M.BAG) < 2*CARD(M.BAG.1)  ⊃  T1.V = MAJORITY(M.BAG))

MAJ.2: axiom
        0 = CARD(M.BAG)  ⊃  BOTTOM(T2.V) = MAJORITY(M.BAG)

MAJ.3: axiom
        ¬(∃ T1.V, T3.V ∀ M.BAG.1:
             (∀ V1.V2: V1.V2 ∈ M.BAG.1  ≡  T1.V = VALUE(V1.V2) ∧ V1.V2 ∈ M.BAG)
                 ⊃
             CARD(M.BAG) < 2*CARD(M.BAG.1))
            ⊃
        BOTTOM(T2.V) = MAJORITY(M.BAG)
```

SUBSECTION 7.8

SPECIFICATION FOR SEQUENCES OF TYPE1 VALUES

```
/* SPECIFICATION FOR SEQUENCES OF TYPE1 VALUES */

using NAT.STP

var TYPE1: type
var X: TYPE1

SEQ: type(TYPE1)

var SEQ1,SEQ2: SEQ(TYPE1)

SEQ.ELEM: (SEQ(TYPE1), NAT) TYPE1
SEQ.LENGTH: (SEQ(TYPE1)) NAT
SEQ.CAT: (SEQ(TYPE1), SEQ(TYPE1)) SEQ(TYPE1)
SEQ.NEW: (TYPE1) SEQ(TYPE1)
MAKESEQ: (TYPE1) SEQ(TYPE1)
SEQ.MEMBER: (TYPE1, SEQ(TYPE1)) BOOL

var Y: NAT

SEQ.LENGTH.AXIOM: axiom
        SEQ.LENGTH(SEQ.CAT(SEQ1, SEQ2)) = SEQ.LENGTH(SEQ1)+SEQ.LENGTH(SEQ2)

SEQ.ELEM.AXIOM2: axiom
        SEQ.ELEM(SEQ.CAT(SEQ1, SEQ2), Y) = X
             ≡
        (Y > SEQ.LENGTH(SEQ1)-INT.NAT(1)
         ∧ SEQ.ELEM(SEQ2, Y-SEQ.LENGTH(SEQ1)) = X)
        ∨ (Y ≤ SEQ.LENGTH(SEQ1)-INT.NAT(1) ∧ SEQ.ELEM(SEQ1, Y) = X)

SEQ.LENGTH.NEW.AXIOM: axiom
        SEQ.LENGTH(SEQ.NEW(X)) = 0

SEQ.MAKESEQ.AXIOM: axiom
        SEQ.LENGTH(SEQ1) = 1 ∧ SEQ.ELEM(SEQ1, INT.NAT(0)) = X
             ≡
        SEQ1 = MAKESEQ(X)

SEQ.MEMBER.AXIOM: axiom
        SEQ.MEMBER(X, SEQ1)
             ≡
        (∃ Y: 0 ≤ Y ∧ Y < SEQ.LENGTH(SEQ1) ∧ SEQ.ELEM(SEQ1, Y) = X)

SEQ.EQUALITY.AXIOM: axiom
        SEQ1 = SEQ2  ≡  (∀ Y: SEQ.ELEM(SEQ1, Y) = SEQ.ELEM(SEQ2, Y))
```

SUBSECTION 7.9

REP AXIOMS

```
/*
********** Replication Specification and
                    Mapping to Input/Output Specification **********
*/

using INTEGER.STP

using SEQ.STP

using SETS.AXIOMS

using PAIROF.STP

var TYPE2: type

var TYPE1: type

REALTIME: type is INTEGER

SUBFRAMETIME: type is INTEGER

INTERVAL: type is PAIR.OF(SUBFRAMETIME, SUBFRAMETIME)

var INTERVAL1: INTERVAL

BEGIN: (INTERVAL1) → SUBFRAMETIME = FIRST(INTERVAL1)

END: (INTERVAL1) → SUBFRAMETIME = SECOND(INTERVAL1)

/* BEGIN: (INTERVAL) → SUBFRAMETIME */

/* END: (INTERVAL) → SUBFRAMETIME */

VALUE: (PAIR1) → TYPE1 = FIRST(PAIR1)

SOURCE: (PAIR1) → TYPE2 = SECOND(PAIR1)

using MAJORITY.STP

FUNCTION.TYPE: type

ITERATION: type is INTEGER

DATAVAL: type

BOTTOMD: → DATAVAL

var D1: DATAVAL

/*
A type system hack: the parameterized BOTTOM
         function applied to any DATAVAL produces the
         BOTTOMD element of the DATAVAL domain
*/

BOTTOM.EQUALITY: axiom
        BOTTOM(D1) = BOTTOMD()
```

```
TASK: type

var K: TASK

var L: TASK

RESULT.SIZE: (TASK) → NAT

DATA: type is SEQ(DATAVAL)

var V: DATA

var V1: DATA

BOTTOM1: (TASK) → DATA

/*
Each element of a DATA BOTTOM1 function is
         a BOTTOMD value -- again a type hack
*/

DATA.BOTTOM: axiom
        1 ≤ Y ∧ Y ≤ RESULT.SIZE(K)  ⊃  SEQ.ELEM(BOTTOM1(K), Y) = BOTTOMD()

/*
Two DATA values V and V1 are equal iff the
         lengths are equal and corresponding elements
         are equal
*/

DATA.EQUALITY: axiom
        V = V1
            ≡
        (∀ Y:
            SEQ.LENGTH(V) = SEQ.LENGTH(V1) ∧ 1 ≤ Y ∧ Y ≤ SEQ.LENGTH(V)
                 ⊃
            SEQ.ELEM(V, Y) = SEQ.ELEM(V1, Y))

PROC: type

GLOBAL.EXEC: → TASK

CLOCK: → TASK

var I: ITERATION

var J: ITERATION

var J1: ITERATION

var T: SUBFRAMETIME

var TT: SUBFRAMETIME

var II: INTERVAL

var P: PROC

var QQ: PROC
var R: PROC

var V.T: PAIR.OF(DATA, TASK)

var V.INPUTS: SET.OF(PAIR.OF(DATA, TASK))

var V.BAG: SET.OF(PAIR.OF(DATA, PROC))

EPSILON: → REALTIME

LAMBDA: → REALTIME

var T1: SUBFRAMETIME

var T2: SUBFRAMETIME

OF: (ITERATION, TASK) → INTERVAL

DW.OF: (ITERATION, TASK) → INTERVAL

DW.FOR.TO.OF: (TASK, ITERATION, TASK) → INTERVAL

TO.OF: (TASK, ITERATION, TASK) → ITERATION

ERROR.REPORTER: (PROC) → TASK

var T.SUB: SUBFRAMETIME

SUB.INCR: (T.SUB) → SUBFRAMETIME = T.SUB+1

SUB.DECR: (T.SUB) → SUBFRAMETIME = T.SUB-1

IC.ERROR.REPORTER: (PROC) → TASK

SAFE: (SUBFRAMETIME) → SET.OF(PROC)

SAFE.FOR: (INTERVAL) → SET.OF(PROC)

CONFIGURATION: (DATA) → SET.OF(PROC)

TASK.SAFE: (TASK, ITERATION) → BOOL

POLL.FOR.OF: (ITERATION, TASK) → SET.OF(PROC)

ON: (TASK, ITERATION, PROC) → SET.OF(DATA)

ON.IN: (TASK, ITERATION, PROC, PROC) → DATA

IN: (TASK, ITERATION, PROC) → DATA

RESULT: (TASK, ITERATION) → SET.OF(DATA)

V.INPUTS.A2: (ITERATION, TASK) → SET.OF(PAIR.OF(DATA, TASK))

APPLY: (FUNCTION.TYPE, SET.OF(PAIR.OF(DATA, TASK))) → DATA

FUNCTION: (TASK) → FUNCTION.TYPE
```

```
/* Defines sequence lengths of functions returning DATA values */

DATA.SIZE.IS.SEQ.LENGTH: axiom
        SEQ.LENGTH(IN(K, I, QQ)) = RESULT.SIZE(K)
        ∧ SEQ.LENGTH(APPLY(FUNCTION(K), V.INPUTS.A2(I, K))) = RESULT.SIZE(K)
        ∧ SEQ.LENGTH(BOTTOM1(K)) = RESULT.SIZE(K)
        ∧ SEQ.LENGTH(ON.IN(K, I, P, QQ)) = RESULT.SIZE(K)

/* Each task output vector has at least one element */

RESULT.SIZE.GREATER.THAN.1: axiom
        RESULT.SIZE(K) ≥ 1

IC: (TASK) → BOOL

ON.DURING: (TASK, ITERATION) → BOOL

SSF: (TASK, TASK) → BOOL

INPUTS: (TASK) → SET.OF(TASK)

REAL.TIME: (SUBFRAMETIME) → REALTIME

REPORTS: (PROC, PROC, ITERATION, TASK) → BOOL

REPORTVAL: (PROC, PROC, ITERATION, TASK) → DATA

ON.DURING: (TASK, ITERATION) → BOOL

TO.OF: (TASK, ITERATION, TASK) → ITERATION

TASK.SAFE: (TASK, ITERATION) → BOOL

RESULT: (TASK, ITERATION) → SET.OF(DATA)

var V.CARD: DATA

var V.CARD1: DATA

var S1: SET.OF(TYPE1)

var S2: SET.OF(TYPE1)

var V2: DATA

var V3: DATA

var V4: DATA

var V.P: PAIR.OF(DATA, PROC)
```

```
D.BAG.L10: (TASK, ITERATION, PROC, NAT) → SET.OF(PAIR.OF(DATAVAL, PROC))

IC.TASK.SAFE: (TASK, ITERATION) → BOOL

IC.TASK.SAFE: (TASK, ITERATION) → BOOL

DECR: (I) → ITERATION = I-1

INCR: (I) → ITERATION = 1+I

/* same as IO.A1.1 */

RP.A1.1: axiom
        SUB.INCR(BEGIN(OF(I, K))) < END(OF(I, K))

/* same as IO.A1.2 */

RP.A1.2: axiom
        END(OF(I, K)) ≤ BEGIN(OF(INCR(I), K))

/*
If a processor P is both in the poll set
         for the i-th iteration of K and safe for
         the DataWindow, then the set On(K,I,P) of 'computed'
         values is defined to consist of exactly the
         On.In(K,I,P,Q) values which safe processors
         have received from P
*/

RP.A2: axiom
        P ∈ POLL.FOR.OF(I, K) ∩ SAFE.FOR(DW.OF(I, K))
             ⊃
        (V ∈ ON(K, I, P)
              ≡
         (∃ QQ: QQ ∈ SAFE.FOR(OF(I, K)) ∧ V = ON.IN(K, I, P, QQ)))

/* Error Reporter Characteristics... */

RP.A7: axiom
        CARD(INPUTS(ERROR.REPORTER(P))) = 0
        ∧ SINGLETON(INPUTS(IC.ERROR.REPORTER(P)), ERROR.REPORTER(P))
        ∧ IC(IC.ERROR.REPORTER(P))
        ∧ SINGLETON(POLL.FOR.OF(I, ERROR.REPORTER(P)), P)
        ∧ I = TO.OF(IC.ERROR.REPORTER(P), I, ERROR.REPORTER(P))

/* More Error Reporter Characteristics */

RP.A8: axiom
        IC.ERROR.REPORTER(P) ∈ INPUTS(GLOBAL.EXEC())
        ∧ BEGIN(OF(I, GLOBAL.EXEC())) < BEGIN(OF(I, IC.ERROR.REPORTER(QQ)))
        ∧ BEGIN(OF(I, IC.ERROR.REPORTER(QQ)))
             < BEGIN(OF(INCR(I), GLOBAL.EXEC()))
```

```
/* Global Executive Characteristics */

RP.A9: axiom
        CONFIGURATION(SELECT(RESULT(GLOBAL.EXEC(), I)))
           ⊆ CONFIGURATION(SELECT(RESULT(GLOBAL.EXEC(), DECR(I))))
        ∧ (END(OF(I, GLOBAL.EXEC())) < BEGIN(OF(J, K))
                  ⊃
           POLL.FOR.OF(J, K) ⊆ CONFIGURATION(SELECT(RESULT(GLOBAL.EXEC(), I))))

/*
The beginning of the DataWindow for input task L
         providing input to the i-th iteration of K is
         equal to the beginning of the corresponding iteration
         of L
*/

RP.D2.1: axiom
        BEGIN(DW.FOR.TO.OF(L, I, K))
           = if L ∈ INPUTS(K)
                then BEGIN(OF(TO.OF(L, I, K), L))
                else BEGIN(OF(I, K))
             end if

/* The end of the same DataWindow is the end of the
         i-th iteration of K
*/

RP.D2.2: axiom
        END(DW.FOR.TO.OF(L, I, K)) = END(OF(I, K))

/*
The overall DataWindow for the i-th iteration of K
         starts sometime after the beginning of the
         DataWindow for each input task
*/

RP.D3.1: axiom
        ¬(BEGIN(DW.FOR.TO.OF(L, I, K)) < BEGIN(DW.OF(I, K)))

/*
The overall DataWindow starts with the DataWindow
         for some (input) task
*/

RP.D3.2: axiom
        ∃ L: BEGIN(DW.FOR.TO.OF(L, I, K)) = BEGIN(DW.OF(I, K))

/* The overall DataWindow ends with the end of the i-th iteration */

RP.D3.3: axiom
        END(DW.OF(I, K)) = END(OF(I, K))
```

```
/*
Mapping: A task is on iff there is at least one processor in
         the poll set
*/

RP.D7: axiom
        ON.DURING(K, I)  ≡  CARD(POLL.FOR.OF(I, K)) > 0

var D.P: PAIR.OF(DATAVAL, PROC)

var D.BAG: SET.OF(PAIR.OF(DATAVAL, PROC))

D.BAG.D4: (TASK, ITERATION, PROC, NAT) → SET.OF(PAIR.OF(DATAVAL, PROC))

/*
Set Abstraction for D.BAG.D4: Consists of all
         <dataval,processor> pairs for the y-th DATA element
         received by processors in the poll set
*/

RP.D4A: axiom
        D.P ∈ D.BAG.D4(K, I, QQ, Y)
             ≡
        (∃ P:
            SEQ.ELEM(ON.IN(K, I, P, QQ), Y) = VALUE(D.P)
            ∧ P = SOURCE(D.P)
            ∧ P ∈ POLL.FOR.OF(I, K))

/*
Definition: For a safe processor QQ, the IN value
         is defined to be equal to the majority of the set
         defined above
*/

RP.D4: axiom
        QQ ∈ SAFE.FOR(OF(I, K)) ∧ 1 ≤ Y ∧ Y ≤ RESULT.SIZE(K)
             ⊃
        SEQ.ELEM(IN(K, I, QQ), Y) = MAJORITY(D.BAG.D4(K, I, QQ, Y))

V.INPUTS.A3: (TASK, ITERATION, PROC) → SET.OF(PAIR.OF(DATA, TASK))

/*
Set abstraction for V.INPUTS.A3(K,I,P): the set of
         <DATA,TASK> pairs where the DATA value is the
          voted IN value for an input task -- all in processor P
*/

RP.A3A: axiom
        V.T ∈ V.INPUTS.A3(K, I, P)
             ≡
        SOURCE(V.T) ∈ INPUTS(K)
        ∧ VALUE(V.T) = IN(SOURCE(V.T), TO.OF(SOURCE(V.T), I, K), P)
```

```
/*
MAIN EXECUTE AXIOM: If processor P is both in
         the poll set and safe for the DataWindow, it
         will produce a single, correct value
*/

RP.A3: axiom
        P ∈ POLL.FOR.OF(I, K) ∩ SAFE.FOR(DW.OF(I, K))
             ⊃
        SINGLETON(ON(K, I, P), APPLY(FUNCTION(K), V.INPUTS.A3(K, I, P)))

/* same as IO.D1 */

RP.D1: axiom
        (L ∈ INPUTS(K) ∧ ¬SSF(L, K)
              ⊃
         ¬(BEGIN(OF(I, K)) < END(OF(TO.OF(L, I, K), L)))
         ∧ BEGIN(OF(I, K)) < END(OF(INCR(TO.OF(L, I, K)), L)))
        ∧ (L ∈ INPUTS(K) ∧ SSF(L, K)
                ⊃
           END(OF(TO.OF(L, I, K), L)) = SUB.INCR(BEGIN(OF(I, K))))

IO.D1: axiom
        (L ∈ INPUTS(K) ∧ ¬SSF(L, K)
              ⊃
         ¬(BEGIN(OF(I, K)) < END(OF(TO.OF(L, I, K), L)))
         ∧ BEGIN(OF(I, K)) < END(OF(INCR(TO.OF(L, I, K)), L)))
        ∧ (L ∈ INPUTS(K) ∧ SSF(L, K)
                ⊃
           END(OF(TO.OF(L, I, K), L)) = SUB.INCR(BEGIN(OF(I, K))))

/*
Set Abstraction D.BAG.L10(K,I,QQ,Y): set of
         <DATAVAL,Processor> pairs  where the DATAVAL
         was received by a processor P which was
         both in the poll set and safe for the appropriate period.
*/

RP.D11: axiom
        D.P ∈ D.BAG.L10(K, I, QQ, Y)
             ≡
        (∃ P:
            P ∈ POLL.FOR.OF(I, K) ∩ SAFE.FOR(DW.OF(I, K))
            ∧ SEQ.ELEM(ON.IN(K, I, P, QQ), Y) = VALUE(D.P)
            ∧ P = SOURCE(D.P))

/*
Mapping: A task K is safe during iteration i iff
         either it is not on or a majority of the processors
         in the poll set are safe for the DataWindow
*/

RP.D9A: axiom
        TASK.SAFE(K, I)
             ≡
        ¬ON.DURING(K, I)
        ∨ CARD(POLL.FOR.OF(I, K))
             < 2*CARD(POLL.FOR.OF(I, K) ∩ SAFE.FOR(DW.OF(I, K)))
```

```
/* same as IO.A3 --this will be left for future proof attempts */

RP.A4: axiom
        IC(K) ∧ IC.TASK.SAFE(K, I)  ⊃  CARD(RESULT(K, I)) = 1

/*
Mapping: An iteractive consistency task K is safe iff
         (2 * size of poll set for both K and input task)
         < (3 * number of safe processors in the poll set for
                either K or input task)
*/

RP.D9B: axiom
        IC.TASK.SAFE(K, I)
             ≡
        ¬ON.DURING(K, I)
        ∨ (IC(K)
           ∧ (L ∈ INPUTS(K)
                    ⊃
              2*CARD(POLL.FOR.OF(TO.OF(L, I, K), L) ∪ POLL.FOR.OF(I, K))
                 < 3
                   *CARD(SAFE.FOR(DW.OF(I, K))
                         ∩ (POLL.FOR.OF(TO.OF(L, I, K), L)
                            ∪ POLL.FOR.OF(I, K)))))

/* same as IO.A4 */

RP.A5: axiom
        IC(K) ∧ SOURCE(V.T) ∈ INPUTS(K) ∧ SINGLETON(V.INPUTS, V.T)
             ⊃
        CARD(INPUTS(K)) = 1
        ∧ (L ∈ INPUTS(K)  ⊃  1 = CARD(POLL.FOR.OF(TO.OF(L, I, K), L)))
        ∧ VALUE(V.T) = APPLY(FUNCTION(K), V.INPUTS)

/*
A safe processor reports any processor from whom it has
         received a value not consistent with its voted value
         -- does not apply to outputs from  interactive consistency tasks
*/

RP.A10: axiom
        P ∈ SAFE.FOR(DW.OF(J, K))
             ⊃
        (¬IC(L)
         ∧ L ∈ INPUTS(K)
         ∧ QQ ∈ POLL.FOR.OF(TO.OF(L, J, K), L)
         ∧ ¬(ON.IN(L, TO.OF(L, J, K), QQ, P) = IN(L, TO.OF(L, J, K), P))
              ≡
         REPORTS(P, QQ, TO.OF(L, J, K), L))

/* An unsafe processor can never again be trusted */

RP.A11: formula
        T1 > T2  ⊃  SAFE(T1) ⊆ SAFE(T2)
```

```
/* same as IO.A6 --not covered by current proof */

RP.A6: axiom
        T2 < T1 ∧ (∀ I: END(OF(I, CLOCK())) ≤ T1  ⊃  TASK.SAFE(CLOCK(), I))
             ⊃
        (T1-T2)*(1-LAMBDA())-EPSILON() < REAL.TIME(T1)-REAL.TIME(T2)
        ∧ REAL.TIME(T1)-REAL.TIME(T2) < (T1-T2)*(1+LAMBDA())+EPSILON()

/* not used in proof */

RP.D8: axiom
        REPORTS(P, QQ, I, K)
             ≡
        (∃ J:
            BEGIN(OF(I, K)) ≤ BEGIN(OF(J, ERROR.REPORTER(P)))
            ∧ BEGIN(OF(DECR(J), ERROR.REPORTER(P))) < END(OF(TO.OF(L, I, K), L))
            ∧ REPORTVAL(P, QQ, I, K) ∈ RESULT(ERROR.REPORTER(P), J))

/* not used in proof */

RP.D10: axiom
        (∀ T: BEGIN(II) ≤ T ∧ T < END(II)  ⊃  P ∈ SAFE(T))  ≡  P ∈ SAFE.FOR(II)

/*
Mapping: A DATA value V is in the result set iff
         it is the voted value from some safe processor
*/

RP.D6: axiom
        V ∈ RESULT(K, I)  ≡  (∃ P: P ∈ SAFE.FOR(OF(I, K)) ∧ V = IN(K, I, P))
```

SUBSECTION 7.10

I O RP DERIVATION

```
/*
********** Proof of Mapping between
                    I/O and Replication Specifications **********
*/

/*
The beginning of a Data Window is earlier or at least equal
         to the beginning of the Execution Window
*/

RP.L1: formula
        BEGIN(OF(I, K)) ≥ BEGIN(DW.FOR.TO.OF(L, I, K))

prove RP.L1
using RP.A1.1 [K ← L,
               I ← TO.OF(L, I, K)]
      RP.D1
      RP.D2.1
```

```
/*
If a time is within the Execution Window, then it must
        be within the Data Window
*/

RP.L2: formula
        BEGIN(OF(I, K)) ≤ T ∧ T < END(OF(I, K))
             ⊃
        BEGIN(DW.OF(I, K)) ≤ T ∧ T < END(DW.OF(I, K))

prove RP.L2
using RP.A1.1
      RP.D3.3
      RP.D3.1
      RP.L1
```

```
/*
If a processor is Safe for the Data Window, it is Safe
         for the Execution Window
*/

RP.L3: formula
        P ∈ SAFE.FOR(DW.OF(I, K))  ⊃  P ∈ SAFE.FOR(OF(I, K))

prove RP.L3
using RP.L2 [T ← *T:3]
      RP.D10 [T ← *T:3,
              II ← DW.OF(I, K)]
      RP.D10 [II ← OF(I, K),
              T ← @:D]
```

```
/*
If a task generates a singleton result value, then safe
        processors will have that value in their In buffer
*/

RP.L4: formula
        L ∈ INPUTS(K)
        ∧ 1 = CARD(RESULT(L, TO.OF(L, I, K)))
        ∧ P ∈ SAFE.FOR(DW.OF(I, K))
             ⊃
        (V ∈ RESULT(L, TO.OF(L, I, K))  ≡  V = IN(L, TO.OF(L, I, K), P))

prove RP.L4
using RP.L7
      CARD.2 [X ← IN(L, TO.OF(L, I, K), P),
              X1 ← V,
              S ← RESULT(L, TO.OF(L, I, K))]
      RP.D6 [I ← TO.OF(L, I, K),
             K ← L,
             V ← IN(L, TO.OF(L, I, K), P)]
      RP.D6 [I ← TO.OF(L, I, K),
             K ← L]
```

```
/*
If a task is on a processor that is Safe for its data window,
         and if all its input tasks are well behaved, the inputs to the
         task will be same as in the IO Model
*/

RP.L5: formula
        (∀ L: L ∈ INPUTS(K)  ⊃  1 = CARD(RESULT(L, TO.OF(L, I, K))))
        ∧ P ∈ SAFE.FOR(DW.OF(I, K))
              ⊃
        (V.T ∈ V.INPUTS.A3(K, I, P)  ≡  V.T ∈ V.INPUTS.A2(I, K))

prove RP.L5 [L ← SOURCE(V.T)]
using RP.A3A
      IO.A2A
      RP.L4 [V ← VALUE(V.T),
             L ← SOURCE(V.T)]
```

```
/* As RP.L5 */

RP.L6: formula
        (∀ L: L ∈ INPUTS(K)  ⊃  1 = CARD(RESULT(L, TO.OF(L, I, K))))
        ∧ P ∈ SAFE.FOR(DW.OF(I, K))
             ⊃
        V.INPUTS.A2(I, K) = V.INPUTS.A3(K, I, P)

prove RP.L6 [L ← *L:2]
using SETEQUALITY [S2 ← V.INPUTS.A3(K, I, P),
                   S1 ← V.INPUTS.A2(I, K),
                   X ← *X:1]
      RP.L5 [V.T ← *X:1]
```

```
/*
If a processor is Safe for the Data Window of a task, it
         is Safe for the Execution Windows of each of that task's
         input tasks.  Needed to prove RP.L4
*/

RP.L7: formula
        P ∈ SAFE.FOR(DW.OF(I, K)) ∧ L ∈ INPUTS(K)
            ⊃
        P ∈ SAFE.FOR(OF(TO.OF(L, I, K), L))

prove RP.L7
using RP.D10 [T ← *T:1,
              II ← OF(TO.OF(L, I, K), L)]
      RP.D10 [T ← *T:1,
              II ← DW.OF(I, K)]
      RP.L2A [T ← *T:1]
```

```
/*
If a processor executes a task, and is Safe for the data
        window of that task, and if all the inputs to the
        task are well behaved, then the task output computed by
        that processor will be the result of applying the task function
        to the correct task inputs.
*/

RP.L8: formula
        P ∈ POLL.FOR.OF(I, K) ∩ SAFE.FOR(DW.OF(I, K))
        ∧ (∀ L: L ∈ INPUTS(K)  ⊃  1 = CARD(RESULT(L, TO.OF(L, I, K))))
             ⊃
        SINGLETON(ON(K, I, P), APPLY(FUNCTION(K), V.INPUTS.A2(I, K)))

prove RP.L8 [L ← *L:2]
using INTERSECT [S1 ← SAFE.FOR(DW.OF(I, K)),
                 S ← POLL.FOR.OF(I, K),
                 X ← P]
      RP.L6
      RP.A3
```

```
/*
...and that output value will be the broadcast value received
         by all processors that are Safe for the execution window of
         the task
*/

RP.L9: formula
        P ∈ POLL.FOR.OF(I, K) ∩ SAFE.FOR(DW.OF(I, K))
        ∧ (∀ L: L ∈ INPUTS(K)  ⊃  1 = CARD(RESULT(L, TO.OF(L, I, K))))
        ∧ QQ ∈ SAFE.FOR(OF(I, K))
             ⊃
        ON.IN(K, I, P, QQ) = APPLY(FUNCTION(K), V.INPUTS.A2(I, K))

prove RP.L9 [L ← *L:5]
using INTERSECT [S1 ← SAFE.FOR(DW.OF(I, K)),
                 S ← POLL.FOR.OF(I, K),
                 X ← P]
      CARD.2 [X1 ← ON.IN(K, I, P, QQ),
              X ← APPLY(FUNCTION(K), V.INPUTS.A2(I, K)),
              S ← ON(K, I, P)]
      RP.L3
      RP.L10
      RP.L8
      RP.A2 [V ← ON.IN(K, I, P, QQ)]
```

```
/*
A result value received from a Safe processor is a member
        of the set of all computer result values
*/

RP.L11: formula
        QQ ∈ SAFE.FOR(OF(I, K)) ∧ D.P ∈ D.BAG.L10(K, I, QQ, Y)
            ⊃
        D.P ∈ D.BAG.D4(K, I, QQ, Y)

prove RP.L11
using INTERSECT [S1 ← SAFE.FOR(DW.OF(I, K)),
                 S ← POLL.FOR.OF(I, K),
                 X ← *P:2]
      RP.D11 [P ← @:D]
      RP.D4A [P ← *P:2]
```

```
var D.P.1: PAIR.OF(DATAVAL, PROC)

D.BAG.L12: (TASK, ITERATION, PROC, NAT) → SET.OF(PAIR.OF(DATAVAL, PROC))

/*
Set abstraction: A result value received from
         a Safe processor is a member of the
         set of result values to be voted on
*/

RP.L12A: axiom
        D.P.1 ∈ D.BAG.L12(K, I, QQ, Y)
            ≡
        SEQ.ELEM(APPLY(FUNCTION(K), V.INPUTS.A2(I, K)), Y) = VALUE(D.P.1)
        ∧ D.P.1 ∈ D.BAG.D4(K, I, QQ, Y)
```

```
/*
If a processor is Safe for the executio window of a task,
        and that generates a singleton result value, then
        the result values received from Safe processors by
        that processor will be correct values
*/

RP.L12R: formula
        QQ ∈ SAFE.FOR(OF(I, K))
        ∧ D.P ∈ D.BAG.L10(K, I, QQ, Y)
        ∧ (∀ L: L ∈ INPUTS(K)  ⊃  1 = CARD(RESULT(L, TO.OF(L, I, K))))
            ⊃
        D.P ∈ D.BAG.L12(K, I, QQ, Y)

prove RP.L12R [L ← *L:3]
using RP.L12A [D.P.1 ← D.P]
      RP.L11
      RP.L9 [P ← *P:4]
      RP.D11 [P ← @:D]
```

```
/* as RP.L12R */

RP.L13: formula
        QQ ∈ SAFE.FOR(OF(I, K))
        ∧ (∀ L: L ∈ INPUTS(K)  ⊃  1 = CARD(RESULT(L, TO.OF(L, I, K))))
             ⊃
        D.BAG.L10(K, I, QQ, Y) ⊆ D.BAG.L12(K, I, QQ, Y)

prove RP.L13 [L ← *L:2]
using SUBSET [S2 ← D.BAG.L12(K, I, QQ, Y),
              X ← *X:1,
              S1 ← D.BAG.L10(K, I, QQ, Y)]
      RP.L12R [D.P ← *X:1]
```

```
/*
A time within the execution window of an input
        task to a task K lies within the data window of task K.
        Used to prove RP.L7
*/

RP.L2A: formula
        BEGIN(OF(TO.OF(L, I, K), L)) ≤ T
        ∧ T < END(OF(TO.OF(L, I, K), L))
        ∧ L ∈ INPUTS(K)
             ⊃
        BEGIN(DW.OF(I, K)) ≤ T ∧ T < END(DW.OF(I, K))

prove RP.L2A
using RP.A1.1
      RP.D3.3
      RP.D3.1
      RP.D2.1
      RP.D1
```

```
/*
If a task executes and is Safe, at least one processor must have
        been Safe for its execution window
*/

RP.L16: formula
        ON.DURING(K, I) ∧ TASK.SAFE(K, I)  ⊃  CARD(SAFE.FOR(OF(I, K))) > 0

prove RP.L16
using CARD.INTERSECTION [S1 ← SAFE.FOR(DW.OF(I, K)),
                         S ← POLL.FOR.OF(I, K)]
      CARD.SUBSET [S2 ← SAFE.FOR(OF(I, K)),
                   S1 ← SAFE.FOR(DW.OF(I, K))]
      SUBSET [S2 ← SAFE.FOR(OF(I, K)),
              X ← *X:3,
              S1 ← SAFE.FOR(DW.OF(I, K))]
      RP.L3 [P ← *X:3]
      RP.D9A
      RP.D7
```

```
/*
A Primary Lemma.  If a task executes and is Safe, and if
        all its inputs are will behaved, a Safe processor voting on the
        broadcast results will obtain the correct result value for
        that task
*/

RP.L14: formula
        TASK.SAFE(K, I)
        ∧ ON.DURING(K, I)
        ∧ QQ ∈ SAFE.FOR(OF(I, K))
        ∧ (∀ L: L ∈ INPUTS(K)  ⊃  1 = CARD(RESULT(L, TO.OF(L, I, K))))
        ∧ 1 ≤ Y
        ∧ Y ≤ RESULT.SIZE(K)
             ⊃
        SEQ.ELEM(APPLY(FUNCTION(K), V.INPUTS.A2(I, K)), Y)
           = MAJORITY(D.BAG.D4(K, I, QQ, Y))

prove RP.L14 [L ← *L:1]
using RP.L13
      RP.D9A
      CARD.D.BAG.D4
      CARD.D.BAG.L10
      CARD.SUBSET [S2 ← D.BAG.L12(K, I, QQ, Y),
                   S1 ← D.BAG.L10(K, I, QQ, Y)]
      MAJ.1 [T1.V ← SEQ.ELEM(APPLY(FUNCTION(K), V.INPUTS.A2(I, K)), Y),
             M.BAG.1 ← D.BAG.L12(K, I, QQ, Y),
             M.BAG ← D.BAG.D4(K, I, QQ, Y)]
      RP.L12A [D.P.1 ← *V1.V2:6]
```

```
/* ... and will place that result value in its IN buffer */

RP.L15: formula
        TASK.SAFE(K, I)
        ∧ ON.DURING(K, I)
        ∧ QQ ∈ SAFE.FOR(OF(I, K))
        ∧ (∀ L: L ∈ INPUTS(K)  ⊃  1 = CARD(RESULT(L, TO.OF(L, I, K))))
             ⊃
        APPLY(FUNCTION(K), V.INPUTS.A2(I, K)) = IN(K, I, QQ)

prove RP.L15 [L ← *L:1]
using RP.L14 [Y ← *Y:3]
      RP.D4 [Y ← *Y:3]
      DATA.EQUALITY [V ← IN(K, I, QQ),
                     Y ← @:D,
                     V1 ← APPLY(FUNCTION(K), V.INPUTS.A2(I, K))]
      DATA.SIZE.IS.SEQ.LENGTH
      RESULT.SIZE.GREATER.THAN.1
```

```
var L1: TASK

/*
Almost there: If a task executes and is Safe, and all its
         inputs are well behaved, its result will be the result of applying
         its function to the correct inputs
*/

RP.L17: formula
        TASK.SAFE(K, I)
        ∧ ON.DURING(K, I)
        ∧ (∀ L: L ∈ INPUTS(K)  ⊃  1 = CARD(RESULT(L, TO.OF(L, I, K))))
        ∧ (∀ L1: L1 ∈ INPUTS(K)  ⊃  1 = CARD(RESULT(L1, TO.OF(L1, I, K))))
             ⊃
        SINGLETON(RESULT(K, I), APPLY(FUNCTION(K), V.INPUTS.A2(I, K)))

prove RP.L17 [L ← *L:6,
              L1 ← *L:7]
using CARD.3 [V.CARD.3 ← APPLY(FUNCTION(K), V.INPUTS.A2(I, K)),
              S ← RESULT(K, I)]
      RP.L16
      CARD.4 [S ← SAFE.FOR(OF(I, K))]
      RP.D6 [P ← *X:3,
             V ← APPLY(FUNCTION(K), V.INPUTS.A2(I, K))]
      RP.D6 [V ← *X:1,
             P ← @:D]
      RP.L15 [QQ ← *X:3]
      RP.L15 [QQ ← *P:5]
```

```
/*
The number of versins of a task's result available for voting
         on is the number of processors executing that task
*/

CARD.D.BAG.D4: formula
        CARD(D.BAG.D4(K, I, QQ, Y)) = CARD(POLL.FOR.OF(I, K))

/*
The number of correct versions of a task's result is the
        number of Safe processors executing that task
*/

CARD.D.BAG.L10: formula
        CARD(D.BAG.L10(K, I, QQ, Y))
           = CARD(POLL.FOR.OF(I, K) ∩ SAFE.FOR(DW.OF(I, K)))

var L2: TASK

NECESSARY.EVIL: formula
        (∀ L: L ∈ INPUTS(K)  ⊃  1 = CARD(RESULT(L, TO.OF(L, I, K))))
             ⊃
        (∀ L1: L1 ∈ INPUTS(K)  ⊃  1 = CARD(RESULT(L1, TO.OF(L1, I, K))))
        ∧ (∀ L2: L2 ∈ INPUTS(K)  ⊃  1 = CARD(RESULT(L2, TO.OF(L2, I, K))))
```

```
/* We now consider tasks that are not currently being executed */

/*
If a task is executed and safe, and has an input task
        that is not being executed, a majority of the result values
        for that not.on task will be nulls
*/

RP.L19: formula
        L ∈ INPUTS(K)
        ∧ ON.DURING(K, I)
        ∧ TASK.SAFE(K, I)
        ∧ ¬ON.DURING(L, TO.OF(L, I, K))
        ∧ 1 ≤ Y
        ∧ Y ≤ RESULT.SIZE(L)
            ⊃
        BOTTOMD() = MAJORITY(D.BAG.D4(L, TO.OF(L, I, K), QQ, Y))

prove RP.L19
using MAJ.2 [M.BAG ← D.BAG.D4(L, TO.OF(L, I, K), QQ, Y),
             T2.V ← D1:2]
      BOTTOM.EQUALITY
      CARD.D.BAG.D4 [K ← L,
                     I ← TO.OF(L, I, K)]
      CARD.6 [S ← POLL.FOR.OF(TO.OF(L, I, K), L)]
      RP.D7 [K ← L,
             I ← TO.OF(L, I, K)]
```

```
/*
... and on a safe processor that null value will
        be placed in the IN buffer
*/

RP.L20: formula
        L ∈ INPUTS(K)
        ∧ ON.DURING(K, I)
        ∧ TASK.SAFE(K, I)
        ∧ ¬ON.DURING(L, TO.OF(L, I, K))
        ∧ QQ ∈ SAFE.FOR(OF(TO.OF(L, I, K), L))
        ∧ 1 ≤ Y
        ∧ Y ≤ RESULT.SIZE(L)
             ⊃
        SEQ.ELEM(BOTTOM1(L), Y) = SEQ.ELEM(IN(L, TO.OF(L, I, K), QQ), Y)

prove RP.L20
using RP.L19
      DATA.BOTTOM [K ← L]
      DATA.EQUALITY [V ← IN(L, TO.OF(L, I, K), QQ),
                     V1 ← BOTTOM1(L)]
      RP.D4 [K ← L,
             I ← TO.OF(L, I, K)]
```

```
/* as RP.L20 */

RP.L21: formula
        L ∈ INPUTS(K)
        ∧ ON.DURING(K, I)
        ∧ TASK.SAFE(K, I)
        ∧ ¬ON.DURING(L, TO.OF(L, I, K))
        ∧ QQ ∈ SAFE.FOR(OF(TO.OF(L, I, K), L))
             ⊃
        BOTTOM1(L) = IN(L, TO.OF(L, I, K), QQ)

prove RP.L21
using RP.L20 [Y ← *Y:2]
      DATA.EQUALITY [V ← IN(L, TO.OF(L, I, K), QQ),
                     Y ← @:D,
                     V1 ← BOTTOM1(L)]
      DATA.SIZE.IS.SEQ.LENGTH [K ← L,
                               I ← TO.OF(L, I, K)]
```

```
/*
Using the above chain of lemmas, we now prove the main
        Execute axiom of the I/O specification ...
*/

prove IO.A2 [L ← *L:2]
using RP.L17
      NECESSARY.EVIL [L1 ← *L:1,
                      L2 ← *L1:1]
```

```
/*
... and the I/O axiom about tasks that are NOT.ON
        during that iteration.
*/

prove IO.A5
using RP.D9A [K ← L,
              I ← TO.OF(L, I, K)]
      RP.L21 [QQ ← *X:9]
      RP.D9A
      RP.L21 [QQ ← *P:6]
      RP.D6 [K ← L,
             I ← TO.OF(L, I, K),
             V ← BOTTOM1(L),
             P ← *X:9]
      RP.D6 [K ← L,
             I ← TO.OF(L, I, K),
             P ← @:D,
             V ← *X:7]
      CARD.3 [V.CARD.3 ← BOTTOM1(L),
              S ← RESULT(L, TO.OF(L, I, K))]
      RP.L21 [QQ ← *X:9]
      CARD.4 [S ← SAFE.FOR(DW.OF(I, K))]
      RP.L7 [P ← *X:9]
      CARD.INTERSECTION [S ← POLL.FOR.OF(I, K),
                         S1 ← SAFE.FOR(DW.OF(I, K))]
      RP.D7
```

SUBSECTION 7.11

SCHEDULE AXIOMS

```
/*
********** Specification And Constraints
                For Schedule Tables            **********
*/

SCHED: (CONFIGS, SUBFRAMETIME, PROC) → SEQ(ACTIVITY)

/*
If L provides input to K, then SSF(L,K) iff all votes on
        L precede Execute on K
*/

BR.A14: axiom
        1 ≤ Y
        ∧ L ∈ INPUTS(K)
        ∧ Y ≤ RESULT.SIZE(L)
        ∧ ACTION(ACTIV2) = VOTE()
        ∧ TASK.ACTION(ACTIV2) = L
        ∧ ELEM.ACTION(ACTIV2) = Y
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
             ⊃
        (SSF(L, K)
             ≡
         (SEQ.MEMBER(ACTIV, SCHED(CON, T.SUB, P))
              ≡
          SEQ.MEMBER(ACTIV2, SCHED(CON, T.SUB, P)))
         ∧ (∃ W:
              SEQ.ELEM(SCHED(CON, T.SUB, P), W) = ACTIV2
              ∧ (∀ Z: Z > W  ⊃  ¬(ACTIV = SEQ.ELEM(SCHED(CON, T.SUB, P), Z)))))

/*
If a vote on L provides input to K in a different frame, there will
        be no subsequent votes on L prior to the Execution of K
*/

var CON1: CONFIGS

BR.A25A: axiom
        L ∈ INPUTS(K)
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ SEQ.MEMBER(ACTIV, SCHED(CON, T.SUB, P))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ START.FRAME(FRAME(T1.SUB)) = TO.OF(L, I, K)*FRAME.SIZE()
        ∧ SEQ.MEMBER(ACTIV2, SCHED(CON2, T1.SUB, P))
        ∧ ((ACTION(ACTIV2) = VOTE() ∧ ELEM.ACTION(ACTIV2) = Y)
           ∨ ACTION(ACTIV2) = DUMMY.VOTE())
        ∧ TASK.ACTION(ACTIV2) = L
        ∧ FRAME(T.SUB) = FRAME(T2.SUB)
        ∧ T1.SUB < T2.SUB
        ∧ T2.SUB ≤ T.SUB
             ⊃
        ¬(∃ ACTIV3:
             TASK.ACTION(ACTIV3) = L
             ∧ ((ACTION(ACTIV3) = VOTE() ∧ ELEM.ACTION(ACTIV3) = Y)
                ∨ ACTION(ACTIV3) = DUMMY.VOTE())
             ∧ SEQ.MEMBER(ACTIV3, SCHED(CON, T2.SUB, P)))
```

```
/* In a subframe, an Execute for K implies no Vote for K  */

BR.A11: axiom
        SEQ.MEMBER(ACTIV, SCHED(CON, T.SUB, P))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
             ⊃
        ¬(SEQ.MEMBER(ACTIV2, SCHED(CON, T.SUB, QQ))
          ∧ ACTION(ACTIV2) = VOTE()
          ∧ TASK.ACTION(ACTIV2) = K)

/* There will be exactly one Execute on K in a Frame */

BR.A12: axiom
        SEQ.MEMBER(ACTIV, SCHED(CON, T.SUB, P))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
             ⊃
        ¬(¬(T1.SUB = T.SUB)
          ∧ FRAME(T1.SUB) = FRAME(T.SUB)
          ∧ ACTION(ACTIV3) = EXECUTE()
          ∧ TASK.ACTION(ACTIV3) = K
          ∧ SEQ.MEMBER(ACTIV3, SCHED(CON, T1.SUB, P)))

/*
There will not be more than one subframe in a Frame
        containing either a Vote or a Dummy Vote
*/

BR.A12A: axiom
        SEQ.MEMBER(ACTIV, SCHED(CON, T.SUB, P))
        ∧ ((ACTION(ACTIV) = VOTE() ∧ ELEM.ACTION(ACTIV) = Y)
           ∨ ACTION(ACTIV) = DUMMY.VOTE())
        ∧ TASK.ACTION(ACTIV) = K
        ∧ FRAME(T1.SUB) = FRAME(T.SUB)
        ∧ TASK.ACTION(ACTIV3) = K
        ∧ ((ACTION(ACTIV3) = VOTE() ∧ ELEM.ACTION(ACTIV3) = Y)
           ∨ ACTION(ACTIV3) = DUMMY.VOTE())
        ∧ SEQ.MEMBER(ACTIV3, SCHED(CON, T1.SUB, P))
             ⊃
        T.SUB = T1.SUB

var T2.SUB: SUBFRAMETIME

/* In no subframe will there be both a Vote and a Dummy Vote */

BR.A12B: axiom
        ¬(SEQ.MEMBER(ACTIV, SCHED(CON, T.SUB, P))
          ∧ ACTION(ACTIV) = VOTE()
          ∧ TASK.ACTION(ACTIV) = K
          ∧ SEQ.MEMBER(ACTIV2, SCHED(CON, T.SUB, P))
          ∧ ACTION(ACTIV2) = DUMMY.VOTE()
          ∧ TASK.ACTION(ACTIV2) = K)
```

```
/* In any frame there will be either a Vote or a Dummy Vote */

BR.A12C: axiom
        ∃ T.SUB, ACTIV:
           START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
           ∧ SEQ.MEMBER(ACTIV, SCHED(CON, T.SUB, P))
           ∧ TASK.ACTION(ACTIV) = K
           ∧ ((ACTION(ACTIV) = VOTE() ∧ (∃ Y: ELEM.ACTION(ACTIV) = Y))
              ∨ ACTION(ACTIV) = DUMMY.VOTE())

/*
If there is an Execute on K, there will be a later vote
        on every processor for every output element
*/

BR.A13A: axiom
        SEQ.MEMBER(ACTIV, SCHED(CON, T.SUB, P))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
             ⊃
        (∀ Y:
            1 ≤ Y
            ∧ Y ≤ RESULT.SIZE(K)
            ∧ (∃ T1.SUB:
                  (∃ ACTIV2:
                      T1.SUB > T.SUB
                      ∧ FRAME(T1.SUB) = FRAME(T.SUB)
                      ∧ SEQ.MEMBER(ACTIV2, SCHED(CON, T1.SUB, QQ))
                      ∧ ACTION(ACTIV2) = VOTE()
                      ∧ TASK.ACTION(ACTIV2) = K
                      ∧ ELEM.ACTION(ACTIV2) = Y)))

/* If no processor Executes K, no one will Vote on K */

BR.A13B: axiom
        (∀ T.SUB, P, ACTIV:
            ¬(START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
              ∧ SEQ.MEMBER(ACTIV, SCHED(CON, T.SUB, P))
              ∧ ACTION(ACTIV) = EXECUTE()
              ∧ TASK.ACTION(ACTIV) = K))
             ⊃
        ¬(SEQ.MEMBER(ACTIV2, SCHED(CON, T1.SUB, QQ))
          ∧ ACTION(ACTIV2) = VOTE()
          ∧ TASK.ACTION(ACTIV2) = K
          ∧ ELEM.ACTION(ACTIV2) = Y
          ∧ START.FRAME(FRAME(T1.SUB)) = I*FRAME.SIZE())

BR.A44: axiom
        SEQ.MEMBER(ACTIV, SCHED(CON, T.SUB, P))
        ∧ ACTION(ACTIV) = VOTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ ELEM.ACTION(ACTIV) = Y
             ⊃
        1 ≤ Y ∧ Y ≤ RESULT.SIZE(K)
```

```
/* A Vote, Dummy Vote or an Execute will be within the Execution
        Window
*/

BR.A16: axiom
        START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ SEQ.MEMBER(ACTIV, SCHED(CON, T.SUB, P))
        ∧ (ACTION(ACTIV) = EXECUTE()
           ∨ ACTION(ACTIV) = VOTE()
           ∨ ACTION(ACTIV) = DUMMY.VOTE())
        ∧ TASK.ACTION(ACTIV) = K
             ⊃
        BEGIN(OF(I, K)) ≤ T.SUB ∧ T.SUB < END(OF(I, K))

BR.A28: axiom
        BEGIN(DW.FOR.TO.OF(L, I, K))
           = if L ∈ INPUTS(K)
                then BEGIN(OF(TO.OF(L, I, K), L))
                else BEGIN(OF(I, K))
             end if

BR.A29: axiom
        END(DW.FOR.TO.OF(L, I, K)) = END(OF(I, K))

BR.A30: axiom
        ¬(BEGIN(DW.FOR.TO.OF(L, I, K)) < BEGIN(DW.OF(I, K)))

BR.A31: axiom
        END(DW.OF(I, K)) = END(OF(I, K))

LAST: (SUBFRAMETIME, TASK) → ITERATION

/* The Last Subframe finished last and the next has not */

BR.A33: axiom
        I = LAST(T.SUB, K)
            ≡
        END(OF(I, K)) ≤ T.SUB ∧ END(OF(INCR(I), K)) > T.SUB

/*
The Last iteration of the Global Exec is constant over
        an Execution Window
*/

BR.A36: axiom
        BEGIN(OF(I, K)) ≤ T.SUB
        ∧ T.SUB < END(OF(I, K))
        ∧ BEGIN(OF(I, K)) ≤ T2.SUB
        ∧ T2.SUB < END(OF(I, K))
             ⊃
        LAST(T.SUB, GLOBAL.EXEC()) = LAST(T2.SUB, GLOBAL.EXEC())
```

```
/* L to I of K is the last completed iteration of L */

BR.A40: axiom
        (L ∈ INPUTS(K) ∧ ¬SSF(L, K)
              ⊃
         ¬(BEGIN(OF(I, K)) < END(OF(TO.OF(L, I, K), L)))
         ∧ BEGIN(OF(I, K)) < END(OF(INCR(TO.OF(L, I, K)), L)))
        ∧ (L ∈ INPUTS(K) ∧ SSF(L, K)
                ⊃
           END(OF(TO.OF(L, I, K), L)) = SUB.INCR(BEGIN(OF(I, K))))
```

SUBSECTION 7.12

PACT AXIOMS

```
/* ********** Activity Specification ********** */

using PAIROF.STP

using SETS.AXIOMS

using SEQ.STP

/* THE FOLLOWING DST IS NECESSARY TO MAP TO THE PRE/POST SPEC */

DATAVAL: type is NAT

ACTIVITIES: type is NAT

CONFIGS: type is DATAVAL

DATA: type is SEQ(DATAVAL)

FRAMETIME: type is INTEGER

var CON2: CONFIGS

using SUBFRAME.AXIOMS

INTERVAL: type is PAIR.OF(SUBFRAMETIME, SUBFRAMETIME)

var INTERVAL1: INTERVAL

using MAJORITY.STP

VALUE: (PAIR1) → TYPE1 = FIRST(PAIR1)

SOURCE: (PAIR1) → TYPE2 = SECOND(PAIR1)

var T.FRAME: FRAMETIME

FRAME: (SUBFRAMETIME) → FRAMETIME

FRAME.INCR: (T.FRAME) → FRAMETIME = T.FRAME+1

START.FRAME: (FRAMETIME) → SUBFRAMETIME

START: (SUBFRAMETIME, PROC) → REALTIME

var T.REAL: REALTIME

var T1.REAL: REALTIME

RPLUS: (T.REAL, T1.REAL) → REALTIME = T.REAL+T1.REAL

INT.TO.REALTIME: (INT1) → REALTIME = INT1

RESULT.SIZE: (TASK) → NAT

FRAME.SIZE: → NAT
```

```
BR.A27: axiom
        FRAME.SIZE() > 0

ITERATION.START: (I) → SUBFRAMETIME = I*FRAME.SIZE()

OVERHEAD: → NAT

BROADCAST.DELAY: → NAT

MAX.INTERVAL: → NAT

MIN.INTERVAL: → NAT

REAL.SAFE: (REALTIME) → SET.OF(PROC)

ACT.SAFE: (SUBFRAMETIME) → SET.OF(PROC)

WORKING: (SUBFRAMETIME) → SET.OF(PROC)

/*
A processor is safe for a subframe if it is
        safe for all contained 'real times'
*/

BR.A32: axiom
        P ∈ WORKING(T.SUB)
             ≡
        (∀ T.REAL:
            START(T.SUB, P) ≤ T.REAL
            ∧ T.REAL ≤ START(SUB.INCR(T.SUB), P)
            ∧ P ∈ REAL.SAFE(T.REAL))

/* State Components */

INPUTIN.OF: (PROC, TASK, REALTIME) → DATA

DATAFILEIN.FOR.ON: (PROC, TASK, PROC, REALTIME) → DATA

/* ------------------ */

/* Declaration of Activity and Activities Types */

ACTIVITIES: type is NAT

ACTIVITY: type

var ACTIV: ACTIVITY

var ACTIV3: ACTIVITY

VOTE: → ACTIVITIES

EXECUTE: → ACTIVITIES

DUMMY.VOTE: → ACTIVITIES

var ACT: ACTIVITIES
```

```
ACTION: (ACTIVITY) → ACTIVITIES

TASK.ACTION: (ACTIVITY) → TASK

ELEM.ACTION: (ACTIVITY) → NAT

/* -------------------- */

var D1: DATAVAL

CONFIG.FIELD: → NAT

/*
One of the fields in the output of the Global Executive
        is a configuration field
*/

GE.CONFIG.FIELD: axiom
        1 ≤ CONFIG.FIELD() ∧ CONFIG.FIELD() ≤ RESULT.SIZE(GLOBAL.EXEC())

/*
A processor will use the configuration field in the
        voted output of the Global Exec in determining current
        configuration.
*/

CONFIG: (T.SUB, P) → CONFIGS
        = SEQ.ELEM(INPUTIN.OF(P, GLOBAL.EXEC(), START(T.SUB, P)),
                   CONFIG.FIELD())

WORKING.DURING: (P, T.SUB) → BOOL = P ∈ WORKING(T.SUB)

SCHED: (CONFIGS, SUBFRAMETIME, PROC) → SEQ(ACTIVITY)

/*
The following axioms and derived functions introduce
        the definition of real time and its relation to subframe
        time.
*/

FINISH: (T.SUB, P) → REALTIME = START(SUB.INCR(T.SUB), P)-OVERHEAD()

BR.A1.A: axiom
        FINISH(T.SUB, P) ≥ START(T.SUB, P)

BR.A1.C: axiom
        START(T.SUB, P)+MIN.INTERVAL() ≤ START(SUB.INCR(T.SUB), P)
        ∧ START(SUB.INCR(T.SUB), P) ≤ START(T.SUB, P)+MAX.INTERVAL()

BR.A1.D: axiom
        BROADCAST.DELAY() < OVERHEAD()

BR.A1.E: axiom
        2*OVERHEAD() < MIN.INTERVAL() ∧ MIN.INTERVAL() ≤ MAX.INTERVAL()

var CON: CONFIGS
```

```
/*
Frame Axiom for INPUT (the 'post vote' buffer) -- if there
        is no Vote on a task K scheduled on a processor which is working during
        that subframe, the INPUT value for K will be unchanged.
*/

BR.A6A: axiom
        WORKING.DURING(P, T.SUB)
        ∧ 1 ≤ Y
        ∧ Y ≤ RESULT.SIZE(K)
        ∧ ¬(∃ ACTIV:
               SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, P), T.SUB, P))
               ∧ ((ACTION(ACTIV) = VOTE()
                   ∧ TASK.ACTION(ACTIV) = K
                   ∧ ELEM.ACTION(ACTIV) = Y)
                  ∨ (ACTION(ACTIV) = DUMMY.VOTE() ∧ TASK.ACTION(ACTIV) = K)))
             ⊃
        SEQ.ELEM(INPUTIN.OF(P, K, START(T.SUB, P)), Y)
           = SEQ.ELEM(INPUTIN.OF(P, K, START(SUB.INCR(T.SUB), P)), Y)

BOTTOM1: (TASK) → DATA

/*
Dummy Vote axiom -- if a working processor has a Dummy Vote
        scheduled for a task K, the INPUT for K at the beginning of
        the following subframe will be BOTTOM.
*/

BR.A6B: axiom
        WORKING.DURING(P, T.SUB)
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, P), T.SUB, P))
        ∧ ACTION(ACTIV) = DUMMY.VOTE()
        ∧ TASK.ACTION(ACTIV) = K
             ⊃
        INPUTIN.OF(P, K, START(SUB.INCR(T.SUB), P)) = BOTTOM1(K)

NOT.ON.FRAME: (TASK, PROC, SUBFRAMETIME) → BOOL

/*
Definition of NOT.ON.FRAME -- a task K is not on during
        a frame, as determined by processor Q at time T.SUB, if
        using his perception of the configuration, there is no
        processor which is scheduled to Execute K during that
        frame.
*/

BR.A22: axiom
        ¬NOT.ON.FRAME(K, QQ, T.SUB)
             ≡
        (∃ T1.SUB, P, ACTIV:
            FRAME(T.SUB) = FRAME(T1.SUB)
            ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T1.SUB, P))
            ∧ ACTION(ACTIV) = EXECUTE()
            ∧ TASK.ACTION(ACTIV) = K)

POLLBY.FOR: (PROC, TASK, SUBFRAMETIME) → SET.OF(PROC)
```

```
/*
Definition of POLLBY -- a processor R is determined
            by Processor P at time T1.SUB to be in the poll set
            for task L iff, according to his configuration at
            that time there is an Execute on R scheduled for that
            frame.
*/

BR.A9A: axiom
        R ∈ POLLBY.FOR(P, L, T1.SUB)
             ≡
        (∃ T.SUB, ACTIV:
            T.SUB < T1.SUB
            ∧ FRAME(T.SUB) = FRAME(T1.SUB)
            ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, P), T.SUB, R))
            ∧ ACTION(ACTIV) = EXECUTE()
            ∧ TASK.ACTION(ACTIV) = L)

D.BAG.A9C: (PROC, TASK, SUBFRAMETIME, NAT) → SET.OF(PAIR.OF(DATAVAL, PROC))

var D.P: PAIR.OF(DATAVAL, PROC)

/*
The set of <processor,dataval> pairs for broadcast output
        from processors in the poll set -- as determined by processor P at
        time T.SUB.
*/

SET.ABSTRACTION.A9C: axiom
        ∀ D.P:
           D.P ∈ D.BAG.A9C(P, K, T.SUB, Y)
                ≡
           SOURCE(D.P) ∈ POLLBY.FOR(P, K, T.SUB)
           ∧ VALUE(D.P)
                = SEQ.ELEM(DATAFILEIN.FOR.ON(P, K, SOURCE(D.P),
                                            START(T.SUB, P)),
                           Y)

D.BAG.MAJ:
   (PROC, TASK, SUBFRAMETIME, NAT, DATAVAL) → SET.OF(PAIR.OF(DATAVAL, PROC))

SET.ABSTRACTION.MAJ: axiom
        ∀ D.P2:
           D.P2 ∈ D.BAG.MAJ(P, K, T.SUB, Y, D2)
                ≡
           D.P2 ∈ D.BAG.A9C(P, K, T.SUB, Y) ∧ D2 = VALUE(D.P2)
```

```
/*
Definition of Vote activity -- a working processor scheduled
        to perform a Vote will have placed the majority of values
        from processors in its poll.by set into INPUT.
*/

BR.A9C: axiom
        WORKING.DURING(P, T.SUB)
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, P), T.SUB, P))
        ∧ ACTION(ACTIV) = VOTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ ELEM.ACTION(ACTIV) = Y
             ⊃
        SEQ.ELEM(INPUTIN.OF(P, K, START(SUB.INCR(T.SUB), P)), Y)
           = MAJORITY(D.BAG.A9C(P, K, T.SUB, Y))

var V.INPUTS: SET.OF(PAIR.OF(DATA, TASK))

/*
Definition of  Execute activity --  Given a working processor QQ
        scheduled to Execute task K and (another) working processor P
        to receive the broadcast, the result of applying the correct
        mathematical function to the Inputs will, according to QQ's
        clock, be present in P at the end of the current subframe plus
        the broadcast delay.  N.B.  the timing is w.r.t. to processor
        performing and broadcasting the computation.
*/

BR.A41: formula
        WORKING.DURING(P, T.SUB)
        ∧ WORKING.DURING(QQ, T.SUB)
        ∧ (∀ V.T:
              V.T ∈ V.INPUTS
                   ≡
              SOURCE(V.T) ∈ INPUTS(K)
              ∧ VALUE(V.T)
                   = INPUTIN.OF(QQ, SOURCE(V.T), START(SUB.INCR(T.SUB), QQ)))
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
             ⊃
        DATAFILEIN.FOR.ON(P, K, QQ,
                          RPLUS(INT.TO.REALTIME(BROADCAST.DELAY()),
                                FINISH(T.SUB, QQ)))
           = APPLY(FUNCTION(K), V.INPUTS)
```

```
/*
Frame axiom for the Datafile -- If processors P and QQ are
        working during a subframe in which QQ is not to Execute K,
        the datafile in P should remain unchanged from the end of
        the previous subframe + broadcast delay until the end of the
        current subframe +broadcast delay (according to QQ's clock)
*/

BR.A42: formula
        WORKING.DURING(P, T.SUB)
        ∧ WORKING.DURING(QQ, T.SUB)
        ∧ ¬(∃ ACTIV:
               SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
               ∧ ACTION(ACTIV) = EXECUTE()
               ∧ TASK.ACTION(ACTIV) = K)
        ∧ RPLUS(INT.TO.REALTIME(BROADCAST.DELAY()), FINISH(SUB.DECR(T.SUB), QQ))
             ≤ T.REAL
        ∧ T.REAL ≤ RPLUS(INT.TO.REALTIME(BROADCAST.DELAY()), FINISH(T.SUB, QQ))
             ⊃
        DATAFILEIN.FOR.ON(P, K, QQ, T.REAL)
           = DATAFILEIN.FOR.ON(P, K, QQ,
                                      RPLUS(INT.TO.REALTIME(BROADCAST.DELAY()),
                                            FINISH(SUB.DECR(T.SUB), QQ)))

VOTE.COST: → NAT

EXECUTE.COST: (TASK) → NAT

COST: (ACTIV) → NAT
        = if ACTION(ACTIV) = VOTE()
             then VOTE.COST()
             else EXECUTE.COST(TASK.ACTION(ACTIV))
          end if

SEQ.COST: (SEQ(ACTIVITY)) → NAT

var SEQ.ACTIV: SEQ(ACTIVITY)

BR.SEQ.COST: axiom
        SEQ.COST(SEQ.CAT(SEQ.ACTIV, MAKESEQ(ACTIV)))
           = SEQ.COST(SEQ.ACTIV)+COST(ACTIV)

BR.A10: axiom
        SEQ.COST(SCHED(CON, T.SUB, P)) ≤ MIN.INTERVAL()-OVERHEAD()

var ACTIV2: ACTIVITY

/* The i-th iteration of a task will take place during the
        i-th frame.
*/

BR.A18: axiom
        START.FRAME(FRAME(BEGIN(OF(I, K)))) = I*FRAME.SIZE()
        ∧ FRAME(BEGIN(OF(I, K))) = FRAME(SUB.DECR(END(OF(I, K))))
```

```
/* The next two axioms establish the well ordering of iterations */

BR.A19.1: axiom
        SUB.INCR(BEGIN(OF(I, K))) < END(OF(I, K))

BR.A19.2: axiom
        END(OF(I, K)) ≤ BEGIN(OF(INCR(I), K))

/* Instance of Induction Scheme Over Integers */

BR.INDUCTION.SUB.TIME.1: formula
        ∀ T.SUB, T1.SUB:
           DATAFILEIN.FOR.ON(QQ, K, P, START(T.SUB, QQ)) = V
           ∧ (∀ T2.SUB:
                 T.SUB ≤ T2.SUB
                 ∧ T2.SUB < T1.SUB
                 ∧ DATAFILEIN.FOR.ON(QQ, K, P, START(T2.SUB, QQ)) = V
                     ⊃
                 DATAFILEIN.FOR.ON(QQ, K, P, START(SUB.INCR(T2.SUB), QQ)) = V)
                ⊃
           DATAFILEIN.FOR.ON(QQ, K, P, START(T1.SUB, QQ)) = V

/* Another instance ... */

BR.INDUCTION.SUB.TIME.2: formula
        ∀ T.SUB, T1.SUB:
           D1 = SEQ.ELEM(INPUTIN.OF(QQ, L, START(T.SUB, QQ)), Y)
           ∧ (∀ T2.SUB:
                 T.SUB ≤ T2.SUB
                 ∧ T2.SUB < T1.SUB
                 ∧ D1 = SEQ.ELEM(INPUTIN.OF(QQ, L, START(T2.SUB, QQ)), Y)
                     ⊃
                 D1 = SEQ.ELEM(INPUTIN.OF(QQ, L, START(SUB.INCR(T2.SUB), QQ)),
                               Y))
                ⊃
           D1 = SEQ.ELEM(INPUTIN.OF(QQ, L, START(T1.SUB, QQ)), Y)

/* Induction Scheme over Iterations - Instance of Integer Induction 1 */

BR.INDUCTION.ITERATION.1: formula
        ∀ I, J:
           J ≥ I
           ∧ BEGIN(OF(I, K)) > T.SUB
           ∧ (∀ J1: BEGIN(OF(J1, K)) > T.SUB  ⊃  BEGIN(OF(INCR(J1), K)) > T.SUB)
                ⊃
           BEGIN(OF(J, K)) > T.SUB
```

```
/*
The next three axioms establish basic properties of START.FRAME
        and FRAME functions.
*/

BR.A21A: axiom
        START.FRAME(FRAME(T.SUB)) < START.FRAME(FRAME.INCR(FRAME(T.SUB)))

BR.A21B: axiom
        START.FRAME(FRAME.INCR(T.FRAME)) = START.FRAME(T.FRAME)+FRAME.SIZE()

BR.A21: axiom
        START.FRAME(FRAME(T.SUB)) ≤ T1.SUB
        ∧ T1.SUB < START.FRAME(FRAME.INCR(FRAME(T.SUB)))
             ≡
        FRAME(T.SUB) = FRAME(T1.SUB)

/* Topology of Interactive Consistency Tasks */

BR.A20: axiom
        IC(K) ∧ SOURCE(V.T) ∈ INPUTS(K) ∧ SINGLETON(V.INPUTS, V.T)
             ⊃
        CARD(INPUTS(K)) = 1
        ∧ (L ∈ INPUTS(K)  ⊃  1 = CARD(POLL.FOR.OF(TO.OF(L, I, K), L)))
        ∧ 3 = CARD(POLL.FOR.OF(I, K))
        ∧ VALUE(V.T) = APPLY(FUNCTION(K), V.INPUTS)

/*
A processor is safe for an iteration iff it is
        safe for every subframe within the execution window.
*/

BR.D1: axiom
        (∀ T.SUB: BEGIN(II) ≤ T.SUB ∧ T.SUB < END(II)  ⊃  P ∈ SAFE(T.SUB))
             ≡
        P ∈ SAFE.FOR(II)

var ACTIV3: ACTIVITY

LAST: (SUBFRAMETIME, TASK) → ITERATION

CLOCK.SAFE: (PROC, SUBFRAMETIME) → BOOL
```

```
/*
Two processors are within skew of each other at time
        T.SUB iff the end of the last subframe + broadcast delay
        w.r.t. one processor's clock is less than the start
        of the current subframe of the other's clock.
*/

WITHIN.SKEW: (P, QQ, T.SUB) → BOOL
        = FINISH(SUB.DECR(T.SUB), P)+BROADCAST.DELAY() ≤ START(T.SUB, QQ)
          ∧ FINISH(SUB.DECR(T.SUB), QQ)+BROADCAST.DELAY() ≤ START(T.SUB, P)

/*
This lemma was proven by hand by Leslie Lamport.  It is the
        basic statement of correctness for the clock task which runs
        on each processor.
*/

BR.LEMMA.FOR.LES.TO.PROVE: formula
        CLOCK.SAFE(P, T.SUB) ∧ CLOCK.SAFE(QQ, T.SUB)
             ⊃
        WITHIN.SKEW(P, QQ, T.SUB)

var R: PROC

/* Draw a picture... */

BR.A35: axiom
        CLOCK.SAFE(P, T.SUB)
             ≡
        IC.TASK.SAFE(CLOCK(), LAST(T.SUB, CLOCK()))
        ∧ P ∈ WORKING(T.SUB)
        ∧ (∀ R:
              (∀ T1.SUB, QQ:
                  T1.SUB ≤ T.SUB
                  ∧ T1.SUB ≥ END(OF(LAST(T.SUB, CLOCK()), CLOCK()))
                  ∧ R ∈ WORKING(T1.SUB)
                  ∧ QQ ∈ SAFE(END(OF(LAST(T.SUB, CLOCK()), CLOCK())))
                  ∧ QQ ∈ WORKING(T1.SUB)
                  ∧ WITHIN.SKEW(R, QQ, T.SUB))
                   ⊃
              WITHIN.SKEW(P, R, T.SUB))

DATA.SIZE.IS.SEQ.LENGTH.2: axiom
        SEQ.LENGTH(INPUTIN.OF(P, K, T.REAL)) = RESULT.SIZE(K)
        ∧ SEQ.LENGTH(DATAFILEIN.FOR.ON(P, K, QQ, T.REAL)) = RESULT.SIZE(K)

DATA.SIZE.IS.SEQ.LENGTH.3: axiom
        SEQ.LENGTH(APPLY(FUNCTION(K), V.INPUTS.A3(K, I, QQ))) = RESULT.SIZE(K)
        ∧ SEQ.LENGTH(APPLY(FUNCTION(K), V.INPUTS)) = RESULT.SIZE(K)

LENGTH.OF.ELEMENTS.OF.ON.IS.LENGTH.OF.ON.IN: axiom
        V ∈ ON(K, I, QQ)  ⊃  SEQ.LENGTH(V) = RESULT.SIZE(K)

using SCHED.AXIOMS

using REACT.MAPPING
```

SUBSECTION 7.13

REACT MAPPING

```
/*
********** Non-Identity Mappings from Activity Model
                   to Replication Model                      **********
*/

/*
A processor QQ is in the global poll set iff there
        exists a processor P safe for the execution window who,
        according to his view of the configuration, decides at the
        time of a scheduled Vote that QQ is in his POLL.BY set.
*/

BR.RE.MAPPING.4: axiom
        QQ ∈ POLL.FOR.OF(I, K)
            ≡
        (∃ P, T.SUB, ACTIV, Y:
            START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
            ∧ 1 ≤ Y
            ∧ Y ≤ RESULT.SIZE(K)
            ∧ P ∈ SAFE.FOR(OF(I, K))
            ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, P), T.SUB, P))
            ∧ ACTION(ACTIV) = VOTE()
            ∧ TASK.ACTION(ACTIV) = K
            ∧ ELEM.ACTION(ACTIV) = Y
            ∧ QQ ∈ POLLBY.FOR(P, K, T.SUB))

/*
A value V is in the global ON set for K of I on P iff
        some safe processor QQ determines that either (1) the task K is on
        for the contained frame and has V as a voted ON.IN value,
        or (2) the task is determined to be NOT.ON and V is bottom.
*/

BR.RE.MAPPING.5: axiom
        V ∈ ON(K, I, P)
            ≡
        (∃ QQ:
            QQ ∈ SAFE.FOR(OF(I, K))
            ∧ ((¬NOT.ON.FRAME(K, QQ, BEGIN(OF(I, K)))
                ∧ (∀ Y:
                      1 ≤ Y ∧ Y ≤ RESULT.SIZE(K)
                          ⊃
                      SEQ.ELEM(V, Y) = SEQ.ELEM(ON.IN(K, I, P, QQ), Y)))
               ∨ (NOT.ON.FRAME(K, QQ, BEGIN(OF(I, K))) ∧ V = BOTTOM1(K))))
```

```
/*
A  V is the ON.IN value in QQ for K of I on P iff
        either (2) the task is determined to be NOT.ON and
        V is bottom, or (2) the task is ON and each component
        of V is equal to the DATAFILE component at the time of a scheduled
        vote.
*/

BR.RE.MAPPING.6: axiom
        V = ON.IN(K, I, P, QQ)
             ≡
        (∀ Y, T.SUB, ACTIV:
            BEGIN(OF(I, K)) ≤ T.SUB
            ∧ T.SUB < END(OF(I, K))
            ∧ 1 ≤ Y
            ∧ Y ≤ RESULT.SIZE(K)
            ∧ ¬NOT.ON.FRAME(K, QQ, BEGIN(OF(I, K)))
            ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
            ∧ ACTION(ACTIV) = VOTE()
            ∧ TASK.ACTION(ACTIV) = K
            ∧ ELEM.ACTION(ACTIV) = Y
                 ⊃
            SEQ.ELEM(V, Y)
               = SEQ.ELEM(DATAFILEIN.FOR.ON(QQ, K, P, START(T.SUB, QQ)), Y))
        ∨ (NOT.ON.FRAME(K, QQ, BEGIN(OF(I, K))) ∧ V = BOTTOM1(K))

/*
The IN value for K of I on P is determined from
        the value in the voted INPUT structure at the time of
        a scheduled vote.
*/

BR.RE.MAPPING.7: axiom
        (∃ T.SUB, ACTIV:
            START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
            ∧ 1 ≤ Y
            ∧ Y ≤ RESULT.SIZE(K)
            ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, P), T.SUB, P))
            ∧ TASK.ACTION(ACTIV) = K
            ∧ ACTION(ACTIV) = VOTE()
            ∧ ELEM.ACTION(ACTIV) = Y
            ∧ SEQ.ELEM(INPUTIN.OF(P, K, START(SUB.INCR(T.SUB), P)), Y) = D1)
              ⊃
        SEQ.ELEM(IN(K, I, P), Y) = D1

/* ... or the value in the INPUT structure at the time of a dummy vote. */

BR.RE.MAPPING.8: axiom
        (∃ T.SUB, ACTIV:
            SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, P), T.SUB, P))
            ∧ ACTION(ACTIV) = DUMMY.VOTE()
            ∧ TASK.ACTION(ACTIV) = K
            ∧ INPUTIN.OF(P, K, START(SUB.INCR(T.SUB), P)) = V)
              ⊃
        IN(K, I, P) = V
```

```
/*
A processor is safe at a subframe time iff it is working,
        the clock is safe, the Global Executive was safe for
        its last iteration, and the INPUT in the processor has
        the global consensus value for the last iteration
        of the Global Executive.
*/

RE.BR.MAPPING.9: axiom
        P ∈ SAFE(T.SUB)
             ≡
        P ∈ WORKING(T.SUB)
        ∧ CLOCK.SAFE(P, T.SUB)
        ∧ TASK.SAFE(GLOBAL.EXEC(), LAST(T.SUB, GLOBAL.EXEC()))
        ∧ SINGLETON(RESULT(GLOBAL.EXEC(), LAST(T.SUB, GLOBAL.EXEC())),
                    INPUTIN.OF(P, GLOBAL.EXEC(), START(T.SUB, P)))
```

SUBSECTION 7.14

REACT LEMMAS

```
/*
**********               Lemmas and Proofs for               **********
         Proof between Replication and Activity Specifications
*/

/*
If processors P and QQ are safe for the execution window of
         the i-th iteration of task K, QQ has (according to his information)
         an Execute scheduled, P has a Vote scheduled (wrt P's information)
         later in the frame, there will be no further Execute scheduled
         on QQ for that frame.
*/

BR.LEMMA.1: formula
        P ∈ SAFE.FOR(OF(I, K))
        ∧ QQ ∈ SAFE.FOR(OF(I, K))
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ FRAME(T.SUB) = FRAME(T1.SUB)
        ∧ SEQ.MEMBER(ACTIV2, SCHED(CONFIG(T1.SUB, P), T1.SUB, P))
        ∧ ACTION(ACTIV2) = VOTE()
        ∧ TASK.ACTION(ACTIV2) = K
        ∧ SUB.INCR(T.SUB) ≤ T2.SUB
        ∧ T2.SUB ≤ T1.SUB
            ⊃
        ¬(ACTION(ACTIV3) = EXECUTE()
          ∧ TASK.ACTION(ACTIV3) = K
          ∧ SEQ.MEMBER(ACTIV3, SCHED(CONFIG(T2.SUB, QQ), T2.SUB, QQ)))

prove BR.LEMMA.1
using BR.A21 [T1.SUB ← T.SUB]
      BR.A21
      BR.A21 [T1.SUB ← T2.SUB]
      BR.A12 [CON ← CONFIG(T2.SUB, QQ),
              T1.SUB ← T2.SUB,
              P ← QQ]
      BR.LEMMA.14
      BR.A16 [P ← QQ,
              CON ← CONFIG(T.SUB, QQ)]
      BR.A16 [P ← QQ,
              T.SUB ← T2.SUB,
              CON ← :4,
              ACTIV ← ACTIV3]
```

```
/*
Given the circumstances of Lemma 1, the datafile in P will
        remain unchanged from the subframe following the Execute
        until the time of the Vote
*/

BR.LEMMA.2: formula
        P ∈ SAFE.FOR(OF(I, K))
        ∧ QQ ∈ SAFE.FOR(OF(I, K))
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ FRAME(T.SUB) = FRAME(T1.SUB)
        ∧ SEQ.MEMBER(ACTIV2, SCHED(CONFIG(T1.SUB, P), T1.SUB, P))
        ∧ ACTION(ACTIV2) = VOTE()
        ∧ TASK.ACTION(ACTIV2) = K
        ∧ DATAFILEIN.FOR.ON(P, K, QQ, START(SUB.INCR(T.SUB), P)) = V
            ⊃
        DATAFILEIN.FOR.ON(P, K, QQ, START(T1.SUB, P)) = V

prove BR.LEMMA.2
using BR.LEMMA.4 [T2.SUB ← *T2.SUB:2]
      BR.INDUCTION.SUB.TIME.1 [T.SUB ← SUB.INCR(T.SUB),
                               P ← QQ,
                               QQ ← P]
```

```
/*
A processor P that is safe for the exection window of the
        i-th iteration of K will be working for each contained subframe
        time.
*/

BR.LEMMA.3: formula
        P ∈ SAFE.FOR(OF(I, K)) ∧ BEGIN(OF(I, K)) ≤ T.SUB ∧ T.SUB < END(OF(I, K))
            ⊃
        WORKING.DURING(P, T.SUB)

prove BR.LEMMA.3
using RE.BR.MAPPING.9
      BR.D1 [II ← OF(I, K)]


/* Analogous to Lemma 3 for the Data Window */

BR.LEMMA.3A: formula
        P ∈ SAFE.FOR(DW.FOR.TO.OF(L, I, K))
        ∧ BEGIN(DW.FOR.TO.OF(L, I, K)) ≤ T.SUB
        ∧ T.SUB < END(DW.FOR.TO.OF(L, I, K))
            ⊃
        WORKING.DURING(P, T.SUB)

prove BR.LEMMA.3A
using RE.BR.MAPPING.9
      BR.D1 [II ← DW.FOR.TO.OF(L, I, K)]
```

```
/*
By Lemma 2 and induction, the Datafile in P at the time of the Vote
        will be same as it was at the beginning of the subframe following
        the Execute on QQ
*/

BR.LEMMA.4: formula
        P ∈ SAFE.FOR(OF(I, K))
        ∧ QQ ∈ SAFE.FOR(OF(I, K))
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ FRAME(T.SUB) = FRAME(T1.SUB)
        ∧ SEQ.MEMBER(ACTIV2, SCHED(CONFIG(T1.SUB, P), T1.SUB, P))
        ∧ ACTION(ACTIV2) = VOTE()
        ∧ TASK.ACTION(ACTIV2) = K
        ∧ SUB.INCR(T.SUB) ≤ T2.SUB
        ∧ T2.SUB < T1.SUB
            ⊃
        DATAFILEIN.FOR.ON(P, K, QQ, START(SUB.INCR(T2.SUB), P))
           = DATAFILEIN.FOR.ON(P, K, QQ, START(T2.SUB, P))

prove BR.LEMMA.4
using BR.LEMMA.3 [T.SUB ← T2.SUB]
      BR.LEMMA.3 [T.SUB ← T2.SUB,
                  P ← QQ]
      BR.A16 [CON ← CONFIG(T1.SUB, P),
              T.SUB ← T1.SUB,
              ACTIV ← ACTIV2]
      BR.LEMMA.1 [T2.SUB ← SUB.INCR(T2.SUB),
                  ACTIV3 ← ACTIV2:5]
      BR.LEMMA.18 [T.SUB ← T2.SUB]
      BR.LEMMA.1 [ACTIV3 ← ACTIV:5]
      BR.D1 [T.SUB ← T2.SUB,
             II ← OF(I, K)]
      BR.D1 [T.SUB ← T2.SUB,
             II ← OF(I, K),
             P ← QQ]
      BR.D1 [II ← OF(I, K),
             T.SUB ← SUB.INCR(T2.SUB)]
      BR.D1 [II ← OF(I, K),
             T.SUB ← SUB.INCR(T2.SUB),
             P ← QQ]
      BR.A16 [P ← QQ,
              CON ← CONFIG(T.SUB, QQ)]
```

```
/*
If QQ has an Execute scheduled for the i-th iteration of K,
        and QQ is safe for the execution window, there will be no
        further Execute scheduled for the remainder of the execution
        window.
*/

BR.LEMMA.5: formula
        QQ ∈ SAFE.FOR(OF(I, K))
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
             ⊃
        ¬(∃ ACTIV2:
             SEQ.MEMBER(ACTIV2,
                        SCHED(CONFIG(SUB.INCR(T.SUB), QQ), SUB.INCR(T.SUB), QQ))
             ∧ ACTION(ACTIV2) = EXECUTE()
             ∧ TASK.ACTION(ACTIV2) = K)
        ∧ BEGIN(OF(I, K)) ≤ SUB.INCR(T.SUB)
        ∧ SUB.INCR(T.SUB) < END(OF(I, K))

prove BR.LEMMA.5
using BR.A13A [CON ← CONFIG(T.SUB, QQ),
               P ← QQ]
      BR.A16 [T.SUB ← *T1.SUB:1,
              ACTIV ← *ACTIV2:1,
              CON ← CONFIG(T.SUB, QQ),
              P ← QQ]
      BR.A12 [P ← QQ,
              ACTIV3 ← ACTIV2,
              CON ← CONFIG(T.SUB, QQ),
              T1.SUB ← SUB.INCR(T.SUB)]
      BR.LEMMA.14 [T2.SUB ← SUB.INCR(T.SUB)]
      BR.A16 [P ← QQ,
              CON ← CONFIG(T.SUB, QQ)]
      BR.A21 [T1.SUB ← T.SUB]
      BR.A21 [T1.SUB ← *T1.SUB:1]
      BR.A21 [T1.SUB ← SUB.INCR(T.SUB)]
```

```
/*
If processor QQ is scheduled to execute task K, and either a Vote
        or a Dummy Vote on an input task L to K is scheduled earlier,
        the voted value in Input at the time of the Execute
        will be the same as that just after the Vote.  The proof of
        this uses Lemma 7 and induction.
*/

BR.LEMMA.6: formula
        QQ ∈ SAFE.FOR(DW.FOR.TO.OF(L, I, K))
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ 1 ≤ Y
        ∧ Y ≤ RESULT.SIZE(K)
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ L ∈ INPUTS(K)
        ∧ SEQ.MEMBER(ACTIV2, SCHED(CONFIG(T1.SUB, QQ), T1.SUB, QQ))
        ∧ TASK.ACTION(ACTIV2) = L
        ∧ ((ACTION(ACTIV2) = VOTE() ∧ ELEM.ACTION(ACTIV2) = Y)
           ∨ ACTION(ACTIV2) = DUMMY.VOTE())
        ∧ START.FRAME(FRAME(T1.SUB)) = TO.OF(L, I, K)*FRAME.SIZE()
        ∧ SEQ.ELEM(INPUTIN.OF(QQ, L, START(SUB.INCR(T1.SUB), QQ)), Y) = D1
            ⊃
        SEQ.ELEM(INPUTIN.OF(QQ, L, START(SUB.INCR(T.SUB), QQ)), Y) = D1

prove BR.LEMMA.6
using BR.INDUCTION.SUB.TIME.2 [T.SUB ← SUB.INCR(T1.SUB),
                               T1.SUB ← SUB.INCR(T.SUB)]
      BR.LEMMA.7 [T2.SUB ← T2.SUB:1]
```

```
/*
If processor QQ is scheduled to execute task K, and either a Vote
        or a Dummy Vote on an input task L to K is scheduled earlier,
        then, for each subframe time between the Vote
        and subsequent Execute, the voted value in Input
        will remained unchanged.          */

BR.LEMMA.7: formula
        QQ ∈ SAFE.FOR(DW.FOR.TO.OF(L, I, K))
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ 1 ≤ Y
        ∧ Y ≤ RESULT.SIZE(K)
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ L ∈ INPUTS(K)
        ∧ SEQ.MEMBER(ACTIV2, SCHED(CONFIG(T1.SUB, QQ), T1.SUB, QQ))
        ∧ TASK.ACTION(ACTIV2) = L
        ∧ ((ACTION(ACTIV2) = VOTE() ∧ ELEM.ACTION(ACTIV2) = Y)
           ∨ ACTION(ACTIV2) = DUMMY.VOTE())
        ∧ START.FRAME(FRAME(T1.SUB)) = TO.OF(L, I, K)*FRAME.SIZE()
        ∧ T1.SUB < T2.SUB
        ∧ T2.SUB ≤ T.SUB
        ∧ D1 = SEQ.ELEM(INPUTIN.OF(QQ, L, START(T2.SUB, QQ)), Y)
            ⊃
        D1 = SEQ.ELEM(INPUTIN.OF(QQ, L, START(SUB.INCR(T2.SUB), QQ)), Y)

prove BR.LEMMA.7
using BR.LEMMA.49 [P ← QQ,
                   ACTIV3 ← ACTIV:6]
      BR.LEMMA.50 [P ← QQ,
                   ACTIV3 ← ACTIV:6]
      BR.LEMMA.51 [P ← QQ,
                   ACTIV3 ← ACTIV:6]
      BR.A25A [P ← QQ,
               CON ← CONFIG(T.SUB, QQ),
               CON2 ← CONFIG(T1.SUB, QQ),
               ACTIV3 ← ACTIV:6]
      BR.A12A [T1.SUB ← T2.SUB,
               ACTIV3 ← ACTIV:6,
               T.SUB ← T1.SUB,
               ACTIV ← ACTIV2,
               CON ← CONFIG(T1.SUB, QQ),
               P ← QQ,
               K ← L]
      BR.A6A [P ← QQ,
              T.SUB ← T2.SUB,
              K ← L]
      BR.LEMMA.3A [T.SUB ← T2.SUB,
                   P ← QQ]
      BR.A28
      BR.A29
      BR.A16 [P ← QQ,
              CON ← CONFIG(T.SUB, QQ)]
      BR.A16 [P ← QQ,
              K ← L,
              I ← TO.OF(L, I, K),
              T.SUB ← T1.SUB,
              ACTIV ← ACTIV2,
              CON ← CONFIG(T1.SUB, QQ)]
```

```
/*
For a processor QQ safe for the data window of the i-th iteration
        of K and scheduled to vote on a task input L and later Execute
        task K, QQ will be working for all intervening subframes.
*/

BR.LEMMA.8: formula
        QQ ∈ SAFE.FOR(DW.FOR.TO.OF(L, I, K))
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ L ∈ INPUTS(K)
        ∧ SEQ.MEMBER(ACTIV2, SCHED(CONFIG(T1.SUB, QQ), T1.SUB, QQ))
        ∧ TASK.ACTION(ACTIV2) = L
        ∧ ((ACTION(ACTIV2) = VOTE() ∧ ELEM.ACTION(ACTIV2) = Y)
           ∨ ACTION(ACTIV2) = DUMMY.VOTE())
        ∧ START.FRAME(FRAME(T1.SUB)) = TO.OF(L, I, K)*FRAME.SIZE()
        ∧ T1.SUB < T2.SUB
        ∧ T2.SUB ≤ T.SUB
             ⊃
        WORKING.DURING(QQ, T2.SUB)

prove BR.LEMMA.8
using BR.LEMMA.11
      RE.BR.MAPPING.9 [T.SUB ← T2.SUB,
                       P ← QQ]
      BR.D1 [II ← DW.FOR.TO.OF(L, I, K),
             P ← QQ,
             T.SUB ← T2.SUB]
```

```
/* No more than one task iteration can take place in one frame */

BR.LEMMA.10: formula
        BEGIN(OF(I, K)) ≤ T.SUB
        ∧ T.SUB < END(OF(I, K))
        ∧ BEGIN(OF(J, K)) ≤ T1.SUB
        ∧ T1.SUB < END(OF(J, K))
        ∧ ¬(I = J)
             ⊃
        ¬(START.FRAME(FRAME(T.SUB)) = START.FRAME(FRAME(T1.SUB)))

prove BR.LEMMA.10
using TIMES.AXIOM.1 [INT1 ← I,
                     INT2 ← J,
                     INT3 ← FRAME.SIZE()]
      BR.A27
      BR.A18
      BR.A18 [I ← J]
      BR.A21 [T.SUB ← BEGIN(OF(I, K)),
              T1.SUB ← SUB.DECR(END(OF(I, K)))]
      BR.A21 [T.SUB ← BEGIN(OF(I, K)),
              T1.SUB ← T.SUB]
      BR.A21 [T.SUB ← BEGIN(OF(J, K)),
              T1.SUB ← SUB.DECR(END(OF(J, K)))]
      BR.A21 [T.SUB ← BEGIN(OF(J, K))]
      BR.A21 [T.SUB ← BEGIN(OF(I, K)),
              T1.SUB ← BEGIN(OF(I, K))]
      BR.A21 [T.SUB ← BEGIN(OF(J, K)),
              T1.SUB ← BEGIN(OF(J, K))]
      BR.A21A
      BR.A21A [T.SUB ← T1.SUB]
      BR.A21 [T1.SUB ← START.FRAME(FRAME(T.SUB))]
      BR.A21 [T.SUB ← T1.SUB,
              T1.SUB ← START.FRAME(FRAME(T1.SUB))]
```

```
/*
If a processor QQ is safe for the data window, and has
        a Vote on an input task L providing input to a later Execute of task
        K, every intervening subframe is within the data window.
*/

BR.LEMMA.11: formula
        QQ ∈ SAFE.FOR(DW.FOR.TO.OF(L, I, K))
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ L ∈ INPUTS(K)
        ∧ SEQ.MEMBER(ACTIV2, SCHED(CONFIG(T1.SUB, QQ), T1.SUB, QQ))
        ∧ TASK.ACTION(ACTIV2) = L
        ∧ ((ACTION(ACTIV2) = VOTE() ∧ ELEM.ACTION(ACTIV2) = Y)
           ∨ ACTION(ACTIV2) = DUMMY.VOTE())
        ∧ START.FRAME(FRAME(T1.SUB)) = TO.OF(L, I, K)*FRAME.SIZE()
        ∧ T1.SUB < T2.SUB
        ∧ T2.SUB ≤ T.SUB
             ⊃
        BEGIN(DW.FOR.TO.OF(L, I, K)) ≤ T2.SUB
        ∧ T2.SUB < END(DW.FOR.TO.OF(L, I, K))

prove BR.LEMMA.11
using BR.A28
      BR.A16 [T.SUB ← T1.SUB,
              P ← QQ,
              ACTIV ← ACTIV2,
              CON ← CONFIG(T1.SUB, QQ),
              I ← TO.OF(L, I, K),
              K ← L]
      BR.A16 [CON ← CONFIG(T.SUB, QQ),
              P ← QQ]
      BR.A29


var T1.FRAME: FRAMETIME
```

```
/* Two frames are equal iff their beginning subframes are equal. */

BR.LEMMA.12: formula
        START.FRAME(FRAME(T.SUB)) = START.FRAME(FRAME(T1.SUB))
             ≡
        FRAME(T.SUB) = FRAME(T1.SUB)

prove BR.LEMMA.12
using BR.A21
      BR.A21 [T1.SUB ← T.SUB]
      BR.A21 [T.SUB ← T1.SUB]
      BR.A21B [T.FRAME ← FRAME(T.SUB)]
      BR.A21B [T.FRAME ← FRAME(T1.SUB)]
```

```
/*
If QQ is safe for the execution window for I of K, then
        the configuration according to processor QQ will be the same
        for any two subframe times within the window.
*/

BR.LEMMA.14: formula
        QQ ∈ SAFE.FOR(OF(I, K))
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ FRAME(T.SUB) = FRAME(T2.SUB)
        ∧ BEGIN(OF(I, K)) ≤ T.SUB
        ∧ T.SUB < END(OF(I, K))
        ∧ BEGIN(OF(I, K)) ≤ T2.SUB
        ∧ T2.SUB < END(OF(I, K))
             ⊃
        CONFIG(T.SUB, QQ) = CONFIG(T2.SUB, QQ)

prove BR.LEMMA.14
using BR.D1 [II ← OF(I, K),
             P ← QQ,
             T.SUB ← T2.SUB]
      BR.D1 [II ← OF(I, K),
             P ← QQ]
      CARD.2 [S ← RESULT(GLOBAL.EXEC(), LAST(T.SUB, GLOBAL.EXEC())),
              X ← INPUTIN.OF(QQ, GLOBAL.EXEC(), START(T.SUB, QQ)),
              X1 ← INPUTIN.OF(QQ, GLOBAL.EXEC(), START(T2.SUB, QQ))]
      RE.BR.MAPPING.9 [P ← QQ]
      RE.BR.MAPPING.9 [P ← QQ,
                       T.SUB ← T2.SUB]
      SEQ.EQUALITY.AXIOM [SEQ1
                              ← INPUTIN.OF(QQ, GLOBAL.EXEC(), START(T.SUB, QQ)),
                          SEQ2
                              ← INPUTIN.OF(QQ, GLOBAL.EXEC(), START(T2.SUB, QQ)),
                          Y ← CONFIG.FIELD()]
      BR.A36
```

```
/*
If P is safe for the I,K execution window, then, for
        any two contained subframes, the decision as to whether
        a given task is 'on' for that frame will be the same
        (see the definition of NOT.ON.FRAME in BR.A22)
*/

BR.LEMMA.16: formula
        P ∈ SAFE.FOR(OF(I, K))
        ∧ BEGIN(OF(I, K)) ≤ T.SUB
        ∧ T.SUB < END(OF(I, K))
        ∧ BEGIN(OF(I, K)) ≤ T1.SUB
        ∧ T1.SUB < END(OF(I, K))
             ⊃
        (NOT.ON.FRAME(K, P, T.SUB)  ≡  NOT.ON.FRAME(K, P, T1.SUB))

prove BR.LEMMA.16
using BR.LEMMA.16A
      BR.LEMMA.16A [T.SUB ← T1.SUB,
                    T1.SUB ← T.SUB]


/* Trivial corollary of Lemma 16 */

BR.LEMMA.16A: formula
        P ∈ SAFE.FOR(OF(I, K))
        ∧ BEGIN(OF(I, K)) ≤ T.SUB
        ∧ T.SUB < END(OF(I, K))
        ∧ BEGIN(OF(I, K)) ≤ T1.SUB
        ∧ T1.SUB < END(OF(I, K))
        ∧ ¬NOT.ON.FRAME(K, P, T.SUB)
             ⊃
        ¬NOT.ON.FRAME(K, P, T1.SUB)

prove BR.LEMMA.16A
using BR.A22 [QQ ← P,
              ACTIV ← @:D]
      BR.LEMMA.14 [QQ ← P,
                   T2.SUB ← T1.SUB]
      BR.LEMMA.21
      BR.LEMMA.21 [T.SUB ← T1.SUB]
      BR.A18
      BR.LEMMA.12 [T1.SUB ← BEGIN(OF(I, K))]
      BR.A22 [T1.SUB ← *T1.SUB:1,
              ACTIV ← *ACTIV:1,
              T.SUB ← T1.SUB,
              QQ ← P,
              P ← *P:1]
```

```
/*
Composing Lemma 16 with the mapping of ON.IN,
        we conclude that if a processor P is safe for
        I of K, and processor P, at time T.SUB determines
        then that the task K was run and there should be
        a Vote activity performed, the ON.IN value
        will be the value in the Datafile at that time.
*/

BR.LEMMA.15: formula
        P ∈ SAFE.FOR(OF(I, K))
        ∧ ¬NOT.ON.FRAME(K, P, T.SUB)
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, P), T.SUB, P))
        ∧ ACTION(ACTIV) = VOTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ ELEM.ACTION(ACTIV) = Y
        ∧ 1 ≤ Y
        ∧ Y ≤ RESULT.SIZE(K)
             ⊃
        SEQ.ELEM(ON.IN(K, I, QQ, P), Y)
           = SEQ.ELEM(DATAFILEIN.FOR.ON(P, K, QQ, START(T.SUB, P)), Y)
prove BR.LEMMA.15
using BR.LEMMA.16 [T1.SUB ← BEGIN(OF(I, K))]
      BR.A16 [CON ← CONFIG(T.SUB, P)]
      BR.RE.MAPPING.6 [V ← ON.IN(K, I, QQ, P),
                       P ← QQ,
                       QQ ← P]
```

```
var V.INPUTS: SET.OF(PAIR.OF(DATA, TASK))

/*
This is an important lemma concerning synchronization
        and availability of broadcast data across processor
        boundaries.  It states that if processor QQ is safe for
        I of K, and processors P and QQ are clock safe and working
        for subframe T.SUB, at which time an Execute is scheduled
        on processor QQ, according to QQ's clock, then the
        Datafile on processor P, according to P's clock, will have the
        correct output value at the beginning of the next subframe.
*/

BR.LEMMA.17X: formula
        QQ ∈ SAFE.FOR(OF(I, K))
        ∧ CLOCK.SAFE(P, SUB.INCR(T.SUB))
        ∧ CLOCK.SAFE(QQ, SUB.INCR(T.SUB))
        ∧ P ∈ WORKING(T.SUB)
        ∧ QQ ∈ WORKING(T.SUB)
        ∧ QQ ∈ WORKING(SUB.INCR(T.SUB))
        ∧ P ∈ WORKING(SUB.INCR(T.SUB))
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ (∀ V.T:
              V.T ∈ V.INPUTS
                   ≡
              SOURCE(V.T) ∈ INPUTS(K)
              ∧ VALUE(V.T)
                   = INPUTIN.OF(QQ, SOURCE(V.T), START(SUB.INCR(T.SUB), QQ)))
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
             ⊃
        DATAFILEIN.FOR.ON(P, K, QQ, START(SUB.INCR(T.SUB), P))
           = APPLY(FUNCTION(K), V.INPUTS)

prove BR.LEMMA.17X [V.T ← *V.T:1]
using BR.A41
      BR.A42 [T.REAL ← START(SUB.INCR(T.SUB), P),
              T.SUB ← SUB.INCR(T.SUB)]
      NAT.NONNEGATIVE [Y ← BROADCAST.DELAY()]
      NAT.NONNEGATIVE [Y ← OVERHEAD()]
      NAT.NONNEGATIVE [Y ← MIN.INTERVAL()]
      BR.A1.D
      BR.A1.E
      BR.A1.C [T.SUB ← SUB.INCR(T.SUB),
               P ← QQ]
      BR.LEMMA.FOR.LES.TO.PROVE [T.SUB ← SUB.INCR(T.SUB)]
      BR.LEMMA.5 [ACTIV2 ← *ACTIV:2]
```

```
/*
This lemma builds on lemma 17x.  Eliminated are various
        assumptions concerning WORKING and CLOCK.SAFE that
        are shown to be satisfied by virtue of other constraints.
*/

BR.LEMMA.17: formula
        P ∈ SAFE.FOR(OF(I, K))
        ∧ QQ ∈ SAFE.FOR(OF(I, K))
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ (∀ V.T:
              V.T ∈ V.INPUTS
                   ≡
              SOURCE(V.T) ∈ INPUTS(K)
              ∧ VALUE(V.T)
                   = INPUTIN.OF(QQ, SOURCE(V.T), START(SUB.INCR(T.SUB), QQ)))
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
             ⊃
        DATAFILEIN.FOR.ON(P, K, QQ, START(SUB.INCR(T.SUB), P))
           = APPLY(FUNCTION(K), V.INPUTS)

prove BR.LEMMA.17 [V.T ← *V.T:1]
using BR.LEMMA.17X
      RE.BR.MAPPING.9
      RE.BR.MAPPING.9 [P ← QQ]
      RE.BR.MAPPING.9 [T.SUB ← SUB.INCR(T.SUB)]
      RE.BR.MAPPING.9 [T.SUB ← SUB.INCR(T.SUB),
                       P ← QQ]
      BR.D1 [II ← OF(I, K)]
      BR.D1 [II ← OF(I, K),
             P ← QQ]
      BR.D1 [II ← OF(I, K),
             T.SUB ← SUB.INCR(T.SUB),
             P ← QQ]
      BR.D1 [II ← OF(I, K),
             T.SUB ← SUB.INCR(T.SUB)]
      BR.LEMMA.5
      BR.A16 [CON ← CONFIG(T.SUB, QQ),
              P ← QQ]
```

```
/*
If processors P and QQ are both safe for two consecutive subframes
        during which QQ does not have an Execute scheduled (according
        to QQ's view of the configuration), then the Datafile in P
        for the results of QQ will remain unchanged.
*/

BR.LEMMA.18: formula
        P ∈ SAFE(T.SUB)
        ∧ QQ ∈ SAFE(T.SUB)
        ∧ P ∈ SAFE(SUB.INCR(T.SUB))
        ∧ QQ ∈ SAFE(SUB.INCR(T.SUB))
        ∧ ¬(∃ ACTIV:
               SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
               ∧ ACTION(ACTIV) = EXECUTE()
               ∧ TASK.ACTION(ACTIV) = K)
        ∧ ¬(∃ ACTIV2:
               SEQ.MEMBER(ACTIV2,
                          SCHED(CONFIG(SUB.INCR(T.SUB), QQ), SUB.INCR(T.SUB),
                                QQ))
               ∧ ACTION(ACTIV2) = EXECUTE()
               ∧ TASK.ACTION(ACTIV2) = K)
             ⊃
        DATAFILEIN.FOR.ON(P, K, QQ, START(SUB.INCR(T.SUB), P))
           = DATAFILEIN.FOR.ON(P, K, QQ, START(T.SUB, P))

prove BR.LEMMA.18 [ACTIV ← *ACTIV:17,
                   ACTIV2 ← *ACTIV:2]
using BR.A42 [T.REAL ← START(T.SUB, P)]
      BR.A42 [T.REAL ← START(SUB.INCR(T.SUB), P),
              T.SUB ← SUB.INCR(T.SUB)]
      BR.A42 [T.REAL
                 ← RPLUS(INT.TO.REALTIME(BROADCAST.DELAY()), FINISH(T.SUB, QQ))]
      RE.BR.MAPPING.9
      RE.BR.MAPPING.9 [P ← QQ]
      RE.BR.MAPPING.9 [T.SUB ← SUB.INCR(T.SUB)]
      RE.BR.MAPPING.9 [P ← QQ,
                       T.SUB ← SUB.INCR(T.SUB)]
      BR.LEMMA.FOR.LES.TO.PROVE
      BR.LEMMA.FOR.LES.TO.PROVE [T.SUB ← SUB.INCR(T.SUB)]
      NAT.NONNEGATIVE [Y ← OVERHEAD()]
      NAT.NONNEGATIVE [Y ← MIN.INTERVAL()]
      NAT.NONNEGATIVE [Y ← BROADCAST.DELAY()]
      BR.A1.A
      BR.A1.C [P ← QQ,
               T.SUB ← SUB.INCR(T.SUB)]
      BR.A1.E
      BR.A1.C [P ← QQ]
      REGRETABLE.EVIL [ACTIV2 ← *ACTIV:1,
                       ACTIV3 ← *ACTIV:3]
```

```
/*
For two times T.SUB and T1.SUB, during which processors
        P and QQ are (respectively) safe during an execution
        window, both processors will reach the same decision
        concerning the system configuration.
*/

BR.LEMMA.19: formula
        P ∈ SAFE(T.SUB)
        ∧ QQ ∈ SAFE(T1.SUB)
        ∧ BEGIN(OF(I, K)) ≤ T.SUB
        ∧ T.SUB < END(OF(I, K))
        ∧ BEGIN(OF(I, K)) ≤ T1.SUB
        ∧ T1.SUB < END(OF(I, K))
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
            ⊃
        CONFIG(T.SUB, P) = CONFIG(T1.SUB, QQ)


prove BR.LEMMA.19
using BR.A36 [T2.SUB ← T1.SUB]
      RE.BR.MAPPING.9
      RE.BR.MAPPING.9 [P ← QQ,
                       T.SUB ← T1.SUB]
      CARD.2 [S ← RESULT(GLOBAL.EXEC(), LAST(T.SUB, GLOBAL.EXEC())),
              X ← INPUTIN.OF(P, GLOBAL.EXEC(), START(T.SUB, P)),
              X1 ← INPUTIN.OF(QQ, GLOBAL.EXEC(), START(T1.SUB, QQ))]
      DATA.EQUALITY [V ← INPUTIN.OF(P, GLOBAL.EXEC(), START(T.SUB, P)),
                     V1 ← INPUTIN.OF(QQ, GLOBAL.EXEC(), START(T1.SUB, QQ)),
                     Y ← CONFIG.FIELD()]
      GE.CONFIG.FIELD
      DATA.SIZE.IS.SEQ.LENGTH.2 [K ← GLOBAL.EXEC(),
                                 T.REAL ← START(T.SUB, P)]
      DATA.SIZE.IS.SEQ.LENGTH.2 [K ← GLOBAL.EXEC(),
                                 P ← QQ,
                                 T.REAL ← START(T1.SUB, QQ)]
```

```
/*
Two times within the same execution window will be within
        the same frame
*/

BR.LEMMA.21: formula
        BEGIN(OF(I, K)) ≤ T.SUB ∧ T.SUB < END(OF(I, K))
             ⊃
        FRAME(T.SUB) = FRAME(BEGIN(OF(I, K)))

prove BR.LEMMA.21
using BR.A18
      BR.A21 [T.SUB ← BEGIN(OF(I, K)),
              T1.SUB ← BEGIN(OF(I, K))]
      BR.A21 [T.SUB ← SUB.DECR(END(OF(I, K))),
              T1.SUB ← SUB.DECR(END(OF(I, K)))]
      BR.A21 [T.SUB ← BEGIN(OF(I, K)),
              T1.SUB ← T.SUB]
```

```
/*
If two processors are safe for the execution window and
        and have Votes scheduled during the same frame on
        the same element of the same task, they will determine
        the same set of processors to be polled.  This again,
        involves proving global consistency. This lemma proves
        only implication.  Equivalence is stated in lemma 25
        and proved by referring two to this lemma.
*/

BR.LEMMA.22: formula
        P ∈ SAFE.FOR(OF(I, K))
        ∧ QQ ∈ SAFE.FOR(OF(I, K))
        ∧ SEQ.MEMBER(ACTIV2, SCHED(CONFIG(T.SUB, P), T.SUB, P))
        ∧ ACTION(ACTIV2) = VOTE()
        ∧ TASK.ACTION(ACTIV2) = K
        ∧ ELEM.ACTION(ACTIV2) = Y
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T1.SUB, QQ), T1.SUB, QQ))
        ∧ ACTION(ACTIV) = VOTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ ELEM.ACTION(ACTIV) = Z
        ∧ FRAME(T.SUB) = FRAME(T1.SUB)
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
              ⊃
        (R ∈ POLLBY.FOR(P, K, T.SUB)  ⊃  R ∈ POLLBY.FOR(QQ, K, T1.SUB))

prove BR.LEMMA.22
using BR.A9A [L ← K,
              T1.SUB ← T.SUB]
      BR.A9A [T.SUB ← *T.SUB:1,
              ACTIV ← *ACTIV:1,
              P ← QQ,
              L ← K]
      BR.LEMMA.23 [T.SUB ← T1.SUB,
                   T1.SUB ← *T.SUB:1,
                   ACTIV ← *ACTIV:1,
                   Y ← Z,
                   ACTIV2 ← ACTIV,
                   P ← QQ,
                   QQ ← R]
      BR.LEMMA.19 [T.SUB ← *T.SUB:1,
                   T1.SUB ← *T.SUB:1]
      BR.D1 [T.SUB ← *T.SUB:1,
             P ← QQ,
             II ← OF(I, K)]
      BR.D1 [T.SUB ← *T.SUB:1,
             II ← OF(I, K)]
      BR.A16 [T.SUB ← *T.SUB:1,
              ACTIV ← *ACTIV:1,
              P ← R,
              CON ← CONFIG(*T.SUB:1, P)]
```

```
/*
If, according to a processor P who has a Vote scheduled on
        task K, there is another processor QQ who has an Execute
        scheduled within the same frame,  the Execute must precede
        the Vote.
*/

BR.LEMMA.23: formula
        P ∈ SAFE.FOR(OF(I, K))
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ SEQ.MEMBER(ACTIV2, SCHED(CONFIG(T.SUB, P), T.SUB, P))
        ∧ ACTION(ACTIV2) = VOTE()
        ∧ TASK.ACTION(ACTIV2) = K
        ∧ ELEM.ACTION(ACTIV2) = Y
        ∧ FRAME(T.SUB) = FRAME(T1.SUB)
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T1.SUB, P), T1.SUB, QQ))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
             ⊃
        T1.SUB < T.SUB

prove BR.LEMMA.23
using BR.A13A [CON ← CONFIG(T1.SUB, P),
               T.SUB ← T1.SUB,
               P ← QQ,
               QQ ← P]
      BR.A12A [CON ← CONFIG(T.SUB, P),
               ACTIV ← ACTIV2,
               T1.SUB ← *T1.SUB:1,
               ACTIV3 ← *ACTIV2:1]
      BR.LEMMA.14 [QQ ← P,
                   T2.SUB ← T1.SUB]
      BR.A16 [ACTIV ← ACTIV2,
              CON ← CONFIG(T.SUB, P)]
      BR.A16 [CON ← CONFIG(T1.SUB, P),
              P ← QQ,
              T.SUB ← T1.SUB]
```

```
/* ...the IFF version of lemma 22. */

BR.LEMMA.25: formula
        P ∈ SAFE.FOR(OF(I, K))
        ∧ QQ ∈ SAFE.FOR(OF(I, K))
        ∧ SEQ.MEMBER(ACTIV2, SCHED(CONFIG(T.SUB, P), T.SUB, P))
        ∧ ACTION(ACTIV2) = VOTE()
        ∧ TASK.ACTION(ACTIV2) = K
        ∧ ELEM.ACTION(ACTIV2) = Y
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T1.SUB, QQ), T1.SUB, QQ))
        ∧ ACTION(ACTIV) = VOTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ ELEM.ACTION(ACTIV) = Z
        ∧ FRAME(T.SUB) = FRAME(T1.SUB)
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
             ⊃
        (R ∈ POLLBY.FOR(P, K, T.SUB)  ≡  R ∈ POLLBY.FOR(QQ, K, T1.SUB))

prove BR.LEMMA.25
using BR.LEMMA.22
      BR.LEMMA.22 [P ← QQ,
                   T.SUB ← T1.SUB,
                   T1.SUB ← T.SUB,
                   ACTIV ← ACTIV2,
                   Z ← Y,
                   Y ← Z,
                   ACTIV2 ← ACTIV,
                   QQ ← P]
```

```
/*
From lemma 25 and the mapping of POLL.FOR.OF,
        we determine that a processor safe for the execution
        window who has a vote scheduled will reach the global
        consensus as to the set of processors to be polled.
*/

BR.LEMMA.26: formula
        P ∈ SAFE.FOR(OF(I, K))
        ∧ SEQ.MEMBER(ACTIV2, SCHED(CONFIG(T.SUB, P), T.SUB, P))
        ∧ ACTION(ACTIV2) = VOTE()
        ∧ TASK.ACTION(ACTIV2) = K
        ∧ ELEM.ACTION(ACTIV2) = Y
        ∧ 1 ≤ Y
        ∧ Y ≤ RESULT.SIZE(K)
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
             ⊃
        (R ∈ POLLBY.FOR(P, K, T.SUB)  ≡  R ∈ POLL.FOR.OF(I, K))
```

```
/*
A processor safe for the execution window who has a Vote
        scheduled will have placed in INPUT the majority value
        of the ON.IN values.
*/

BR.LEMMA.27: formula
        QQ ∈ SAFE.FOR(OF(I, K))
        ∧ 1 ≤ Y
        ∧ Y ≤ RESULT.SIZE(K)
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
        ∧ ACTION(ACTIV) = VOTE()·
        ∧ TASK.ACTION(ACTIV) = K
        ∧ ELEM.ACTION(ACTIV) = Y
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
            ⊃
        SEQ.ELEM(INPUTIN.OF(QQ, K, START(SUB.INCR(T.SUB), QQ)), Y)
           = MAJORITY(D.BAG.D4(K, I, QQ, Y))

prove BR.LEMMA.27
using BR.A9C [P ← QQ]
      SETEQUALITY [S1 ← D.BAG.D4(K, I, QQ, Y),
                   S2 ← D.BAG.A9C(QQ, K, T.SUB, Y),
                   X ← *X:2]
      SET.ABSTRACTION.A9C [P ← QQ,
                           D.P ← *X:2]
      BR.A16 [P ← QQ,
              CON ← CONFIG(T.SUB, QQ)]
      BR.D1 [II ← OF(I, K),
             P ← QQ]
      RE.BR.MAPPING.9 [P ← QQ]
      BR.LEMMA.26 [P ← QQ,
                   ACTIV2 ← ACTIV,
                   R ← SOURCE(*X:2)]
      RP.D4A [P ← SOURCE(*X:2),
              D.P ← *X:2]
      BR.LEMMA.28 [P ← QQ]
      BR.LEMMA.15 [P ← QQ,
                   QQ ← SOURCE(*X:2)]
```

```
/*
If a safe processor decides that a Vote activity for K
        is scheduled, the task must be ON that frame, as
        determined during that subframe.
*/

BR.LEMMA.28: formula
        P ∈ SAFE.FOR(OF(I, K))
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, P), T.SUB, P))
        ∧ ACTION(ACTIV) = VOTE()
        ∧ ELEM.ACTION(ACTIV) = Y
        ∧ TASK.ACTION(ACTIV) = K
             ⊃
        ¬NOT.ON.FRAME(K, P, T.SUB)

prove BR.LEMMA.28
using BR.A13B [ACTIV2 ← ACTIV,
               CON ← CONFIG(T.SUB, P),
               T1.SUB ← T.SUB,
               QQ ← P]
      BR.A22 [T1.SUB ← *T.SUB:1,
              ACTIV ← *ACTIV:1,
              QQ ← P,
              P ← *P:1]
      BR.LEMMA.12 [T1.SUB ← *T.SUB:1]
      BR.LEMMA.14 [QQ ← P,
                   T2.SUB ← *T.SUB:1]
      BR.A16 [CON ← CONFIG(T.SUB, P)]
      BR.A16 [T.SUB ← *T.SUB:1,
              ACTIV ← *ACTIV:1,
              P ← *P:1,
              CON ← CONFIG(T.SUB, P)]
```

```
/*
An extension of lemma 28, ... the task must be ON
        as determined at the beginning of the execution
        window.
*/

BR.LEMMA.29: formula
        P ∈ SAFE.FOR(OF(I, K))
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, P), T.SUB, P))
        ∧ ACTION(ACTIV) = VOTE()
        ∧ ELEM.ACTION(ACTIV) = Y
        ∧ TASK.ACTION(ACTIV) = K
             ⊃
        ¬NOT.ON.FRAME(K, P, BEGIN(OF(I, K)))

prove BR.LEMMA.29
using BR.A13B [ACTIV2 ← ACTIV,
               CON ← CONFIG(T.SUB, P),
               T1.SUB ← T.SUB,
               QQ ← P]
      BR.A22 [T1.SUB ← T.SUB:1,
              ACTIV ← :1,
              QQ ← P,
              T.SUB ← BEGIN(OF(I, K)),
              P ← :1]
      BR.LEMMA.12 [T1.SUB ← T.SUB:1,
                   T.SUB ← :0]
      BR.LEMMA.14 [QQ ← P,
                   T2.SUB ← T.SUB:1]
      BR.A16 [CON ← CONFIG(T.SUB, P)]
      BR.A16 [T.SUB ← :1,
              ACTIV ← :1,
              P ← :1,
              CON ← CONFIG(T.SUB, P)]
      BR.LEMMA.14 [QQ ← P,
                   T2.SUB ← BEGIN(OF(I, K))]
      BR.A18
      BR.LEMMA.12 [T1.SUB ← BEGIN(OF(I, K))]
      BR.LEMMA.12 [T.SUB ← *T.SUB:1,
                   T1.SUB ← BEGIN(OF(I, K))]
```

```
/*
If a task is ON, as determined by a safe processor P at the
        beginning of the execution window, then there exists,
        according to P, a Vote activity scheduled within that
        frame.  Thus, a safe processor will vote on all tasks
        it views to be in the configuration.
*/

BR.LEMMA.30: formula
        ¬NOT.ON.FRAME(K, P, BEGIN(OF(I, K)))
        ∧ 1 ≤ Y
        ∧ Y ≤ RESULT.SIZE(K)
        ∧ P ∈ SAFE.FOR(OF(I, K))
             ⊃
        (∃ T.SUB, ACTIV:
            START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
            ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, P), T.SUB, P))
            ∧ ACTION(ACTIV) = VOTE()
            ∧ ELEM.ACTION(ACTIV) = Y
            ∧ TASK.ACTION(ACTIV) = K)

prove BR.LEMMA.30 [ACTIV ← *ACTIV2:2,
                   T.SUB ← *T1.SUB:2]
using BR.A22 [T.SUB ← BEGIN(OF(I, K)),
              T1.SUB ← @:D,
              P ← @:D,
              ACTIV ← @:D,
              QQ ← P]
      BR.A13A [CON ← CONFIG(BEGIN(OF(I, K)), P),
               T.SUB ← *T1.SUB:1,
               P ← *P:1,
               QQ ← P,
               ACTIV ← *ACTIV:1]
      BR.A16 [CON ← CONFIG(BEGIN(OF(I, K)), P),
              T.SUB ← *T1.SUB:2,
              ACTIV ← *ACTIV2:2]
      BR.A18
      BR.A16 [CON ← CONFIG(BEGIN(OF(I, K)), P),
              T.SUB ← *T1.SUB:1,
              P ← *P:1,
              ACTIV ← *ACTIV:1]
      BR.A19.1
      BR.LEMMA.14 [T.SUB ← BEGIN(OF(I, K)),
                   QQ ← P,
                   T2.SUB ← *T1.SUB:1]
      BR.LEMMA.14 [T.SUB ← *T1.SUB:1,
                   QQ ← P,
                   T2.SUB ← *T1.SUB:2]
```

```
/*
For a safe processor QQ and task K which QQ determines
        at the beginning of the execution window is in the configuration,
        The IN value will be the majority of ON.IN values.  Thus
        the Vote activity satisfies the 'voted input' criterion
        of the Replication specification.
*/

BR.LEMMA.31: formula
        QQ ∈ SAFE.FOR(OF(I, K))
        ∧ 1 ≤ Y
        ∧ Y ≤ RESULT.SIZE(K)
        ∧ ¬NOT.ON.FRAME(K, QQ, BEGIN(OF(I, K)))
            ⊃
        SEQ.ELEM(IN(K, I, QQ), Y) = MAJORITY(D.BAG.D4(K, I, QQ, Y))

prove BR.LEMMA.31
using BR.LEMMA.30 [P ← QQ]
      BR.LEMMA.27 [ACTIV ← *ACTIV:1,
                   T.SUB ← *T.SUB:1]
      BR.RE.MAPPING.7 [P ← QQ,
                       T.SUB ← *T.SUB:1,
                       D1 ← SEQ.ELEM(INPUTIN.OF(QQ, K,
                                                START(SUB.INCR(*T.SUB:1), QQ)),
                                  Y),
                       ACTIV ← *ACTIV:1]
```

```
/*
If a safe processor determines at the beginning of the
        execution window that a task is not ON, then there
        will be a DUMMY VOTE scheduled sometime within the frame.
*/

BR.LEMMA.32: formula
        QQ ∈ SAFE.FOR(OF(I, K))
        ∧ 1 ≤ Y
        ∧ Y ≤ RESULT.SIZE(K)
        ∧ NOT.ON.FRAME(K, QQ, BEGIN(OF(I, K)))
            ⊃
        (∃ T1.SUB, ACTIV2:
            START.FRAME(FRAME(T1.SUB)) = I*FRAME.SIZE()
            ∧ SEQ.MEMBER(ACTIV2, SCHED(CONFIG(BEGIN(OF(I, K)), QQ), T1.SUB, QQ))
            ∧ TASK.ACTION(ACTIV2) = K
            ∧ ACTION(ACTIV2) = DUMMY.VOTE())

prove BR.LEMMA.32 [T1.SUB ← *T.SUB:4,
                   ACTIV2 ← *ACTIV:4]
using BR.A22 [T.SUB ← BEGIN(OF(I, K)),
              T1.SUB ← *T.SUB:2,
              P ← *P:2,
              ACTIV ← *ACTIV:2]
      BR.A13B [T1.SUB ← *T.SUB:4,
               ACTIV2 ← *ACTIV:4,
               Y ← *Y:4,
               CON ← CONFIG(BEGIN(OF(I, K)), QQ)]
      BR.A18
      BR.A12C [P ← QQ,
               CON ← CONFIG(BEGIN(OF(I, K)), QQ)]
      BR.LEMMA.12 [T.SUB ← *T.SUB:2,
                   T1.SUB ← BEGIN(OF(I, K))]
```

```
/*
The counterpart to Lemma 31 -- this is the case where the
        task K is determined to be NOT.ON during the frame.
        This lemma, together will Lemma 31 proves RP.D4 of
        the Replication specification.
*/

BR.LEMMA.33: formula
        QQ ∈ SAFE.FOR(OF(I, K))
        ∧ 1 ≤ Y
        ∧ Y ≤ RESULT.SIZE(K)
        ∧ NOT.ON.FRAME(K, QQ, BEGIN(OF(I, K)))
             ⊃
        SEQ.ELEM(IN(K, I, QQ), Y) = MAJORITY(D.BAG.D4(K, I, QQ, Y))

prove BR.LEMMA.33
using BR.LEMMA.35
      MAJ.2 [M.BAG ← D.BAG.D4(K, I, QQ, Y),
             T2.V ← D1:3]
      BOTTOM.EQUALITY
      DATA.BOTTOM
      BR.LEMMA.36
```

```
/*
If a safe processor QQ determines at the beginning
        of the execution window that K is NOT.ON, then
        the global POLL.FOR.OF set will be empty.
*/

BR.LEMMA.34: formula
        QQ ∈ SAFE.FOR(OF(I, K)) ∧ NOT.ON.FRAME(K, QQ, BEGIN(OF(I, K)))
             ⊃
        (∀ P: ¬(P ∈ POLL.FOR.OF(I, K)))

/*
Proof Outline: From AC, apply Forward Br.a22.  Then use conclusion
        to satisfy Antecendent of A13b. Obtain negation of Backward consequent
        of Mapping.4 by: using backward antecent of Mapping.4,
        prove a16 (employing also lemma.12),
        and lemma.19 (employing also a19.1, rp.d1 and a18),
        resulting in negation of backward antecedent of
        mapping.4
*/

prove BR.LEMMA.34
using BR.A22 [T.SUB ← BEGIN(OF(I, K)),
              T1.SUB ← *T.SUB:2,
              P ← *P:2,
              ACTIV ← *ACTIV:2]
      BR.A13B [T1.SUB ← *T.SUB:4,
               ACTIV2 ← *ACTIV:4,
               Y ← *Y:4,
               QQ ← *P:4,
               CON ← CONFIG(BEGIN(OF(I, K)), QQ)]
      BR.A18
      BR.RE.MAPPING.4 [QQ ← P,
                       ACTIV ← @:D,
                       Y ← @:D,
                       T.SUB ← @:D]
      BR.A16 [T.SUB ← *T.SUB:4,
              ACTIV ← *ACTIV:4,
              P ← *P:4,
              CON ← CONFIG(*T.SUB:4, *P:4)]
      BR.D1 [II ← OF(I, K),
             T.SUB ← *T.SUB:4,
             P ← *P:4]
      BR.D1 [II ← OF(I, K),
             T.SUB ← BEGIN(OF(I, K)),
             P ← QQ]
      BR.LEMMA.19 [P ← QQ,
                   T.SUB ← BEGIN(OF(I, K)),
                   QQ ← *P:4,
                   T1.SUB ← *T.SUB:4]
      BR.A19.1
      BR.LEMMA.12 [T.SUB ← *T.SUB:2,
                   T1.SUB ← BEGIN(OF(I, K))]
```

```
/*
...and furthermore, the cardinality of the set of ON.IN
        values for processors in the poll set will therefore be 0.
*/

BR.LEMMA.35: formula
        QQ ∈ SAFE.FOR(OF(I, K)) ∧ NOT.ON.FRAME(K, QQ, BEGIN(OF(I, K)))
             ⊃
        0 = CARD(D.BAG.D4(K, I, QQ, Y))

prove BR.LEMMA.35
using BR.LEMMA.34 [P ← *P:2]
      RP.D4A [D.P ← *X:4,
              P ← @:D]
      CARD.6 [S ← D.BAG.D4(K, I, QQ, Y)]
      CARD.4 [S ← D.BAG.D4(K, I, QQ, Y)]
```

```
/*
More about the case when the task is determined to
        be NOT.ON ... the IN value of the Replication level
        will be the bottom value -- according to the definition
        of majority and the formation of the ON.IN bag.
*/

BR.LEMMA.36: formula
        QQ ∈ SAFE.FOR(OF(I, K))
        ∧ 1 ≤ Y
        ∧ Y ≤ RESULT.SIZE(K)
        ∧ NOT.ON.FRAME(K, QQ, BEGIN(OF(I, K)))
             ⊃
        IN(K, I, QQ) = BOTTOM1(K)

/*
Use antecedent of conclusion to satisfy antecedent
        of Lemma.32.  Using conclusion of lemma.32, lemma.14
       (requiring  a16, a19.1, lemma.12(requiring a18)),
       and re.br.mapping.7 (requiring
       br.d1) to satisfy the antecent of a6b.  Use conclusion of
       a6b, together with lemma.14, to satify the antecent of
       mapping.8 to conclude consequent of conclusion.
*/

prove BR.LEMMA.36
using BR.LEMMA.32
      BR.A6B [P ← QQ,
              T.SUB ← *T1.SUB:1,
              ACTIV ← *ACTIV2:1]
      BR.LEMMA.14 [T.SUB ← *T1.SUB:1,
                   T2.SUB ← BEGIN(OF(I, K))]
      BR.D1 [II ← OF(I, K),
             P ← QQ,
             T.SUB ← *T1.SUB:1]
      RE.BR.MAPPING.9 [P ← QQ,
                       T.SUB ← *T1.SUB:1]
      BR.A16 [T.SUB ← *T1.SUB:1,
              P ← QQ,
              CON ← CONFIG(BEGIN(OF(I, K)), QQ),
              ACTIV ← *ACTIV2:1]
      BR.A19.1
      BR.RE.MAPPING.8 [T.SUB ← *T1.SUB:1,
                       ACTIV ← *ACTIV2:1,
                       V ← BOTTOM1(K),
                       P ← QQ]
      BR.A18
      BR.LEMMA.12 [T.SUB ← BEGIN(OF(I, K)),
                   T1.SUB ← *T1.SUB:1]
```

```
/*
If a processor is in the global poll set for I of K, and is safe
        for the execution window, then there exists an
        Execute activity sometime within that frame for K.
*/

BR.LEMMA.37: formula
        P ∈ POLL.FOR.OF(I, K) ∧ P ∈ SAFE.FOR(OF(I, K))
             ⊃
        (∃ T.SUB, ACTIV:
            START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
            ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, P), T.SUB, P))
            ∧ ACTION(ACTIV) = EXECUTE()
            ∧ TASK.ACTION(ACTIV) = K)

prove BR.LEMMA.37 [T.SUB ← *T.SUB:2,
                   ACTIV ← *ACTIV:2]
using BR.RE.MAPPING.4 [QQ ← P,
                       Y ← @:D]
      BR.A9A [R ← P,
              P ← *P:1,
              L ← K,
              T1.SUB ← *T.SUB:1]
      BR.D1 [II ← OF(I, K),
             T.SUB ← *T.SUB:2]
      BR.D1 [II ← OF(I, K),
             T.SUB ← *T.SUB:2,
             P ← *P:1]
      BR.LEMMA.19 [QQ ← *P:1,
                   T1.SUB ← *T.SUB:2,
                   T.SUB ← *T.SUB:2]
      BR.A16 [T.SUB ← *T.SUB:2,
              CON ← CONFIG(*T.SUB:2, *P:1),
              ACTIV ← *ACTIV:2]
      BR.A16 [T.SUB ← *T.SUB:1,
              P ← *P:1,
              ACTIV ← *ACTIV:1,
              CON ← CONFIG(*T.SUB:1, *P:1)]
      BR.LEMMA.12 [T.SUB ← *T.SUB:1,
                   T1.SUB ← *T.SUB:2]
```

```
/*
If a safe processor has a Dummy Vote scheduled and determines
        that the task is NOT.ON, then the INPUT value is the
        majority value.  This extends lemma 36.
*/

BR.LEMMA.38: formula
        QQ ∈ SAFE.FOR(OF(I, K))
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
        ∧ ACTION(ACTIV) = DUMMY.VOTE()
        ∧ 1 ≤ Y
        ∧ Y ≤ RESULT.SIZE(K)
        ∧ TASK.ACTION(ACTIV) = K
        ∧ NOT.ON.FRAME(K, QQ, BEGIN(OF(I, K)))
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
            ⊃
        SEQ.ELEM(INPUTIN.OF(QQ, K, START(SUB.INCR(T.SUB), QQ)), Y)
           = MAJORITY(D.BAG.D4(K, I, QQ, Y))

prove BR.LEMMA.38
using BR.A6B [P ← QQ]
      BR.D1 [II ← OF(I, K),
             P ← QQ]
      RE.BR.MAPPING.9 [P ← QQ]
      BR.A19.1
      BR.RE.MAPPING.8 [V ← BOTTOM1(K),
                        P ← QQ]
      BR.LEMMA.35
      MAJ.2 [M.BAG ← D.BAG.D4(K, I, QQ, Y),
             T2.V ← D1:8]
      BOTTOM.EQUALITY
      BR.A16 [CON ← CONFIG(T.SUB, QQ),
              P ← QQ]
      DATA.BOTTOM
      BR.A16 [P ← QQ,
              CON ← CONFIG(BEGIN(OF(I, K)), QQ)]
      BR.LEMMA.14 [T2.SUB ← BEGIN(OF(I, K))]
      BR.A18
      BR.LEMMA.12 [T.SUB ← BEGIN(OF(I, K)),
                   T1.SUB ← T.SUB]
```

```
/*
If a safe processor QQ is scheduled to Execute task K
        and one of its input tasks L is NOT.ON, the value
        in INPUT.IN at the beginning of the subframe after the Execute
        will be the correct IN value.  Note that because of the way
        the Execute is defined at this level, we want to use the input
        values present after the Execute. This allows a Vote and subsequent
        Execute within the same subframe.
*/

BR.LEMMA.39: formula
        QQ ∈ SAFE.FOR(DW.OF(I, K))
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ 1 ≤ Y
        ∧ Y ≤ RESULT.SIZE(L)
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ L ∈ INPUTS(K)
        ∧ NOT.ON.FRAME(L, QQ, BEGIN(OF(TO.OF(L, I, K), L)))
             ⊃
        SEQ.ELEM(INPUTIN.OF(QQ, L, START(SUB.INCR(T.SUB), QQ)), Y)
           = SEQ.ELEM(IN(L, TO.OF(L, I, K), QQ), Y)

prove BR.LEMMA.39
using BR.LEMMA.32 [I ← TO.OF(L, I, K),
                   K ← L]
      BR.LEMMA.14 [K ← L,
                   I ← TO.OF(L, I, K),
                   T.SUB ← BEGIN(OF(TO.OF(L, I, K), L)),
                   T2.SUB ← *T1.SUB:1]
      BR.A18 [I ← TO.OF(L, I, K),
              K ← L]
      BR.A19.1 [I ← TO.OF(L, I, K),
                K ← L]
      BR.LEMMA.12 [T.SUB ← BEGIN(OF(TO.OF(L, I, K), L)),
                   T1.SUB ← *T1.SUB:1]
      BR.A16 [T.SUB ← *T1.SUB:1,
              CON ← CONFIG(BEGIN(OF(TO.OF(L, I, K), L)), QQ),
              ACTIV ← *ACTIV2:1,
              K ← L,
              I ← TO.OF(L, I, K),
              P ← QQ]
      BR.LEMMA.38 [ACTIV ← *ACTIV2:1,
                   T.SUB ← *T1.SUB:1,
                   K ← L,
                   I ← TO.OF(L, I, K)]
      BR.LEMMA.33 [I ← TO.OF(L, I, K),
                   K ← L]
      BR.LEMMA.6 [T1.SUB ← *T1.SUB:1,
                  ACTIV2 ← *ACTIV2:1,
                  D1 ← SEQ.ELEM(IN(L, TO.OF(L, I, K), QQ), Y)]
      RP.L7 [P ← QQ]
      SCHED.LEMMA.1
SCHED.LEMMA.1: formula
        QQ ∈ SAFE.FOR(DW.OF(I, K)) ∧ L ∈ INPUTS(K)
             ⊃
        QQ ∈ SAFE.FOR(DW.FOR.TO.OF(L, I, K))


prove SCHED.LEMMA.1
using RP.D10 [II ← DW.OF(I, K),
              P ← QQ,
              T ← *T:2]
      RP.D10 [II ← DW.FOR.TO.OF(L, I, K),
              P ← QQ,
              T ← @:D]
      RP.D3.1
      RP.D3.3
      RP.D2.2
```

```
/*
The parallel argument to Lemma 39, this time where input
        task L is ON during the frame.
*/

BR.LEMMA.40: formula
        QQ ∈ SAFE.FOR(DW.OF(I, K))
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ 1 ≤ Y
        ∧ Y ≤ RESULT.SIZE(L)
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ L ∈ INPUTS(K)
        ∧ ¬NOT.ON.FRAME(L, QQ, BEGIN(OF(TO.OF(L, I, K), L)))
             ⊃
        SEQ.ELEM(INPUTIN.OF(QQ, L, START(SUB.INCR(T.SUB), QQ)), Y)
           = SEQ.ELEM(IN(L, TO.OF(L, I, K), QQ), Y)

prove BR.LEMMA.40
using BR.LEMMA.30 [I ← TO.OF(L, I, K),
                   K ← L,
                   P ← QQ]
      BR.A18 [I ← TO.OF(L, I, K),
              K ← L]
      BR.LEMMA.27 [ACTIV ← *ACTIV:1,
                   T.SUB ← *T.SUB:1,
                   K ← L,
                   I ← TO.OF(L, I, K)]
      BR.LEMMA.31 [I ← TO.OF(L, I, K),
                   K ← L]
      BR.LEMMA.6 [T1.SUB ← *T.SUB:1,
                  ACTIV2 ← *ACTIV:1,
                  D1 ← SEQ.ELEM(IN(L, TO.OF(L, I, K), QQ), Y)]
      RP.L7 [P ← QQ]
      SCHED.LEMMA.1
```

```
/*
Lemma 40 states that each element Y in the INPUT will be
        map to the IN value of the Replication specification.
        Here we state that the aggregate INPUT will thus also map.
*/

BR.LEMMA.41: formula
        QQ ∈ SAFE.FOR(DW.OF(I, K))
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ L ∈ INPUTS(K)
            ⊃
        INPUTIN.OF(QQ, L, START(SUB.INCR(T.SUB), QQ))
           = IN(L, TO.OF(L, I, K), QQ)

prove BR.LEMMA.41
using DATA.SIZE.IS.SEQ.LENGTH [K ← L,
                               I ← TO.OF(L, I, K)]
      DATA.SIZE.IS.SEQ.LENGTH.2 [P ← QQ,
                                 K ← L,
                                 T.REAL ← START(SUB.INCR(T.SUB), QQ)]
      DATA.EQUALITY [V ← INPUTIN.OF(QQ, L, START(SUB.INCR(T.SUB), QQ)),
                     V1 ← IN(L, TO.OF(L, I, K), QQ),
                     Y ← @:D]
      BR.LEMMA.40 [Y ← *Y:3]
      BR.LEMMA.39 [Y ← *Y:3]
```

```
/*
Continuing, having established that the correct input values
        will be present at the time of an execute, we now state that
        the DATAFILE just after execution in every safe processor will
        have the correct mathematical function performed on the
        specified inputs.
*/

BR.LEMMA.42: formula
        P ∈ SAFE.FOR(OF(I, K))
        ∧ QQ ∈ SAFE.FOR(DW.OF(I, K))
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
            ⊃
        DATAFILEIN.FOR.ON(P, K, QQ, START(SUB.INCR(T.SUB), P))
           = APPLY(FUNCTION(K), V.INPUTS.A3(K, I, QQ))

prove BR.LEMMA.42
using RP.A3A [P ← QQ,
              V.T ← *V.T:3]
      BR.LEMMA.41 [L ← SOURCE(*V.T:3)]
      BR.LEMMA.17 [V.INPUTS ← V.INPUTS.A3(K, I, QQ)]
      BR.A16 [CON ← CONFIG(T.SUB, QQ),
              P ← QQ]
      RP.L3 [P ← QQ]
      BR.LEMMA.3 [P ← QQ]
      BR.LEMMA.3
```

```
/*
... and therefore the ON.IN value present in every processor
        will thus reflect this correctly computed value.
*/

BR.LEMMA.43: formula
        P ∈ SAFE.FOR(OF(I, K))
        ∧ QQ ∈ SAFE.FOR(DW.OF(I, K))
        ∧ 1 ≤ Y
        ∧ Y ≤ RESULT.SIZE(K)
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
            ⊃
        SEQ.ELEM(ON.IN(K, I, QQ, P), Y)
           = SEQ.ELEM(APPLY(FUNCTION(K), V.INPUTS.A3(K, I, QQ)), Y)

prove BR.LEMMA.43
using BR.LEMMA.42
      BR.A16 [CON ← CONFIG(T.SUB, QQ),
              P ← QQ]
      RP.L3 [P ← QQ]
      BR.LEMMA.3 [P ← QQ]
      BR.LEMMA.3
      BR.A13A [CON ← CONFIG(T.SUB, QQ),
               P ← QQ,
               QQ ← P]
      BR.LEMMA.19 [P ← QQ,
                   QQ ← P,
                   T1.SUB ← *T1.SUB:6]
      BR.A16 [CON ← CONFIG(T.SUB, QQ),
              T.SUB ← *T1.SUB:6,
              ACTIV ← *ACTIV2:6]
      BR.D1 [T.SUB ← *T1.SUB:6,
             II ← OF(I, K)]
      BR.D1 [II ← OF(I, K),
             P ← QQ]
      BR.LEMMA.2 [T1.SUB ← *T1.SUB:6,
                  ACTIV2 ← *ACTIV2:6,
                  V ← APPLY(FUNCTION(K), V.INPUTS.A3(K, I, QQ))]
      DATA.SIZE.IS.SEQ.LENGTH.3
      DATA.SIZE.IS.SEQ.LENGTH.2 [T.REAL ← START(*T1.SUB:6, P)]
      DATA.EQUALITY [V ← DATAFILEIN.FOR.ON(P, K, QQ, START(*T1.SUB:6, P)),
                     V1 ← APPLY(FUNCTION(K), V.INPUTS.A3(K, I, QQ))]
      BR.LEMMA.15 [ACTIV ← *ACTIV2:6,
                   T.SUB ← *T1.SUB:6]
      BR.LEMMA.28 [ACTIV ← *ACTIV2:6,
                   T.SUB ← *T1.SUB:6]
      BR.LEMMA.12 [T1.SUB ← *T1.SUB:6]
```

```
/*
... and therefore the correct answer will be in the set of
        ON values, given the mapping of ON.IN to ON values.
*/

BR.LEMMA.44: formula
        QQ ∈ SAFE.FOR(DW.OF(I, K))
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
             ⊃
        APPLY(FUNCTION(K), V.INPUTS.A3(K, I, QQ)) ∈ ON(K, I, QQ)

prove BR.LEMMA.44
using BR.A22 [T1.SUB ← T.SUB,
              T.SUB ← BEGIN(OF(I, K)),
              P ← QQ,
              QQ ← QQ]
      BR.LEMMA.19 [P ← QQ,
                   T1.SUB ← BEGIN(OF(I, K)),
                   QQ ← QQ]
      BR.A16 [P ← QQ,
              CON ← CONFIG(T.SUB, QQ)]
      BR.D1 [II ← OF(I, K),
             P ← QQ]
      RP.L3 [P ← QQ]
      BR.A19.1
      BR.D1 [II ← OF(I, K),
             P ← QQ,
             T.SUB ← BEGIN(OF(I, K))]
      BR.RE.MAPPING.5 [P ← QQ,
                       QQ ← QQ,
                       Y ← @:D,
                       V ← APPLY(FUNCTION(K), V.INPUTS.A3(K, I, QQ))]
      BR.LEMMA.43 [Y ← *Y:8,
                   P ← QQ]
      BR.A18
      BR.LEMMA.12 [T1.SUB ← BEGIN(OF(I, K))]
```

```
/*
Finally, ... the ON set will contain only the correct value.
        Citing this and lemma 37 leads to the proof of RP.A3 -- the
        main Execute axiom of the Replication specification.
*/

BR.LEMMA.45: formula
        QQ ∈ SAFE.FOR(DW.OF(I, K))
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
             ⊃
        SINGLETON(ON(K, I, QQ), APPLY(FUNCTION(K), V.INPUTS.A3(K, I, QQ)))

prove BR.LEMMA.45
using BR.LEMMA.44
      CARD.3 [V.CARD.3 ← APPLY(FUNCTION(K), V.INPUTS.A3(K, I, QQ)),
              S ← ON(K, I, QQ)]
      BR.RE.MAPPING.5 [V ← *X:2,
                       Y ← *Y:5,
                       P ← QQ]
      BR.LEMMA.43 [P ← *QQ:3,
                   Y ← *Y:5]
      DATA.EQUALITY [V ← *X:2,
                     Y ← @:D,
                     V1 ← @V.CARD.3:2]
      LENGTH.OF.ELEMENTS.OF.ON.IS.LENGTH.OF.ON.IN [V ← *X:2]
      DATA.SIZE.IS.SEQ.LENGTH.3
      BR.A22 [T1.SUB ← T.SUB,
              T.SUB ← BEGIN(OF(I, K)),
              QQ ← *QQ:3,
              P ← QQ]
      BR.LEMMA.12 [T1.SUB ← BEGIN(OF(I, K))]
      BR.LEMMA.19 [T1.SUB ← BEGIN(OF(I, K)),
                   P ← QQ,
                   QQ ← *QQ:3]
      BR.D1 [II ← OF(I, K),
             P ← QQ]
      BR.D1 [II ← OF(I, K),
             T.SUB ← BEGIN(OF(I, K)),
             P ← *QQ:3]
      RP.L3 [P ← QQ]
      BR.A19.1
      BR.A16 [P ← QQ,
              CON ← CONFIG(T.SUB, QQ)]
      BR.A18
```

```
/* The remaining lemmas are used to prove lemma 7. */

/*
An input task executes either during the same frame as
        its input task or during the previous frame.  The iteration
        numbers are thus off by at most one.
*/

BR.LEMMA.46: formula
        L ∈ INPUTS(K)  ⊃  TO.OF(L, I, K) = I ∨ TO.OF(L, I, K)+1 = I

prove BR.LEMMA.46
using BR.A18
      BR.A18 [K ← L,
              I ← INCR(TO.OF(L, I, K))]
      BR.LEMMA.52 [T.SUB ← BEGIN(OF(I, K)),
                   T1.SUB ← SUB.DECR(END(OF(INCR(TO.OF(L, I, K)), L))),
                   J ← INCR(TO.OF(L, I, K))]
      BR.LEMMA.52 [I ← INCR(TO.OF(L, I, K)),
                   J ← TO.OF(L, I, K),
                   T.SUB ← SUB.DECR(END(OF(INCR(TO.OF(L, I, K)), L))),
                   T1.SUB ← SUB.DECR(END(OF(TO.OF(L, I, K), L)))]
      BR.A18 [K ← L,
              I ← TO.OF(L, I, K)]
      BR.LEMMA.52 [T1.SUB ← BEGIN(OF(I, K)),
                   T.SUB ← SUB.DECR(END(OF(TO.OF(L, I, K), L))),
                   I ← TO.OF(L, I, K),
                   J ← I]
      BR.A40
```

```
/*
If an input task L executes within the previous frame,
        then the next iteration of L can't end until after the
        beginning of K -- this establishes that the TO.OF function
        picks up the value present in the state.
*/

BR.LEMMA.47: formula
        L ∈ INPUTS(K) ∧ INCR(TO.OF(L, I, K)) = I
             ⊃
        BEGIN(OF(I, K)) < END(OF(I, L))

prove BR.LEMMA.47
using BR.A40
      BR.A18
      BR.A18 [I ← TO.OF(L, I, K),
              K ← L]
      BR.A27
      TIMES.AXIOM.1 [INT1 ← I,
                     INT2 ← TO.OF(L, I, K),
                     INT3 ← FRAME.SIZE()]
```

```
var CON1: CONFIGS

/*
Draw a picture... Consider an Execute for the
        i-th iteration of task K and a Vote
        for the corresponding input task L.  Let there be another
        Vote on L scheduled after the first vote
        and before the Execute.  We state that either the second Vote
        is outside the execution window and in the same frame as the
        Execute on K, or there is in fact only one Vote.  This involves
        a case split on whether the corresponding Vote on L is in the
        same frame as the Execute of K or in the previous frame.
*/

BR.LEMMA.48: axiom
        P ∈ SAFE.FOR(DW.FOR.TO.OF(L, I, K))
        ∧ L ∈ INPUTS(K)
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, P), T.SUB, P))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ START.FRAME(FRAME(T1.SUB)) = TO.OF(L, I, K)*FRAME.SIZE()
        ∧ SEQ.MEMBER(ACTIV2, SCHED(CONFIG(T1.SUB, P), T1.SUB, P))
        ∧ ((ACTION(ACTIV2) = VOTE() ∧ ELEM.ACTION(ACTIV2) = Y)
           ∨ ACTION(ACTIV2) = DUMMY.VOTE())
        ∧ TASK.ACTION(ACTIV2) = L
        ∧ FRAME(T.SUB) = FRAME(T2.SUB)
        ∧ T2.SUB > T1.SUB
        ∧ T2.SUB ≤ T.SUB
        ∧ TASK.ACTION(ACTIV3) = L
        ∧ ((ACTION(ACTIV3) = VOTE() ∧ ELEM.ACTION(ACTIV3) = Y)
           ∨ ACTION(ACTIV3) = DUMMY.VOTE())
        ∧ SEQ.MEMBER(ACTIV3, SCHED(CONFIG(T2.SUB, P), T2.SUB, P))
             ⊃
        (BEGIN(OF(I, K)) ≤ T2.SUB ∧ T2.SUB ≤ T.SUB ∧ T.SUB < END(OF(I, K)))
        ∨ (BEGIN(OF(I, L)) ≤ T2.SUB
           ∧ T2.SUB < BEGIN(OF(I, K))
           ∧ BEGIN(OF(I, K)) < END(OF(I, L)))
```

```
prove BR.LEMMA.48
using BR.LEMMA.47
      BR.LEMMA.46
      BR.A40
      BR.A16 [CON ← CONFIG(T.SUB, P)]
      BR.A16 [T.SUB ← T2.SUB,
              ACTIV ← ACTIV3,
              CON ← CONFIG(T2.SUB, P),
              K ← L]
      BR.LEMMA.19 [QQ ← P,
                   T.SUB ← T1.SUB,
                   I ← TO.OF(L, I, K),
                   T1.SUB ← T2.SUB,
                   K ← L]
      BR.A16 [T.SUB ← T1.SUB,
              K ← L,
              CON ← CONFIG(T1.SUB, P),
              ACTIV ← ACTIV2,
              I ← TO.OF(L, I, K)]
      BR.LEMMA.21 [T.SUB ← T2.SUB,
                   K ← L,
                   I ← TO.OF(L, I, K)]
      BR.A19.1 [K ← L,
                I ← TO.OF(L, I, K)]
      BR.A12A [T.SUB ← T1.SUB,
               CON ← CONFIG(T1.SUB, P),
               K ← L,
               ACTIV ← ACTIV2,
               T1.SUB ← T2.SUB]
      BR.A18
      BR.A18 [I ← TO.OF(L, I, K),
              K ← L]
      BR.LEMMA.12 [T.SUB ← BEGIN(OF(I, K)),
                   T1.SUB ← BEGIN(OF(TO.OF(L, I, K), L))]
      BR.LEMMA.21 [T.SUB ← T1.SUB,
                   I ← TO.OF(L, I, K),
                   K ← L]
      BR.LEMMA.21 [T.SUB ← T2.SUB,
                   I ← TO.OF(L, I, K),
                   K ← L]
      BR.A28
      BR.A29
      BR.D1 [T.SUB ← T1.SUB,
             II ← DW.FOR.TO.OF(L, I, K)]
      BR.D1 [T.SUB ← T2.SUB,
             II ← DW.FOR.TO.OF(L, I, K)]
```

```
/*
In the situation described in the previous lemma, the
        configuration at the time of the Vote will be the same
        as at the time of the Execute, even though the Vote
        and Execute may be in different frames.  This follows
        because of schedule constaints, lemma 48, and misc.
*/

BR.LEMMA.49: axiom
        P ∈ SAFE.FOR(DW.FOR.TO.OF(L, I, K))
        ∧ L ∈ INPUTS(K)
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, P), T.SUB, P))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ START.FRAME(FRAME(T1.SUB)) = TO.OF(L, I, K)*FRAME.SIZE()
        ∧ SEQ.MEMBER(ACTIV2, SCHED(CONFIG(T1.SUB, P), T1.SUB, P))
        ∧ ((ACTION(ACTIV2) = VOTE() ∧ ELEM.ACTION(ACTIV2) = Y)
           ∨ ACTION(ACTIV2) = DUMMY.VOTE())
        ∧ TASK.ACTION(ACTIV2) = L
        ∧ FRAME(T.SUB) = FRAME(T2.SUB)
        ∧ T2.SUB > T1.SUB
        ∧ T2.SUB ≤ T.SUB
        ∧ TASK.ACTION(ACTIV3) = L
        ∧ ((ACTION(ACTIV3) = VOTE() ∧ ELEM.ACTION(ACTIV3) = Y)
           ∨ ACTION(ACTIV3) = DUMMY.VOTE())
        ∧ SEQ.MEMBER(ACTIV3, SCHED(CONFIG(T2.SUB, P), T2.SUB, P))
            ⊃
        CONFIG(T.SUB, P) = CONFIG(T2.SUB, P)

prove BR.LEMMA.49
using BR.LEMMA.19 [QQ ← P,
                   T1.SUB ← BEGIN(OF(I, K))]
      BR.LEMMA.19 [QQ ← P,
                   T1.SUB ← T2.SUB]
      BR.LEMMA.19 [QQ ← P,
                   K ← L,
                   T.SUB ← T2.SUB,
                   T1.SUB ← BEGIN(OF(I, K))]
      BR.LEMMA.48
      BR.A40
      BR.A19.2
      BR.A28
      BR.A29
      BR.A16 [CON ← CONFIG(T.SUB, P)]
      BR.A16 [K ← L,
              T.SUB ← T1.SUB,
              ACTIV ← ACTIV2,
              I ← TO.OF(L, I, K),
              CON ← CONFIG(T1.SUB, P)]
      BR.A19.1
      BR.D1 [II ← DW.FOR.TO.OF(L, I, K)]
      BR.D1 [II ← DW.FOR.TO.OF(L, I, K),
             T.SUB ← T2.SUB]
      BR.D1 [II ← DW.FOR.TO.OF(L, I, K),
             T.SUB ← BEGIN(OF(I, K))]
      BR.A19.1 [K ← L,
                I ← TO.OF(L, I, K)]
```

```
/*
Furthermore, ... the configuration at the time of the
        possible two Votes will be the same.
*/

BR.LEMMA.50: axiom
        P ∈ SAFE.FOR(DW.FOR.TO.OF(L, I, K))
        ∧ L ∈ INPUTS(K)
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, P), T.SUB, P))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ START.FRAME(FRAME(T1.SUB)) = TO.OF(L, I, K)*FRAME.SIZE()
        ∧ SEQ.MEMBER(ACTIV2, SCHED(CONFIG(T1.SUB, P), T1.SUB, P))
        ∧ ((ACTION(ACTIV2) = VOTE() ∧ ELEM.ACTION(ACTIV2) = Y)
           ∨ ACTION(ACTIV2) = DUMMY.VOTE())
        ∧ TASK.ACTION(ACTIV2) = L
        ∧ FRAME(T1.SUB) = FRAME(T2.SUB)
        ∧ T2.SUB > T1.SUB
        ∧ T2.SUB ≤ T.SUB
        ∧ TASK.ACTION(ACTIV3) = L
        ∧ ((ACTION(ACTIV3) = VOTE() ∧ ELEM.ACTION(ACTIV3) = Y)
           ∨ ACTION(ACTIV3) = DUMMY.VOTE())
        ∧ SEQ.MEMBER(ACTIV3, SCHED(CONFIG(T2.SUB, P), T2.SUB, P))
             ⊃
        CONFIG(T1.SUB, P) = CONFIG(T2.SUB, P)


prove BR.LEMMA.50
using BR.A16 [CON ← CONFIG(T.SUB, P)]
      BR.A16 [K ← L,
              I ← TO.OF(L, I, K),
              T.SUB ← T1.SUB,
              ACTIV ← ACTIV2,
              CON ← CONFIG(T1.SUB, P)]
      BR.A16 [K ← L,
              I ← TO.OF(L, I, K),
              T.SUB ← T2.SUB,
              ACTIV ← ACTIV3,
              CON ← CONFIG(T2.SUB, P)]
      BR.LEMMA.19 [QQ ← P,
                   K ← L,
                   I ← TO.OF(L, I, K),
                   T.SUB ← T1.SUB,
                   T1.SUB ← T2.SUB]
      BR.A19.1
      BR.A40
      BR.A28
      BR.A29
      BR.D1 [T.SUB ← T1.SUB,
             II ← DW.FOR.TO.OF(L, I, K)]
      BR.D1 [T.SUB ← T2.SUB,
             II ← DW.FOR.TO.OF(L, I, K)]
```

```
/*
...And, either the two Votes are in the same frame or the
        second Vote is in the same frame as the Execute.
*/

BR.LEMMA.51: axiom
        P ∈ SAFE.FOR(DW.FOR.TO.OF(L, I, K))
        ∧ L ∈ INPUTS(K)
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, P), T.SUB, P))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ START.FRAME(FRAME(T1.SUB)) = TO.OF(L, I, K)*FRAME.SIZE()
        ∧ SEQ.MEMBER(ACTIV2, SCHED(CONFIG(T1.SUB, P), T1.SUB, P))
        ∧ ((ACTION(ACTIV2) = VOTE() ∧ ELEM.ACTION(ACTIV2) = Y)
           ∨ ACTION(ACTIV2) = DUMMY.VOTE())
        ∧ TASK.ACTION(ACTIV2) = L
        ∧ T2.SUB > T1.SUB
        ∧ T2.SUB ≤ T.SUB
        ∧ TASK.ACTION(ACTIV3) = L
        ∧ ((ACTION(ACTIV3) = VOTE() ∧ ELEM.ACTION(ACTIV3) = Y)
           ∨ ACTION(ACTIV3) = DUMMY.VOTE())
        ∧ SEQ.MEMBER(ACTIV3, SCHED(CONFIG(T2.SUB, P), T2.SUB, P))
            ⊃
        FRAME(T1.SUB) = FRAME(T2.SUB) ∨ FRAME(T.SUB) = FRAME(T2.SUB)


prove BR.LEMMA.51
using BR.LEMMA.46
      BR.LEMMA.12
      BR.A21 [T.SUB ← T1.SUB]
      BR.A21 [T1.SUB ← T.SUB]
      BR.A21 [T1.SUB ← T2.SUB]
      BR.A21 [T.SUB ← T1.SUB,
              T1.SUB ← T2.SUB]
      BR.A21B [T.FRAME ← FRAME(T1.SUB)]
      TIMES.AXIOM.2 [INT1 ← TO.OF(L, I, K),
                     INT2 ← FRAME.SIZE()]
```

```
/* Later iterations of a task are performed during later subframes. */

BR.LEMMA.52: formula
        START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ START.FRAME(FRAME(T1.SUB)) = J*FRAME.SIZE()
        ∧ I > J
             ⊃
        T.SUB > T1.SUB


prove BR.LEMMA.52
using BR.A21 [T1.SUB ← T.SUB]
      BR.A21B [T.FRAME ← FRAME(T.SUB)]
      BR.A21 [T.SUB ← T1.SUB]
      BR.A21B [T.FRAME ← FRAME(T1.SUB)]
      NAT.NONNEGATIVE [Y ← FRAME.SIZE()]
      TIMES.AXIOM.3 [INT1 ← I,
                     INT2 ← J,
                     INT3 ← FRAME.SIZE()]
      TIMES.AXIOM.1 [INT1 ← I,
                     INT2 ← J,
                     INT3 ← FRAME.SIZE()]
      BR.LEMMA.12
      BR.A21 [T1.SUB ← T.SUB,
              T.SUB ← T1.SUB]
```

```
/*
Given a pair of safe processors, the ON.IN value received
        will be among the values mapped up to ON.
*/

BR.LEMMA.53: formula
        P ∈ SAFE.FOR(OF(I, K)) ∧ QQ ∈ SAFE.FOR(OF(I, K))
            ⊃
        ON.IN(K, I, P, QQ) ∈ ON(K, I, P)


prove BR.LEMMA.53
using BR.RE.MAPPING.5 [V ← ON.IN(K, I, P, QQ),
                       Y ← @:D]
      BR.RE.MAPPING.6 [V ← BOTTOM1(K),
                       Y ← @:D,
                       T.SUB ← @:D,
                       ACTIV ← @:D]
```

```
/* As the formula name implies ... */

REGRETABLE.EVIL: formula
        ¬(∃ ACTIV:
             SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
             ∧ ACTION(ACTIV) = EXECUTE()
             ∧ TASK.ACTION(ACTIV) = K)
             ⊃
        ¬(∃ ACTIV2:
             SEQ.MEMBER(ACTIV2, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
             ∧ ACTION(ACTIV2) = EXECUTE()
             ∧ TASK.ACTION(ACTIV2) = K)
        ∧ ¬(∃ ACTIV3:
               SEQ.MEMBER(ACTIV3, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
               ∧ ACTION(ACTIV3) = EXECUTE()
               ∧ TASK.ACTION(ACTIV3) = K)
```

```
/*
**********       Proofs of Replication Model Axioms       **********
*/

prove RP.D4
using BR.LEMMA.33
      BR.LEMMA.31

prove RP.A3
using INTERSECT [X ← P,
                 S ← POLL.FOR.OF(I, K),
                 S1 ← SAFE.FOR(DW.OF(I, K))]
      RP.L3
      BR.LEMMA.37
      BR.LEMMA.45 [QQ ← P,
                   T.SUB ← *T.SUB:3,
                   ACTIV ← *ACTIV:3]

prove RP.A2 [QQ ← *QQ:3]
using INTERSECT [S ← POLL.FOR.OF(I, K),
                 S1 ← SAFE.FOR(DW.OF(I, K)),
                 X ← P]
      BR.LEMMA.34 [QQ ← @QQ:C]
      BR.RE.MAPPING.5 [Y ← *Y:4,
                       QQ ← @:D]
      DATA.EQUALITY [V1 ← ON.IN(K, I, P, *QQ:3),
                     Y ← @:D]
      LENGTH.OF.ELEMENTS.OF.ON.IS.LENGTH.OF.ON.IN [QQ ← P]
      DATA.SIZE.IS.SEQ.LENGTH [QQ ← *QQ:3]
      BR.LEMMA.34 [QQ ← *QQ:C]
      BR.RE.MAPPING.5 [QQ ← *QQ:C,
                       Y ← @Y:D]
      DATA.EQUALITY [V1 ← ON.IN(K, I, P, *QQ:C),
                     Y ← *Y:8]
      DATA.SIZE.IS.SEQ.LENGTH [QQ ← *QQ:C]
```

SUBSECTION 7.15

PREPOST AXIOMS

```
/* ********** Prepost Level Axioms derived by Program Proof ********** */

CARDINALITY: (S) → NAT = CARD(S)

ARRAY.TYPE: type(TYPE1) is SEQ(TYPE1)

INDEX: (ARRAY.TYPE(TYPE1), NAT) → TYPE1

TASK: type is NAT

PROC: type is NAT

ACTIVITIES: type is NAT

NULL.ENTRY: () → ACTIVITIES = 0

STATE: type is PAIR.OF(SUBFRAMETIME, PROC)

var STATE.SIFT: STATE

MAKE.STATE: (T.SUB, P) → STATE = MAKE.PAIR(T.SUB, P)

S.SUB: (STATE.SIFT) → SUBFRAMETIME = FIRST(STATE.SIFT)

S.PROC: (STATE.SIFT) → PROC = SECOND(STATE.SIFT)

NEXT: (STATE.SIFT) → STATE
        = MAKE.STATE(SUB.INCR(S.SUB(STATE.SIFT)), S.PROC(STATE.SIFT))

P.SUBFRAMETIME: type is NAT

var T.PSUB: P.SUBFRAMETIME

BOTTOM_VAL: () → DATAVAL = BOTTOMD()

INSET: (X, S) → BOOL = X ∈ S

MOD.FRAMESIZE: (T.SUB) → P.SUBFRAMETIME = MOD(T.SUB, FRAME.SIZE())

SUBFRAME: (STATE.SIFT) → P.SUBFRAMETIME = MOD.FRAMESIZE(S.SUB(STATE.SIFT))

P.SUBFRAMETIME.BOUNDS: axiom
        T.PSUB < FRAME.SIZE()

P.CONFIG: (STATE) → CONFIGS

var TYPE3: type

RECORD.TYPE: type(TYPE1, TYPE2, TYPE3)

ACTIVITY.RECORD.TYPE: type is RECORD.TYPE(ACTIVITIES, TASK, NAT)

SCHED_TABLE:
   → ARRAY.TYPE(ARRAY.TYPE(ARRAY.TYPE(ARRAY.TYPE(ACTIVITY.RECORD.TYPE))))

TABLE_LENGTH: (ARRAY.TYPE(ACTIVITY.RECORD.TYPE)) → NAT

var TAB: ARRAY.TYPE(ACTIVITY.RECORD.TYPE)
```

```
MAX_ACTIVITIES: → NAT
```

```
DOT.TASKNAME: (ACTIVITY.RECORD.TYPE) → TASK

DOT.ACTIVITY: (ACTIVITY.RECORD.TYPE) → ACTIVITIES

DOT.ELEM: (ACTIVITY.RECORD.TYPE) → NAT

INPUT: (STATE) → ARRAY.TYPE(ARRAY.TYPE(DATAVAL))

DATAFILE: (STATE) → ARRAY.TYPE(ARRAY.TYPE(ARRAY.TYPE(DATAVAL)))

P.INPUTS: → ARRAY.TYPE(ARRAY.TYPE(NAT))

POLL: → ARRAY.TYPE(ARRAY.TYPE(ARRAY.TYPE(BOOL)))

RESULT_SIZE: → ARRAY.TYPE(NAT)

NULL_TASK: → TASK

MAX.ACTIVITIES: → NAT

var INP: ARRAY.TYPE(ARRAY.TYPE(DATAVAL))

TASK_RESULTS: (TASK, ARRAY.TYPE(ARRAY.TYPE(DATAVAL))) → ARRAY.TYPE(DATAVAL)

VOTE: → ACTIVITIES

DUMMY_VOTE: () → ACTIVITIES = DUMMY.VOTE()

VIRT_TO_REAL: (STATE) → ARRAY.TYPE(PROC)

REAL_TO_VIRT: (STATE) → ARRAY.TYPE(PROC)

var TI: TASK

var EI: NAT

var JI: NAT

var PROCS: SET.OF(PROC)

var MAJ_PROCS: SET.OF(PROC)

var MAJI: DATAVAL

var MAJI2: DATAVAL

var TASKI: TASK

var JI1: NAT

SET.FN#9: (DATAVAL, STATE, NAT, TASK, CONFIGS) → SET.OF(PROC)

MAX_PROCESSORS: → NAT
```

```
/* Schedule table axiom */

SCHED.TABLE.LENGTH.AXIOM: axiom
        TABLE_LENGTH(TAB) = Y
             ≡
        DOT.ACTIVITY(INDEX(TAB, Y+INT.NAT(1))) = NULL.ENTRY()
        ∧ (∀ Z: Z ≤ Y ∧ Z > 0  ⊃  ¬(DOT.ACTIVITY(INDEX(TAB, Z)) = NULL.ENTRY()))


/* Set Abstraction definition */

SET.ABSTR#9: axiom
        INSET(QQ, SET.FN#9(MAJI, STATE.SIFT, EI, TI, CON))
             ≡
        QQ ≥ 1
        ∧ QQ ≤ MAX_PROCESSORS()
        ∧ INDEX(INDEX(INDEX(POLL(), CON), INDEX(REAL_TO_VIRT(STATE.SIFT), QQ)),
                TI)
        ∧ MAJI = INDEX(INDEX(INDEX(DATAFILE(STATE.SIFT), QQ), TI), EI)

SET.FN#10: (TASK, CONFIGS, STATE) → SET.OF(PROC)

/* Another set abstraction */

SET.FN.AXIOM.10: axiom
        ∀ P, TI, CON:
           INSET(P, SET.FN#10(TI, CON, STATE.SIFT))
                ≡
           P ≥ 1
           ∧ P ≤ MAX_PROCESSORS()
           ∧ INDEX(INDEX(INDEX(POLL(), CON),
                         INDEX(REAL_TO_VIRT(STATE.SIFT), P)),
                   TI)
```

```
/*
This formula, characterizing the effect of
        having no Vote within a frame has been proven
        of the code by verification condition generation
*/

VOTE.FRAME.AXIOM: axiom
        ∀ TI, EI:
           ¬(∃ JI:
                WORKING.DURING(S.PROC(STATE.SIFT), S.SUB(STATE.SIFT))
                ∧ JI ≥ 1
                ∧ JI ≤ MAX_ACTIVITIES()
                ∧ TI = DOT.TASKNAME(INDEX(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                            INDEX
                                                            (REAL_TO_VIRT
                                                             (STATE.SIFT),
                                                             S.PROC(STATE.SIFT))
                                                            ),
                                                      P.CONFIG(STATE.SIFT)),
                                                SUBFRAME(STATE.SIFT)),
                                          JI))
                ∧ ((DOT.ACTIVITY(INDEX(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                         INDEX
                                                         (REAL_TO_VIRT
                                                          (STATE.SIFT),
                                                          S.PROC(STATE.SIFT))),
                                                   P.CONFIG(STATE.SIFT)),
                                             SUBFRAME(STATE.SIFT)),
                                       JI))
                       = VOTE()
                    ∧ EI = DOT.ELEM(INDEX(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                            INDEX
                                                            (REAL_TO_VIRT
                                                             (STATE.SIFT),
                                                             S.PROC(STATE.SIFT))
                                                            ),
                                                      P.CONFIG(STATE.SIFT)),
                                                SUBFRAME(STATE.SIFT)),
                                          JI)))
                   ∨ DOT.ACTIVITY(INDEX(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                          INDEX
                                                          (REAL_TO_VIRT
                                                           (STATE.SIFT),
                                                           S.PROC(STATE.SIFT))),
                                                    P.CONFIG(STATE.SIFT)),
                                              SUBFRAME(STATE.SIFT)),
                                        JI))
                       = DUMMY_VOTE()))
               ⊃
           INDEX(INDEX(INPUT(NEXT(STATE.SIFT)), TI), EI)
              = INDEX(INDEX(INPUT(STATE.SIFT), TI), EI)
```

```
/*
This axiom describing the effect of a Vote on
         the state has been proven of the Pascal code.
*/

VOTE.ACTIVITY: axiom
        ∀ TI, EI:
           (∃ JI:
               WORKING.DURING(S.PROC(STATE.SIFT), S.SUB(STATE.SIFT))
               ∧ JI ≥ 1
               ∧ JI ≤ MAX_ACTIVITIES()
               ∧ DOT.ACTIVITY(INDEX(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                      INDEX
                                                      (REAL_TO_VIRT(STATE.SIFT),
                                                       S.PROC(STATE.SIFT))),
                                                P.CONFIG(STATE.SIFT)),
                                          SUBFRAME(STATE.SIFT)),
                                    JI))
                    = VOTE()
               ∧ TI = DOT.TASKNAME(INDEX(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                           INDEX
                                                           (REAL_TO_VIRT
                                                            (STATE.SIFT),
                                                            S.PROC(STATE.SIFT)))
                                                     ,
                                                     P.CONFIG(STATE.SIFT)),
                                               SUBFRAME(STATE.SIFT)),
                                         JI))
               ∧ EI = DOT.ELEM(INDEX(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                       INDEX
                                                       (REAL_TO_VIRT(STATE.SIFT)
                                                        , S.PROC(STATE.SIFT))),
                                                 P.CONFIG(STATE.SIFT)),
                                           SUBFRAME(STATE.SIFT)),
                                     JI)))
                ⊃
           if ∃ MAJI2:
                 CARDINALITY(SET.FN#9(MAJI2, STATE.SIFT, EI, TI,
                                      P.CONFIG(STATE.SIFT)))
                 *2
                    > CARDINALITY(SET.FN#10(TI, P.CONFIG(STATE.SIFT),
                                            STATE.SIFT))
              then
                 ∀ MAJI:
                    CARDINALITY(SET.FN#9(MAJI, STATE.SIFT, EI, TI,
                                         P.CONFIG(STATE.SIFT)))
                    *2
                       > CARDINALITY(SET.FN#10(TI, P.CONFIG(STATE.SIFT),
                                               STATE.SIFT))
                         ⊃
                    INDEX(INDEX(INPUT(NEXT(STATE.SIFT)), TI), EI) = MAJI
              else BOTTOMD() = INDEX(INDEX(INPUT(NEXT(STATE.SIFT)), TI), EI)
           end if
```

```
/* Similarly for the Dummy Vote */

DUMMY_VOTE.ACTIVITY: axiom
        ∀ TI, EI:
           (∃ JI:
               WORKING.DURING(S.PROC(STATE.SIFT), S.SUB(STATE.SIFT))
               ∧ JI ≥ 1
               ∧ JI ≤ MAX_ACTIVITIES()
               ∧ DOT.ACTIVITY(INDEX(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                       INDEX
                                                       (REAL_TO_VIRT(STATE.SIFT),
                                                        S.PROC(STATE.SIFT))),
                                                 P.CONFIG(STATE.SIFT)),
                                           SUBFRAME(STATE.SIFT)),
                                     JI))
                    = DUMMY_VOTE()
               ∧ TI = DOT.TASKNAME(INDEX(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                            INDEX
                                                            (REAL_TO_VIRT
                                                             (STATE.SIFT),
                                                             S.PROC(STATE.SIFT)))
                                                      ,
                                                      P.CONFIG(STATE.SIFT)),
                                                SUBFRAME(STATE.SIFT)),
                                          JI)))
           ∧ EI ≥ 1
           ∧ EI ≤ INDEX(RESULT_SIZE(), TI)
               ⊃
           INDEX(INDEX(INPUT(NEXT(STATE.SIFT)), TI), EI) = BOTTOM_VAL()
```

```
/* and for the lack of an Execute during a frame */

EXECUTE.FRAME.AXIOM: axiom
        ∀ P, TI, EI:
           WORKING.DURING(S.PROC(STATE.SIFT), S.SUB(STATE.SIFT))
           ∧ ¬(∃ JI:
                  JI ≥ 1
                  ∧ JI ≤ MAX_ACTIVITIES()
                  ∧ DOT.ACTIVITY(INDEX(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                        INDEX
                                                        (REAL_TO_VIRT
                                                         (STATE.SIFT),
                                                         S.PROC(STATE.SIFT))),
                                                  P.CONFIG(STATE.SIFT)),
                                            SUBFRAME(STATE.SIFT)),
                                       JI))
                       = EXECUTE()
                  ∧ DOT.TASKNAME(INDEX(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                        INDEX
                                                        (REAL_TO_VIRT
                                                         (STATE.SIFT),
                                                         S.PROC(STATE.SIFT))),
                                                  P.CONFIG(STATE.SIFT)),
                                            SUBFRAME(STATE.SIFT)),
                                       JI))
                       = TI)
                ⊃
           INDEX(INDEX(INDEX(DATAFILE(NEXT(MAKE.STATE(S.SUB(STATE.SIFT),
                                                     S.PROC(STATE.SIFT))))),
                             S.PROC(STATE.SIFT)),
                       TI),
                 EI)
              = INDEX(INDEX(INDEX(DATAFILE(MAKE.STATE(S.SUB(STATE.SIFT),
                                                     S.PROC(STATE.SIFT))),
                                  S.PROC(STATE.SIFT)),
                            TI),
                      EI)
```

```
/* and for an Execute activity */

EXECUTE.ACTIVITY: axiom
        ∀ TI, INP:
           WORKING.DURING(S.PROC(STATE.SIFT), S.SUB(STATE.SIFT))
           ∧ (∃ JI:
                 JI ≥ 1
                 ∧ JI ≤ MAX_ACTIVITIES()
                 ∧ DOT.ACTIVITY(INDEX(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                        INDEX
                                                        (REAL_TO_VIRT
                                                         (STATE.SIFT),
                                                         S.PROC(STATE.SIFT))),
                                                  P.CONFIG(STATE.SIFT)),
                                            SUBFRAME(STATE.SIFT)),
                                      JI))
                      = EXECUTE()
                 ∧ DOT.TASKNAME(INDEX(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                        INDEX
                                                        (REAL_TO_VIRT
                                                         (STATE.SIFT),
                                                         S.PROC(STATE.SIFT))),
                                                  P.CONFIG(STATE.SIFT)),
                                            SUBFRAME(STATE.SIFT)),
                                      JI))
                      = TI
                 ∧ (∀ TASKI, JI1, EI:
                       EI ≥ 1 ∧ EI ≤ INDEX(RESULT_SIZE(), TASKI)
                            ⊃
                       (INDEX(INDEX(P.INPUTS(), TI), JI1) = TASKI
                        ∧ ¬(TASKI = NULL_TASK())
                             ⊃
                        INDEX(INDEX(INP, JI1), EI)
                           = INDEX(INDEX(INPUT(NEXT(STATE.SIFT)), TASKI), EI))))
                ⊃
           INDEX(INDEX(DATAFILE(NEXT(MAKE.STATE(S.SUB(STATE.SIFT),
                                                S.PROC(STATE.SIFT)))),
                       S.PROC(STATE.SIFT)),
                 TI)
              = TASK_RESULTS(TI, INP)
```

```
SIFT.PARAMETER.INVARIANT#7: axiom
        VOTE() > 0 ∧ DUMMY_VOTE() > 0 ∧ EXECUTE() > 0

SIFT.PARAMETER.INVARIANT#8: axiom
        ¬(VOTE() = DUMMY_VOTE())
        ∧ ¬(VOTE() = EXECUTE())
        ∧ ¬(DUMMY_VOTE() = EXECUTE())

SIFT.PARAMETER.INVARIANT#17: axiom
        ∀ P, CON, T.PSUB, JI, Y:
           DOT.ACTIVITY(INDEX(INDEX(INDEX(INDEX(SCHED_TABLE(), P), CON),
                                         T.PSUB),
                                   JI))
              = 0
           ∧ Y > JI
               ⊃
           DOT.ACTIVITY(INDEX(INDEX(INDEX(INDEX(SCHED_TABLE(), P), CON),
                                         T.PSUB),
                                   Y))
              = 0

using PREPOST.MAPPING
```

SUBSECTION 7.16

PREPOST MAPPING

```
/*
********** Mapping from Prepost Specification
                    to Activity Specification           **********
*/

PP.MAPPING.1: axiom
        RESULT.SIZE(K) = INDEX(RESULT_SIZE(), K)

PP.MAPPING.2: axiom
        CONFIG(T.SUB, P) = P.CONFIG(MAKE.STATE(T.SUB, P))

PP.MAPPING.3: axiom
        Y ≥ 1
        ∧ Y ≤ TABLE_LENGTH(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                               INDEX
                                               (REAL_TO_VIRT(MAKE.STATE(T.SUB,
                                                                         P)),
                                                P)),
                                         P.CONFIG(MAKE.STATE(T.SUB, P))),
                                  SUBFRAME(MAKE.STATE(T.SUB, P))))
              ⊃
        ACTION(SEQ.ELEM(SCHED(CONFIG(T.SUB, P), T.SUB, P), Y-INT.NAT(1)))
           = DOT.ACTIVITY(INDEX(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                    INDEX
                                                    (REAL_TO_VIRT
                                                     (MAKE.STATE(T.SUB, P)),
                                                     P)),
                                              P.CONFIG(MAKE.STATE(T.SUB, P))),
                                        SUBFRAME(MAKE.STATE(T.SUB, P))),
                                 Y))
        ∧ TASK.ACTION(SEQ.ELEM(SCHED(CONFIG(T.SUB, P), T.SUB, P), Y-INT.NAT(1)))
             = DOT.TASKNAME(INDEX(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                      INDEX
                                                      (REAL_TO_VIRT
                                                       (MAKE.STATE(T.SUB, P)),
                                                       P)),
                                                P.CONFIG(MAKE.STATE(T.SUB, P))),
                                          SUBFRAME(MAKE.STATE(T.SUB, P))),
                                   Y))
        ∧ ELEM.ACTION(SEQ.ELEM(SCHED(CONFIG(T.SUB, P), T.SUB, P), Y-INT.NAT(1)))
             = DOT.ELEM(INDEX(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                  INDEX
                                                  (REAL_TO_VIRT(MAKE.STATE(T.SUB,

                                                                            P
)),

                                                   P)),
                                            P.CONFIG(MAKE.STATE(T.SUB, P))),
                                      SUBFRAME(MAKE.STATE(T.SUB, P))),
                               Y))
```

```
PP.MAPPING.4: axiom
        SEQ.LENGTH(SCHED(CONFIG(T.SUB, P), T.SUB, P))
           = TABLE_LENGTH(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                            INDEX(REAL_TO_VIRT(MAKE.STATE(T.SUB,

                                                                                 P
)),
                                                        P)),
                                               P.CONFIG(MAKE.STATE(T.SUB, P))),
                                         SUBFRAME(MAKE.STATE(T.SUB, P))))

PP.MAPPING.5: axiom
        1 ≤ Y ∧ Y ≤ RESULT.SIZE(K)
             ⊃
        SEQ.ELEM(DATAFILEIN.FOR.ON(P, K, QQ, START(T.SUB, P)), Y)
           = SEQ.ELEM(INDEX(INDEX(DATAFILE(MAKE.STATE(T.SUB, P)), QQ), K), Y)
        ∧ SEQ.ELEM(INDEX(INDEX(DATAFILE(MAKE.STATE(T.SUB, P)), QQ), K), Y)
             = INDEX(INDEX(INDEX(DATAFILE(MAKE.STATE(T.SUB, P)), QQ), K), Y)

PP.MAPPING.6: axiom
        WORKING.DURING(P, T.SUB)
        ∧ WORKING.DURING(QQ, T.SUB)
        ∧ 1 ≤ Y
        ∧ Y ≤ RESULT.SIZE(K)
             ⊃
        SEQ.ELEM(DATAFILEIN.FOR.ON(P, K, QQ, START(T.SUB, P)), Y)
           = SEQ.ELEM(INDEX(INDEX(DATAFILE(MAKE.STATE(T.SUB, QQ)), QQ), K), Y)
        ∧ SEQ.ELEM(INDEX(INDEX(DATAFILE(MAKE.STATE(T.SUB, QQ)), QQ), K), Y)
             = INDEX(INDEX(INDEX(DATAFILE(MAKE.STATE(T.SUB, QQ)), QQ), K), Y)

PP.MAPPING.7: axiom
        1 ≤ Y ∧ Y ≤ RESULT.SIZE(K)
             ⊃
        SEQ.ELEM(INPUTIN.OF(P, K, START(T.SUB, P)), Y)
           = SEQ.ELEM(INDEX(INPUT(MAKE.STATE(T.SUB, P)), K), Y)
        ∧ SEQ.ELEM(INDEX(INPUT(MAKE.STATE(T.SUB, P)), K), Y)
             = INDEX(INDEX(INPUT(MAKE.STATE(T.SUB, P)), K), Y)

CONVERT.REP: (SET.OF(PAIR.OF(DATA, TASK))) → ARRAY.TYPE(ARRAY.TYPE(DATAVAL))

PP.MAPPING.10: axiom
        MAKE.PAIR(V, L) ∈ V.INPUTS
             ≡
        (1 ≤ EI
         ∧ EI ≤ RESULT.SIZE(L)
         ∧ INDEX(INDEX(P.INPUTS(), K), JI1) = L
         ∧ ¬(L = NULL_TASK())
              ⊃
         INDEX(INDEX(CONVERT.REP(V.INPUTS), JI1), EI) = SEQ.ELEM(V, EI))

PP.MAPPING.8: axiom
        APPLY(FUNCTION(K), V.INPUTS) = TASK_RESULTS(K, CONVERT.REP(V.INPUTS))

PP.MAPPING.9: axiom
        L ∈ INPUTS(K)
             ≡
        (∃ EI: INDEX(INDEX(P.INPUTS(), K), EI) = L ∧ ¬(L = NULL_TASK()))
```

```
PP.MAPPING.11: axiom
        TABLE_LENGTH(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                       INDEX(REAL_TO_VIRT(MAKE.STATE(T.SUB, P)),
                                             P)),
                                 P.CONFIG(MAKE.STATE(T.SUB, P))),
                           SUBFRAME(MAKE.STATE(T.SUB, P))))
           ≤ MAX_ACTIVITIES()

PP.MAPPING.12: axiom
        QQ ∈ POLLBY.FOR(P, K, T.SUB)
             ≡
        INDEX(INDEX(INDEX(POLL(), P.CONFIG(MAKE.STATE(T.SUB, P))),
                    INDEX(REAL_TO_VIRT(MAKE.STATE(T.SUB, P)), QQ)),
              K)

PP.MAPPING.13: axiom
        1 ≤ P ∧ P ≤ MAX_PROCESSORS()
```

SUBSECTION 7.17

PREPOST LEMMAS

```
/*
*********          Lemmas and Proofs between                    **********
              Activity and PrePost Specifications
*/

PP.LEMMA.1: formula
        P ∈ SAFE.FOR(OF(I, K))
        ∧ QQ ∈ SAFE.FOR(OF(I, K))
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
             ⊃
        WORKING.DURING(S.PROC(MAKE.STATE(T.SUB, QQ)),
                       S.SUB(MAKE.STATE(T.SUB, QQ)))
        ∧ (∃ JI:
              JI ≥ 1
              ∧ JI ≤ MAX_ACTIVITIES()
              ∧ DOT.ACTIVITY(INDEX(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                      INDEX
                                                      (REAL_TO_VIRT
                                                       (MAKE.STATE(T.SUB, QQ)),
                                                       S.PROC(MAKE.STATE(T.SUB,
                                                                         QQ)))),
                                                P.CONFIG(MAKE.STATE(T.SUB, QQ))),
                                          SUBFRAME(MAKE.STATE(T.SUB, QQ))),
                                    JI))
                   = EXECUTE()
              ∧ DOT.TASKNAME(INDEX(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                      INDEX
                                                      (REAL_TO_VIRT
                                                       (MAKE.STATE(T.SUB, QQ)),
                                                       S.PROC(MAKE.STATE(T.SUB,
                                                                         QQ)))),
                                                P.CONFIG(MAKE.STATE(T.SUB, QQ))),
                                          SUBFRAME(MAKE.STATE(T.SUB, QQ))),
                                    JI))
                   = K)
```

```
prove PP.LEMMA.1 [JI ← *Y:1+INT.NAT(1)]
using SEQ.MEMBER.AXIOM [SEQ1 ← SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ),
                        X ← ACTIV,
                        Y ← @:D]
      SCHED.TABLE.LENGTH.AXIOM [Y ← TABLE_LENGTH(@TAB:2),
                                Z ← @:D,
                                TAB
                                  ← INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                      INDEX
                                                      (REAL_TO_VIRT
                                                       (MAKE.STATE(T.SUB, QQ)),
                                                       QQ)),
                                                P.CONFIG(MAKE.STATE(T.SUB,
                                                                    QQ))),
                                          SUBFRAME(MAKE.STATE(T.SUB, QQ)))]
      SIFT.PARAMETER.INVARIANT#7
      SIFT.PARAMETER.INVARIANT#17 [P ← INDEX(REAL_TO_VIRT(MAKE.STATE(T.SUB,
                                                                    QQ)),
                                             QQ),
                                   CON ← CONFIG(T.SUB, QQ),
                                   T.PSUB ← SUBFRAME(MAKE.STATE(T.SUB, QQ)),
                                   JI ← TABLE_LENGTH(@TAB:2)+INT.NAT(1),
                                   Y ← JI:C]
      PP.MAPPING.11 [P ← QQ]
      PP.MAPPING.4 [P ← QQ]
      BR.D1 [II ← OF(I, K),
             P ← QQ]
      BR.A16 [P ← QQ,
              CON ← CONFIG(T.SUB, QQ)]
      RE.BR.MAPPING.9 [P ← QQ]
      PAIR.AXIOM.2 [X1 ← T.SUB,
                    X2 ← QQ]
      PP.MAPPING.3 [Y ← *Y:1+INT.NAT(1),
                    P ← QQ]
      PP.MAPPING.2 [P ← QQ]
      PP.MAPPING.1
PP.LEMMA.2: formula
        WORKING.DURING(QQ, T.SUB)
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
        ∧ ACTION(ACTIV) = VOTE()
        ∧ ELEM.ACTION(ACTIV) = Y
        ∧ TASK.ACTION(ACTIV) = K
             ⊃
        (∃ JI:
            JI ≥ 1
            ∧ JI ≤ MAX_ACTIVITIES()
            ∧ DOT.ACTIVITY(INDEX(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                   INDEX
                                                   (REAL_TO_VIRT
                                                    (MAKE.STATE(T.SUB, QQ)),
                                                    S.PROC(MAKE.STATE(T.SUB,
                                                                      QQ)))),
                                             P.CONFIG(MAKE.STATE(T.SUB, QQ))),
                                       SUBFRAME(MAKE.STATE(T.SUB, QQ))),
                                 JI))
                 = VOTE()
            ∧ DOT.TASKNAME(INDEX(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                   INDEX
                                                   (REAL_TO_VIRT
                                                    (MAKE.STATE(T.SUB, QQ)),
                                                    S.PROC(MAKE.STATE(T.SUB,
                                                                      QQ)))),
                                             P.CONFIG(MAKE.STATE(T.SUB, QQ))),
                                       SUBFRAME(MAKE.STATE(T.SUB, QQ))),
                                 JI))
                 = K
            ∧ DOT.ELEM(INDEX(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                               INDEX
                                               (REAL_TO_VIRT(MAKE.STATE(T.SUB,
                                                                        QQ)),
                                                S.PROC(MAKE.STATE(T.SUB, QQ)))),
                                         P.CONFIG(MAKE.STATE(T.SUB, QQ))),
                                   SUBFRAME(MAKE.STATE(T.SUB, QQ))),
                             JI))
                 = Y)
```

```
prove PP.LEMMA.2 [JI ← *Y:1+INT.NAT(1)]
using SEQ.MEMBER.AXIOM [SEQ1 ← SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ),
                        X ← ACTIV,
                        Y ← @:D]
      SCHED.TABLE.LENGTH.AXIOM [Y ← TABLE_LENGTH(@TAB:2),
                                Z ← @:D,
                                TAB
                                   ← INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                        INDEX
                                                        (REAL_TO_VIRT
                                                         (MAKE.STATE(T.SUB, QQ)),
                                                         QQ)),
                                                  P.CONFIG(MAKE.STATE(T.SUB,
                                                                      QQ))),
                                            SUBFRAME(MAKE.STATE(T.SUB, QQ)))]
      SIFT.PARAMETER.INVARIANT#7
      SIFT.PARAMETER.INVARIANT#17 [P ← INDEX(REAL_TO_VIRT(MAKE.STATE(T.SUB,
                                                                      QQ)),
                                              QQ),
                                   CON ← CONFIG(T.SUB, QQ),
                                   T.PSUB ← SUBFRAME(MAKE.STATE(T.SUB, QQ)),
                                   JI ← TABLE_LENGTH(@TAB:2)+INT.NAT(1),
                                   Y ← JI:C]
      PP.MAPPING.11 [P ← QQ]
      PP.MAPPING.4 [P ← QQ]
      PAIR.AXIOM.2 [X1 ← T.SUB,
                    X2 ← QQ]
      PP.MAPPING.3 [Y ← *Y:1+INT.NAT(1),
                    P ← QQ]
      PP.MAPPING.2 [P ← QQ]
      PP.MAPPING.1
PP.LEMMA.3: formula
        WORKING.DURING(QQ, T.SUB)
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
        ∧ ACTION(ACTIV) = DUMMY_VOTE()
        ∧ TASK.ACTION(ACTIV) = K
             ⊃
        (∃ JI:
            JI ≥ 1
            ∧ JI ≤ MAX_ACTIVITIES()
            ∧ DOT.ACTIVITY(INDEX(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                   INDEX
                                                   (REAL_TO_VIRT
                                                    (MAKE.STATE(T.SUB, QQ)),
                                                    S.PROC(MAKE.STATE(T.SUB,
                                                                      QQ)))),
                                             P.CONFIG(MAKE.STATE(T.SUB, QQ))),
                                       SUBFRAME(MAKE.STATE(T.SUB, QQ))),
                                 JI))
                 = DUMMY_VOTE()
            ∧ DOT.TASKNAME(INDEX(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                   INDEX
                                                   (REAL_TO_VIRT
                                                    (MAKE.STATE(T.SUB, QQ)),
                                                    S.PROC(MAKE.STATE(T.SUB,
                                                                      QQ)))),
                                             P.CONFIG(MAKE.STATE(T.SUB, QQ))),
                                       SUBFRAME(MAKE.STATE(T.SUB, QQ))),
                                 JI))
                 = K)
```

```
prove PP.LEMMA.3 [JI ← *Y:1+INT.NAT(1)]
using SEQ.MEMBER.AXIOM [SEQ1 ← SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ),
                        X ← ACTIV,
                        Y ← @:D]
      SCHED.TABLE.LENGTH.AXIOM [Y ← TABLE_LENGTH(@TAB:2),
                                Z ← @:D,
                                TAB
                                   ← INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                       INDEX
                                                       (REAL_TO_VIRT
                                                        (MAKE.STATE(T.SUB, QQ)),
                                                        QQ)),
                                                 P.CONFIG(MAKE.STATE(T.SUB,
                                                                     QQ))),
                                           SUBFRAME(MAKE.STATE(T.SUB, QQ)))]
      SIFT.PARAMETER.INVARIANT#7
      SIFT.PARAMETER.INVARIANT#17 [P ← INDEX(REAL_TO_VIRT(MAKE.STATE(T.SUB,
                                                                     QQ)),
                                             QQ),
                                   CON ← CONFIG(T.SUB, QQ),
                                   T.PSUB ← SUBFRAME(MAKE.STATE(T.SUB, QQ)),
                                   JI ← TABLE_LENGTH(@TAB:2)+INT.NAT(1),
                                   Y ← JI:C]
      PP.MAPPING.11 [P ← QQ]
      PP.MAPPING.4 [P ← QQ]
      PAIR.AXIOM.2 [X1 ← T.SUB,
                    X2 ← QQ]
      PP.MAPPING.3 [Y ← *Y:1+INT.NAT(1),
                    P ← QQ]
      PP.MAPPING.2 [P ← QQ]
      PP.MAPPING.1
PP.LEMMA.4: axiom
        JI > TABLE_LENGTH(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                             INDEX(REAL_TO_VIRT(MAKE.STATE(T.SUB,
                                                                           P)),
                                                   P)),
                                       P.CONFIG(MAKE.STATE(T.SUB, P))),
                                 SUBFRAME(MAKE.STATE(T.SUB, P))))
        ∧ JI ≤ MAX_ACTIVITIES()
             ⊃
        DOT.ACTIVITY(INDEX(INDEX(INDEX(INDEX(SCHED_TABLE(),
                                              INDEX
                                              (REAL_TO_VIRT(MAKE.STATE(T.SUB,
                                                                       P)),
                                               S.PROC(MAKE.STATE(T.SUB, P)))),
                                        P.CONFIG(MAKE.STATE(T.SUB, P))),
                                  SUBFRAME(MAKE.STATE(T.SUB, P))),
                            JI))
           = NULL.ENTRY()
```

```
PP.LEMMA.5: 'formula
        (∀ V.T:
            V.T ∈ V.INPUTS
                 ≡
            SOURCE(V.T) ∈ INPUTS(K)
            ∧ VALUE(V.T)
                 = INPUTIN.OF(QQ, SOURCE(V.T), START(SUB.INCR(T.SUB), QQ)))
        ∧ EI ≥ 1
        ∧ EI ≤ INDEX(RESULT_SIZE(), TASKI)
        ∧ INDEX(INDEX(P.INPUTS(), K), JI1) = TASKI
        ∧ ¬(TASKI = NULL_TASK())
             ⊃
        INDEX(INDEX(CONVERT.REP(V.INPUTS), JI1), EI)
           = INDEX(INDEX(INPUT(NEXT(MAKE.STATE(T.SUB, QQ))), TASKI), EI)

prove PP.LEMMA.5 [V.T
                     ← MAKE.PAIR(INPUTIN.OF(QQ, TASKI,
                                           START(SUB.INCR(T.SUB), QQ)),
                                TASKI)]
using PAIR.AXIOM.2 [X1 ← T.SUB,
                    X2 ← QQ]
      PAIR.AXIOM.2 [X1 ← INPUTIN.OF(QQ, TASKI, START(SUB.INCR(T.SUB), QQ)),
                    X2 ← TASKI]
      PP.MAPPING.1 [K ← TASKI]
      PP.MAPPING.9 [L ← TASKI,
                    EI ← JI1]
      PP.MAPPING.10 [L ← TASKI,
                     EI ← EI,
                     V ← INPUTIN.OF(QQ, TASKI, START(SUB.INCR(T.SUB), QQ))]
      PP.MAPPING.7 [Y ← EI,
                    T.SUB ← SUB.INCR(T.SUB),
                    P ← QQ,
                    K ← TASKI]
```

```
PP.LEMMA.7: formula
        P ∈ SAFE.FOR(OF(I, K))
        ∧ QQ ∈ SAFE.FOR(OF(I, K))
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ (∀ V.T:
              V.T ∈ V.INPUTS
                   ≡
              SOURCE(V.T) ∈ INPUTS(K)
              ∧ VALUE(V.T)
                   = INPUTIN.OF(QQ, SOURCE(V.T), START(SUB.INCR(T.SUB), QQ)))
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
             ⊃
        INDEX(INDEX(DATAFILE(MAKE.STATE(SUB.INCR(T.SUB), QQ)), QQ), K)
           = TASK_RESULTS(K, CONVERT.REP(V.INPUTS))

prove PP.LEMMA.7 [V.T ← *V.T:1]
using PP.LEMMA.5 [EI ← *EI:2,
                  TASKI ← *TASKI:2,
                  JI1 ← *JI1:2]
      EXECUTE.ACTIVITY [STATE.SIFT ← MAKE.STATE(T.SUB, QQ),
                        JI ← *JI:3,
                        TI ← K,
                        INP ← CONVERT.REP(V.INPUTS)]
      PP.LEMMA.1
      PAIR.AXIOM.2 [X1 ← T.SUB,
                    X2 ← QQ]
```

```
PP.LEMMA.8: formula
        P ∈ SAFE.FOR(OF(I, K))
        ∧ QQ ∈ SAFE.FOR(OF(I, K))
        ∧ START.FRAME(FRAME(T.SUB)) = I*FRAME.SIZE()
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ))
        ∧ ACTION(ACTIV) = EXECUTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ INDEX(INDEX(DATAFILE(MAKE.STATE(SUB.INCR(T.SUB), QQ)), QQ), K)
             = TASK_RESULTS(K, CONVERT.REP(V.INPUTS))
             ⊃
        DATAFILEIN.FOR.ON(P, K, QQ, START(SUB.INCR(T.SUB), P))
           = APPLY(FUNCTION(K), V.INPUTS)

prove PP.LEMMA.8
using PP.MAPPING.8
      PP.MAPPING.5 [Y ← *Y:3,
                    T.SUB ← SUB.INCR(T.SUB)]
      DATA.EQUALITY [V1 ← DATAFILEIN.FOR.ON(P, K, QQ,
                                            START(SUB.INCR(T.SUB), P)),
                     Y ← @:D,
                     V ← APPLY(FUNCTION(K), V.INPUTS)]
      PP.MAPPING.6 [Y ← *Y:3,
                    T.SUB ← SUB.INCR(T.SUB)]
      BR.LEMMA.5
      BR.D1 [II ← OF(I, K),
             T.SUB ← SUB.INCR(T.SUB)]
      BR.D1 [II ← OF(I, K),
             P ← QQ,
             T.SUB ← SUB.INCR(T.SUB)]
      RE.BR.MAPPING.9 [T.SUB ← SUB.INCR(T.SUB)]
      RE.BR.MAPPING.9 [P ← QQ,
                       T.SUB ← SUB.INCR(T.SUB)]
      DATA.SIZE.IS.SEQ.LENGTH.3
      DATA.SIZE.IS.SEQ.LENGTH.2 [T.REAL ← START(SUB.INCR(T.SUB), P)]
      SEQ.EQUALITY.AXIOM [SEQ1 ← APPLY(FUNCTION(K), V.INPUTS),
                          SEQ2 ← TASK_RESULTS(K, CONVERT.REP(V.INPUTS)),
                          Y ← *Y:3]
      SEQ.EQUALITY.AXIOM [SEQ1
                             ← INDEX(INDEX(DATAFILE(NEXT(MAKE.STATE(T.SUB,
                                                                    QQ))),
                                           QQ),
                                     K),
                          SEQ2 ← TASK_RESULTS(K, CONVERT.REP(V.INPUTS)),
                          Y ← *Y:3]
      PAIR.AXIOM.2 [X1 ← T.SUB,
                    X2 ← QQ]
      PAIR.AXIOM.2 [X1 ← T.SUB,
                    X2 ← QQ]
```

```
PP.LEMMA.10: axiom
        WORKING.DURING(P, T.SUB)
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, P), T.SUB, P))
        ∧ ACTION(ACTIV) = VOTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ ELEM.ACTION(ACTIV) = Y
             ⊃
        CARD(D.BAG.A9C(P, K, T.SUB, Y))
           ≐ CARD(SET.FN#10(K, P.CONFIG(MAKE.STATE(T.SUB, P)),
                            MAKE.STATE(T.SUB, P)))

prove PP.LEMMA.10
using SET.ABSTRACTION.A9C [D.P ← MAKE.PAIR(*D1:2, *QQ:2)]
      SET.CONSTRUCTION.LEMMA
      PAIR.AXIOM.2 [X1 ← D1:2,
                    X2 ← QQ:2]
      PP.MAPPING.12 [QQ ← *QQ:2]
      SET.FN.AXIOM.10 [P ← *QQ:2,
                       STATE.SIFT ← MAKE.STATE(T.SUB, P),
                       TI ← K,
                       CON ← P.CONFIG(MAKE.STATE(T.SUB, P))]
      PP.MAPPING.13 [P ← *QQ:2]
```

```
var R1: PROC

var D.P1: PAIR.OF(DATAVAL, PROC)

var D.P2: PAIR.OF(DATAVAL, PROC)

var D.BAG.1: SET.OF(PAIR.OF(DATAVAL, PROC))

var D2: DATAVAL

/*
The set of processors computing the same value can be represented
        either as a set of <proc,value> pairs or as a set of procs.
*/

SET.CONSTRUCTION.LEMMA: formula
        (∀ D1, QQ:
            MAKE.PAIR(D1, QQ) ∈ D.BAG.A9C(P, K, T.SUB, Y)
                ≡
            QQ
            ∈ SET.FN#10(K, P.CONFIG(MAKE.STATE(T.SUB, P)), MAKE.STATE(T.SUB, P))
            ∧ D1 = SEQ.ELEM(DATAFILEIN.FOR.ON(P, K, QQ, START(T.SUB, P)), Y))
             ⊃
        CARD(D.BAG.A9C(P, K, T.SUB, Y))
           = CARD(SET.FN#10(K, P.CONFIG(MAKE.STATE(T.SUB, P)),
                            MAKE.STATE(T.SUB, P)))


SET.CONSTRUCTION.LEMMA.2: formula
        (∀ D.P:
            D.P ∈ D.BAG
                ≡
            SOURCE(D.P)
            ∈ SET.FN#9(VALUE(D.P), MAKE.STATE(T.SUB, P), Y, K,
                       P.CONFIG(MAKE.STATE(T.SUB, P))))
             ⊃
        CARD(D.BAG.MAJ(P, K, T.SUB, Y, D2))
           = CARD(SET.FN#9(D2, MAKE.STATE(T.SUB, P), Y, K,
                           P.CONFIG(MAKE.STATE(T.SUB, P))))
```

```
PP.LEMMA.13: formula
        1 ≤ Y ∧ Y ≤ RESULT.SIZE(K)
             ⊃
        CARD(D.BAG.MAJ(P, K, T.SUB, Y, D2))
           = CARD(SET.FN#9(D2, MAKE.STATE(T.SUB, P), Y, K,
                           P.CONFIG(MAKE.STATE(T.SUB, P))))

prove PP.LEMMA.13
using SET.CONSTRUCTION.LEMMA.2 [D.BAG ← D.BAG.A9C(P, K, T.SUB, Y)]
      PP.MAPPING.13 [P ← SOURCE(*D.P:1)]
      SET.ABSTRACTION.A9C [D.P ← *D.P:1]
      SET.ABSTR#9 [QQ ← SOURCE(*D.P:1),
                   MAJI ← VALUE(*D.P:1),
                   STATE.SIFT ← MAKE.STATE(T.SUB, P),
                   EI ← Y,
                   TI ← K,
                   CON ← P.CONFIG(MAKE.STATE(T.SUB, P))]
      PP.MAPPING.12 [QQ ← SOURCE(*D.P:1)]
      PP.MAPPING.5 [QQ ← SOURCE(*D.P:1)]
PP.LEMMA.14: formula
        WORKING.DURING(P, T.SUB)
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, P), T.SUB, P))
        ∧ ACTION(ACTIV) = VOTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ ELEM.ACTION(ACTIV) = Y
        ∧ 1 ≤ Y
        ∧ Y ≤ RESULT.SIZE(K)
        ∧ (∃ MAJI2:
              CARDINALITY(SET.FN#9(MAJI2, MAKE.STATE(T.SUB, P), Y, K,
                                   P.CONFIG(MAKE.STATE(T.SUB, P))))
              *2
                 > CARDINALITY(SET.FN#10(K, P.CONFIG(MAKE.STATE(T.SUB, P)),
                                         MAKE.STATE(T.SUB, P))))
        ∧ (∀ MAJI:
              CARDINALITY(SET.FN#9(MAJI, MAKE.STATE(T.SUB, P), Y, K,
                                   P.CONFIG(MAKE.STATE(T.SUB, P))))
              *2
                 > CARDINALITY(SET.FN#10(K, P.CONFIG(MAKE.STATE(T.SUB, P)),
                                         MAKE.STATE(T.SUB, P)))
                   ↄ
              INDEX(INDEX(INPUT(NEXT(MAKE.STATE(T.SUB, P))), K), Y) = MAJI)
             ↄ
        SEQ.ELEM(INPUTIN.OF(P, K, START(SUB.INCR(T.SUB), P)), Y)
           = MAJORITY(D.BAG.A9C(P, K, T.SUB, Y))

prove PP.LEMMA.14 [MAJI ← *MAJI2:C]
using PP.LEMMA.13 [D2 ← *MAJI2:C]
      PP.LEMMA.10
      MAJ.1 [M.BAG ← D.BAG.A9C(P, K, T.SUB, Y),
             M.BAG.1 ← D.BAG.MAJ(P, K, T.SUB, Y, *MAJI2:C),
             T1.V ← *MAJI2:C]
      SET.ABSTRACTION.MAJ [D.P2 ← *V1.V2:3,
                           D2 ← *MAJI2:C]
      PP.MAPPING.7 [T.SUB ← SUB.INCR(T.SUB)]
      PAIR.AXIOM.2 [X1 ← T.SUB,
                    X2 ← P]
```

```
PP.LEMMA.15: formula
        WORKING.DURING(P, T.SUB)
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, P), T.SUB, P))
        ∧ ACTION(ACTIV) = VOTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ ELEM.ACTION(ACTIV) = Y
        ∧ 1 ≤ Y
        ∧ Y ≤ RESULT.SIZE(K)
        ∧ ¬(∃ MAJI2:
               CARDINALITY(SET.FN#9(MAJI2, MAKE.STATE(T.SUB, P), Y, K,
                                    P.CONFIG(MAKE.STATE(T.SUB, P))))
               *2
                  > CARDINALITY(SET.FN#10(K, P.CONFIG(MAKE.STATE(T.SUB, P)),
                                          MAKE.STATE(T.SUB, P))))
        ∧ BOTTOMD() = INDEX(INDEX(INPUT(NEXT(MAKE.STATE(T.SUB, P))), K), Y)
             ⊃
        SEQ.ELEM(INPUTIN.OF(P, K, START(SUB.INCR(T.SUB), P)), Y)
           = MAJORITY(D.BAG.A9C(P, K, T.SUB, Y))

prove PP.LEMMA.15 [MAJI2 ← *T1.V:3]
using PP.LEMMA.13 [D2 ← *T1.V:3]
      PP.LEMMA.10
      MAJ.3 [M.BAG.1 ← D.BAG.MAJ(P, K, T.SUB, Y, *T1.V:3),
             M.BAG ← D.BAG.A9C(P, K, T.SUB, Y),
            .T2.V ← BOTTOMD()]
      SET.ABSTRACTION.MAJ [D.P2 ← *V1.V2:3,
                           D2 ← *T1.V:3]
      PP.MAPPING.7 [T.SUB ← SUB.INCR(T.SUB)]
      PAIR.AXIOM.2 [X1 ← T.SUB,
                    X2 ← P]
      BOTTOM.EQUALITY [D1 ← BOTTOMD()]
```

```
PP.LEMMA.16: formula
        WORKING.DURING(P, T.SUB)
        ∧ SEQ.MEMBER(ACTIV, SCHED(CONFIG(T.SUB, P), T.SUB, P))
        ∧ ACTION(ACTIV) = VOTE()
        ∧ TASK.ACTION(ACTIV) = K
        ∧ ELEM.ACTION(ACTIV) = Y
        ∧ 1 ≤ Y
        ∧ Y ≤ RESULT.SIZE(K)
             ⊃
        SEQ.ELEM(INPUTIN.OF(P, K, START(SUB.INCR(T.SUB), P)), Y)
           = MAJORITY(D.BAG.A9C(P, K, T.SUB, Y))

prove PP.LEMMA.16
using PP.LEMMA.2 [QQ ← P]
      VOTE.ACTIVITY [JI ← *JI:1,
                     MAJI2 ← *MAJI2:5,
                     MAJI ← *MAJI:3,
                     STATE.SIFT ← MAKE.STATE(T.SUB, P),
                     EI ← Y,
                     TI ← K]
      PP.LEMMA.14 [MAJI2 ← *MAJI2:5]
      VOTE.ACTIVITY [JI ← *JI:1,
                     MAJI2 ← *MAJI2:5,
                     MAJI ← *MAJI:3,
                     STATE.SIFT ← MAKE.STATE(T.SUB, P),
                     EI ← Y,
                     TI ← K]
      PP.LEMMA.15
      PAIR.AXIOM.2 [X1 ← T.SUB,
                    X2 ← P]
```

```
/*
     Proofs of Activity Specification Axioms and Lemmas
*/

prove BR.A6A [ACTIV
                 ← SEQ.ELEM(SCHED(CONFIG(T.SUB, P), T.SUB, P),
                            *JI:2-INT.NAT(1))]
using SEQ.MEMBER.AXIOM [SEQ1 ← SCHED(CONFIG(T.SUB, P), T.SUB, P),
                        X ← @ACTIV:C,
                        Y ← *JI:2-INT.NAT(1)]
      VOTE.FRAME.AXIOM [STATE.SIFT ← MAKE.STATE(T.SUB, P),
                        TI ← K,
                        EI ← Y]
      PP.MAPPING.4
      PAIR.AXIOM.2 [X1 ← T.SUB,
                    X2 ← P]
      PP.MAPPING.3 [Y ← *JI:2]
      PP.MAPPING.2
      PP.MAPPING.7
      PP.MAPPING.7 [T.SUB ← SUB.INCR(T.SUB)]
      SCHED.TABLE.LENGTH.AXIOM [Y ← TABLE_LENGTH(@TAB:9),
                                Z ← @:D,
                                TAB
                                   ← INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                       INDEX
                                                       (REAL_TO_VIRT
                                                        (MAKE.STATE(T.SUB, P)),
                                                        P)),
                                                 P.CONFIG(MAKE.STATE(T.SUB,
                                                                     P))),
                                           SUBFRAME(MAKE.STATE(T.SUB, P)))]
      SIFT.PARAMETER.INVARIANT#7
      SIFT.PARAMETER.INVARIANT#17 [P ← INDEX(REAL_TO_VIRT(MAKE.STATE(T.SUB, P)),
                                             P),
                                   CON ← CONFIG(T.SUB, P),
                                   T.PSUB ← SUBFRAME(MAKE.STATE(T.SUB, P)),
                                   JI ← TABLE_LENGTH(@TAB:9)+INT.NAT(1),
                                   Y ← *JI:2]
      PP.MAPPING.11
      PP.MAPPING.1
```

```
prove BR.LEMMA.18 [ACTIV
                      ← SEQ.ELEM(SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ),
                                 *JI:2-INT.NAT(1))]
using SEQ.MEMBER.AXIOM [SEQ1 ← SCHED(CONFIG(T.SUB, QQ), T.SUB, QQ),
                        X ← @ACTIV:C,
                        Y ← *JI:2-INT.NAT(1)]
      EXECUTE.FRAME.AXIOM [STATE.SIFT ← MAKE.STATE(T.SUB, QQ),
                           TI ← K,
                           EI ← *Y:3]
      DATA.EQUALITY [V ← DATAFILEIN.FOR.ON(P, K, QQ, START(SUB.INCR(T.SUB), P)),
                     V1 ← DATAFILEIN.FOR.ON(P, K, QQ, START(T.SUB, P)),
                     Y ← @:D]
      DATA.SIZE.IS.SEQ.LENGTH.2 [T.REAL ← START(SUB.INCR(T.SUB), P)]
      DATA.SIZE.IS.SEQ.LENGTH.2 [T.REAL ← START(T.SUB, P)]
      PAIR.AXIOM.2 [X1 ← T.SUB,
                    X2 ← QQ]
      PP.MAPPING.3 [Y ← *JI:2,
                    P ← QQ]
      PP.MAPPING.2 [P ← QQ]
      PP.MAPPING.6 [Y ← *Y:3]
      PP.MAPPING.6 [T.SUB ← SUB.INCR(T.SUB),
                    Y ← *Y:3]
      RE.BR.MAPPING.9
      RE.BR.MAPPING.9 [T.SUB ← SUB.INCR(T.SUB)]
      RE.BR.MAPPING.9 [P ← QQ]
      RE.BR.MAPPING.9 [P ← QQ,
                       T.SUB ← SUB.INCR(T.SUB)]
      SCHED.TABLE.LENGTH.AXIOM [Y ← TABLE_LENGTH(@TAB:15),
                                Z ← @:D,
                                TAB
                                  ← INDEX(INDEX(INDEX(SCHED_TABLE(),
                                                      INDEX
                                                      (REAL_TO_VIRT
                                                       (MAKE.STATE(T.SUB, QQ)),
                                                       QQ)),
                                                P.CONFIG(MAKE.STATE(T.SUB,
                                                                    QQ))),
                                          SUBFRAME(MAKE.STATE(T.SUB, QQ)))]
      SIFT.PARAMETER.INVARIANT#7
      SIFT.PARAMETER.INVARIANT#17 [P ← INDEX(REAL_TO_VIRT(MAKE.STATE(T.SUB,
                                                                    QQ)),
                                           QQ),
                                  CON ← CONFIG(T.SUB, QQ),
                                  T.PSUB ← SUBFRAME(MAKE.STATE(T.SUB, QQ)),
                                  JI ← TABLE_LENGTH(@TAB:15)+INT.NAT(1),
                                  Y ← *JI:2]
      PP.MAPPING.11 [P ← QQ]
      PP.MAPPING.4 [P ← QQ]
      PP.MAPPING.1


prove BR.LEMMA.17 [V.T ← *V.T:1]
using PP.LEMMA.7
      PP.LEMMA.8
```

```
prove BR.A9C
using PP.LEMMA.16
      BR.A44 [CON ← CONFIG(T.SUB, P)]



prove BR.A6B
using PP.LEMMA.3 [QQ ← P]
      DUMMY_VOTE.ACTIVITY [JI ← *JI:1,
                           EI ← *Y:4,
                           STATE.SIFT ← MAKE.STATE(T.SUB, P),
                           TI ← K]
      DATA.BOTTOM [Y ← *Y:4]
      DATA.EQUALITY [V ← INPUTIN.OF(P, K, START(SUB.INCR(T.SUB), P)),
                     V1 ← BOTTOM1(K),
                     Y ← @:D]
      DATA.SIZE.IS.SEQ.LENGTH
      DATA.SIZE.IS.SEQ.LENGTH.2 [T.REAL ← START(SUB.INCR(T.SUB), P)]
      PP.MAPPING.7 [Y ← *Y:4,
                    T.SUB ← SUB.INCR(T.SUB)]
      PAIR.AXIOM.2 [X1 ← T.SUB,
                    X2 ← P]
      PP.MAPPING.1
```

```
8_STATUS(IO.A2)

Proved Using the Following Axioms and Unproved Formulas:

----- Axioms -----

BOTTOM.EQUALITY (20)
BR.A1.A (4)
BR.A1.C (8)
BR.A1.E (4)
BR.A12 (12)
BR.A12A (36)
BR.A12C (10)
BR.A13A (38)
BR.A13B (38)
BR.A16 (300)
BR.A18 (244)
BR.A19.1 (74)
BR.A19.2 (8)
BR.A21 (944)
BR.A21B (432)
BR.A22 (108)
BR.A25A (8)
BR.A27 (8)
BR.A28 (32)
BR.A29 (32)
BR.A36 (192)
BR.A40 (48)
BR.A44 (10)
BR.A9A (42)
BR.D1 (440)
BR.RE.MAPPING.4 (26)
BR.RE.MAPPING.5 (8)
BR.RE.MAPPING.6 (14)
BR.RE.MAPPING.7 (6)
BR.RE.MAPPING.8 (10)
DATA.BOTTOM (20)
DATA.EQUALITY (114)
DATA.SIZE.IS.SEQ.LENGTH (20)
DATA.SIZE.IS.SEQ.LENGTH.2 (190)
DATA.SIZE.IS.SEQ.LENGTH.3 (10)
DUMMY_VOTE.ACTIVITY (10)
EXECUTE.ACTIVITY (4)
GE.CONFIG.FIELD (84)
IO.A2A (2)
LENGTH.OF.ELEMENTS.OF.ON.IS.LENGTH.OF.ON.IN (4)
MAJ.1 (22)
MAJ.2 (10)
MAJ.3 (10)
NAT.NONNEGATIVE (60)
PAIR.AXIOM.2 (132)
PP.MAPPING.1 (46)
PP.MAPPING.10 (4)
PP.MAPPING.11 (32)
PP.MAPPING.12 (60)
PP.MAPPING.13 (60)
PP.MAPPING.2 (32)
PP.MAPPING.3 (32)
```

SUBSECTION 7.18

PROOF-STATUS 4

```
42_STATUS(BR.LEMMA.17)


Proved Using the Following Axioms and Unproved Formulas:

----- Axioms -----

BR.A12 (1)
BR.A13A (1)
BR.A16 (3)
BR.A21 (3)
BR.A36 (1)
BR.D1 (5)
DATA.EQUALITY (1)
DATA.SIZE.IS.SEQ.LENGTH.2 (1)
DATA.SIZE.IS.SEQ.LENGTH.3 (1)
EXECUTE.ACTIVITY (1)
PAIR.AXIOM.2 (6)
PP.MAPPING.1 (2)
PP.MAPPING.10 (1)
PP.MAPPING.11 (1)
PP.MAPPING.2 (1)
PP.MAPPING.3 (1)
PP.MAPPING.4 (1)
PP.MAPPING.5 (1)
PP.MAPPING.6 (1)
PP.MAPPING.7 (1)
PP.MAPPING.8 (1)
PP.MAPPING.9 (1)
RE.BR.MAPPING.9 (5)
SCHED.TABLE.LENGTH.AXIOM (1)
SEQ.EQUALITY.AXIOM (3)
SEQ.MEMBER.AXIOM (1)
SIFT.PARAMETER.INVARIANT#17 (1)
SIFT.PARAMETER.INVARIANT#7 (1)


----- Unproved Formulas -----

CARD.2 (1)
```

SUBSECTION 7.19

PROOF-STATUS 3

```
33_STATUS(IO.A2)


Proved Using the Following Axioms and Unproved Formulas:


----- Axioms -----

BOTTOM.EQUALITY (20)
BR.A1.C (4)
BR.A1.D (4)
BR.A1.E (4)
BR.A12 (16)
BR.A12A (36)
BR.A12C (10)
BR.A13A (42)
BR.A13B (38)
BR.A16 (308)
BR.A18 (244)
BR.A19.1 (74)
BR.A19.2 (8)
BR.A21 (956)
BR.A21B (432)
BR.A22 (108)
BR.A25A (8)
BR.A27 (8)
BR.A28 (32)
BR.A29 (32)
BR.A36 (196)
BR.A40 (48)
BR.A44 (10)
BR.A9A (42)
BR.D1 (452)
BR.RE.MAPPING.4 (26)
BR.RE.MAPPING.5 (8)
BR.RE.MAPPING.6 (14)
BR.RE.MAPPING.7 (6)
BR.RE.MAPPING.8 (10)
DATA.BOTTOM (20)
DATA.EQUALITY (114)
DATA.SIZE.IS.SEQ.LENGTH (20)
DATA.SIZE.IS.SEQ.LENGTH.2 (194)
DATA.SIZE.IS.SEQ.LENGTH.3 (6)
DUMMY_VOTE.ACTIVITY (10)
EXECUTE.FRAME.AXIOM (4)
GE.CONFIG.FIELD (84)
IO.A2A (2)
LENGTH.OF.ELEMENTS.OF.ON.IS.LENGTH.OF.ON.IN (4)
MAJ.1 (22)
MAJ.2 (10)
MAJ.3 (10)
NAT.NONNEGATIVE (60)
PAIR.AXIOM.2 (112)
PP.MAPPING.1 (42)
PP.MAPPING.11 (32)
PP.MAPPING.12 (60)
PP.MAPPING.13 (60)
PP.MAPPING.2 (32)
PP.MAPPING.3 (32)
PP.MAPPING.4 (32)
PP.MAPPING.5 (30)
PP.MAPPING.6 (8)
PP.MAPPING.7 (56)
RE.BR.MAPPING.9 (476)
RESULT.SIZE.GREATER.THAN.1 (2)
RP.A1.1 (44)
RP.A3A (6)
RP.D1 (27)
RP.D10 (70)
RP.D11 (4)
RP.D2.1 (27)
RP.D2.2 (8)
RP.D3.1 (35)
RP.D3.3 (35)
RP.D4A (22)
RP.D6 (6)
RP.D7 (1)
RP.D9A (3)
RP.L12A (4)
SCHED.TABLE.LENGTH.AXIOM (32)
SEQ.EQUALITY.AXIOM (112)
SEQ.MEMBER.AXIOM (32)
SET.ABSTR#9 (30)
SET.ABSTRACTION.A9C (70)
SET.ABSTRACTION.MAJ (30)
SET.FN.AXIOM.10 (30)
SIFT.PARAMETER.INVARIANT#17 (32)
SIFT.PARAMETER.INVARIANT#7 (32)
TIMES.AXIOM.1 (56)
TIMES.AXIOM.2 (8)
TIMES.AXIOM.3 (48)
VOTE.ACTIVITY (20)
VOTE.FRAME.AXIOM (8)


----- Unproved Formulas -----

BR.A41 (4)
BR.A42 (4)
BR.INDUCTION.SUB.TIME.1 (4)
BR.INDUCTION.SUB.TIME.2 (8)
BR.LEMMA.FOR.LES.TO.PROVE (4)
CARD.2 (200)
CARD.3 (3)
CARD.4 (11)
CARD.6 (10)
CARD.D.BAG.D4 (2)
CARD.D.BAG.L10 (2)
CARD.INTERSECTION (1)
CARD.SUBSET (3)
INTERSECT (10)
NECESSARY.EVIL (1)
RP.L10 (2)
SET.CONSTRUCTION.LEMMA (30)
SET.CONSTRUCTION.LEMMA.2 (30)
SETEQUALITY (12)
SUBSET (3)
```

```
34_STATUS(IO.A5)


Proved Using the Following Axioms and Unproved Formulas:

----- Axioms -----

BOTTOM.EQUALITY (9)
BR.A12A (6)
BR.A12C (3)
BR.A13A (9)
BR.A13B (9)
BR.A16 (42)
BR.A18 (30)
BR.A19.1 (9)
BR.A21 (99)
BR.A21B (42)
BR.A22 (24)
BR.A36 (33)
BR.A44 (3)
BR.A9A (12)
BR.D1 (72)
BR.RE.MAPPING.4 (6)
BR.RE.MAPPING.6 (3)
BR.RE.MAPPING.7 (3)
BR.RE.MAPPING.8 (3)
DATA.BOTTOM (9)
DATA.EQUALITY (18)
DATA.SIZE.IS.SEQ.LENGTH (6)
DATA.SIZE.IS.SEQ.LENGTH.2 (21)
DUMMY_VOTE.ACTIVITY (3)
GE.CONFIG.FIELD (9)
MAJ.1 (6)
MAJ.2 (6)
MAJ.3 (3)
PAIR.AXIOM.2 (30)
PP.MAPPING.1 (9)
PP.MAPPING.11 (6)
PP.MAPPING.12 (18)
PP.MAPPING.13 (18)
PP.MAPPING.2 (6)
PP.MAPPING.3 (6)
PP.MAPPING.4 (6)
PP.MAPPING.5 (9)
PP.MAPPING.7 (12)
RE.BR.MAPPING.9 (72)
RP.A1.1 (1)
RP.D1 (1)
RP.D10 (2)
RP.D2.1 (1)
RP.D3.1 (1)
RP.D3.3 (1)
RP.D4A (6)
RP.D6 (2)
RP.D7 (4)
RP.D9A (2)
SCHED.TABLE.LENGTH.AXIOM (6)
SEQ.EQUALITY.AXIOM (24)
SEQ.MEMBER.AXIOM (6)
```

```
SET.ABSTR#9 (9)
SET.ABSTRACTION.A9C (21)
SET.ABSTRACTION.MAJ (9)
SET.FN.AXIOM.10 (9)
SIFT.PARAMETER.INVARIANT#17 (6)
SIFT.PARAMETER.INVARIANT#7 (6)
VOTE.ACTIVITY (6)


----- Unproved Formulas -----

CARD.2 (33)
CARD.3 (1)
CARD.4 (4)
CARD.6 (6)
CARD.D.BAG.D4 (3)
CARD.INTERSECTION (1)
SET.CONSTRUCTION.LEMMA (9)
SET.CONSTRUCTION.LEMMA.2 (9)
SETEQUALITY (3)
```

```
35_STATUS(BR.LEMMA.17)


Proved Using the Following Axioms and Unproved Formulas:

----- Axioms -----

BR.A1.C (1)
BR.A1.D (1)
BR.A1.E (1)
BR.A12 (2)
BR.A13A (2)
BR.A16 (5)
BR.A21 (6)
BR.A36 (2)
BR.D1 (8)
NAT.NONNEGATIVE (3)
RE.BR.MAPPING.9 (8)
SEQ.EQUALITY.AXIOM (2)


----- Unproved Formulas -----

BR.A41 (1)
BR.A42 (1)
BR.LEMMA.FOR.LES.TO.PROVE (1)
CARD.2 (2)
```

SUBSECTION 7.20

PROOF-STATUS 1

```
10_STATUS(IO.A2)


Proved Using the Following Axioms and Unproved Formulas:


----- Axioms -----

BOTTOM.EQUALITY (20)
BR.A12 (12)
BR.A12A (36)
BR.A12C (10)
BR.A13A (38)
BR.A13B (38)
BR.A16 (300)
BR.A18 (244)
BR.A19.1 (74)
BR.A19.2 (8)
BR.A21 (944)
BR.A21B (432)
BR.A22 (108)
BR.A25A (8)
BR.A27 (8)
BR.A28 (32)
BR.A29 (32)
BR.A36 (192)
BR.A40 (48)
BR.A44 (10)
BR.A9A (42)
BR.D1 (440)
BR.RE.MAPPING.4 (26)
BR.RE.MAPPING.5 (8)
BR.RE.MAPPING.6 (14)
BR.RE.MAPPING.7 (6)
BR.RE.MAPPING.8 (10)
DATA.BOTTOM (20)
DATA.EQUALITY (118)
DATA.SIZE.IS.SEQ.LENGTH (20)
DATA.SIZE.IS.SEQ.LENGTH.2 (198)
DATA.SIZE.IS.SEQ.LENGTH.3 (10)
DUMMY_VOTE.ACTIVITY (10)
EXECUTE.ACTIVITY (4)
EXECUTE.FRAME.AXIOM (4)
GE.CONFIG.FIELD (84)
IO.A2A (2)
LENGTH.OF.ELEMENTS.OF.ON.IS.LENGTH.OF.ON.IN (4)
MAJ.1 (22)
MAJ.2 (10)
MAJ.3 (10)
NAT.NONNEGATIVE (48)
PAIR.AXIOM.2 (136)
PP.MAPPING.1 (50)
PP.MAPPING.10 (4)
PP.MAPPING.11 (36)
PP.MAPPING.12 (60)
PP.MAPPING.13 (60)
PP.MAPPING.2 (36)
PP.MAPPING.3 (36)
PP.MAPPING.4 (36)
PP.MAPPING.5 (34)
PP.MAPPING.6 (12)
PP.MAPPING.7 (60)
PP.MAPPING.8 (4)
PP.MAPPING.9 (4)
RE.BR.MAPPING.9 (464)
RESULT.SIZE.GREATER.THAN.1 (2)
RP.A1.1 (44)
RP.A3A (6)
RP.D1 (27)
RP.D10 (70)
RP.D11 (4)
RP.D2.1 (27)
RP.D2.2 (8)
RP.D3.1 (35)
RP.D3.3 (35)
RP.D4A (22)
RP.D6 (6)
RP.D7 (1)
RP.D9A (3)
RP.L12A (4)
SCHED.TABLE.LENGTH.AXIOM (36)
SEQ.EQUALITY.AXIOM (116)
SEQ.MEMBER.AXIOM (36)
SET.ABSTR#9 (30)
SET.ABSTRACTION.A9C (70)
SET.ABSTRACTION.MAJ (30)
SET.FN.AXIOM.10 (30)
SIFT.PARAMETER.INVARIANT#17 (36)
SIFT.PARAMETER.INVARIANT#7 (36)
TIMES.AXIOM.1 (56)
TIMES.AXIOM.2 (8)
TIMES.AXIOM.3 (48)
VOTE.ACTIVITY (20)
VOTE.FRAME.AXIOM (8)


----- Unproved Formulas -----

BR.INDUCTION.SUB.TIME.1 (4)
BR.INDUCTION.SUB.TIME.2 (8)
CARD.2 (196)
CARD.3 (3)
CARD.4 (11)
CARD.6 (10)
CARD.D.BAG.D4 (2)
CARD.D.BAG.L10 (2)
CARD.INTERSECTION (1)
CARD.SUBSET (3)
INTERSECT (10)
NECESSARY.EVIL (1)
RP.L10 (2)
SET.CONSTRUCTION.LEMMA (30)
SET.CONSTRUCTION.LEMMA.2 (30)
SETEQUALITY (12)
SUBSET (3)
```

```
13_STATUS(IO.A5)

Proved Using the Following Axioms and Unproved Formulas:

----- Axioms -----

BOTTOM.EQUALITY (9)
BR.A12A (6)
BR.A12C (3)
BR.A13A (9)
BR.A13B (9)
BR.A16 (42)
BR.A18 (30)
BR.A19.1 (9)
BR.A21 (99)
BR.A21B (42)
BR.A22 (24)
BR.A36 (33)
BR.A44 (3)
BR.A9A (12)
BR.D1 (72)
BR.RE.MAPPING.4 (6)
BR.RE.MAPPING.6 (3)
BR.RE.MAPPING.7 (3)
BR.RE.MAPPING.8 (3)
DATA.BOTTOM (9)
DATA.EQUALITY (18)
DATA.SIZE.IS.SEQ.LENGTH (6)
DATA.SIZE.IS.SEQ.LENGTH.2 (21)
DUMMY_VOTE.ACTIVITY (3)
GE.CONFIG.FIELD (9)
MAJ.1 (6)
MAJ.2 (6)
MAJ.3 (3)
PAIR.AXIOM.2 (30)
PP.MAPPING.1 (9)
PP.MAPPING.11 (6)
PP.MAPPING.12 (18)
PP.MAPPING.13 (18)
PP.MAPPING.2 (6)
PP.MAPPING.3 (6)
PP.MAPPING.4 (6)
PP.MAPPING.5 (9)
PP.MAPPING.7 (12)
RE.BR.MAPPING.9 (72)
RP.A1.1 (1)
RP.D1 (1)
RP.D10 (2)
RP.D2.1 (1)
RP.D3.1 (1)
RP.D3.3 (1)
RP.D4A (6)
RP.D6 (2)
RP.D7 (4)
RP.D9A (2)
SCHED.TABLE.LENGTH.AXIOM (6)
SEQ.EQUALITY.AXIOM (24)
SEQ.MEMBER.AXIOM (6)
```

```
SET.ABSTR#9 (9)
SET.ABSTRACTION.A9C (21)
SET.ABSTRACTION.MAJ (9)
SET.FN.AXIOM.10 (9)
SIFT.PARAMETER.INVARIANT#17 (6)
SIFT.PARAMETER.INVARIANT#7 (6)
VOTE.ACTIVITY (6)


----- Unproved Formulas -----

CARD.2 (33)
CARD.3 (1)
CARD.4 (4)
CARD.6 (6)
CARD.D.BAG.D4 (3)
CARD.INTERSECTION (1)
SET.CONSTRUCTION.LEMMA (9)
SET.CONSTRUCTION.LEMMA.2 (9)
SETEQUALITY (3)
```

CHAPTER 8

OVERVIEW OF CODE VERIFICATION

Overview of Code Verification

The verified code for the SIFT Executive is not the code that executes on the SIFT system as delivered. The running versions of the SIFT Executive contain optimizations and special code relating to the messy interface to the hardware broadcast interface and to packing of data to conserve space in the store of the BDX930 processors. The running code was in fact developed prior to and without consideration of any mechanical verification. This was regarded as necessary experimentation with the SIFT hardware and special-purpose Pascal compiler. New sections of Pascal code, entirely independent of the running SIFT Executive, were written for the code verification from the lowest-level design specifications. Exactly those sections of code were constructed to correspond to the PrePost specifications. The Prepost specifications, in turn, specify only what was needed to prove I/O level properties addressed by the design verification.

The Pascal code sections cover: the selection of a schedule from the global executive broadcast, scheduling, dispatching, three way voting, and error reporting actions of the SIFT Executive. Not included in these sections of Pascal code are: the global executive, five way voting, clock synchronization, interactive consistency, low level broadcasting, and program loading, initialization, and schedule construction. The original intention was to augment the proven code in order to produce and deliver a running, proven system. Due to lack of project resources, this integration did not occur.

The STP system was developed and used to prove properties of axiomatically specified design specifications. It does not however support deduction of properties of Pascal programs. Relatively late in the project, a Pascal/HDM verification tool was developed to make this code proof capability available to the project. The tool accepts (precondition/postcondition) specifications written in HDM's SPECIAL language, together with a Pascal program, and produces an STP theory containing the generated verification conditions. The verification conditions were then proven in STP, with some assistance from the tool.

The complexity of the interface between SPECIAL, Pascal, and STP led to unwieldy interaction and a lack of confidence in the outcome of the code proof effort. In some cases, manual intervention was used to separate long verification conditions into manageable pieces, etc. Consequently, it is not possible to ascribe to the code proof the degree of confidence one might have in the design verification. However, the relationship between the PrePost Specification and the code sections is very close, and inspection of them will show that the specification requires the code to 'do what it does'. Virtually all interesting aspects of the design have been incrementally introduced and incorporated into the proof prior to the code level. For this reason, we feel that the actual code proof, while of course a necessary step in the complete exercise, is less valuable than design proof of the algorithms.

The code proof was performed for the sections of Pascal code described earlier, containing the selection of a schedule from the global executive broadcast, scheduling, dispatching, three way voting, and error reporting actions of the SIFT Executive. The size of these sections of code cannot be compared to the size of other SIFT code, written on an entirely different basis. Indeed, their size is not relevant; it is their functionality, of maintaining consistency and correct results in an asynchronous multicomputer system containing faults, that is important. These sections of code alone suffice to substantiate the I/O Specification property verified by the design verification. We intend to repeat this code verification, rigorously, using a new specification and verification system.

The code verification did not consider the code of the global executive, five way voting, clock synchronization, interactive consistency, low level broadcasting, and program loading, initialization, and schedule construction. It is our opinion that the code for the global executive, five way voting, interactive consistency, and schedule construction, is straight-forward and presents no special verification problems beyond those of any other program code. We have no plans to verify the program loading and initialization code. The difficulty of verifying the low level broadcast code, and the clock synchronization code, will depend very much on the model assumed for the 'Pascal' broadcasting hardware. Undoubtedly, they can be done but may be very messy. The proof of the clock synchronization code (only 35 lines of declarations and statements in Pascal) is probably beyond the state of the art in any available verification system without the use of hierarchical design verification.

Chapter 9 provides an introduction to the HDM-Pascal Code Verification System, and Chapter 10 describes the System for automatically constructing Verification Condition Generators, as used to build the HDM-Pascal VCG. Chapter 11 describes the operation of the Code Verification System and Chapter 12 contains the User Manual. Chapter 13 contains a simple example, and Chapter 14 describes the actual code verification done for SIFT.

CHAPTER 9

HDM-PASCAL CODE VERIFICATION SYSTEM – GENERAL INTRODUCTION

A Summary of HDM

This chapter provides a brief summary of the Hierarchical Development Methodology. HDM decomposes the design of a system into a hierarchy of abstract machines, linearly ordered with a different abstract machine at each *level* in the hierarchy. Each abstract machine in the hierarchy is dependent only on the functionality of lower-level machines. Each abstract machine provides all of the facilities (operations and abstract data structures) that are needed to realize (i.e., to implement operations of and to represent the data structures of) the machine at the next higher level. The facilities of the highest-level abstract machine, and only those of that machine, are visible to a user of the system. The lowest-level machine, denoted as the primitive machine, contains facilities that the designer deems as primitive, e.g., the hardware on which the system is running or a programming language. A machine is itself decomposed into *modules*, each module having operations and data structures which typically define a single abstract data concept. As in the Parnas module concept, the module is the *programming unit* of HDM; each of the modules may be independently implemented. The programs implementing a module can access the data structures of their own abstract machine, but not those of lower-level machines. Lower-level data structures may be modified only by the execution of lower-level operations. Thus the internal details of a module remain hidden from above the module.

In HDM there is a clear separation of the aspects of system realization into *stages*, as follows:

1. Conceptualization of the system.

2. Definition of the functions of the external interface and the structuring of those functions into a hierarchy of abstract machines, each consisting of one or more modules.

3. Adding further abstract machines to the structure of the entire system, including modules within the hierarchy that are not externally visible.

4. Formal specification of each module.

5. Formal representation of the data structures of each machine in terms of those of the modules at the next lower level.

6. Abstract implementation of the operations of each module, i.e., writing an abstract program for each abstract machine written in terms of the operations at the next lower level.

7. Coding, or transforming the abstract programs into efficient executable programs.

Parnas [12] has characterized software development as a sequence of decisions, where it is likely that decision di is dependent on earlier decisions d1, ..., d(i-1). What Parnas recognized as vital is that there is a proper order for decisions, namely the earlier decisions have the greater impact on the ultimate success of the system. Thus it is vital to identify the important decisions and to evaluate them critically. HDM has been designed to formalize this decision model.

Each of the stages of HDM involves the making of decisions, and HDM provides languages to express these decisions. Those decisions associated with stages (1) through (4) are generally considered as ***design***. Those associated with stage (5) and with stages (6) and (7) involve ***representation*** and ***implementation***, respectively. The decisions made from stage (1) to stage (7) are roughly in order of decreasing importance. For example, whether or not to use paging involves a design decision, and is clearly more important than how to store the page table --which is a representation decision. The algorithm for page replacement is an implementation decision. This approach contrasts with the current approach to software realization in which the program itself is used to capture all of the decisions of design, representation, and implementation. In a system designed according to HDM, the four stages would largely be pursued in order. Thus, all of the design decisions should be made before the representation or implementation is attempted. However, backtracking is normally expected. In addition, it is not implied that a designer first considers the highest abstract machine, then the next highest and so on, i.e., top-down design. We would expect that attention would be given to several abstract machines at a time, i.e. when a designer conceives of a particular abstract machine at a position in the hierarchy, he might also have in mind lower level abstract machines to implement that machine. It is also possible for the design to be accomplished top-down while the implementation proceeds bottom-up.

Module specification (stage 4) involves the expression of the intent of a module, independent of its implementation. The language SPECIAL (SPECIfication and Assertion Language) ([16], [13]) is used for this purpose and enables the concise and formal description of a module. SPECIAL is also used for writing intermodule representations (stage 5), which we call ***mapping functions***. The intermodule implementation programs (stage 6) are called ***abstract programs***, since each can be viewed as running on an abstract machine whose operations they invoke. Abstract programs are intended to be directly compiled into executable code (stage 7). The language used for writing abstract programs can be extremely simple since most of the complexity of the programs is embodied within the abstract machine operations invoked by the programs. We have developed a clean simple language (ILPL -- Intermediate Level Programming Language) to describe abstract programs. Alternatively, programs could be written directly in a modern programming language such as Ada, Euclid, or Modula.

The first three stages of HDM are fundamental to the development. The decisions precisely formulated for these stages provide an early documentation of a system, prior to implementation, and significantly more understandable than the implementation. They thus provide the basis for good implementation. The results of these stages also provide the assertions that define what correctness means for the system. Since each stage of HDM has an appropriate formal language for expressing the decisions made at that stage, machine checking is possible. Existing tools accomplish some types of machine checking for these stages.

The specifications for the highest-level abstract machine are a concise description of the system as seen by the user, but only in terms of those facilities that are relevant to the specifications, i.e., implementation details are omitted from the specification. In addition, the module specifications and mapping functions are used [14] to formulate assertions for the proof of the abstract programs. This report is concerned primarily with the design aspects of HDM, although references are included that discuss techniques for verification in HDM.

HDM is a new synthesis of several promising approaches to software design. It has been developed to address deficiencies in the current software practice. It has been clearly influenced by the concepts of hierarchical programming and its extensions, in particular the important principles of hierarchical design, of doing design prior to implementation, of decomposing a system into small manageable pieces, and of carrying out a proof of correctness simultaneous with design. Although these principles are well-known, they are difficult to apply to real systems. The key to the effectiveness of HDM is that it offers a practical means for constructing, manipulating, evolving, and maintaining formal program abstractions. This property is absent in current structured programming methodologies, and present in only primitive form in modern programming languages. Formal abstraction provides the mechanism for verification, separation of specifications and implementation, variations in the order of binding design decisions, family design, and other desiderata of modern system development.

At present, HDM is evolving and does not yet possess all of the on-line aids that would ease its routine use. For the immediate future (say the next two years), it will see its greatest use in systems where correctness is of extreme concern. We anticipate that in the future, HDM-like methodologies will be an important approach to the design of general software.

This document is intended to serve as an overview of HDM, describing in some detail most of the features needed to design and implement systems. Some attempt is made to justify particular features and to compare HDM with other approaches, but this report is not intended to be a complete survey on design methodologies. A more complete description of HDM can be found in the three-volume HDM Handbook [15, 17, 8].

Chapter 3 of this report discusses the uses of HDM in the development of secure systems and subsystems. Chapter 4 presents an example of the use of HDM, organized according to the stages of HDM outlined above. Chapter 5 presents part of the design of a secure data management system as an illustration of HDM's usefulness in designing a secure application system.

2. The Use of HDM for Attaining Security

This chapter presents some of the important aspects of HDM for developing software satisfying security requirements.

The attainment of security requires an overall perspective on the system needs. In general, it is very difficult (if not impossible) to enforce elaborate security policies in an application environment if the underlying system is insecure -- unless the application environment is extremely restrictive (e.g., has no sharing of resources, or hides all of the facilities of the underlying system). Thus it is necessary to consider the security provided by the operating system, not just the security provided by an application environment. Further, the attainment of security can be adversely affected by improper design, by poor choice of programming language, and by improper implementation. Any weak link could provide a critical flaw.

The ways in which HDM contributes to the attainment of secure software have been considered at length in [11]. These issues are only summarized here.

Suitability for verification is the major factor that differentiates HDM from the wealth of development methodologies. Since correctness is so critical for security, formal verification of security properties is considered mandatory for certain systems. HDM was developed to address the need for verifying large systems. HDM organizes the development into stages, the system into a hierarchy of abstract machines, and the machines into modules to produce units small enough and well structured enough to be amenable to verification. A verification methodology based upon this approach has been developed [14]. However, even if formal verification is not attempted, the precision and discipline imposed by HDM encourage sound design and implementation. The concentration on careful design and matching implementation, and the potential for analysis throughout make HDM an excellent choice when security is an issue.

2.1 Current Uses of HDM for Security

HDM is being applied to the design of several systems with critical requirements. These include secure systems designed at SRI, namely the Provably Secure Operating System (PSOS [10], [5]) and a secure real-time operating system (TACEXEC [4]). (HDM is also being used for NASA by SRI in the development of SIFT, an ultrareliable fault-tolerant computer system [18].)

HDM is being used outside of SRI as well. The Ford Aerospace and Communications Corporation is developing a system [KSOS] whose user interface is compatible with UNIX (Registered Trademark of Bell Laboratories) and which is based on a security kernel [9]. The security of the KSOS design is being subjected to formal proofs that the specifications are consistent with a formal model for multilevel security [1].

At the time of writing, all of the kernel specifications have been subjected to the proof process, and the proofs have pointed out the flaws remaining in the design. Honeywell is using HDM on its own version of KSOS [KSOS-6], and has used it in the past for a flight-control system [2] and for the design and proof of a secure kernel for a Multics-like system (together with the MITRE Corp.) [7]. In addition, there have been and are various other experimental uses of HDM.

Until now, most of the applications of HDM have been to operating systems or kernels in which there are extremely critical requirements. In many of these efforts, verification is an important consideration.

2.2 Requirements

A system should be designed with a clear understanding of what requirements it is to meet, particularly with regard to security. It is desirable to have a precise definition of what it means for a system to be "secure". For example, the PSOS design permits the implementation of highly sophisticated security policies; various properties of the basic PSOS protection mechanism have been formalized. The KSOS design has a security kernel which provides enforcement of a multilevel security policy (under which information at a given security level cannot filter down to a lower level). A formal requirement that the specification for each kernel function satisfies this model is being used for the proofs mentioned above. (An earlier version is given in [3].) It is also applicable to trusted processes that are authorized to selectively violate the security of the kernel.

2.3 Design

HDM enforces constraints on the way in which a design is defined, although it does not essentially constrain what the design can achieve or what functionality can be implemented. Use of hierarchical design structure and formal specifications for each module in the hierarchy contributes to the avoidance of many types of security flaws commonly found in the design and implementation of existing systems. For example, the notion of abstract machine specificatin verification are appropriate. In general, verifiability is greatly enhanced by the use of HDM [14]. The staged decomposition of the development process permits design proofs to be carried out before implementation is attempted (providing a formal means for early evaluation of the design), and then permits proofs of program correctness. The hierarchical decomposition of the design into levels of abstract machines is particularly valuable in simplifying both the design proofs and the program proofs. Use of formally based languages of HDM is vital to both types of proof. Design proofs demonstrate a formal consistency between the formal specifications (in SPECIAL) and a formal model (e.g., a model of the security requirements). These specifications also form a basis for the program proofs, verifying that the program implementing a module specification is consistent with its specification. As noted above, the choice of programming language can greatly influence the feasibility of verification.

3. A Simple Example of the Use of HDM

In this chapter, HDM is used to describe a complete -- although very simple -- system: a "stack" module implemented in terms of an "array" module. The discussion is organized into seven sections: a review of HDM, and one section for each stage outlined in Chapter 2.

In HDM, a system evolves from an initial concept to verified executable code as a sequence of "decisions". In each stage of the development process, the system developer makes a series of decisions. The stages are ordered so that improper decisions tend to be exposed early, and therefore can be corrected early.

The verification aspects of HDM are found in [14]. Some aspects of verification are discussed below in connection with the first six stages.

A primary concern is to illustrate the staged, decision-oriented development of a system using the three languages of HDM -- HSL (the Hierarchy Specification Language), SPECIAL, and ILPL. Brief introductions to these languages are given to produce a reasonably self-contained description. However, the simplicity of the example does not properly illustrate many of the advantages of HDM as applied to complex systems. More details on HDM and a more complex example appear in [15, 17, 8].

3.1 Review of the Mechanisms of HDM

In HDM, a system is realized as a linear hierarchy (a sequence) of *abstract machines*, sometimes called *levels*. The top level is called the *user-interface*, while the bottom level is called the *primitive machine*. These two machines together are called the *extreme* machines. The remaining levels are called *intermediate machines*. Each machine provides *operations*, each of which has a unique name and arguments. An operation is *invoked*, similar to a subroutine call in a conventional programming language, by associating values for the operation's arguments. The invocation of an operation can return a value and/or modify the *internal state* (abbreviated as *state*) of the machine, as reflected by the values of the machine's *abstract data structures*. As discussed later, the "return" of an operation can be either a value or an "exception", the latter corresponding to one of a number of conditions that are defined for the module.

The "user-interface" provides the operations that are available to the user of the system. The operations of the "primitive machine" are typically constructs of a programming language and possibly some of the hardware operations.

A machine *specification* characterizes the value returned and the new state for each possible machine operation and each possible state of the machine. The specification describes the *functional behavior* of a system (returned values for all input combinations), but not necessarily the performance of the system or the resources consumed by its execution.

The realization of a machine (not the primitive machine, hereafter noted as machine i) is a two step process. First, the abstract data structures of a machine i (i not 1) are *represented* by those of the next lower-level machine i-1. Second, each of the operations of a machine i (i not 1) is *implemented* as a program in terms of the operations of machine i-1. The collection of implementations for all machines excluding the primitive machine constitutes the *system implementation.*

A machine is sometimes decomposed into simpler units called *modules.* For the purposes of this discussion, a module may itself be viewed as a machine; however, in reality a module's specification need not be self-contained, unlike that of a machine.

Clearly, system implementation is the desired end-product of the system development process. However, its emergence takes place only at stage 6. In the five previous stages, important decisions are made that logically progress toward the end product.

3.2 Stage 1 -- Conceptualization

In stage 1, the problem to be solved is formulated in general terms. Typically, the statement is in terms of constraints imposed on the extreme machines, and of the performance expected from the system. Currently, English is employed as the description medium, although consideration is being given to a formal language for conceptualization. For our single example, we will utilize the Conceptualization stage to provide informal descriptions of the extreme machines.

The user interface provides a collection of individually accessible stacks, manipulatable by conventional stack operations. The primitive machine consists of a collection of individually accessible arrays, as provided by a conventional high-level programming language. This example is developed according to the stages of HDM. The completed example is presented in the following figures.

Figure 3-1: Specification of the STACKS Module

```
MODULE stacks  $( maintains a fixed number of stacks of integers,
              each of the same fixed maximum size)

   TYPES

stack_name: DESIGNATOR $( names for stacks) ;

   PARAMETERS

INTEGER max_stack_size $( maximum size for a given stack) ;


   FUNCTIONS

VFUN ptr(stack_name s) -> INTEGER i;  $( stack pointer, or
                       number of elements, of stack s)
   HIDDEN;
   INITIALLY
     i = 0;

VFUN stack_val(stack_name s; INTEGER i) -> INTEGER v;
  $( v is the ith value of stack s)
   HIDDEN;
   INITIALLY
     v = ?;

OFUN push(stack_name s; INTEGER v);
  $( puts the value v on top of stack s)
   EXCEPTIONS
     stack_overflow : ptr(s) = max_stack_size;
   EFFECTS
     'stack_val(s, 'ptr(s)) = v;
     'ptr(s) = ptr(s) + 1;

OVFUN pop(stack_name s) -> INTEGER v;
  $( pops the stack s and returns the old top)
   EXCEPTIONS
     stack_underflow : ptr(s) = 0;
   EFFECTS
     'stack_val(s, ptr(s)) = ?;
     'ptr(s) = ptr(s) - 1;
     v = stack_val(s, ptr(s));

END_MODULE
```

Figure 3-2: Specification of the ARRAYS Module

```
  MODULE arrays $( maintains a fixed number of fixed-size
                integer arrays)


  TYPES

array_name: DESIGNATOR;


  PARAMETERS

INTEGER array_size $( the number of elements in an array);


  FUNCTIONS

VFUN access_array(array_name a; INTEGER i) -> INTEGER v;
  $( returns element i of array a)
  EXCEPTIONS
   array_bounds : i < 0 OR i > array_size - 1;
  INITIALLY
   v = 0;

OFUN change_array(array_name a; INTEGER i, v);
  $( changes the ith value of array a to v)
  EXCEPTIONS
   array_bounds: i < 0 OR i > array_size - 1;
  EFFECTS
   'access_array(a, i) = v;

END_MODULE
```

Figure 3-3: Mappings for STACKS and ARRAYS

```
   MAP stacks TO arrays;


   EXTERNALREFS

   FROM stacks:
stack_name: DESIGNATOR;
INTEGER max_stack_size;
VFUN ptr( stack_name s) -> INTEGER i;
VFUN stack_val( stack_name s; INTEGER i) -> INTEGER v;

   FROM arrays:
array_name: DESIGNATOR;
INTEGER array_size;
VFUN access_array( array_name a; INTEGER i) -> INTEGER v;


   INVARIANTS

FORALL array_name a: access_array(a, 0) <= array_size - 1
                        AND
                     access_array(a, 0) >= 0;


   MAPPINGS

stack_name: array_name;

max_stack_size: array_size - 1;

ptr( stack_name s): access_array(s, 0);

stack_val( stack_name s; INTEGER i):
     IF 1 <= i AND i <= access_array(s,0)
       THEN access_array(s, i)
       ELSE ?;


   END_MAP
```

Figure 3-4: Abstract Implementation of the STACKS Module

```
IMPLEMENTATION stacks IN TERMS OF arrays;

   EXTERNALREFS

   FROM stacks:
stack_name: DESIGNATOR;
INTEGER max_stack_size;
OFUN push(stack_name s; INTEGER v);
OVFUN pop(stack_name s) -> INTEGER v;
   FROM arrays:
array_name: DESIGNATOR;
INTEGER array_size;
VFUN access_array(array_name a; INTEGER i) -> INTEGER v;
OFUN change_array(array_name a; INTEGER i, v);

   TYPE MAPPINGS
stack_name: array_name;

   INITIALIZATION
BEGIN
  max_stack_size <- array_size - 1;
END;

   IMPLEMENTATIONS

OPROG push(stack_name s; INTEGER v);
DECLARATIONS
  INTEGER i;
BEGIN
  i <- access_array(s, 0) + 1;
  EXECUTE change_array(s, i, v) THEN
    ON array_bounds : RAISE(stack_overflow);
    ON NORMAL: ;  END;
  change_array(s, 0, i);
END;

OVPROG pop(stack_name s) -> INTEGER v;
DECLARATIONS
  INTEGER i;
BEGIN
  i <- access_array(s, 0);
  IF i = 0 THEN RAISE(stack_underflow); FI;
  change_array(s, 0, i-1);
  v <- access_array(s, i);
  RETURN(v);
END;
END_IMPLEMENTATION
```

3.3 Stage 2 -- Extreme Machine Interface Design

In stage 2, more detail is developed for the two extreme machines, concerned primarily with the decomposition of these machines into modules and the selection of the operations of the constituent modules. An *interface description* is derived for each module, specifying the module's operations and providing supporting information. The interface description is sometimes [6] referred to as the "syntax" of a module, in contrast to the specification (stage 4) which is referred to as the "semantics".

For our example, each (extreme) machine is a single module: "stacks" for the "user-interface", and "arrays" for the "primitive machine". Hence we here refer to "stacks" and "arrays" both as machines and as modules.

3.3.1 Interface Description for "stacks"

```
      MODULE stacks

   stack-name: DESIGNATOR

   INTEGER max_stack_size

   OFUN push(stack_name s; INTEGER v )
   OVFUN pop(stack_name s) -> INTEGER v
```

Some brief remarks about the syntax of SPECIAL are appropriate. First, all reserved words are in caps. Second, SPECIAL is a "typed" language in that a type is associated with each item when declared, thus permitting subsequent appearances of the items in a specification (see stage 4) to be checked for consistency with their declared type. For present purposes, a *type* is a set of values. The type INTEGER (a primitive type of SPECIAL) has as values all of the integers -- positive and negative (including zero). The type BOOLEAN (also a primitive type of SPECIAL) has as values TRUE and FALSE. Although not needed for this example, there are additional primitive types. New types, e.g., sets, vectors, structures (records), subtypes, may also be constructed out of existing types.

One or more types noted as ***designator types*** can be associated with a module. The values of these types, called ***designators***, serve as names for abstract objects of the module. The interface description of a module lists all of its designator types. For example, the "stacks" module interface description declares the designator type "stack_name" (an abbreviation for name-of-stack).

Following the de the module being specified, and additional functions declared to produce a more readable specification. The following are examples of types of expressions supported by SPECIAL.

1. Arithmetic Expressions

The value returned by an arithmetic expression is of type INTEGER or REAL. An arithmetic expression is a single constant, a variable or a user-defined function of type INTEGER or REAL, or is built out of existing arithmetic expressions using the operations "+", "*", "-", "/".

2. Boolean Expressions

The value returned by a boolean expression is of type BOOLEAN. The constants TRUE and FALSE are boolean expressions, as are variables and functions declared to be of type BOOLEAN. The operations AND, OR, "~" (NOT) and "=>" (IMPLIES) are used to build up boolean expressions from existing boolean expressions.

3. Relational Expressions

Using the infix relational operators (namely "=", "<", "<=", ">", ">=", "~="), boolean expressions are constructed from existing expressions. For "=" (or "~="), the resulting expression is of the form A = B (or A ~= B) where A and B are required to have the same type. For the other operators, each of the two component expressions is required to be of type INTEGER or REAL.

4. Conditional Expressions

A conditional expression is of the form IF P THEN Q ELSE R, where P is of type boolean, and Q and R are of the same arbitrary type. The type of the resulting expression is the type of Q (or R).

5. Quantified Expressions

To express properties relating to a large number of values, SPECIAL provides quantified expressions that are in the first-order predicate calculus. The universal quantified statement is written as

FORALL x | P(x): Q(x)

or

FORALL x: P(x)=>Q(x).

The meaning is "For all values of x such that P(x) is true, Q(x) is also true." Clearly, P(x) and Q(x) are of type BOOLEAN, as is the type of resulting expression. The variable x can be of any type, usually declared prior to its introduction in the specification.

The existentially quantified statement is written as

EXISTS x | P(x): Q(x),

which has the meaning "There exists a value x such that, if P(x) is true, then Q(x) is also true."

3.3.2 Role of "?" in SPECIAL

SPECIAL provides the particular value UNDEFINED (abbreviated as "?") to stand for "no value". It is used in a specification where the designer wishes to associate the absence of a meaningful value with a data structure. (UNDEFINED should not be confused with "don't care", which stands for some value.) UNDEFINED is only used in a specification, not in an implementation; no operation can return "?" as a value. For purposes of establishing type matching rules, however, "?" is assumed to be a value of every type.

3.3.3 Specification of 'stacks'

Now we are ready to discuss the SPECIAL specification of the module "stacks". This specification consists of three *paragraphs*: TYPES, PARAMETERS, and FUNCTIONS. More complex modules would require additional paragraphs, omitted here for simplicity.

1. TYPES paragraph

Here the types referred to in the specification are declared. It is required that all designator types (e.g., "stacks" for this module) be declared, but the declaration of other types can be deferred until the first appearance of an item of that type. Note that comments -- $(This is a comment) -- can appear anywhere in a specification.

2. PARAMETERS paragraph

All of the parameters are listed as they appear in the interface description of the module.

3. FUNCTIONS paragraph

Most of the functionally interesting information in a module specification is embodied in the FUNCTIONS paragraph. Each of the operations of the module ("push" and "pop" for the module "stacks") is listed and individually specified. In addition, other functions, typically V-functions corresponding to data structures, are introduced to assist in the specification of the operations. It is emphasized that, except for the primitive machine, the data structures serve only for purposes of specification.

We separately consider V-functions and O- and OV-functions.

a. Specification of V-functions

For purpose of specification, a V-function returns a value and never causes a state change. A V-function is classified as [primitive or derived] and [visible or hidden]. Thus a V-function is one of four flavors, identified by the combination of reserved words that appear in its specification.

The *primitive* V-functions -- "ptr" and "stack_val" for the "stacks" module -- correspond to the module's data structures. Their specification requires the association of an initial value with each possible argument value. That is, all primitive functions are defined to be "total", although many argument values correspond to physically meaningless conditions. For such conditions, the value of the function is usually "?". The expression following INITIALLY specifies the initial value. The primitive v-function "stack_val" returns the INTEGER v corresponding to the i-th location in stacks. We have decided that the initial value v of "stack-val" for any stacks is to be "?" for all i. The expression

```
v = ?
```

which is understood to mean

```
FORALL s; i: stack_val(s, i)=?
```

captures this decision. Note that in general the expression need not determine a unique initial value for a primitive V-function.

The other primitive V-function, "ptr" returns the value i of the stack pointer for stack s. The initial value of "ptr" is 0 for all stacks, reflecting the decision that all stacks are to be initially empty.

A *hidden* V-function cannot be called from outside the module, i.e., it is not an operation. The reserved word HIDDEN in the V-function specification declares the function to be hidden. Clearly, "stack_val" should be hidden since only the top element of the stack is to be accessible. However, some designs for a stack allow the pointer to be accessible.

The *visible* V-functions are operations that return a value, but do not cause a state change. They are identified by the absence of the word HIDDEN in the specification. As is the case for all operations, the specification can indicate a list of exception conditions. Since the "stacks" module has no visible V-functions, we defer discussion of exception conditions to the next section.

The value of a *derived* V-function is specified in terms of the values of the primitive V-functions. In the specification of a derived V-function, an expression that defines the returned value appears following the reserved word DERIVATION.

Because a V-function can serve multiple roles (say as an operation and a data structure), the length of a SPECIAL specification can be reduced, as compared with an alternative specification technique in which operations and data structures are separately specified.

b. Specification of O- and OV-functions

All O- and OV-functions are state-changing operations. An operation can return one of n exceptions ex1, ex2, ..., exn (we use the descriptive term "raise" in referring to exceptional returns), or can return "normally". No state change occurs when an operation invocation raises an exception. A value-returning operation (V- or OV-function) will return an actual value upon the NORMAL return; an O-function merely returns. Exception returns are a way of associating particular events with classes of states and values of the operation's arguments. In the specification of an operation, the specification of each exception condition consists of a name (typically a mnemonic for the condition) followed by a boolean expression that characterizes the condition. The list of exception conditions follows the reserved word EXCEPTIONS.

The behavior of an operation that has n exception conditions is determined as follows: if the expression corresponding to ex1 evaluates to true, then the first exception is raised; if the expression corresponding to ex1 evaluates to false and the expression corresponding to ex2 evaluates to true, then the second exception is raised; ...; finally, if the expressions corresponding to ex1, ..., exn evaluate to false, the operation returns normally.

For the O-function "push", there is the single exception condition, specified as:

    stack_overflow: ptr(s) = max_stack_size

The expression evaluates to true when the number of elements in the stack is equal to the maximum size of a stack.

Following the reserved word EFFECTS, the state changes that can occur as associated with

O- and OV-functions, together with the value corresponding to the NORMAL return of an OV-function, are specified. The specification consists of a collection of boolean expressions, each called an *effect* (in which the order of presentation is irrelevant). Semantically, the collection of effects should be read as a single expression which is the conjunction of the expressions corresponding to each of the effects. An effect can reference the following: arguments to the operation, values of primitive V-functions before the invocation ("old" values) of the operation, and value that primitive V-functions will obtain after the invocation ("new" values). In the specification, a single quote, "'" , preceding a primitive V-function indicates the value of the V-function after the invocation. The collection of effects defines the new value of each primitive V-function in terms of old values and argument values in the following way: the feasible new values for the primitive V- functions are those for which each of the effects evaluates to TRUE. Thus the specifications need not be *deterministic*, i.e., they need not define a unique new value for each primitive V-function argument list. However, the specifications for our simple example are deterministic.

When the new value of a primitive V-function for some argument is not constrained by the specification, it is assumed that the new value is identical to the old value.

For "push", the effects are:

```
    'stack_val(s, 'ptr(s)) = v;
    'ptr(s) = ptr(s) + 1;
```

They constrain the new value of "ptr(s)" to be the old value incremented by one, and the new value of the pointer for s to be the value v pushed onto the stack. Note that since the effects do not constrain the values of stack_val(s,i) for i ~= 'ptr(s), such values remain unchanged.

We will not burden the reader with a discussion of the effects for "pop", except for a few remarks. First, note that the returned value v is specified to be the INTEGER on the top of the stack in the old state. Second, the location at the top of the stack is the old state changed to be "?". It should be clear that this latter state change is apparent only in the specification. The implementation need not be concerned with this apparent storing of "?".

3.3.4 Specification of "arrays"

Since the specification of the module "arrays" is relatively straightforward, only a few clarifying remarks are necessary. The V-function "access_array" serves both as the principal data structure of the module and as an operation. Its invocation raises an exception if the actual argument i is out of bounds. Thus, although the function is defined to be total, its representation (for example, as a data structure in a programming language) need only account for values of i that are within bounds.

3.4 Stage 5 -- Data Representation

3.4.1 Overview of Module Representation

In this stage, the primary concern is with representing the data structures of each machine (other than the primitive machine) in terms of the data structure of the next lower-level machine. The description of the representation of a machine m in SPECIAL is denoted as the "m mapping". As with a module specification, a mapping can be checked for self-consistency, but also for consistency with the module specifications, interface description, and hierarchy description.

A mapping, similar to a module specification, does not act as executable code. Instead, a mapping is a formal description, serving as a record of the representation decisions and as an input to a verification system. Thus the representations are conveniently described using the SPECIAL expression mechanism.

Since the hierarchy for our example contains only two levels, only one mapping is required, for "stacks". The mapping contains three paragraphs: EXTERNALREFS, INVARIANTS, and MAPPINGS. (For more complicated systems, additional paragraphs would be required.) Before discussing the mapping in detail, it is appropriate to present informally the basic representation decisions.

3.4.2 Representation Decisions for "stacks"

Each stack of integers is represented as an integer array. The current value of the stack pointer for stacks is the value in the 0-th location of the array a corresponding to s. Each of the "defined" elements in stack s -- those in position l, 2, ..., ptr(s) -- are in corresponding positions of array. Thus the locations of array a starting with location ptr(s) + 1 hold values that are inconsequential to the "stacks" module. Since all locations of the array except the 0-th are available to hold stack elements, the maximum stack size is the array size minus one.

3.4.3 EXTERNALREFS Paragraph

In the EXTERNALREFS paragraph are listed all the items of the upper level that are to be represented, and those of the lower level that are the targets of the representation. For both levels, the items of concern are primitive V-functions, parameters, and designator types. Clearly, the primitive V-functions and parameters are of concern here as they are the data structures of the respective machines. However, the mapping must also consider the designator types of the upper modules, since they embody a set of values that have meaning only at the upper module, and thus are part of the data of that module. The inclusion of type information here (although redundant with information in the module specification) permits the type checking of a mapping as a self-contained unit.

3.4.4 MAPPINGS Paragraph

In this paragraph the representation decisions that were informally presented above are precisely formulated. Each upper-level data item is separately represented, that is, associated with an expression in terms of lower-level data items. The expression associated with an upper-level data item can be viewed as a definition of that item in terms of the data items at the next lower level.

The first of the mappings

```
stack_name: array_name
```

captures the decision that the type "stack_name" is to be represented by the type "array_name". It is understood that each designator s of "stack_name" is to be represented by a unique designator a of "array_name", although at this point it is not necessary to define precisely the correspondence between values of the two types. In general, a designator type of the upper level can be represented by any type of the lower level. Thus, designators can be represented (for example) by integers; indeed, assuming that a primitive machine supporting designators is not available, the ultimate representation of designators is likely to be in terms of such primitive data types as integers, characters, or machine words.

The second of the mappings

```
max_stack_size: array_size-1
```

captures the decision that the maximum number of stack elements is one less than the size of an array.

The third of the mappings

```
ptr(stack_name s): access_array(s, 0)
```

captures the decision that the stack pointer is stored in the 0-th location of the corresponding array. Note that s, declared to be of type "stack_name", appears in the defining expression in a context in which "stack_name" has no meaning. Clearly, s in the defining expression refers to the unique "array_name" designator corresponding to s. In general, when an argument a of some type t associated with the upper level appears in the defining expression, it is assumed to be the unique element a' that is the representation of a. Thus the type of a in the defining expression is t', where t' is the type that represents t.

The fourth of the mappings

```
    stack_val(stack_name s; INTEGER i) :
      IF i > 0 AND i <= access_array (s, 0)
        THEN access_array (s, i)
        ELSE ?
```

captures the decision that "defined" elements of the stack appear in corresponding elements of the array. For i corresponding to an undefined stack element, the expression must evaluate to "?"

3.4.5 INVARIANTS Paragraph

This paragraph contains boolean expressions (invariants) in terms of the lower level that are intended to be true after the execution of a program that implements an operation of the upper level machine. In effect, the invariants express constraints on the lower level state. It should be understood that the invariants are expected to be satisfied by any program referring to the operations of the lower-level machines. Generally, many invariants can be posed, but only those that assist in the verification, that are significant in the documentation of the system, or that simplify the implementation are included.

The single invariant of our example

```
FORALL array_name a:
  access_array(a, 0) <= array_size-1
      AND
  access_array(a, 0) >= 0
```

constrains the value in the 0-th location of all arrays to be bounded by 0 and array_size-1. Since the stack pointer is stored in the 0-th location of the corresponding array, this invariant indeed seems reasonable.

3.5 Stage 6 -- Abstract Implementation

In stage 6 each machine (other than the primitive machine) is implemented in terms of the machine at the next lower level. For machine i, the implementation consists of

- An *initialization* program whose execution causes the state of machine i-1 to become a state that maps (up) to the initial state of i; A program for each operation of i that satisfies its specifications.

All programs of the implementation of i reference operations of i-1.

For expressing the implementation programs, we have developed the language ILPL (Intermediate Level Programming Language). Although, in principle, almost any programming language could be used to express machine implementations, ILPL is particularly well-suited in that its syntax, type checking rules, and model of computation are compatible with the other languages of HDM.

We will not present a detailed description of ILPL, but instead illustrate some of its features in connection with the implementation of "stacks". First, we present a brief overview of the language.

3.5.1 Overview of ILPL

ILPL is an extremely simple imperative language, avoiding many of the complex features of high-order programming languages. The main purpose of ILPL is to describe a sequence of calls to operations. Some of the significant features of ILPL are the following:

- Simple argument passing discipline: In ILPL, all arguments are passed by "value". Of the conventional schemes for passing arguments -- by "value", by "reference", and by "name" -- "call by value" is conceptually the simplest. It has several advantages in implementing secure systems, including the avoidance of a wide class of security bugs referred to as "time-of-creation to time-of-use" modifications [11].

- Limited built-in data structures: In HDM, most of the data structures are provided by specified modules. Thus, ILPL need provide only a few simple kinds of data types, namely integers, characters, booleans, vectors, and structures (records).

- Controlled side-effects: Since an ILPL program consists mainly of calls to operations of a machine, the only side-effects are changes to V-functions as portrayed in the specifications of modules.

- Simple storage allocation: The only allocation carried out in the execution of an ILPL program is for local variables. Any dynamic allocation of objects is reserved for the modules that maintain such objects.

- No design aids in the language: Since HDM separates design and implementation decisions into distinct stages, all descriptions relating to design are expressed in SPECIAL or HSL.

- Structured exception handling: The program implementing an operation has multiple return points, one corresponding to the normal return and the remainder corresponding to the exceptional returns. A program referencing an operation "handles" any of the possible returns -- exceptional or normal -- for that operation.

- Type compatibility with SPECIAL: ILPL provides only a subset of the types of SPECIAL, essentially those that are easily implemented. Among those omitted is the "set". However, ILPL does support designator types, enforcing the same protection rules for designators as SPECIAL.

The implementation of "stacks" contains four paragraphs: EXTERNALREFS, TYPE MAPPINGS, INITIALIZATION, and IMPLEMENTATIONS, discussed next.

3.5.2 EXTERNALREFS Paragraph

All of the operations of both levels are listed. Also included are the parameters of both machines (since they can be referenced as operations) and the designator types. Complete type information is given here for all arguments and results, even though it duplicates information in the module specifications, allowing the implementations to be complete for purposes of type checking.

3.5.3 TYPE MAPPINGS Paragraph

The mappings of the designator types of the upper machine are listed. Again, this information has already appeared (in the representation), but its appearance here means that the implementation is self-contained.

3.5.4 INITIALIZATION Paragraph

The "initialization" program is given which when executed will drive the lower-level machine to a state that maps to the desired initial state of the upper-level machine. For the example, the initialization program has only to establish a value for the "stacks" parameter "max_stack_size". Note that the image of the initial state of "arrays" is such that "ptr(s)" has the initial value 0.

The reader might wonder how the initial value of "?" for stack_val(s, i) is realized. Recall that the representation for "stack_val" is

```
stack_val(s; i) :
  IF i >= 1 AND i <= access_array(s, 0)
    THEN access_array(s, i)
    ELSE ?
```

But the initial value of access_array(s, 0) is 0, in which case the expression following IF is false for all i. Hence, for the initial state of "arrays" the representation of "stack_val" becomes

```
stack_val(s, i) = ?
```

which is the initial value desired.

3.5.5 IMPLEMENTATIONS Paragraph

Following are the programs that implement the operations of "stacks". The informal description of the program for "push" should serve as a documentation of the program, and assist a reader in grasping the syntax of ILPL.

```
    INFORMAL DESCRIPTION OF "push"
Retrieve the 0-th element of the array (p, the stack pointer);
If i=p+1 is beyond the array bounds (and thus exceeds the
  maximum stack size), raise the "stack_overflow" exception
  and exit;
Modify the i-th location in the array to be v (push v onto
  the stack);
Modify the 0-th location in the array to be i (increment the
  the stack pointer);
```

For the actual program, the first statement is

```
i <- access_array(s, 0) + 1
```

No exception is expected from the invocation of access_array(s, 0), since the second argument (0) is clearly in bounds. The second statement

```
EXECUTE change_array(s, i, v) THEN
  ON array_bounds : RAISE(stack_overflow);
  ON NORMAL : ;
  END;
```

illustrates the mechanism for exception handling in ILPL. Following EXECUTE is a reference to an operation, change_array(s, i, v), that can lead to an exception, in this case "array_bounds". The text following ON has the following meaning: If the "array_bounds" exception is raised as a result of the invocation of "change_array", then the "stack_overflow" exception is raised as the termination of the program for "push". (If the "change_array" operation had more exceptions than were expected, they would be accommodated by additional "ON" terms.) If the "array_bounds" exception is not triggered, then the invocation of "change_array" terminates "normally" by storing v in the i-th location of array s.

To complete the description of the program, no exception is expected for the statement

```
  change_array(s, 0, i),
```

since it is seen that i is within bounds.

3.6 Stage 7 -- Coding

The abstract programs associated with stage 6 must ultimately be transformed into efficiently executable programs. That is the task of stage 7. In general, the task may be accomplished automatically or manually. The choice may rest on the actual hardware and on the tools available for compiling or assembling code. The development of automatic tools for accomplishing stage 7 is encouraged.

References

[1] T. Berson and J. Barksdale.
KSOS: Development Methodology for a Secure Operating System.
In *NCC '79, NY NY.* AFIPS, June, 1979.

[2] W. E. Boebert, J. M. Kamrad, E. R. Rang.
Analytic Validation of Flight Hardware.
Technical Report, Honeywell, 77SRC63, Systems and Research Center, Minneapolis, Minnesota, September, 1977.

[3] R. J. Feiertag, K. N. Levitt, L. Robinson.
Proving Multilevel Security of a System Design.
In *Sixth Symp. on Operating System Principles, 16-18 November 1977*, pages 57-67. ACM, November, 1977.
In Operating Systems Review, Vol. 11, No. 5.

[4] R.J.Feiertag.
TACEXEC.
Technical Report, SRI International, Menlo Park CA, Final Report, April, 1979.

[5] R.J. Feiertag and P.G. Neumann.
The Foundations of a Provably Secure Operating System (PSOS).
In *National Computer Conference 1978, Vol. 48*, pages 115-120. AFIPS, 1979.

[6] J. V. Guttag.
The specification and application to programming of abstract data types.
PhD thesis, Department of Computer Science, University of Toronto, 1975.
Computer Science Research Group Tech. Report CSRG-59.

[7] Honeywell Corp.
Project Guardian Final Report.
Technical Report, Honeywell Information Systems, Inc., Federal Systems Division, McLean VA, September, 1977.
ESD-TR-78-115.

[8] K. N. Levitt, L. Robinson, B. A. Silverberg.
The HDM Handbook, Vol III: A Detailed Example in the Use of HDM.
Technical Report, SRI International, June, 1979.

[9] E.J. McCauley and P. Drongowski.
KSOS: Design of a Secure Operating System.
In *NCC '79, NY NY.* AFIPS, June, 1979.

[10] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson.
A Provably Secure Operating System: the System, Its Applications, and Proofs.
Technical Report, SRI International, May, 1980.
This is the Second Edition, replacing the 1977 edition.

[11] P.G. Neumann.
Computer Security Evaluation.
In *National Computer Conference 1978, Vol. 47*, pages 1087-1095. AFIPS, 1978.

[12] D. L. Parnas.
On a 'Buzzword': Hierarchical Structure.
In *Information Processing 74*, pages 336-339. IFIP, North-Holland Pub. Co., Amsterdam, 1974.

[13] L. Robinson, K. N. Levitt, P. G. Neumann, and A. R. Saxena.
A Formal Methodology for the Design of Operating System Software.
Prentice-Hall, New York, 1977, .

[14] L. Robinson and K. N. Levitt.
Proof Techniques for Hierarchically Structured Programs.
Comm. ACM 20(4):271-283, April, 1977.

[15] L. Robinson.
The HDM Handbook, Vol I: The Foundations of HDM.
Technical Report, SRIInternational", June, 1979.

[16] O. M. Roubine and L. Robinson.
The SPECIAL Reference Manual.
Technical Report, SRI International, CSL-45, January, 1977.

[17] B. A. Silverberg, L. Robinson, K. N. Levitt.
The HDM Handbook, Vol II: The Languages and Tools of HDM.
Technical Report, SRI International, June, 1979.

[18] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar Smith, R. E. Shostak, and C. B. Weinstock.
SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control.
Proc. IEEE 66(10):1240-1255, October, 1978.

CHAPTER 10

META-VERIFICATION CONDITION GENERATOR; HOARE RULES FOR PASCAL

Automatic Construction of Verification Condition Generators From Hoare Logics

Mark Moriconi and Richard L. Schwartz

Abstract. We define a method for mechanically constructing verification condition generators from a useful class of Hoare logics. Any verification condition generator constructed by our method is shown to be sound and deduction-complete with respect to the associated Hoare logic. The method has been implemented.

1. Introduction

A *verification condition generator* (VCG), a central component in a program verification system, reduces the question of whether a program is consistent with its specifications to that of whether certain logical formulas are theorems in an underlying theory. VCGs must embody the semantics of the programming language; for the most part, they have been seen as implementations of Hoare-style axiomatic semantics. In the past, all such VCGs have been hand-coded for a specific language, with no formal guarantee that they accurately reflect the axiomatic language definition. The new contributions of this paper are (i) a general method for constructing VCGs mechanically from a useful class of Hoare logics and (ii) a formal basis for the method that provides the needed correspondence between a VCG and the axiomatic definition on which it is based.

Our theoretical results show that any VCG constructed by our method accurately reflects the axiomatic definition of the programming language. In other words, any such VCG is *sound* and *deduction-complete* with respect to the Hoare axiomatization of the language. Of course, it is still necessary to establish the soundness and relative completeness of the axiomatic definition with respect to an operational model [2].

In the process of trying to prove that our method has these properties, we found some subtle limitations of the commonly used implementation strategy for Hoare logics. This led us to identify precise conditions on Hoare logics under which this strategy will produce correct VCGs. Roughly the entire Pascal [5] and Euclid [7] axiomatizations satisfy our constraints, for example. We discuss the practical limitations of this work and propose extensions in the conclusion.

Our method consists of two main steps. An axiomatic definition is first transformed into a normal form from which we then derive a recursively defined VCG. The method has been implemented to form a meta verification condition generator, called MetaVCG. If supplied with an axiomatic language definition satisfying our constraints, MetaVCG will automatically produce a VCG for the language.

After introducing some of the basic concepts to be used in this paper, we present the normal form for rules (Section 3), the less constrained rule form for user-defined rules (Section 4), and then the main soundness and completeness results (Section 5).

2. Background and Overview

2.1. The Basic Method

In 1969 Hoare [4] introduced the style of axiomatic semantics frequently used to define programming languages. Hoare's approach is to regard program text as specifying a relation between assertions. The notation $P\{S\}Q$ is used to mean that "if *precondition* P is true before execution of program fragment S, and if execution of S terminates, then *postcondition* Q is true upon its completion". The meaning of every simple statement (such as assignment) is defined by an axiom schema and every compound statement (such as composition) is defined by an inference rule schema. A logical system containing Hoare axiom and deduction schemas for all syntactic forms in a programming language constitutes a partial-correctness *axiomatic definition* or *axiomatization* of the language [8].

The role of a VCG in a program verification system is to reduce the correctness of a sentence $P\{S\}Q$ to the correctness of some number of lemmas, called *verification conditions*, in the underlying theory. The provability of these lemmas is intended to be sufficient to guarantee that an axiomatic proof using the Hoare system could be constructed.

Verification condition generation is typically performed using a recursively defined *predicate transformer* $\mathrm{pre}(S,Q)$, which maps a program fragment S and a postcondition Q into a precondition. The function pre computes an assertion *sufficient* to guarantee that Q will be derivable as a postcondition (i.e., that $\mathrm{pre}(S,Q)\{S\}Q$ is provable). The provability of the verification condition $P \supset \mathrm{pre}(S,Q)$ in the underlying logic is thus sufficient to show that $P\{S\}Q$ is provable within any Hoare system containing the rule of consequence. A predicate transformer that produces preconditions which are both *necessary* and *sufficient* to derive Q as the postcondition is said to compute the *weakest derivable precondition*, and is denoted wdp(S,Q).

Our notion of wdp should not be confused with Dijkstra's notion of weakest liberal precondition wlp [3]. Weakest in our context is with respect to provability from the axiomatic definition, while Dijkstra's notion of weakest is relative to an interpretive model.

We now show how to derive wdp from a Hoare axiomatization of a programming language. Suppose that the normal form characterization of rules is

$$\frac{P_1\{S_1\}\mathcal{Q}_1, \ldots, P_n\{S_n\}\mathcal{Q}_n,\ \Gamma}{\mathcal{P}\{S\}\,Q} \tag{1}$$

which permits the deduction of $\mathcal{P}\{S\}\,Q$ from the $n+1$ premises. For the moment, let each $\mathcal{Q}_i$, Γ, and $\mathcal{P}$ stand for arbitrary logical expressions involving predicate symbols $P_1,\ldots,P_n,Q$ and formulas in the underlying theory. Examples of rules of this form are the axiom for assignment (without side effects)

$$P[x \leftarrow e]\,\{x := e\}\,P \tag{2}$$

where $P[x \leftarrow e]$ indicates the proper substitution of the expression e for each occurence of the variable x in P, and the rule of inference for statement composition

$$\frac{P\{S_1\}R,\ R\{S_2\}Q}{P\{S_1;S_2\}Q} \tag{3}$$

Given any rule of form (1), wdp can be defined as follows:

$$\mathrm{wdp}(S,Q) = \mathcal{P}[P_1 \leftarrow \mathrm{wdp}(S_1,\mathcal{Q}_1), \ldots, P_n \leftarrow \mathrm{wdp}(S_n,\mathcal{Q}_n)] \wedge (\forall \bar{v})\, \Gamma\, [P_1 \leftarrow \mathrm{wdp}(S_1,\mathcal{Q}_1), \ldots, P_n \leftarrow \mathrm{wdp}(S_n,\mathcal{Q}_n)] \tag{4}$$

where $[P_1 \leftarrow t_1,...,P_n \leftarrow t_n]$ denotes n proper substitutions carried out *sequentially* in a left-to-right order, and $\bar{v}$ is the set of all free logical variables in Γ. For example, the predicate transformer corresponding to assignment axiom (2) would be

$$wdp(x:=e,P) = P[x \leftarrow e]$$

and the one corresponding to composition rule (3) would be

$$wdp(S_1;S_2,Q) = P\,[P \leftarrow wdp(S_1,R),\ R \leftarrow wdp(S_2,Q)] = wdp(S_1,wdp(S_2,Q)) \qquad (5)$$

Notice that these are the predicate transformers usually associated with assignment and composition. In fact, the predicate transformers produced by wd: ɹe the ones commonly used to mechanize Hoare logics.

2.2. The Two Main Problems

As just discussed, a VCG reduces the question of whether a sentence $P\{S\}Q$ is a theorem in Hoare logic to the question of whether $P \supset wdp(S,Q)$ is a theorem in the underlying theory (e.g., first-order logic). Two important questions naturally arise:

1. Soundness. Does the VCG accurately reflect the semantics of the programming language as embodied by the associated Hoare logic? In other words, if $P \supset wdp(S,Q)$ is provable in the underlying theory, is $P\{S\}Q$ provable in the Hoare logic?

2. Completeness. Is a VCG as "powerful" as the Hoare logic from which it is derived? In other words, if $P\{S\}Q$ is provable, is $P \supset wdp(S,Q)$ also provable?

More formally, we must show that

$$\vdash_{\mathcal{H}} P\{S\}Q \quad \text{iff} \quad \vdash_{\mathcal{T}} P \supset wdp_{\mathcal{H}}(S,Q)$$

where $\mathcal{H}$ is a Hoare axiom system, $\mathcal{T}$ is the underlying logical theory, $wdp_{\mathcal{H}}(S,Q)$ is the predicate transformer derived from $\mathcal{H}$ as prescribed by (4), and $P \supset wdp_{\mathcal{H}}(S,Q)$ is the formula in $\mathcal{T}$ produced by the VCG. This theorem does not hold for arbitrary $\mathcal{H}$, as explained later. Thus, the problem is to find a sufficient set of constraints on $\mathcal{H}$ that does not unreasonably restrict the expressiveness of the resulting logics. The general rule form constraints of Section 4 have this property.

2.3. A Unifying Model

We need a conceptual model that connects formal axiomatic proofs and VCGs based on wdp. Such a model is provided by viewing a VCG as an *automatic proof constructor* for Hoare logic. An axiomatic proof of a sentence $P\{S\}Q$ consists of a sequence of steps where the last step is $P\{S\}Q$, and each previous step is either an instance of an axiom schema, a theorem in the underlying logic, or follows from previous steps by applying an instance of a rule of inference. In our model, a VCG constructs such proofs, using wdp to find instantiations for free predicate symbols in axioms and rules of inference. For any sentence $P\{S\}Q$, the basic strategy is to instantiate precondition P with $wdp(S,Q)$.

This model is illustrated below, where annotations (indicated by lines with roman numbering) relate the strategy used by the VCG in attempting to construct the formal proof of the sentence $\alpha\{z:=1;y:=z+1\}\beta$. Indentation indicates the nesting of recursive calls on wdp.

i. Select and instantiate composition rule for "$z:=1;y:=z+1$" with:

$S_1 \leftarrow z:=1,\ S_2 \leftarrow y:=z+1,\ Q \leftarrow \beta,\ R \leftarrow wdp(y:=z+1,Q)=\beta[y \leftarrow z+1],$
$P \leftarrow wdp(z:=1,R)=wdp(z:=1,wdp(y:=z+1,\beta))=(\beta[y \leftarrow z+1])[z \leftarrow 1]$

ii. Apply assignment axiom for "$y:=z+1$" with:

$x \leftarrow y,\ e \leftarrow z+1,\ P \leftarrow \beta$

1. $\beta[y \leftarrow z+1]\{y:=z+1\}\beta$ assignment

iii. Apply assignment axiom for "$z:=1$" with:

$x \leftarrow z,\ e \leftarrow 1,\ P \leftarrow \beta[y \leftarrow z+1],$

2. $(\beta[y \leftarrow z+1])[z \leftarrow 1]\ \{z:=1\}\ \beta[y \leftarrow z+1]$ assignment

3. $(\beta[y \leftarrow z+1])[z \leftarrow 1]\ \{z:=1;y:=z+1\}\ \beta$ composition (1,2)

4. $\alpha \supset (\beta[y \leftarrow z+1])[z \leftarrow 1]$ lemma

5. $\alpha\ \{z:=1;y:=z+1\}\ \beta$ consequence (4)

The overall proof strategy of the VCG is to select and instantiate the rule of inference that applies to the outermost syntactic structure in the program fragment (step i), satisfy its premises (steps ii and iii), and then conclude its conclusion (line 3). The VCG begins the proof by selecting the rule of composition (3) and performing the instantiations indicated above. This is a *valid instantiation* because it binds all free symbols in the rule. To see this, notice that when the substitution $[P \leftarrow wdp(z:=1,R),\ R \leftarrow wdp(y:=z+1,\beta)]$ is applied to the composition rule, we get $wdp(y:=z+1,\beta)$ for R and $wdp(z:=1,wdp(y:=z+1,\beta))$ for P. Next, the VCG must prove both premises of the rule, namely $wdp(y:=z+1,\beta)\{y:=z+1\}\beta$ and $wdp(z:=1,wdp(y:=z+1,\beta))\{z:=1\}wdp(y:=z+1,\beta)$. Expanding the definition of wdp, we see that both are instances of assignment axiom (2), yielding lines 1 and 2 of the formal proof. Having satisfied the premises of the composition rule, the VCG concludes line 3 of the proof.

Lines 4 and 5 are instances of a two-line scheme that completes every proof done by the VCG. Line 4 corresponds to the formation of $P \supset wdp(S,Q)$, which must be provable in the underlying theory for this to be a valid proof. Line 5 then follows immediately by the rule of consequence (ROC):

$$\frac{P \supset R,\quad R\{S\}T,\quad T \supset Q}{P\ \{S\}\ Q}$$

Although VCGs normally produce only line 4 as output, their output could just as easily be the entire axiomatic proof.

It should be pointed out that our model accurately describes the VCG only because of the restrictions we place on Hoare logics. The model clearly is inadequate for arbitrary Hoare logics. For example, it is easy to state a rule that requires the invention of an inductive assertion, which wdp is incapable of doing. The normal form constraints given in the next section were carefully chosen so that the VCG can always properly apply rules (consistency) and so that the instantiations computed by wdp will always be the weakest derivable instantiations (completeness).

This model provides the needed conceptual link between previous work on VCGs and Hoare logic. In the past, the VCG *itself* has been taken as the definition of the programming language. It is usually the case that the predicate transformer serves as the definition of each construct in the language, since there is often no axiomatic definition of the language. In some cases, however, there does exist an axiomatic definition of the programming language, but no formal correspondence between it and the predicate transformer is demonstrated. In both cases, we are left with the VCG as a *de facto* standard when automated proofs are attempted.

The only attempt at validating the view of a VCG as a proof constructor for an associated axiom system is the work of Igarashi, London, and Luckham [6]. In their paper, they give an axiomatic definition of a small language and an associated VCG. While they do not demonstrate a formal correspondence between their recursively specified VCG and their axiom system, they do prove that their axiom system is interderivable with another axiom system that more directly reflects the instantiation strategy employed by their VCG.

3. The Normal Form for Rules

This section defines a set of constraints on Hoare axioms and rules of inference under which the desired consistency and completeness property holds. Rules satisfying these constraints will be called *normal form rules.*

3.1. Notational Conventions and Preliminary Definitions

We will be defining properties of partially interpreted axiom and inference rule schemes, and must therefore carefully distinguish among three levels of discourse. In defining the normal form, we will use metavariables $\mathcal{P}$, $\mathcal{Q}$, $\mathcal{R}$, ... (with or without subscripts) to denote partially interpreted, standard first-order formulas. These formulas can contain uninterpreted predicate symbols P, Q, R, ... (with or without subscripts) and formulas from the underlying theory. For example, $\mathcal{P}$ could denote P, $P \wedge x=5$, or $x=5$. We assume that uninterpreted predicate variables P, Q, R, ... may be instantiated by formulas in the underlying theory. For example, P could be instantiated as $x=5$, but not $Q \wedge x=5$.

We will make use of a binary relation $\Leftarrow$ on uninterpreted predicate symbols. For a Hoare sentence of the form

$$\mathcal{P}(P_1, ..., P_m)\,\{S\}\,\mathcal{Q}(Q_1, ..., Q_n)$$

where predicate symbols $P_1,...,P_m$ and $Q_1,...,Q_n$ are logically free in $\mathcal{P}$ and $\mathcal{Q}$, respectively, we have

$$P_i \Leftarrow Q_j \text{, for } i \in \{1,...,m\} \text{ and } j \in \{1, ..., n\}$$

Intuitively, a relation $P \Leftarrow Q$ should be thought of as indicating that the binding of predicate symbol P is dependent upon the binding of predicate symbol Q. The relation $\Leftarrow$ is defined with respect to a set of Hoare sentences in the obvious way. The notation $\overset{+}{\Leftarrow}$ denotes the *transitive closure* of $\Leftarrow$. Whenever we have $H \overset{+}{\Leftarrow} T$, H will be called the *head* of the dependency chain and T the *tail.*

Similarly, for a rule of the form given in (1), we employ the relation $\ll$ to define the dependence of a proof concerning S on proofs concerning $S_1, ..., S_n$. In particular, we have $S \ll S_i$, for $i \in \{1,...,n\}$. For a Hoare axiom system, we define the transitive closure $\ll +$ in the obvious manner.

We use the function FreePreds to denote the set of *logically free* predicate symbols in a formula, a Hoare sentence, or a Hoare rule. FreePreds applied to a formula denotes its logically free symbols. FreePreds applied to a Hoare sentence $\mathcal{P}\{S\}\mathcal{Q}$ is simply the union of FreePreds($\mathcal{P}$) and FreePreds($\mathcal{Q}$), and FreePreds applied to an inference rule is the union of the predicate symbols obtained by applying FreePreds to each premise and the conclusion of the rule.

We will use the function FragVars to denote the set of "fragment variables" in the language fragment S of a Hoare sentence $P\{S\}Q$. For example, FragVars applied to "if B then S_1 else S_2 fi" has the value $\{B,S_1,S_2\}$. If applied to an entire Hoare rule, FragVars yields a set containing the fragment variables from every Hoare sentence in the rule.

Lastly, we use these two functions in defining the notion of a bound occurrence of an uninterpreted predicate symbol in a rule. For a rule R, a predicate symbol in FreePreds(R) is *bound in R* if and only if it is in FragVars(R). Otherwise, the occurrence is said to be *free in R*. Intuitively, we are carefully distinguishing those logically free variables that are bound in the program fragment when a rule is applied (i.e., those bound in the rule) from those that must be bound by wdp.

3.2. The Constraints

We will state the complete definition of the normal form and then explain it in detail.

The Normal Form

A *normal form rule* is any instance N of

$$\frac{P_1\{S_1\}\mathcal{Q}_1 \;\;....,\; P_n\{S_n\}\mathcal{Q}_n \;,\; \Gamma}{\mathcal{P}\{S\}\,Q}$$

that satisfies the following constraints:

1. $P_1, ..., P_n$ and Q are predicate symbols free in N.

2. Γ is a sentence in the underlying theory whose logically free predicate symbols can include only those in FreePreds(N) or FragVars(S).

3. The fragment variables of each S_i in the premises must be bound in S. That is, it must be the case that $\cup_{1\leq i\leq n}\text{FragVars}(S_i)\subseteq\text{FragVars}(S)$.

4. *Dependency ordering.* The Hoare-sentence premises of N must satisfy two dependency constraints.

   a. $P_i \overset{+}{\doteq} P_j \;\supset\; i<j$

   b. $T\overset{+}{\doteq}U \wedge \neg(\exists R)U\overset{+}{\doteq}R \;\supset\; U\equiv Q \vee U$ bound in N

5. *Monotonicity.* Let $\mathcal{P}[P\leftarrow\text{false}, P\in s]$ denote $\mathcal{P}$ with the proper substitution of false for each predicate P in the set s. Then, the following constraint on $\mathcal{P}$ must be satisfied:

$$\mathcal{P}[P_1,...,P_n,Q\leftarrow\text{true}] \;\vee\; \forall s\subseteq\{P_1,...,P_n,Q\}\; \neg\mathcal{P}[P\leftarrow\text{false}, P\in s])$$

This constraint must hold for Γ (with $\mathcal{P}$ replaced by Γ) and for each $\mathcal{Q}_i$ (with $\mathcal{P}$ replaced by $\mathcal{Q}_i$).

For axioms, this definition collapses to sentences of the form $\mathcal{P}(Q)\{S\}Q$, where postcondition Q is the only predicate symbol that can be free in the axiom and the following constraint must be satisfied: $\mathcal{P}[Q\leftarrow\text{true}] \vee \neg\mathcal{P}[Q\leftarrow\text{false}]$.

Two constraints are placed on a collection of normal form rules: (i) Any terminal string σ in the programming language can be an instance of at most one language fragment S defined by by a normal form axiom or inference rule. (ii) The relation $\ll+$ must be irreflexive. (We show later that this will guarantee termination of wdp.) Also, accompanying every normal form system are the ROC and the axiom false$\{S\}Q$.

Constraints 4 and 5 require further explanation. Constraint 4 ensures that wdp will be able to compute instantiations for all free, uninterpreted predicate symbols in rules using left-to-right substitution of wdp$(S_i,\mathcal{Q}_i)$ for each $\mathcal{P}_i$. This is done by first imposing restrictions on where free predicate symbols can occur in rules, and then placing constraints on some of these symbols based on dependency considerations. Constraint 4a requires an ordering of free predicate symbols that is made apparent by the following schema:

$$\frac{P_1\{S_1\}\mathcal{Q}_1(P_2,\ldots,P_n),\ \ldots,\ P_i\{S_i\}\mathcal{Q}_i(P_{i+1},\ldots,P_n),\ \ldots,\ P_n\{S_n\}\mathcal{Q}_n}{\mathcal{P}(P_1,\ldots,Q)\,\{S\}\,Q}$$

This says that every precondition of a Hoare sentence premise of an inference rule can depend only upon preconditions occurring in subsequent premises. This has the effect of eliminating dependency cycles, such as a premise of the form $P\{\ldots\}P$ (as in the case of the "unasserted" while statement) or a pair of premises of the form $P\{\ldots\}R$ and $R\{\ldots\}P$. In neither case would wdp be able to find an instantiation for the repeated symbol P. Also note that 4a requires not only that a proper ordering of premises exists, but that premises actually be placed in the prescribed order. For example, if the premises for of composition rule (3) were reversed, it would not satisfy 4a.

Given this ordering, constraint 4b ensures that the tail of every dependency chain is either expressible as a function of postcondition Q or is bound in a program fragment. In the hypothesis of 4b, U is the tail of a dependency chain $T \overset{+}{=} U$ which does not also occur as the head of another chain (i.e., there is no other R such that $U \overset{+}{=} R$). The conclusion of 4b says that every such U must be either the postcondition Q or a fragment variable in N, both of which are bound without the use of wdp when a rule is applied. Composition rule (3), for example, satisfies this constraint, since postcondition Q is the only tail not also occurring as the head of a dependency chain. In contrast, a rule containing premises $P\{\ldots\}T$, $S\{\ldots\}R$, and $R\{\ldots\}Q$ would not be allowed, unless T were bound in a program fragment. Otherwise, wdp would not compute an instantiation for T.

Constraint 5 is necessary for completeness. Recall that the completeness of a VCG hinges upon its ability to compute the weakest derivable precondition wdp(S,Q) for a given S and Q. As the simplest example of a rule for which wdp cannot compute the weakest derivable precondition, consider the axiom $\neg Q\{S\}Q$. From this axiom, it is possible to prove true{S}true using the ROC. The predicate transformer associated with this axiom by (4) is $\mathrm{wdp}(S,Q)=\neg Q$, meaning that $\mathrm{wdp}(S,\mathrm{true})=\mathrm{false}$. But true (not false) is the weakest derivable precondition. This same sort of difficulty can result from interactions among several different rules.

Therefore, Constraint 5 imposes a monotonicity constraint on rules, which eliminates rules in which certain "changes of sign" exist between the preconditions of the premises and the precondition in the conclusion. The first disjunct of 5 says that an inference rule that does not have a sign change is acceptable. That is, if the truth of $\mathcal{P}$ follows from the truth of $P_1, \ldots, P_n$ and Q, the rule is acceptable. The second disjunct states that a sign change in an inference rule is acceptable if the falsity of $\mathcal{P}$ is *independent* of the free variables in the rule. More precisely, it says that a rule is acceptable if there are no truth values assignments to $P_1, \ldots, P_n$ and Q that will make $\mathcal{P}$ true. Whenever this is the case, we know that any sign change is a function of predicate symbols bound in the language fragment; it turns out that this does not result in incompleteness. The axiom $\neg Q\{S\}Q$ above does not satisfy this constraint.

A normal form definition of a simple language is given in Figure 1; the general form of this definition is given in the next section. Although the while rule N4 and the conditional rule N5 may appear unusual, their general rule form is the common one. Also note that the procedure declaration and call rules (N7 and N8, respectively) use assertion-language functions to handle the association between procedure declaration and call. The predicate boundP(p,Q) is used in N8 to test whether there is an expression of the form bind(p,<*assertion,assertion,variable*>) in Q before total functions getpre, getpost, and getvars are applied to retrieve binding information at the point of call. A more elegant approach to handling this contextual information is suggested in the conclusion. For pedagogical reasons, we assume in our simple language that expressions have no side effects, procedures are nonrecursive, procedure as parameters and aliasing in procedure calls are prohibited, and global variables are disallowed.

Axioms

N1. simple assignment: $P[x \leftarrow e]\ \{x := e\}\ P$

N2. array assignment: $P[a \leftarrow \mathrm{arrayUpdate}(a,e_1,e_2)]\ \{a[e_1] := e_2\}\ P$

N3. empty statement: $P\{\ \}P$

Rules of inference

N4. iteration:

$$\frac{P_1\{S\}P,\quad P \wedge \neg B \supset Q,\quad P \wedge B \supset P_1}{P\ \{\textbf{while}\ B\ \textbf{assert}\ P\ \textbf{do}\ S\ \textbf{od}\}\ Q}$$

N5. conditional:

$$\frac{P_1\{S_1\}Q,\quad P_2\{S_2\}Q}{B \supset P_1 \wedge \neg B \supset P_2\ \{\textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\}\ Q}$$

N6. compound statements:

$$\frac{P\{S_1\}R,\quad R\{S_2\}Q}{P\{S_1; S_2\}\ Q}$$

N7. procedure declaration:

$$\frac{U\{S_1\}Q,\quad R\{S_2\}T \wedge \mathrm{bind}(p,\langle P,Q,x\rangle),\quad P \supset U}{R\{\textbf{begin proc}\ p(\textbf{var}\ x) = \textbf{pre}\ P;\ \textbf{post}\ Q;\ S_1\ \textbf{corp};\ S_2\ \textbf{end}\}T}$$

N8. procedure call:

$$\frac{\mathrm{boundP}(p,Q)}{(\forall \bar{v})(\mathrm{getpre}(p,Q)[\mathrm{getvars}(p,Q) \leftarrow a] \wedge \mathrm{getpost}(p,Q)[\mathrm{getvars}(p,Q) \leftarrow x']) \supset Q[a \leftarrow x']\ \{p(a)\}\ Q}$$

Figure 1: Example normal form language definition.

4. An Equivalent Rule Form With Fewer Constraints

So far, we have explained the normal form for rules and how to transform them into a VCG. This section presents the remaining part of our method, which is motivated by the practical concern of wanting to impose as few constraints as possible on rules written by users of MetaVCG. The general rule form defined below allows considerable flexibility in stating premises to inference rules -- premises need not be ordered and may have more general preconditions. This rule form has the important property that any rule satisfying its constraints can be mechanically transformed into an equivalent normal form rule. The normal form rules of Figure 1 that are more conveniently expressed in this general rule form are contained in Figure 2, and the transformation between the two rule forms is defined in the appendix.

The General Rule Form

A *general form rule* is any instance G of

$$\frac{\mathcal{S}_1, \ldots, \mathcal{S}_n, \Gamma}{\mathcal{P}\{S\}\ Q}$$

that satisfies normal form constraints 1-3 and 4b, where:

1. Each premise $\mathcal{S}$ is a Hoare sentence of one of the following forms.

a. $\mathcal{R}\ \{S\}\ \mathcal{Q}$ b. $\mathcal{F}\ \{S\}\ \mathcal{Q}$ c. $\mathcal{R} \wedge \mathcal{F}\ \{S\}\ \mathcal{Q}$

where, in all three cases, $\mathcal{R}$ is a metavariable evaluating to a single predicate symbol free in G, $\mathcal{F}$ is a metavariable evaluating to a formula not containing any predicate symbols free in G, and $\mathcal{Q}$ is a metavariable.

2. The relation $\overset{+}{=}$ must be irreflexive with respect to $\mathcal{S}_1, \ldots, \mathcal{S}_n$.

3. Let $\bar{r}$ be the set of predicate symbols free in the preconditions of $\mathcal{S}_1, \ldots, \mathcal{S}_n$.
Then, the following constraint on $\mathcal{P}$ must hold:

$$\mathcal{P}[P \leftarrow \mathrm{true},\ P \in \bar{r} \cup \{Q\}] \quad \vee \quad \forall s \subseteq \bar{r} \cup \{Q\}\ \neg\mathcal{P}[P \leftarrow \mathrm{false},\ P \in s]$$

This constraint must hold for Γ (with $\mathcal{P}$ replaced by Γ) and for each $\mathcal{Q}_i$ (with $\mathcal{P}$ replaced by $\mathcal{Q}_i$).

G1. Iteration:

$$\frac{P\land B\{S\}P,\ P\land\neg B\supset Q}{P\ \{\text{while } B \text{ assert } P \text{ do } S \text{ od}\}\ Q}$$

G2. Conditional:

$$\frac{P\land B\{S_1\}Q,\ P\land\neg B\{S_2\}Q}{P\ \{\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}\}\ Q}$$

G3. Procedure declaration:

$$\frac{P\{S_1\}Q,\quad R\{S_2\}T\land \mathrm{bind}(p,\langle P,Q,x\rangle)}{R\{\text{begin proc } p(\text{var } x)=\text{pre } P;\ \text{post } Q;\ S_1 \text{ corp};\ S_2 \text{ end}\}T}$$

Figure 2: Acceptable renditions of awkward normal form rules.

The interesting constraints are the first two. Constraint 1 gives the user considerable flexibility in expressing the Hoare sentence premises of an inference rule by lifting three normal form restrictions. Constraint 1a allows duplicate free predicate symbols as preconditions, and 1c allows a combination of (possibly duplicate) free and bound predicate symbols. Rules G1 and G2 illustrate the utility of this weakening of the normal form constraints. Constraint 1b allows preconditions whose logically free variables are bound in the rule, as illustrated by G3.

Constraint 2 is the only dependency constraint. It says that the Hoare sentence premises of an inference rule can be unordered, provided there are no dependency cycles. This is in contrast to normal form constraint 4a, which requires a very particular ordering of premises.

The collection of general form axioms and rules of inference must satisfy the two overall constraints given for the normal form system.

5. Formal Basis for the Method

To demonstrate that a VCG constructed by our method is sound and deduction-complete with respect to a general form axiomatic definition $\mathcal{G}$, we prove the following theorem.

Theorem: Let $\mathcal{G}'$ be any general form axiom system $\mathcal{G}$ augmented by the rule of consequence and the axiom false{S}Q, and let τ denote the transformation from $\mathcal{G}$ to the normal form, and suppose that $\mathcal{T}$ is a complete (perhaps noneffective) proof system for the underlying theory. Then

$$\vdash_{\mathcal{G}'} P\{S\}Q \quad \text{iff} \quad \vdash_{\mathcal{T}} P\supset wdp_{\tau(\mathcal{G})}(S,Q)\ .$$

The proof is done in two steps, showing first that

$$\vdash_{\mathcal{G}'} P\{S\}Q \quad \text{iff} \quad \vdash_{\tau(\mathcal{G})'} P\{S\}Q$$

and then that

$$\vdash_{\mathcal{N}'} P\{S\}Q \quad \text{iff} \quad \vdash_{\mathcal{T}} P\supset wdp_{\mathcal{N}}(S,Q)$$

where $\mathcal{N}'$ is any normal form axiom system $\mathcal{N}$ augmented by the ROC and the axiom false{S}Q. The former lemma demonstrates that a general form system $\mathcal{G}$ is sound and deduction-complete with respect to the normal form representation of $\mathcal{G}$ under τ. Its proof is tedious but routine and will not be given here. The second lemma, which we prove here, establishes that VCGs constructed by our method are sound and deduction-complete with respect to any normal form system $\mathcal{N}'$.

When wdp(S,Q) appears in a formula, there is an implicit assertion that it terminates and denotes a formula in the underlying theory. As part of the completeness proof, we will prove that, whenever a sentence P{S}Q is provable in $\mathcal{N}'$, wdp(S,Q) always in fact terminates and produces a formula in $\mathcal{T}$.

5.1. Main Soundness Result

In this section we prove the consistency of a VCG with respect to its associated normal form axiom system augmented by the ROC. The soundness lemma to be proved is:

$$\text{If} \quad \vdash_{\mathcal{T}} P \supset wdp_{\mathcal{N}}(S,Q) \quad \text{then} \quad \vdash_{\mathcal{N}'} P\{S\}Q \quad .$$

Henceforth, we will usually omit explicit reference to theories and will use wdp(S,Q) is an abbreviation for $wdp_{\mathcal{N}}(S,Q)$.

The proof is by induction on the depth of recursive application of wdp. In terms of our proof constructor model, we must show that wdp properly applies the axioms and rules of inference defining each construct of the programming language. Each recursive application of wdp must correspond to a valid proof step. We show that, for each S defined by an axiom, wdp(S,Q){S}Q is provable, and that for each S defined by an inference rule, wdp(S,Q){S}Q is provable whenever $wdp(S_i,\mathcal{Q}_i)\{S_i\}\mathcal{Q}_i$ is provable for each premise of the rule. This demonstrates that wdp(S,Q){S}Q holds for any construct S; the hypothesis and the ROC can then be used to obtain the desired conclusion.

As the base case for the induction, we consider the situation in which S is defined by an axiom of the form $\mathcal{P}(Q)\{S\}Q$. By the definition of wdp, we get wdp(S,Q){S}Q , from which the desired conclusion follows.

We now show that wdp properly applies inference rules defining the composite constructs of the language. This means that, for any normal form inference rule N, (i) wdp must find a valid instantiation of N and (ii) if $wdp(S_i,\mathcal{Q}_i)$ finds a valid instantiation for each of the n Hoare-sentence premises, then wdp(S,Q){S}Q follows. To establish (i) we must show that the left-to-right substitution

$$[P_1 \leftarrow wdp(S_1,\mathcal{Q}_1), ..., P_n \leftarrow wdp(S_n,\mathcal{Q}_n)] \tag{6}$$

binds all free predicate symbols in N. Recall that the premise of a normal form inference rule consists of n Hoare-sentence premises of the form $P_i\{S_i\}\mathcal{Q}_i$ and a sentence Γ in the underlying theory. Normal form constraints 4a and 4b require that each $\mathcal{Q}_i$ contain as logically free predicate symbols only $P_{i+1},...,P_n$, Q or predicate symbols bound in S. Further, these are the only symbols that can be free in $wdp(S_i,\mathcal{Q}_i)$. The successive left-to-right substitutions given in (6) will then eliminate each P_i in the Hoare-sentence premises. This leaves as logically free predicate symbols only Q and those bound in S, all of which are bound whenever a rule is applied. It follows from normal form constraint 2 that (6) also binds all free symbols in Γ.

We next establish (ii). We take as inductive hypotheses

$$wdp(S_1,\mathcal{Q}_1)\{S_1\}\mathcal{Q}_1, ..., wdp(S_n,\mathcal{Q}_n)\{S_n\}\mathcal{Q}_n \tag{7}$$

i.e., that wdp generates valid preconditions for each Hoare-sentence premise. Now let $\mathcal{Q}_i'$, $\mathcal{P}'$, and Γ′ stand for $\mathcal{Q}_i$, $\mathcal{P}$, and Γ under (6). More specifically, $\mathcal{Q}_i'$ is

$$\mathcal{Q}_i\,[P_{i+1} \leftarrow wdp(S_{i+1},\mathcal{Q}_{i+1}), ..., P_n \leftarrow wdp(S_n,\mathcal{Q}_n)] \quad , \tag{8}$$

$\mathcal{P}'$ is

$$\mathcal{P}[P_1 \leftarrow wdp(S_1,\mathcal{Q}_1), ..., P_n \leftarrow wdp(S_n,\mathcal{Q}_n)] \quad ,$$

and Γ′ is defined analogously to $\mathcal{P}'$. From our previous analysis of dependency constraints, we know that the only free predicate symbol in $\mathcal{Q}_i'$ is Q. $\mathcal{Q}_i'$ is thus a valid instantiation for $\mathcal{Q}_i$. Using this instantiation and our inductive hypothesis (7), we can conclude

$$\mathrm{wdp}(S_1,Q'_1)\{S_1\}Q'_1, \ldots, \mathrm{wdp}(S_n,Q'_n)\{S_n\}Q'_n \quad . \tag{9}$$

We now show that our VCG is sound independent of whether $(\forall\bar{v})\Gamma'$ is provable in $\mathcal{T}$. First suppose that $\vdash(\forall\bar{v})\Gamma'$. This coupled with (9) satisfies all the premises of N, allowing us to conclude $\mathcal{P}'\{S\}Q$, from which we obtain $\mathcal{P}'\wedge(\forall\bar{v})\Gamma'\{S\}Q$, which according to the definition of wdp is the same as $\mathrm{wdp}(S,Q)\{S\}Q$. Now assume that $\neg(\forall\bar{v})\Gamma'$. Then, from the definition of wdp, we see that $\mathrm{wdp}(S,Q)\supset\mathrm{false}$. In this case we can use the axiom $\mathrm{false}\{S\}Q$ and the ROC to conclude $\mathrm{wdp}(S,Q)\{S\}Q$.

The above induction argument shows that $\mathrm{wdp}(S,Q)\{S\}Q$ is provable for all S. The final step is to observe that our assumption that $P\supset\mathrm{wdp}(S,Q)$ and the ROC can now be used to conclude $\vdash_{\mathcal{N}'} P\{S\}Q$. ▮

5.2. Main Completeness Result

In this section we prove the completeness of a predicate transformer system produced by our method with respect to the sentences derivable from the axiom system $\mathcal{N}'$. In particular, we prove that

$$\vdash_{\mathcal{N}'} P\{S\}Q \quad \text{implies} \quad \vdash_{\mathcal{T}} P\supset\mathrm{wdp}(S,Q) \tag{10}$$

The proof is by induction on the number of recursive calls on wdp. We must show that, for any provable sentence $P\{S\}Q$, wdp can construct a proof using the weakest derivable instantiations. As the base case for our induction, we show that wdp computes the weakest derivable preconditions when S is defined by an axiom. The induction step considers the situation in which S is defined by a rule of inference. We prove inductively that if wdp generates the weakest precondition for each Hoare-sentence in the premises of the rule, then it will generate the weakest derivable precondition for the S defined by the rule. This is sufficient to show establish our theorem, since each premise used in the application of an inference rule is deduced from application of another inference rule or follows from an axiom.

Before presenting the main proof, we first establish that the function $\mathrm{wdp}(S,Q)$ always terminates. If S is defined by an axiom, wdp terminates because it involves no recursion. For S defined by an inference rule N, the termination of $\mathrm{wdp}(S,Q)$ depends upon the termination of each $\mathrm{wdp}(S_i,Q_i)$ in the n Hoare-sentence premises of N. Our overall system constraint that $\ll+$ is irreflexive guarantees that the proof of a sentence concerning S cannot depend upon satisfying a premise concerning S. The sequence of inferences attempting to satisfy the premises of N must therefore be finite, and thus the computation of $\mathrm{wdp}(S,Q)$ must also be finite.

Case 1 (*S defined by axiom*). Our theorem clearly holds if S is proved using $\mathrm{false}\{S\}Q$ (since P must be false). Now suppose S is proved using normal form axiom $\mathcal{P}(Q)\{S\}Q$, whose corresponding predicate transformer is $\mathrm{wdp}(S,Q)=\mathcal{P}(Q)$. Since this axiom uniquely defines S (excluding $\mathrm{false}\{S\}Q$ from consideration), it must be applied in any proof of $P\{S\}Q$. In the most general setting, a proof of $P\{S\}Q$ would involve showing that $P\supset\mathcal{P}(R)$ and $R\supset Q$, and then using the instance $\mathcal{P}(R)\{S\}R$ of our axiom and the ROC to conclude $P\{S\}Q$. Thus, our theorem (10) holds for axioms if we can show that $P\supset\mathrm{wdp}(S,R)\supset\mathrm{wdp}(S,Q)$.

We first observe that $P\supset\mathrm{wdp}(S,R)$ follows from the definition of wdp for S and the fact that $P\supset\mathcal{P}(R)$. We now show that $\mathrm{wdp}(S,R)\supset\mathrm{wdp}(S,Q)$ -- or equivalently $\mathcal{P}(R)\supset\mathcal{P}(Q)$ -- follows from the fact that $R\supset Q$ and the monotonicity normal form constraint. There are two ways in which $R\supset Q$ can hold. If R is true, $\mathcal{P}(R)\supset\mathcal{P}(Q)$ clearly holds since Q must also be true. Now assume that R is false and $\mathcal{P}(R)$ is true. Since our monotonicity constraint requires that $\mathcal{P}(\mathrm{true})\vee\neg\mathcal{P}(\mathrm{false})$, $\mathcal{P}(\neg R)$ must also be true. Hence, the truth of $\mathcal{P}$ is thus independent of the truth value of the free predicate symbol, and $\mathcal{P}(Q)$ must also be true. Thus wdp(S,Q) is the weakest derivable precondition if S is defined by an axiom. ∎

Case 2 (*S defined by a rule of inference*). Suppose that S is defined by normal form inference rule N. Our inductive hypothesis asserts that, from postcondition Q, wdp generates the weakest derivable instantiation for each precondition in the premises of N. That is, we assume that

$$P_1 \supset wdp(S_1,\mathcal{Q}'_1(Q)) \;,...., \; P_n \supset wdp(S_n,\mathcal{Q}'_n(Q)) \qquad (11)$$

where $\mathcal{Q}'_i$ is defined as before (8) and will contain only postcondition Q as a free variable.

We begin by observing that any proof of P{S}Q must necessarily follow a certain pattern, and then prove inductively that corresponding to any such proof is a proof of $P \supset wdp(S,Q)$ in $\mathcal{T}$. The general form of any proof of P{S}Q using inference rule N must proceed as follows: Since S can be defined only by N, we know that instantiating predicate symbols $P_1, ..., P_n$ and postcondition Q will yield a sentence of the form R{S}U such that $P \supset R$ and $U \supset Q$. The ROC would then be used to conclude P{S}Q. This argument can be characterized more formally as applying some substitution $[P_1 \leftarrow R_1, ..., P_n \leftarrow R_n, Q \leftarrow U]$ to N to obtain

$$\frac{R_1\{S_1\}T_1, ..., R_n\{S_n\}T_n \, , \gamma}{R\{S\}U}$$

such that the premises $R_1\{S_1\}T_1, ..., R_n\{S_n\}T_n$ and γ are satisfied and where $P \supset R$ and $U \supset Q$. Using the ROC, we conclude P{S}Q.

This observation guides the subsequent argument, which shows inductively that $R \supset wdp(S,U)$ given that $P \supset R$. A similar argument can be used to show that $wdp(S,U) \supset wdp(S,Q)$ given that $U \supset Q$. Combining these arguments yields $P \supset wdp(S,Q)$, which means that our method is complete for S defined by an inference rule.

We begin by instantiating our inductive hypothesis (11) with $[P_1 \leftarrow R_1, ..., P_n \leftarrow R_n, Q \leftarrow U]$, yielding

$$R_1 \supset wdp(S_1,\mathcal{Q}'_1(U)) \;,...., \; R_n \supset wdp(S_n,\mathcal{Q}'_n(U)) \qquad (12)$$

We proceed to show that wdp computes the weakest derivable precondition for S and Q, i.e.,

$$\vdash R \;\supset\; wdp(S,U) \quad . \qquad (13)$$

Noting that R is obtained by the proper substitution of $R_1, ..., R_n$ and U into precondition $\mathcal{P}$ of N, and expanding the definition of wdp, we will prove (13) by showing in two independent steps that

$$\mathcal{P}[P_1 \leftarrow R_1, ..., P_n \leftarrow R_n, Q \leftarrow U] \;\supset\; \mathcal{P}[P_1 \leftarrow wdp(S_1,\mathcal{Q}_1), ..., P_n \leftarrow wdp(S_n,\mathcal{Q}_n), Q \leftarrow U] \qquad (14)$$

and that

$$\gamma \;\supset\; (\forall \bar{v})\Gamma\,[P_1 \leftarrow wdp(S_1,\mathcal{Q}_1), ..., P_n \leftarrow wdp(S_n,\mathcal{Q}_n), Q \leftarrow U] \quad . \qquad (15)$$

Choose any assignment of truth values to $R_1, ..., R_n$ such that the antecedent of (14) holds. For any true R_i, $wdp(S_i,\mathcal{Q}'_i(U))$ must be true by the inductive hypothesis. Hence, $\mathcal{P}[P_1 \leftarrow R_1, ..., P_{i-1} \leftarrow R_{i-1}, P_i \leftarrow wdp(S_i,\mathcal{Q}'_i(U)), P_{i+1} \leftarrow R_{i+1}, ..., P_n \leftarrow R_n, Q \leftarrow U]$ must also be true.

Now consider any false R_i such that the antecedent of (14) holds. Recall that our monotonicity constraint requires

$$\mathcal{P}[P_1,...,P_n,Q \leftarrow true] \;\vee\; \forall s \subseteq \{P_1,...,P_n,Q\} \; \neg\mathcal{P}[P \leftarrow false, P \in s] \quad .$$

The first disjunct must hold (since, by hypothesis, there exists a false interpretation of an R_i rendering $\mathcal{P}$ true). But this implies that $\mathcal{P}$ is true irrespective of the truth value of R_i. Therefore, $\mathcal{P}[P_i \leftarrow wdp(S_i,\mathcal{Q}'_i(U))]$ will be true irrespective of the truth value of $wdp(S_i,\alpha_i'(U))$. This completes the proof of (14). The proof of (15) is exactly analogous, since Γ may contain only the free predicate symbols assumed in the proof of (14) and must satisfy an analogous monotonicity constraint.

This completes the proof of $R \supset wdp(S,U)$ at (13). Recall that, in order to complete the entire proof, we must consider derivations of $P\{S\}Q$ that use the second half of the ROC. Specifically, we must show that, given the fact that $U \supset Q$, $wdp(S,U) \supset wdp(S,Q)$. The requires an inductive argument following the reasoning used in the preceding one for (13). Expanding the definitions of $wdp(S,U)$ and $wdp(S,Q)$, we can show that each $wdp(S_i,\alpha_i'(U)) \supset wdp(S_i,\alpha_i'(Q))$, and thus that $wdp(S,U) \supset wdp(S,Q)$. Our hypothesis that $P \supset R$ and the fact that $R \supset wdp(S,U)$ can be used to conclude $P \supset R \supset wdp(S,U) \supset wdp(S,Q)$, thereby completing our proof of Case 2. Combining this with Case 1 demonstrates the completeness of of wdp with respect to the augmented normal form axiom system $\mathcal{N}'$. ∎

6. Conclusion and Future Work

The practical significance of this work is that it is now possible to correctly mechanize a useful class of Hoare logics by automated means. A VCG constructed by our method can serve not only as a central component of a program verification system, but it can also serve as a vehicle for "debugging" axiomatic definitions and exploring the semantic effect of various language design decisions.

It should be pointed out that a VCG produced by our method is correct *only if* the axiomatic definition of the programming language is correct. Thus, the remaining step in validating a verification system as a basis for reasoning about program behavior is to prove that the axiomatic definition is consistent and relatively complete with respect to an interpretive (operational) language model, as done by [2, 1].

Our theoretical results demonstrate that the traditional VCG paradigm is correct when certain constraints are placed on the rule forms in the associated Hoare logic. In addition, we found that MetaVCG's soundness depends upon the presence of $false\{S\}Q$ as an axiom, which brings out a interesting anomaly in the commonly used VCG paradigm. Recall that wdp is constructed from an inference rule by conjoining the premise Γ in the underlying theory to the precondition in the conclusion. In essence, Γ is "carried back" through the proof by wdp rather than occurring as a proof step to justify the application of the inference rule. As a simple example, instead of applying a rule with premise Γ and conclusion $\mathcal{P}\{S\}Q$, the VCG in effect applies the axiom $\mathcal{P} \wedge (\forall \bar{v})\Gamma\{S\}Q$. However, moving Γ from the premise to the precondition in the conclusion is not in general valid. While nothing can be proven from the original inference rule if Γ is unsatisfiable, $wdp(S,Q)\{S\}Q$ can be proven from the rule which the VCG actually applies. This is sound only if $false\{S\}Q$ is independently provable. Clearly, attempting to prove $false\{S\}Q$ when S is defined by an inference rule with an unsatisfiable premise is pathological. Nevertheless, it does illustrate that the paradigm only works if either $false\{S\}Q$ is an axiom or, for every inference rule, all premises in the underlying theory are satisfiable.

The expressiveness of our present theory is in principle sufficient for defining the semantics of real programming languages, provided the assertion language is rich enough. However, as a practical matter, it lacks the expressive power necessary to deal adequately with "context-dependent" semantics, such as full static scope, aliasing, side effects, exceptions, and procedures as parameters. Our method does well with context-independent properties of language semantics, but transfers much of the burden for defining context-dependent semantics to functions embedded in the assertion language.

We are exploring several extensions to our method, all subject to the constraint that they preserve its soundness and completeness. We are investigating the use of an enriched Hoare sentence $C/P\{S\}Q$, as described in [1, 10]. The additional C component expresses static information concerning program structure. This would allow us to reason about context directly within the Hoare logic, rather than having to embed contextual information in the assertion language (as was done in procedure declaration and call rules G3 and N8). Employing the context component, the revised procedure rules might be of the form:

$$\frac{C \cup \{\langle p, \mathrm{proc}(x)\mathrm{pre}\ P, \mathrm{post}\ Q\rangle\} \ /\ P\{S_1\}Q, \quad R\{S_2\}T}{C\ /\ R\{\textbf{begin proc}(\textbf{var}\ x) = \textbf{pre}\ P;\ \textbf{post}Q;\ S_1\ \textbf{corp};\ S_2\ \textbf{end}\}T}$$

$$C \cup \{p, \mathrm{proc}(x)\mathrm{pre}\ R, \mathrm{post}\ T\rangle\} \ /\ (\forall \bar{v})(R[x \leftarrow a] \wedge T[x \leftarrow x']) \supset Q[a \leftarrow x']\ \{p(a)\}\ Q$$

The use of the context component in the first rule allows us to define that the context for elaboration of the block body S_2 and procedure body S_1 is the surrounding context C augmented by the local block declaration for procedure p. The procedure call axiom then defines that the meaning of a procedure call is determined from the context at the point of call. Recursive procedures are handled by these rules.

Our current constraints allow only rules that have an inherent "backward" orientation and to which a backward predicate transformer semantics can be assigned. Our theory (and implementation) could be adapted to handle forward-oriented rules with a forward predicate transformer semantics as well. Based on analysis of predicate dependencies, MetaVCG could choose the appropriate substitution direction, provided that all of the rules have the same orientation. For example, the forward-oriented Algol 68 axiomatization [9] could be handled. Further extensions to handle axiom systems with no consistent orientation are also being studied.

Finally, we are exploring methods of introducing early interpretation of functions in the underlying theory to allow, for example, for interleaved generation and simplification of verification conditions. At present, all functions in the underlying theory (including proper substitution) remain uninterpreted throughout verification condition generation.

Acknowledgments. We would like to thank Larry Flon who, together with the first author, developed the initial concept of MetaVCG. Valuable comments on this paper were provided by Wolf Polak, Bernie Elspas, Pepe Meseguer, Steve Crocker, and Joe Goguen. This work was supported in part by NASA-Langley Research Center under Contract NAS1-15528 and Rome Air Development Center under Contract F30602-78-C-0031.

References

1. E.M. Clarke, Jr. Programming language constructs for which it is impossible to obtain good Hoare axiom systems. *Journal of the ACM*, 26,1, pp. 129-147, January 1979.

2. S.A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computing*, Vol. 7, No. 1, pp. 70-90, February 1978.

3. E.W. Dijkstra. *A discipline of programming*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976.

4. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, vol. 12, no. 10, pp. 576-580, October 1969.

5. C.A.R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2, 4, pp. 335-355, 1973.

6. S. Igarashi, R.L. London, D.C. Luckham. Automatic program verification I: A logical basis and its implementation. *Acta Informatica*, 4, pp. 145-182, 1975.

7. R.L. London, J.V. Guttag, J.J. Horning, B.W. Lampson, J.G. Mitchell, and G.J. Popek. Proof rules for the programming language Euclid. *Acta Informatica*, 10, pp. 1-26, 1978.

8. A. Meyer and J. Halpern. Axiomatic definitions of programming languages: a theoretical assessment. *Seventh Annual ACM Symposium on Principles of Programming Languages*, pp. 203-212, January 1980.

9. R. Schwartz. An axiomatic semantic definition of Algol 68. Ph.D. thesis, UCLA Computer Science Department Report UCLA-34-P214-75, August 1978.

10. R. Schwartz. An Axiomatic Treatment of Algol 68 Routines. *Proceedings of the International Conference on Automata, Languages and Programming*, Springer Verlag Lecture Notes in Computer Science, July 1979.

Appendix: Equivalence-Preserving Transformation to Normal Form

The transformation from the general rule form of Section 4 to the more constrained normal form of Section 3 is defined as follows. First sort the rule according to the three classes of allowable premises, yielding a schema of the form

$$\frac{\mathcal{P}_1\{S_1\}\mathcal{Q}_1, \ldots, \mathcal{P}_j\{S_j\}\mathcal{Q}_j, \mathcal{R}_{j+1}\{S_{j+1}\}\mathcal{Q}_{j+1}, \ldots, \mathcal{R}_k\{S_k\}\mathcal{Q}_k, \quad \mathcal{R}_{k+1} \wedge \mathcal{P}_{k+1}\{S_{k+1}\}\mathcal{Q}_{k+1}, \ldots, \mathcal{R}_n \wedge \mathcal{P}_n\{S_n\}\mathcal{Q}_n, \Gamma}{\mathcal{P}\{S\}\, Q}$$

We now define two functions:

$$\text{Duplicates}(i) \; = \; \{m : |\mathcal{R}_m| = |\mathcal{R}_i|,\; j+1 \leq m \leq n\}, \quad \text{for } j+1 \leq i \leq n$$

where, for a metavariable $\mathcal{R}$, $|\mathcal{R}|$ denotes the partially interpreted first-order formula bound to $\mathcal{R}$, and

$$\text{MkFormula}(i) \; = \; P_i \;(\text{for } j+1 \leq i \leq k) \;\text{ and }\; |\mathcal{P}_i| \supset P_i \;(\text{for } k+1 \leq i \leq n)$$

Now rewrite the sorted schema above as

$$\frac{P_1\{S_1\}\mathcal{Q}_1, \ldots, P_n\{S_n\}\mathcal{Q}_n, \Gamma \wedge (|\mathcal{P}_1| \supset P_1) \wedge \ldots \wedge (|\mathcal{P}_j| \supset P_j)}{\mathcal{P}\{S\}\, Q}$$

with the subsequent overall proper substitution

$$[\,|\mathcal{R}_i| \leftarrow \bigwedge_{k \in \text{Duplicates}(i)} \text{MkFormula}(k)], \quad \text{for } j+1 \leq i \leq n$$

The last step is to reorder the premises of this rule to satisfy normal form constraint 4a, which can always be done.

Example Input to Meta VCG: Definition of a Pascal Subset

Dwight Hare

```
[ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  ~~~~~~~~~~~~~~~~~~~~~ FUNCTION DEFINITION ~~~~~~~~~~~~~~~~~~~~~~~~~~~

    The function definition rule.  The preconditions of a function
    are the invariants of the current module and map and the assertions of
    the associated specification for this function.  The post condition
    are the effects of the function. ]

  P { statement :: compound.stmt } POSTCONDITIONS.OF (function.id.decl)
----------------------------------------------------------------------------
PRECONDITIONS.OF (function.id.decl) => BEGINNING.STATE.OF(P)
    { routine :: FUNCTION function.id.decl opt.formals : pas.type.id;
                          local.constants local.variables compound.stmt } Q;



[ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  ~~~~~~~~~~~~~~~~~~~~~ PROCEDURE DEFINITION ~~~~~~~~~~~~~~~~~~~~~~~~~~~

    The PROCEDURE definition rule.  The preconditions of a PROCEDURE
    are the invariants of the current module and map and the assertions of
    the associated specification for this PROCEDURE.  The post condition
    is the effects and the map invariants. ]

  P { statement :: compound.stmt } POSTCONDITIONS.OF (procedure.id.decl)
----------------------------------------------------------------------------
PRECONDITIONS.OF (procedure.id.decl) => BEGINNING.STATE.OF(P)
       { routine :: PROCEDURE procedure.id.decl opt.formals;
                         local.constants local.variables compound.stmt } Q;


[ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  ~~~~~~~~~~~~~~~~~~~~~ STATEMENT LIST ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    The STATEMENTLIST rule.  A statement list is a statement
    followed by a statement list.  In order to process the statements
    backwards, the statement list is processed and then the statement. ]


   P { statement :: statement } Q & R { stmt.list :: stmt.list } P
-----------------------------------------------------------------
   R { stmt.list :: stmt.list; statement } Q;


[ Another STATEMENTLIST rule.  This is for the last statement of the
   statement list.  The statement is processed. ]


   P { statement :: statement } Q
----------------------------------------
   P { stmt.list :: statement } Q;



[ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
   --------------------- ASSERT ---------------------------------------

   The ASSERT statement rule.  The current assertion must imply the
   post condition and the assertion is passed up as the pre condition. ]

   SPECS.OF(SAME.STATE.OF(specexp)) => Q
----------------------------------------------------
   SPECS.OF(SAME.STATE.OF(specexp)) { statement :: ASSERT specexp } Q;



[ -------------------------------------------------------------------
   --------------------- COMPOUND -------------------------------------

   The COMPOUND statement rule.  The body of the statement is processed ]


   P { stmt.list :: stmt.list } Q
---------------------------------------
   P { statement :: BEGIN stmt.list END } Q;


[ -------------------------------------------------------------------
   --------------------- IF -------------------------------------------

   The IF statement rule.  The expression is processed for overflow, etc.
   Three paths are generated, the test expression being true and the
   execution of the THEN statement implies the post condition and the
   test expression being false and the execution of the ELSE statement
   implies the post condition. ]

   P { statement :: statement1 } Q
&  R { statement :: statement2 } Q
&  S { pas.exp :: pas.exp } TRUE
---------------------------------------
S & (SPECS.OF(pas.exp) => P) & (NOT SPECS.OF(pas.exp) => R)
       { statement :: IF pas.exp THEN statement1 ELSE statement2 } Q;



[ -------------------------------------------------------------------
   --------------------- WHILE ----------------------------------------

   The WHILE statement rule.  Three paths are generated from the WHILE.
   The invariant assertion and the negation of the while test implies
   the post condition (terminating condition).  The invariant and the
   test condition being true through the statement implies the post
   condition.  The precondition of the rule is that the test expression
   is proper and the invariant is true.  ]

                  [ The path around the loop.  The invariant is pushed
                    through the body.  The invariant and the test expression
                    imply the result of the path. ]
   P { statement :: statement } SPECS.OF(SAME.STATE.OF(specexp1))
&  SPECS.OF(SAME.STATE.OF(specexp1)) AND SPECS.OF(pas.exp) => P

                 $( The termination condition.  The post condition for
                    the body is that the counting expression at the
                 end of the body is less than that at the start. )
&  R { statement :: statement }     SPECS.OF(specexp2) < COUNTING.EXP
                                AND SPECS.OF(specexp2) >= 0
&  RENAME.COUNTING.EXP(SPECS.OF(SAME.STATE.OF(specexp1)) AND SPECS.OF(pas.exp)
                           AND COUNTING.EXP = SPECS.OF(specexp2) => R)

&  S { pas.exp :: pas.exp } TRUE

               $( The path exiting the loop. )
&  SPECS.OF(SAME.STATE.OF(specexp1)) AND NOT SPECS.OF(pas.exp) => Q
----------------------------------------
S & SPECS.OF(SAME.STATE.OF(specexp1))
       { statement :: WHILE pas.exp ASSERT specexp1 DECREASING specexp2
                                DO statement } Q;


[ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  ~~~~~~~~~~~~~~~~~~~~~ REPEAT ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  The REPEAT statement rule.  The loop invariant is the post condition
  to the body.  This invariant condition and the test expression implies
  the post condition and the invariant condition and the negation of the
  test expression implies the precondition of the statement.  The
  precondition of the repeat statement and the test on the properness
  of the test expression is the precondition of the statement. ]


                [ The path around the loop.  The invariant is pushed through
                  the body.  The invariant and the negation of the test
                  expression imply the path through the body. ]
  P { stmt.list :: stmt.list } SPECS.OF(SAME.STATE.OF(specexp1))
&  SPECS.OF(SAME.STATE.OF(specexp1)) AND NOT SPECS.OF(pas.exp) => P

               $( The termination condition.  The post condition for
                  the body is that the counting expression at the
                  end of the body is less than that at the start. )
&  R { stmt.list :: stmt.list }     SPECS.OF(specexp2) < COUNTING.EXP
                                AND SPECS.OF(specexp2) >= 0
&  RENAME.COUNTING.EXP(SPECS.OF(SAME.STATE.OF(specexp1))
                           AND NOT SPECS.OF(pas.exp)
                           AND COUNTING.EXP = SPECS.OF(specexp2) => R)

&  S { pas.exp :: pas.exp } TRUE

               $( The exit path from the statement.  The invariant
                  and the test expression implies the post condition. )
&  SPECS.OF(SAME.STATE.OF(specexp1)) AND SPECS.OF(pas.exp) => Q
----------------------------------------------------------------------
               [ The entering path goes through the body once always ]
S & P { statement :: REPEAT stmt.list UNTIL pas.exp
                                ASSERT specexp1 DECREASING specexp2 } Q;



[ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  ~~~~~~~~~~~~~~~~~~~~~ ASSIGNMENT ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  The ASSIGNMENT statement rule.  The variable and the expression are
  checked for proper semantics and the precondition of the assignment
```

```
   is its post condition with the state of the assigned variable changed
   to reflect the assigned value.  ]

   P { variable.access :: variable.access } TRUE
&  S { pas.exp :: pas.exp } P
----------------------------------------
S AND IN.ASSIGN.BOUNDS (SPECS.OF(pas.exp), variable.access)
  & ASSIGN.RULE (variable.access, SPECS.OF(pas.exp), Q)
        { statement :: variable.access := pas.exp } Q;


[ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  ~~~~~~~~~~~~~~~~~~~~~ ARRAY ACCESS ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

   The ARRAY INDEXING rule.  The indexing expression must be proper and
   the array variable must be valid.  The precondition of this reference
   is the post condition, the validity tests, and the fact that the
   indexing expression is in the index set of the array.  ]

   P { pas.exp :: pas.exp } Q
&  S { variable.access :: variable.access } P
----------------------------------------
S AND IN.ARRAY.BOUNDS (SPECS.OF(pas.exp), variable.access)
        { variable.access :: variable.access [ pas.exp ] } Q;


[ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  ~~~~~~~~~~~~~~~~~~~~~ ARRAY REFERENCING ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

   The ARRAY INDEXING rule.  The indexing expression must be proper and
   the array variable must be valid.  The precondition of this reference
   is the post condition, the validity tests, and the fact that the
   indexing expression is in the index set of the array.  ]

   P { pas.exp :: pas.exp } Q
&  S { variable.ref :: variable.ref } P
----------------------------------------
S AND IN.ARRAY.BOUNDS (SPECS.OF(pas.exp), variable.ref)
        { variable.ref :: variable.ref [ pas.exp ] } Q;


[ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  ~~~~~~~~~~~~~~~~~~~~~ RECORD ACCESS ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

   The RECORD INDEXING rule.  The record variable is checked for proper
   semantics.  The precondition of the record reference is the post
   condition and the validity check on the variable.  ]

   P { variable.access :: variable.access } Q
---------------------------------------------------------
   P { variable.access :: variable.access . field.id } Q;


[ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  ~~~~~~~~~~~~~~~~~~~~~ RECORD REF ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
   The RECORD INDEXING rule.  The record variable is checked for proper
   semantics.  The precondition of the record reference is the post
   condition and the validity check on the variable.  ]

   P { variable.ref :: variable.ref } Q
-------------------------------------------------------
   P { variable.ref :: variable.ref . field.id } Q;



[ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   ~~~~~~~~~~~~~~~~~~~~~ PROCEDURE ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

   The PROCEDURE CALL statement rule.  The actuals are checked as being
   proper expressions.  The precondition for the procedure call is that
   the expressions are proper; the values of the actuals are in the range
   of the formals; and that the effects of the procedure implies the
   post condition.  All of the variables (in the actual list or referenced
   globally) which are possibly modified by the invocation of the procedure,
   must be renamed to signify their new values.  ]

   P { pas.exp.list :: pas.exp.list } TRUE
--------------------------------------------
P AND ACTUALS.IN.RANGE (procedure.id, SPECS.OF(pas.exp.list))
  AND PRECONDITIONS.FOR (procedure.id, SPECS.OF(pas.exp.list))
&    POSTCONDITIONS.FOR (procedure.id, SPECS.OF(pas.exp.list))
  => UPDATE.STATE (procedure.id, SPECS.OF(pas.exp.list), Q)
        { statement :: procedure.id ( pas.exp.list ) } Q;




[ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   ~~~~~~~~~~~~~~~~~~~~~ FOR ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

   The FOR statement (TO) rule.  In this rule, tests are used
   to assert that the FOR loop is executed at least once.  If the
   loop is executed then the invariant with the loop variable substituted
   by the final loop value implies the for loop post condition (with
   the loop variable renamed in the post condition).  The
   invariant pushed through the statement implies the invariant with
   the loop variable bumped by one.  The precondition of the for loop
   is that the range of values taken on by the loop variable is in its
   value set and that if the loop is not executed, then the post condition
   (with the loop variable renamed) is true and that if the loop
   is executed then the invariant is true with the identifier replaced
   by its initial loop value.  ]

                   [ The path around the loop.  The invariant and the assertion
                     that the loop variable is within the range of the loop
                     implies the path through the loop to the invariant. ]
   P { statement :: statement } SPECS.OF(SAME.STATE.OF(specexp))
&          SPECS.OF(SAME.STATE.OF(specexp))
       AND SPECS.OF(loop.var) >= SPECS.OF(pas.expl)
       AND SPECS.OF(loop.var) < SPECS.OF(pas.exp2)
    => ASSIGN.RULE(loop.var, SPECS.OF(loop.var) + 1, P)
&  S { pas.exp :: pas.expl } TRUE
&  R { pas.exp :: pas.exp2 } S
```

```
                $( The exit path from the loop.  The for stmt is
                   assumed to have executed at least once and the
                   loop variable in the invariant is replaced by
                   the final expression.  The invariant implies
                   the path exiting from the loop.  )
&           SPECS.OF(pas.exp2) >= SPECS.OF(pas.expl)
        => ASSIGN.RULE (loop.var, SPECS.OF(pas.exp2),
                        SPECS.OF(SAME.STATE.OF(specexp))
                          => UPDATE.VAR.STATE (loop.var, Q))
------------------------------------------------------------
                [ The precondition of the loop.  The possible range
                  of values of the loop variable must include the range
                  of the loop ... ]
  R AND LOOPVAR.IN.RANGE (loop.var, SPECS.OF(pas.expl), SPECS.OF(pas.exp2))

                $( ...and if the loop is not executed then the post
                   condition is true (with the loop var renamed) ... )
&    SPECS.OF(pas.expl) > SPECS.OF(pas.exp2)
  => UPDATE.VAR.STATE (loop.var, Q)

                $( ...and if the loop is executed, then the invariant
                   pushed through the statement with the loop variable
                   replaced by its initial value is asserted.  )
&    SPECS.OF(pas.exp2) >= SPECS.OF(pas.expl)
  => ASSIGN.RULE (loop.var, SPECS.OF(pas.expl), P)

        { statement :: FOR loop.var := pas.expl TO pas.exp2
                         ASSERT specexp DO statement } Q;




[ The FOR statement (DOWNTO) rule.  This rule is the same as the
  above TO version of the FOR rule with the following changes:
  The test of whether or not the loop is executed uses less-than rather
  than greater-than.  The inductive step decrements the loop variable
  instead of incrementing it.  ]

                [ The path around the loop.  The invariant and the assertion
                  that the loop variable is within the range of the loop
                  implies the path through the loop to the invariant. ]
  P { statement :: statement } SPECS.OF(SAME.STATE.OF(specexp))
&        SPECS.OF(SAME.STATE.OF(specexp))
     AND SPECS.OF(loop.var) <= SPECS.OF(pas.expl)
     AND SPECS.OF(loop.var) > SPECS.OF(pas.exp2)
   => ASSIGN.RULE(loop.var, SPECS.OF(loop.var) - 1, P)
&  S { pas.exp :: pas.expl } TRUE
&  R { pas.exp :: pas.exp2 } S

                $( The exit path from the loop.  The for stmt is
                   assumed to have executed at least once and the
                   loop variable in the invariant is replaced by
                   the final expression.  The invariant implies
                   the path exiting from the loop.  )
&           SPECS.OF(pas.exp2) <= SPECS.OF(pas.expl)
        => ASSIGN.RULE (loop.var, SPECS.OF(pas.exp2),
                        SPECS.OF(SAME.STATE.OF(specexp))
```

```
                            => UPDATE.VAR.STATE (loop.var, Q))
-------------------------------------------------------------
                [ The precondition of the loop.  The possible range
                   of values of the loop variable must include the range
                   of the loop ... ]
  R AND LOOPVAR.IN.RANGE (loop.var, SPECS.OF(pas.exp2), SPECS.OF(pas.exp1))

                $( ...and if the loop is not executed then the post
                   condition is true (with the loop var renamed) ... )
&    SPECS.OF(pas.exp1) < SPECS.OF(pas.exp2)
  => UPDATE.VAR.STATE (loop.var, Q)

                $( ...and if the loop is executed, then the invariant
                   pushed through the statement with the loop variable
                   replaced by its initial value is asserted.  )
&    SPECS.OF(pas.exp2) <= SPECS.OF(pas.exp1)
  => ASSIGN.RULE (loop.var, SPECS.OF(pas.exp1), P)

        { statement :: FOR loop.var := pas.exp1 DOWNTO pas.exp2
                         ASSERT specexp DO statement } Q;




[ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   ~~~~~~~~~~~~~~~~~~~~~~ CASE ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

   The CASE statement rule.  The selector expression is processed for being
   proper.  The case.selects are processed.  This is the
   recursive rule to handle the case.selects of the CASE statement.
   The statement on the selector is processed and the case statement
   rule is recursed to handle the other case selects.  The post condition
   of this rule is the results of the other case selects AND that the
   selector expression being in the label list implies the precondition
   of the statement.  ]

   S { pas.exp :: pas.exp } TRUE
&  P { case.vcg.stmt :: CASE pas.exp OF case.selects END } Q
-------------------------------------------------------------
S AND IN.ALLCASE.LIST (SPECS.OF(pas.exp), case.selects) & P
        { statement :: CASE pas.exp OF case.selects END } Q;



   R { statement :: statement } Q
&  P { case.vcg.stmt :: CASE pas.exp OF case.selects END } Q
----------------------------------------
INCASELIST (SPECS.OF(pas.exp), const.list) => R & P
        { case.vcg.stmt :: CASE pas.exp OF
                case.selects ; const.list : statement END } Q;



[ The last case selector is handled by this rule.  It states that the
   fact that the selector is in the label list implies the precondition
   of the selected statement.  ]

  P { statement :: statement } Q
----------------------------------------
```

```
INCASELIST(SPECS.OF(pas.exp), const.list) => P
        { case.vcg.stmt :: CASE pas.exp OF const.list : statement END } Q;




[ -----------------------------------------------------------------------
  ---------------------- BINARY OP --------------------------------------

   The BINARY expression rules.  Each expression is evaluated for proper
   semantics.  The precondition of these rules is the post condition and
   that each of the subexpressions is in the machine's value range.  ]

   P { pas.exp :: pas.exp1 } Q
&  R { pas.exp :: pas.exp2 } P
-----------------------------------------
R { pas.exp :: pas.exp1 AND pas.exp2 } Q;


   P { pas.exp :: pas.exp1 } Q
&  R { pas.exp :: pas.exp2 } P
-----------------------------------------
R { pas.exp :: pas.exp1 OR pas.exp2 } Q;


   P { pas.exp :: pas.exp1 } Q
&  R { pas.exp :: pas.exp2 } P
-----------------------------------------
R AND IN.MACHINE.RANGE(SPECS.OF(rule.form))
     { pas.exp :: pas.exp1 * pas.exp2 } Q;


   P { pas.exp :: pas.exp1 } Q
&  R { pas.exp :: pas.exp2 } P
-----------------------------------------
R AND NOT (SPECS.OF(pas.exp2) = 0)
     { pas.exp :: pas.exp1 DIV pas.exp2 } Q;


   P { pas.exp :: pas.exp1 } Q
&  R { pas.exp :: pas.exp2 } P
-----------------------------------------
R AND NOT (SPECS.OF(pas.exp2) = 0)
     { pas.exp :: pas.exp1 MOD pas.exp2 } Q;


   P { pas.exp :: pas.exp1 } Q
&  R { pas.exp :: pas.exp2 } P
-----------------------------------------
R AND IN.MACHINE.RANGE(SPECS.OF(rule.form))
     { pas.exp :: pas.exp1 + pas.exp2 } Q;


   P { pas.exp :: pas.exp1 } Q
&  R { pas.exp :: pas.exp2 } P
-----------------------------------------
R AND IN.MACHINE.RANGE(SPECS.OF(rule.form))
     { pas.exp :: pas.exp1 - pas.exp2 } Q;
   P { pas.exp :: pas.expl } Q
&  R { pas.exp :: pas.exp2 } P
----------------------------------------
R { pas.exp :: pas.expl = pas.exp2 } Q;


   P { pas.exp :: pas.expl } Q
&  R { pas.exp :: pas.exp2 } P
----------------------------------------
R { pas.exp :: pas.expl <> pas.exp2 } Q;


   P { pas.exp :: pas.expl } Q
&  R { pas.exp :: pas.exp2 } P
----------------------------------------
R { pas.exp :: pas.expl <= pas.exp2 } Q;


   P { pas.exp :: pas.expl } Q
&  R { pas.exp :: pas.exp2 } P
----------------------------------------
R { pas.exp :: pas.expl < pas.exp2 } Q;


   P { pas.exp :: pas.expl } Q
&  R { pas.exp :: pas.exp2 } P
----------------------------------------
R { pas.exp :: pas.expl > pas.exp2 } Q;


   P { pas.exp :: pas.expl } Q
&  R { pas.exp :: pas.exp2 } P
----------------------------------------
R { pas.exp :: pas.expl >= pas.exp2 } Q;



[ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   ~~~~~~~~~~~~~~~~~~~~~~ UNARY OP ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

   The UNARY expression rules.  The expression is checked for proper
   semantics and the precondition is that check and the fact that the
   expression does not overflow.  ]

   P { pas.exp :: pas.exp } Q
----------------------------------------
P AND IN.MACHINE.RANGE(SPECS.OF(rule.form))
     { pas.exp :: - pas.exp } Q;


   P { pas.exp :: pas.exp } Q
----------------------------------------
P { pas.exp :: NOT pas.exp } Q;
```

```
[ ---------------------------------------------------------------------
  ---------------------- EXPRESSION LIST ------------------------------

  The EXPRESSION LIST rule.  Each expression of the expression list
  is checked for proper semantics.  The precondition is this check.  ]

   P { pas.exp :: pas.exp } Q
&  R { pas.exp.list :: pas.exp.list } P
----------------------------------------
   R { pas.exp.list :: pas.exp.list, pas.exp } Q;



[ This rule is an expression list containing one expression  ]

   P { pas.exp :: pas.exp } Q
----------------------------------------
   P { pas.exp.list :: pas.exp } Q;



[ ---------------------------------------------------------------------
  ---------------------- FUNCTION CALL --------------------------------

  The FUNCTION CALL expression rule.  The actuals are checked for
  proper semantics.  The precondition of the function call is the
  test of the actuals and the condition that the actuals be in the value
  range of the formals.  The function invocation is specified to be
  equal to its specification derivation.  ]

   P { pas.exp.list :: pas.exp.list } Q
-------------------------------------------------------
P AND ACTUALS.IN.RANGE (function.id, SPECS.OF(pas.exp.list))
  AND PRECONDITIONS.FOR (function.id, SPECS.OF(pas.exp.list))
      { pas.exp :: function.id ( pas.exp.list ) } Q;
```

CHAPTER 11

HDM-PASCAL CODE VERIFICATION SYSTEM – DESCRIPTION OF OPERATION

The PASCAL-HDM Verification System

1. Introduction

This document describes the PASCAL-HDM verification system. This system supports the mechanical generation of verification conditions from PASCAL programs and HDM-SPECIAL specifications using the Floyd-Hoare axiomatic method [2]. Tools are provided to parse programs and specifications, check their static semantics, generate verification conditions from Hoare rules, and translate the verification conditions appropriately for proof using the Shostak Theorem Prover [7].

This document is mostly an overview and assumes that the reader is well acquainted with the languages involved and the theory of program verification. Hence an understanding is assumed of the PASCAL manual [3], the HDM handbook [6, 8, 4], verification techniques [2], mathematical logic and induction [1], and the Shostak Theorem Prover [7]. In addition, it would be most helpful if the reader is practiced in programming, hopefully in the PASCAL computer language and has at least studied program specifications and preferably written some. Experiences in proving theorems either using a mechanical theorem prover or by hand is also useful.

This document explains the differences between standard PASCAL and the language handled by this system. This consists mostly of restrictions to the standard language definition, the only extensions or modifications being the addition of specifications to the code and the change requiring the reference to a function of no arguments to have empty parentheses (this was done to remove a very irritating syntactic anomaly from the language). Even the ridiculous expression precedence rules of PASCAL have been scrupulously followed. This means that different operator precedence is in effect inside the specifications of the code. Other than these changes, it is hoped that any program which parses and checks in this system will compile without error.

The syntax and semantics of HDM-SPECIAL has been modified, subseted, and extended as necessary to support code proofs and to allow formal manipulation. This is detailed in sections 2 and 3. Examples of PASCAL programs and their specifications are shown in appendix A.

Section 3 explains the detailed theory of verification in this system and how verification conditions are generated. This includes explanations of the meaning and use of each part of HDM-SPECIAL for code proofs and the method of generating verification conditions.

The verification conditions are proven using the Shostak Theorem Prover which requires a certain form describing the context and syntax of the mathematical formulae. How the transformation is done and the resulting form is described in section 4.

Finally section 5 describes how to use the system.

2. Language Definitions

The actual grammar of the subset of PASCAL handled is shown in appendix B. The form of the grammar is a slightly nonstandard BNF form. Nonterminals in the grammar are represented as lowercase words, terminal symbols are delineated by double quote marks. the grammar consists of a list of productions, each production consists of a nonterminal being defined, the symbol '::=' and a list of alternative definitions, separated by a '|'. Each alternative is a sequence (possibly empty) of terminals and nonterminals with repetitions of sequences denoted by a sequence enclosed in '{' and '}'. The list of alternatives is terminated by a ';'. Hence a PASCAL procedure call might be written as:

```
proc.call ::= id "(" exp {"," exp} ")" ;
```

The semantics of the PASCAL constructs are the same as the usual semantics except for some restrictions. These restrictions are motivated from two different sources. The first set of restrictions are necessary to allow for a formal definition of the semantics of PASCAL. Such things as floating point arithmetic cannot be axiomatized and must be removed from the language. Most of the restrictions are reflected in the grammar definition and the rest are the usual restrictions imposed on the language for verification purposes. The other restrictions to the language are restrictions on where variables may be referenced or modified and what routines may be called where. These restrictions check that the PASCAL implementation corresponds to the structure of the HDM specification. These two sets of restrictions are checked separately.

The HDM-SPECIAL language grammar is given later. The syntax corresponds closely to handbook HDM but has some changes as well as omissions. The intended semantics of HDM-SPECIAL is described in the next section dealing with verification condition generation. 3. Verification Condition Generation

There are three main semantic aspects of a PASCAL program to be dealt with during verification condition generation. These are the control flow of the program, the change of state caused by operations, and the satisfaction of certain necessary conditions for proper execution and termination of the program. These are very different aspects of the program and are described and handled separately.

The flow of control through a PASCAL program (or the path analysis) is solely dependent on the semantics of PASCAL statements which change the flow and are not dependent on the program specifications. This control flow is described using weakest precondition Hoare rules and manipulated using the MetaVCG [5]. The Hoare rules used by the system are shown later. The action of the MetaVCG is described in the referenced paper and produces verification conditions through a simple pattern matching and substitution algorithm which uses only a PASCAL program segment and the Hoare rules. the verification conditions which result from applying the Hoare rules to an example PASCAL program are shown later.

Various conditions are necessary to ensure the absence of execution time errors and proper execution. Possible execution time errors include overflow, underflow, array index out of bounds, and assignment out of range. Proper execution consists mainly of proving the preconditions to functions and procedures. These conditions are referred to in functional terms throughout the Hoare rules. Each function is expanded out into a boolean expression before the verification condition is proven.

The state during the execution of a program is specified and changed as indicated by and according to the HDM model. Only two PASCAL statements are allowed to change the state, the assignment and procedure call statements. The procedure call is the main point during verification condition generation where the semantics of PASCAL and SPECIAL interact. There are references in the Hoare rules to functions which capture these semantics. These functions are listed at the end of the chapter.

At this point, it is relevant to discuss the semantics of a SPECIAL module specification and the correlation between specifications and code. A module definition consists of types, parameters, definitions, assertions, invariants, and functions.

The types are as described in the HDM handbook except that the only types supported are INTEGER, BOOLEAN, enumerated types, VECTOR, STRUCT, and SET. there is no DESIGNATOR type or UNION types. These types are meant to correspond to the PASCAL types INTEGER, BOOLEAN, scalar, ARRAY, and RECORD types. There is no correspondence to the SPECIAL SET type which can only be used for specification purposes. Whenever a correspondence between a PASCAL and SPECIAL declaration is necessary, the above correspondence between types is required.

Parameters come in two flavors, symbolic and functional. Although symbolic parameters are equivalent to constant functional parameters, they are distinguished because of the difference in their correspondence to the PASCAL. Symbolic parameters are considered to be constant values and if they have an implementation, then the implemented value must be kept constant. The obvious correspondence of symbolic parameters is to PASCAL CONST declarations. However, PASCAL CONSTs are only allowed to be simple scalar types so it is also allowed to have a symbolic parameter correspond to a PASCAL global variable with the restriction that the variable never be modified. Functional parameters are considered to be uninterpreted mathematical function symbols and never have an implemented correspondence. Functions which are meant to be implemented must be specified in the FUNCTIONS paragraph.

The ASSERTIONS paragraph of handbook HDM has been extended into two paragraphs, the ASSERTIONS and the INVARIANTS. The assertions are restrictions on the value space of the parameters only. Only constant values of SPECIAL including the symbolic and functional parameters of the module can be referenced. Symbolic parameters need not be restricted in the assertions since their value can be restricted in the mapping. However, there is little point in failing to restrict a functional parameter since without the restriction, the function can

THIS PAGE INTENTIONALLY LEFT BLANK

have any value and little can be proven about it. In code proofs, assertions can be assumed to be always true since if the assertions are initially true and the values are constant, then they must be always true. Assertions about implemented constants must be shown to be satisfied by the actual implemented value. Hence such assertions are guaranteed to be consistent.

It should be noted with some caution that assertions about uninterpreted symbolic parameters or parameter functions can be inconsistent in such a way as to render the proof trivial. Care must be taken to assure that there exists a mathematical function which satisfies all of the constraints. Mechanical assistance in this process is not currently available.

INVARIANTS are constraints on the values allowed to be taken on by the state functions (VFUNs). This might include value range restrictions or constraints on the relative values of a set of state functions. These invariants must be true in the initial state of the state functions and must be proved to be true after each invocation of a state changing operation of the module. This is done by assuming the invariants to be true in the state before the invocation of the operation and assuming the effects of the operation imply the invariants in the post state. To help in the proof of the code, the invariants can be assumed to be true in the state at the beginning of the code. The invariants of the lower machine can be assumed to be true in all states of the upper machine.

The FUNCTIONS paragraph is used to specify the state and operation functions. The state functions capture the state of the module and intuitively correspond to the state of the machine during execution. This correspondence is never actually specified but for proof purposes is assumed. The operations of the module provide the only way to modify the state of the module. These operations must correspond directly to an operation in the implementation. An operation which is either not implemented or not specified cannot be proven.

A VFUN (value-function) is used to represent a state function of the module. It has optional parameters and represents a set of values depending on the state the module is in. Handbook HDM allows VFUNs to be either a state function or an operation and distinguishes between HIDDEN and VISIBLE, and between PRIMITIVE and DERIVED. As will be described later, all aspects of the state are HIDDEN. For the sake of simplicity, all VFUNs are PRIMITIVE with OVFUNs being used in place of DERIVED VFUNs. This means that all VFUNs are state functions. Besides formal parameters, VFUNs have an optional INITIALLY clause which is used to specify the initial value of the VFUN. The initialization of the implementation state is described later. These INITIALLYs are used to prove the INVARIANTS as described above.

The operations of a module are specified as OFUNs (operation functions) or OVFUNs (operation-value functions). They differ only in that an OVFUN returns a value while an OFUN does not. There are three clauses of an operation, the preconditions, the exception conditions, and the postconditions. The preconditions are the ASSERTIONS and are constraints on the allowed values of the input parameters and the state of the module. The EXCEPTIONS are a set of conditions under which the operation is aborted during execution. The postconditions are the EFFECTS and describe the effect the operation has on the state of the module.

The meaning of the operation specification is most easily described in terms of its correlation with the corresponding PASCAL operation. An operation in PASCAL is represented as a PROCEDURE or a FUNCTION. Usually, an operation is implemented as a PROCEDURE though some simple operations can be FUNCTIONS. there is a strict correspondence required between the specification and the code in order for the proof process to work. The correspondence in the header of an operation is a name correspondence. The name of the OFUN or OVFUN must be the same as the name of the PROCEDURE or FUNCTION. The input parameters must agree in number, name and type. The output parameter, if any, must agree in name and in type. The way a value is returned in an implementation depends on whether a FUNCTION or PROCEDURE is used. The value of a PROCEDURE is returned through an additional VAR parameter of the same name and type as the returned symbol in the OVFUN. In the implementation, the value is returned through this parameter. In PASCAL, the type of the returned value of a FUNCTION given in the header must be the same type as the returned symbol in the OVFUN. The value to be returned is indicated by an assignment to the name of the FUNCTION as in normal PASCAL. Because of restrictions inherent in the proof process, a PASCAL FUNCTION is only allowed as the implementation for an OVFUN under very restrictive circumstances. Under some assumptions, an implemented expression can be treated as a purely mathematical expression which is very convenient for proof purposes. Some of these assumptions are checked during verification, such as the absence of errors like overflow. Some assumptions such as the totality of functions and the commutivity of certain mathematical operations are ensured through restrictions on FUNCTIONS.

As for the restrictions under which a FUNCTION can be the implementation of an OVFUN: first, PASCAL only allows a FUNCTION to return a simple scalar type, therefore the OVFUN being implemented must return such a simple type. Functions are not allowed to have side effects (and therefore cannot change the state of the module) for they could possibly invalidate the assumptions of the commutivity of certain mathematical operations. To keep them total, OVFUNs having a FUNCTION implementation cannot have exceptions and must therefore always return a value.

The ASSERTIONS of an operation is a set of preconditions which must be satisfied at the invocation of the operation. These preconditions must be proven at every invocation of this operation to assure that the partial function which the operation implements has a computable value. These preconditions are assumed to be true when proving the operation.

The EXCEPTIONS section allow the specification of abnormal returns from the operation. This usually occurs because the current state of the module prevents the completion of the operation. This differs from the assertions in that preconditions are proved to be true while exceptions are proved to be handled correctly during execution. The preferred way for the programming language to handle the exception condition is by an automatic change in the flow of control as in ADA. PASCAL does not support the handling of exceptions so the actual use appears to be more like a multi-return mechanism. The EXCEPTIONS section consists of a list of exception conditions, each condition having three parts, the name of the exception, the condition under which the exception is raised, and a postcondition describing the change of state associated with the exception return (note that this can include a specification of the returned value). The allowance for a change of state on an exception is a relaxation of the constraints on conditions described in the HDM Handbook [6, 8, 4]. This postcondition is optional, and if missing, indicates that no state change occurred.

The method for associating the specified exceptions with the implemented program is rather arbitrary and unesthetic due to the complete lack of any facility in PASCAL. The approach chosen is meant to be flexible, simple, and not dependent on very much additional proof mechanism. To indicate and handle exceptions, it is necessary to communicate to the calling environment which, if any, exception has been raised. This is done through a special global variable EXC which must be declared of the correct type in the implementation. If the operation has exceptions, then the variable EXC is set to either NORMAL_RETURN in the case where there is no exception raised or to the name of the raised exception, as indicated in the specification of the exception conditions. Therefore, the variable EXC must be declared as an scalar type consisting of the name NORMAL_RETURN and all of the exception names in any implemented operation. On return from an operation which may have raised an exception, the program may either test the EXC variable or may assume that no exception was raised. In the latter case, it will be necessary to prove that no exception occurred in order to benefit from the effects of the operation in the proof. The actual postcondition which is created during the verification condition generation process is a combination of the EXCEPTIONS and EFFECTS sections and is described immediately below.

The postconditions of an operation is a specification of the effects the operation has on the state of the module. For the normal return this is captured in the EFFECTS section. The usual and most usable form is to specify the new state of each state function in terms of the old state function, universally quantified over the formal parameters. The actual postcondition is a combination of the exceptions and the effects. Each set of effects, the normal ones and each of the exception effects are guarded by the value of the special variable EXC in the form of an implication. For each possible change of state, there is assumed a clause of the form "EXC = name => effect". The values that EXC might take on under various conditions is specified in an IF expression. In the order that the EXCEPTIONS are listed (order is important), each exception condition is the conditional expression of an IF with the then part specifying the resulting value of EXC. the final else specifies that "EXC = NORMAL_RETURN". Hence the IF expression looks like:

IF cond1 THEN EXC = name1 ELSE IF cond2 THEN EXC = name2 ... ELSE EXC = NORMAL_RETURN

If the normal return effects are necessary in a proof, they can be extracted from the implication described above by proving that EXC = NORMAL_RETURN. This can be done by either testing for the equality in the code or by proving that none of the exception conditions could have occurred thereby reducing the above IF expression to the desired equality. Thus the code can efficiently ignore the possibility of an exception if it can be proven to be impossible.

Throughout these specifications of the change of state to VFUNs, it is implicit that no specification of the new value of a VFUN implies that no change occurred. This is supported by automatically creating the expressions which explicitly state this.

A program is verified by proving that the accumulated state changes from each procedure call implies the desired change of state for the entire routine. Each procedure call describes its associated state change by specifying the new state of each state function in terms of the state just before the procedure call. Hence a series of procedure calls describes a progression of ever new states which are entered as a result of each procedure call. Intermediate and final assertions for the code still speak in terms of the new state and the old state. During execution of the code, many states are entered however. There is therefore a concept of the current state in the verification condition generation process. The effects of an operation are considered to be universally quantified over the state and reference the current and previous state. When a procedure call is encountered, the current state is considered changed to a new state and the effects of the operation are instantiated to the current and previous states where all quoted references to a state function are references to the current state and all unquoted references are to the previous state.

The quoted or unquoted reference to a state function is only sufficient to distinguish two states while many states may be encountered during a routine. For this reason, the notation for the state of a VFUN is changed during the verification condition generation process. The state of a VFUN is indicated by extra parameters to the VFUN. These extra parameters indicate the kind of state and the current "state" of the state. There are different states for each lower module and a different state for the path which starts at the beginning of a program. When a state change occurs, the state is modified by embedding it in a call to the function NEXT. Hence if the state is "NEXT (STACK_MOD.STATE)" then the new state would be "NEXT (NEXT (STACK_MOD.STATE))". The state parameters which each VFUN has depends on the module containing that VFUN. VFUNs of the lower modules have one state, the state of that module. VFUNs of the upper module have a set of states, one for each lower module. This is because the state of the upper module is completely dependent on the state of the lower modules. Also, since definitions are allowed to reference VFUNs, they also have state parameters. 4. Transformation to STP

The verification system produces formulae which if theorems, demonstrate that the code is consistent with the specifications. These formulae may be inspected by the user and accepted as theorems, or they may be given to a mechanical theorem prover for proof. An interface has been provided to the Shostak Theorem Prover (STP) which uses a decision procedure for linear arithmetic on ground formulae. The theorem prover will attempt to produce a proof automatically, but it is usually necessary for the user to help the theorem prover with various instantiations and axioms. The following discussion will presume some understanding of the syntax and method of STP.

The complete environment of a verification condition must be passed to STP. The LEF command of STP assumes the file to be a list of STP commands such as DA (declare axiom), DF (define formula), PR (prove). The file is passed through once, so objects must be declared before they are referenced. Hence the verification system produces a file of declarations for the theorem prover of each object which is referenced directly or indirectly by the verification conditions to be proven.

The objects to be declared fall into categories: types, variables, constants (functions of zero or more arguments), definitions, axioms, formulae, and calls to prove.

Types are declared with the STP command DT. The types are the primitive types of the theorem prover: INTEGER, BOOLEAN, STATE, and any scalar types. INTEGER and BOOLEAN are already primitive types in STP (BOOLEAN being called BOOL). The scalar types are defined as primitive types and axioms given about the scalar names of the type.

The variables of STP are typed names which appear as "free" variables in the declarations and formulae and appear in DSV declarations. For example, the formal names of a definition are "free" variables, as are universally quantified variables. All of the names which appear in such a position are declared as STP variables.

Constant functions are those objects which have a fixed, though possibly unknown value and are declared in a DS. The local variables of the vcgened routine are constant functions of no arguments. The VFUNs of the lower modules are functions with state arguments. Also included are recursive definitions or definitions with free variables, as described below.

Definitions include the special definitions found in the modules and map. PASCAL functions can be considered to have a definition if the EFFECTS of the corresponding OVFUN is of the form "v = def". Upper VFUNs are defined in terms of the mappings. All of these SPECIAL definitions are translated into STP definitions (DD) unless they violate one of the constraints on STP definitions. Such restrictions include no recursive or mutually recursive definitions and no free variables in definitions. When a SPECIAL definition cannot be expressed as an STP definition, it is declared as a constant function (DS) and the definition expressed as an axiom.

Axioms appear as a DA in STP and arise from a variety of sources in the verification system. Axioms include the assertions of the modules and the map, the invariants of the lower modules, and the hypotheses of the verification condition to be proven. Also included are the bodies of SPECIAL definitions which have not been translated into an STP definition.

Formulae are declared with a DF and are the conclusions of verification conditions to be proven. They are proven using the PR command.

The STP file produced for the verification conditions of PUSH are shown later on in the chapter.

REFERENCES

[1]

Robert Boyer and J Strother Moore.
A Computational Logic.
Academic Press, 1979.

[2]

R.W. Floyd.
Assigning meaning to programs.
Mathematical Aspects to Computer Science 19:19-32, 1968.

[3]

K. Jensen and N. Wirth.
User Manual and Report.
Springer-Verlag, 1974.

[4]

K. Levitt, L. Robinson, and B. Silverberg.
HDM Handbook,Volume III: A Detailed Example in the Use of HDM.
CSL Report 95 for Project 4828, SRI International, June, 1979.

[5]

M. Moriconi and R. Schwartz.
Automatic Construction of Verification Condition Generators from Hoare Logics.
Technical Report CSL-125, SRI International, April, 1981.
Presented at the 8th International Colloquium on Automata, Languages, and Programming, Haifa, Israel and Springer-Verlag Lecture Notes in Computer Science, July, 1981.

[6]

L.Robinson.
HDM Handbook, Volume 1: The Foundations of HDM.
CSL Report 93 for Project 4828, SRI International, June, 1979.

[7]

R. Shostak, R. Schwartz and P.M. Melliar-Smith.
STP: A Mechanized Logic for Specification and Verification.
1981.
Submitted to Working Conference on the Formal Description of Programming Concepts.

[8]

B.Silverberg, L.Robinson, and K. Levitt.
HDM Handbook, Volume II: The Languages and Tools of HDM.
CSL Report 94 for Project 4828, SRI International, June, 1979.

CHAPTER 12

HDM-PASCAL CODE VERIFICATION SYSTEM – USERS MANUAL

HDM/Pascal Verification System User's Manual

Dwight Hare

The HDM/Pascal verification system is a tool for proving the correctness of programs written in Pascal and specified in the Hierarchical Development Methodology (HDM). This document assumes an understanding of Pascal, HDM, program verification, and the STP system.

The steps toward verification which this tool provides is parsing programs and specifications, checking the static semantics, and generating verification conditions. Some support functions are provided such as maintaining a database, status management, and editing.

The system runs under the TOPS-20 and TENEX operating systems and is written in INTERLISP. However, no knowledge is assumed of these operating systems or of INTERLISP. The system requires three executable files, HDMVCG, PARSE, and STP. Optionally, the editor EMACS should be on the system in order for the editor to work. The file HDMVCG is invoked to run the system. It uses the files PARSE and STP as lower forks to perform the functions of parsing and proving.

When the system is invoked, the user is at command level. The commands which can be executed at command level are described below. The command scanner accepts input a character at a time and beeps if any character is not legal at that point. A question mark is accepted almost anywhere and responds with all of the available options. When a part of a command or name is recognized, the unique part is filled out automatically. Even though a part has been recognized, the user is allowed to keep typing the characters of the command or name. This is terminated when a space or carriage return is typed. Most commands have several parts to them, each separated by explanatory material in parentheses. An example command with the parts typed by the user in UPPER case or user typed blanks as underscores:

```
   PArse_(language) Pascal_(from file) STACK.PAS_
```

The command can be terminated by a blank or a carriage return. The entry of a command name or a subcommand name can be aborted with <delete> or <rubout> (ascii code 177 octal) whereupon the current subcommand will be reprompted. Commands can be aborted completely by typing control-D. This returns to the command level either while entering a command or while the command is executing. The system is not entirely safe from problems arising from user aborts except during command input. Such aborts should be used sparingly. When the user is prompted for a file name, full file recognition is supported with <escape> and control-F. Wildcard selections are not acceptable.

The system maintains a database of global objects which can be referenced in commands. These global objects include the Pascal global constants, types, variables, routines, and the Special modules, maps, parameters, types, definitions, vfuns, ofuns, and ovfuns. Objects can be added to the database at any time with the "Parse" command or the "Edit" command (by editing an object and changing the name). Redefining an existing entry will have the effect of removing any VCs which have been generated but any other objects which depend on the changed object must be reprocessed at the responsibility of the user.

The system uses temporary scratch files to communicate between the subforks and to hold the database. The database and parse temporary files are opened in the user's directory and will be automatically deleted in time. The files used to communicate events between the system and STP are not temporary files so that proofs can be reattempted at the user's leisure. These files should be manually deleted when they are no longer needed.

Any problems with this system should be reported to:

Dwight Hare
Arpanet address: DHARE@SRI-CSL
EL395
SRI International
333 Ravenswood Ave.
Menlo Park, Cal. 94025

Command Summary

On-line documentation can be gotten through the ? and HELP commands. A question mark can be typed during the completion of any command except during file name input. All of the available options are given in response. The help command is a top level command and gives a short description of each top level system command. The help command takes one argument, the name of a system command.

The main system commands are for parsing in code and specification files, checking the input, generating verification conditions, and proving the VCs. The command for parsing files is PARSE. It takes two subcommands, the language being parsed, either PASCAL or SPECIAL and a file to parse from. The effect of this command is to add the objects contained in the file to the database.

The CHECK command takes the name of an object and a property to check, either the PASCAL, the SPECIAL, the HDM, or the CORRESPONDENCE. The Pascal and Special checks are that the semantics of the object do not violate the Pascal or Special semantics. This involves mostly type checking and other constraints imposed in the language definition. These semantics are defined in a pseudo denotational semantic way and explained in another document. When an error is discovered, it is presented as a violation of an assertion of the semantic definition. At present, the user interface is crude and it takes some experience to discover the source of the error. During this phase, information is gathered into a symbol table which is used by the other two checks and by the vcgen process. The HDM check is concerned with discovering violations in the hierarchy. This includes checking that the hierarchy is a proper tree and that the pascal program doesn't violate the hierarchy. The correspondence check is between the Pascal and Special forms of an object. An object which has a Special representation as a PARAMETER can only have a corresponding Pascal representation as a CONST. A Special OFUN or OVFUN corresponds to a Pascal FUNCTION or PROCEDURE. Special TYPES correspond to equivalently defined Pascal TYPES.

After the Pascal and Special have been parsed and checked, verification conditions can be generated. VCs are generated for operations or modules. An operation is a Special OFUN or OVFUN and a Pascal FUNCTION or PROCEDURE. The command to generate verification conditions is VCGEN. The vcgen process normally has three steps, metavcg, postprocessing, and simplifying. These steps can be done separately if desired by invoking the METAVCG, POSTPROCESS, and SIMPLIFY commands. The vcgen of an operation consists of taking the Pascal routine through the Pascal Hoare rules with the metavcg. This serves to transform the code into Special formulas which can be further processed without regard to the initial Pascal form. The postprocess phase makes the state and state changes explicit and expands functions referenced in the Hoare rules. The simplify phase performs relatively trivial localized simplifying transformations to the VCs to make them more readable. The VC generation for a module does not have a metavcg step for there is no Pascal code involved. The module invariants and assertions must be proved to be true in the initial state and after each operation of the module.

After VCs have been generated, they can be proven in STP through the PROVE command. The prove command takes the name of an object for which VCs have been generated. This command generates a file of events suitable for STP and starts up the STP program on the file. Control is returned to the system by the LOGOUT function performed in STP. This is usually done automatically unless a lisp abort is done during proof. No record is kept currently of any proofs done.

The status of the system can be obtained through two commands. STATUS shows the current status of any or all objects in the system. It takes two arguments, the name of an object and the aspect being inquired about. This includes what has been parsed and checked and whether VCs have been generated. The SHOW command can be used to print the Special or Pascal forms of the object and the VCs. It takes the name of the object and the form to be printed.

The database can be modified (other than by parsing in files) by some database commands. The database can be cleared and put into the initial state with the FLUSH command which takes no arguments. The current state of the system can be saved on a file with the SAVE command. This command requests if the database is to be flushed after saving and prompts for a filename to save the state on. If the database is to be flushed, the current disk file which holds the database becomes the save file. Hence saving and flushing is much more efficient than saving and maintaining the state. This would be the expected mode when the system is saved and exited to be continued another time. The RESTORE command restores the state of a system previously saved with the save command. It asks if the save file can be overwritten and prompts for the name of the save file. If the save file can be overwritten, then it can become the current database which is much quicker and more efficient than not allowing the save file to be modified. This would normally be done when it is desired to continue work on a previously saved system. The EDIT command can be used to edit the Pascal or Special form of an object in the database. This is done by pretty printing the form into an emacs buffer and dropping the user down into emacs. The normal emacs commands work except the <control-T> command which is modified to allow commands to be transmitted back to the system. The command <control-T> <control-T> has the effect of a single <control-T> in emacs. The command <control-T><control-Z> returns to the system without any action being taken. The command <control-T><control-E> exits emacs and calls the parser on the edited buffer with the parsing starting from where the cursor was left. It is usual that the entire buffer is to be parsed, so the user should be careful to put the cursor at the top of the buffer before exiting. If emacs does not exist on the site then the edit command will not work properly.

A session in the system can be logged with the LOG command. It takes the name of a file to log on. A log file can be closed with the CLOSE command.

Various operating system functions can be performed from inside of the system. The QUIT command exits the system and returns to the exec level. The EXEC or PUSH command forks off an inferior exec if the site has an EXEC program on the <SYSTEM> directory. The DIRECTORY command takes a file specification (without any escape conventions but with wild cards) and prints the directory listing. The CONNECT command takes the name of a directory and attempts to connect to that directory requesting a password if necessary. The DAYTIME command shows the current day and time. The SYSTAT command does an exec systat if it exists. The TYPE command takes a file name and types that file to the terminal.

The ; command allows comments to be typed for links or logs. It ignores everything typed up to a carriage return.

Parts Unimplemented

Currently the following are not implemented. The datatypes ARRAY, RECORD, and SET are not handled properly. They should not be used. The HDM and Correspondence checks are not done and any errors in the code or specifications will cause unusual and incorrect behaviour during vcgen. The special STP symbols such as PLUS, IMPLIES, BOOL, etc are not checked for and any program which uses these names will cause errors during proof time.

CHAPTER 13

HDM-PASCAL CODE VERIFICATION SYSTEM – SIMPLE EXAMPLES

THIS PAGE INTENTIONALLY LEFT BLANK

```
MODULE stack

PARAMETERS INTEGER max_stack_size;

ASSERTIONS max_stack_size > 0 AND max_stack_size < maxint;

INVARIANTS ptr() >= 0 AND ptr() <= max_stack_size;

FUNCTIONS

        VFUN stack_val (INTEGER arg) -> INTEGER v;

        VFUN ptr () -> INTEGER v;
          initially v = 0;

        OFUN push (INTEGER v);
          EXCEPTIONS
            full_stack : ptr () >= max_stack_size-1;
          EFFECTS
            'ptr () = ptr () + 1;
            FORALL INTEGER j :
                'stack_val(j) = IF j = ptr () + 1 THEN v
                                ELSE stack_val(j) END_IF END_FORALL;

        OVFUN pop () -> INTEGER v;
          EXCEPTIONS
             empty_stack : ptr() = 0;
          EFFECTS
             v = stack_val(ptr());
             'ptr() = ptr() - 1;

END_MODULE

MODULE array_mod

  PARAMETERS INTEGER Maxarraysize;

  ASSERTIONS Maxarraysize > 0 AND Maxarraysize < maxint;

  FUNCTIONS

    VFUN read (INTEGER arg) -> INTEGER v;

    OFUN write_op (INTEGER arg, val);
        ASSERTIONS
          arg >= 0 AND arg < Maxarraysize;
        EFFECTS
          FORALL INTEGER j :
                'read (j) = IF j = arg THEN val ELSE read(j) END_IF END_FORALL;

    OVFUN read_op (INTEGER arg) -> INTEGER v;
        ASSERTIONS
          arg >= 0 AND arg < Maxarraysize;
        EFFECTS
          v = read(arg);

END_MODULE

MAP stack TO array_mod;
```

```
    MAPPINGS
        ptr () : read(0);
        stack_val (INTEGER arg) : IF arg > 0 THEN read(arg) ELSE 0 END_IF;
        max_stack_size = Maxarraysize - 1;

END_MAP
```

```
CONST
    max_stack_size = 10;
    Maxarraysize = 11;

TYPE exc_kinds = (normal_return, full_stack, empty_stack);

VAR exc : exc_kinds;

FUNCTION read_op (arg : INTEGER) : INTEGER;
BEGIN END;

PROCEDURE write_op (arg, val : INTEGER);
BEGIN END;

PROCEDURE stack_init;
BEGIN
  write_op(0, 0)
END;

PROCEDURE push (v : INTEGER);
VAR pointer : INTEGER;
BEGIN
  pointer := read_op(0);
  IF pointer >= max_stack_size-1
        THEN exc := full_stack
        ELSE
          BEGIN
            write_op(0, pointer+1);
            write_op(pointer+1, v);
            exc := normal_return
          END
END;

PROCEDURE pop (VAR v : INTEGER);
VAR pointer : INTEGER;
BEGIN
  pointer := read_op(0);
  IF pointer = 0 THEN exc := empty_stack
    ELSE
      BEGIN
        v := read_op(pointer);
        write_op(0, pointer - 1);
        exc := normal_return
      END
END;
```

```
MODULE list

  TYPES list_ptr : INTEGER;

  PARAMETERS INTEGER max_list_ptrs, max_atoms;

  DEFINITIONS

    BOOLEAN atomp (list_ptr x) IS
        x >= 0 AND x < max_atoms;

    BOOLEAN iscell (list_ptr x) IS
        EXISTS list_ptr x1 : EXISTS list_ptr x2 :
                cell (x1, x2, x) END_EXISTS END_EXISTS;

  ASSERTIONS max_list_ptrs >= max_atoms AND max_list_ptrs < MAXINT
             AND max_atoms >= 0;

  INVARIANTS

    FORALL list_ptr x1 : FORALL list_ptr x2 : FORALL list_ptr x :
      FORALL list_ptr y1 : FORALL list_ptr y2 :
        cell (x1, y1, x) AND cell (x2, y2, x) => x1 = x2 AND y1 = y2
            END_FORALL END_FORALL END_FORALL END_FORALL END_FORALL;

  FUNCTIONS

    VFUN cell (list_ptr x1, x2, x) -> BOOLEAN b;
      INITIALLY b = FALSE;

    VFUN num_of_cells () -> INTEGER v;
      INITIALLY v = 0;

    OVFUN lcons (list_ptr x1, x2) -> list_ptr x;
      EXCEPTIONS
        storage_full : num_of_cells() = max_list_ptrs;
      ASSERTIONS
        atomp (x1) OR iscell (x1);
        atomp (x2) OR iscell (x2);
      EFFECTS
        'cell (x1, x2, x);
        FORALL list_ptr z1 : FORALL list_ptr z2 : FORALL list_ptr z :
                    (x ~= z => 'cell (z1, z2, z) = cell (z1, z2, z))
                AND (z1 ~= x1 OR z2 ~= x2 => NOT 'cell (z1, z2, x))
                AND NOT cell(z1, z2, x)
           END_FORALL END_FORALL END_FORALL;

    OVFUN lcar (list_ptr x) -> list_ptr x1;
      ASSERTIONS
        iscell (x);
      EFFECTS
        EXISTS list_ptr z2 : cell (x1, z2, x) END_EXISTS;

    OVFUN lcdr (list_ptr x) -> list_ptr x2;
      ASSERTIONS
        iscell (x);
      EFFECTS
        EXISTS list_ptr z1 : cell (z1, x2, x) END_EXISTS;
    OVFUN consp (list_ptr x) -> BOOLEAN b;
      EFFECTS b = iscell (x);

END_MODULE



MODULE ulist

  TYPES ulist_ptr : INTEGER;

  PARAMETERS INTEGER max_ulist_ptrs, max_uatoms;

  DEFINITIONS

    BOOLEAN uatomp (ulist_ptr x) IS
        x >= 0 AND x < max_uatoms;

    BOOLEAN isucell (ulist_ptr x) IS
        EXISTS ulist_ptr z1 : EXISTS ulist_ptr z2 :
           ucell (z1, z2) = x END_EXISTS END_EXISTS;

  ASSERTIONS max_ulist_ptrs >= max_uatoms AND max_ulist_ptrs < MAXINT
             AND max_uatoms >= 0;

  INVARIANTS

    FORALL ulist_ptr x1 : FORALL ulist_ptr x2 :
        FORALL ulist_ptr y1 : FORALL ulist_ptr y2 :
             ucell (x1, y1) = ucell (x2, y2) => x1 = x2 AND y1 = y2
        END_FORALL END_FORALL END_FORALL END_FORALL;
    num_of_ucells() >= 0 AND num_of_ucells() <= max_ulist_ptrs;
       num_of_ucells() = 0
    => FORALL INTEGER x1 : FORALL INTEGER x2 : ucell(x1, x2) = 0
         END_FORALL END_FORALL;

  FUNCTIONS

    VFUN ucell (ulist_ptr x1, x2) -> ulist_ptr x;
        INITIALLY x = 0;

    VFUN num_of_ucells () -> INTEGER v;
        INITIALLY v = 0;

    OVFUN ucons (ulist_ptr x1, x2) -> ulist_ptr x;
      EXCEPTIONS
        ustorage_full : num_of_cells() = max_ulist_ptrs;
      ASSERTIONS
        uatomp (x1) OR isucell (x1);
        uatomp (x2) OR isucell (x2);
      EFFECTS
        FORALL ulist_ptr z1 : FORALL ulist_ptr z2 :
          IF ucell(x1, x2) = 0 THEN
              'ucell (z1, z2) =
                  IF z1 = x1 AND z2 = x2 THEN x
                  ELSE ucell (z1, z2) END_IF AND
              ucell(z1, z2) ~= x
          ELSE 'ucell (z1, z2) = ucell (z1, z2) AND
               ucell (x1, x2) = x END_IF
          END_FORALL END_FORALL;

    OVFUN ucar (ulist_ptr x) -> ulist_ptr x1;
      ASSERTIONS
        isucell (x);
      EFFECTS
        EXISTS ulist_ptr z2 : ucell (x1, z2) = x END_EXISTS;

    OVFUN ucdr (ulist_ptr x) -> ulist_ptr x2;
      ASSERTIONS
        isucell (x);
      EFFECTS
        EXISTS ulist_ptr z1 : ucell (z1, x2) = x END_EXISTS;

    OVFUN uconsp (ulist_ptr x) -> BOOLEAN b;
      EFFECTS b = isucell (x);

END_MODULE



MODULE search

  TYPES table : INTEGER; ptr : INTEGER;

  PARAMETERS table primary_table;
             INTEGER max_tables, table_size;

  INVARIANTS
    tablep (primary_table);
    FORALL integer tbl : FORALL integer key :
                NOT tablep(tbl) => get (key, tbl) = 0
        END_FORALL END_FORALL;

  FUNCTIONS

    VFUN get (INTEGER key; table tbl) -> INTEGER val;
      INITIALLY val = 0;

    VFUN tablep (table tbl) -> BOOLEAN b;
      INITIALLY b = (tbl = primary_table);

    OVFUN newtable () -> table tbl;
      EXCEPTIONS
        no_more_tables : num_of_tables() = max_tables;
      EFFECTS
        FORALL table tbli :
           'tablep (tbli) = IF tbli = tbl THEN TRUE ELSE tablep (tbli)
         END_IF END_FORALL;
        NOT tablep(tbl);

    OFUN save (INTEGER key; ptr value; table tbl);
      EXCEPTIONS
        table_full : num_of_entries(tbl) = max_table_entries;
      ASSERTIONS
        tablep(tbl) AND get (key, tbl) = 0;
      EFFECTS
        FORALL INTEGER i : FORALL table tbli :
          'get (i, tbli) =
             IF i = key AND tbli = tbl THEN value
                ELSE get (i, tbli) END_IF
        END_FORALL END_FORALL;

    OVFUN getop (INTEGER key; table tbl) -> ptr value;
      EXCEPTIONS
        not_found : get(key, tbl) = 0;
      ASSERTIONS tablep (tbl);
      EFFECTS value =  get (key, tbl);

END_MODULE



MAP ulist TO search, list;

  ASSERTIONS

    FORALL INTEGER x : FORALL table tbl :
            tablep(get (x, primary_table))
        AND get (x, primary_table) ~= primary_table
        AND (get (x, primary_table) ~= 0 => atomp (x) OR iscell (x))
        AND (   get (x, tbl) ~= 0 AND tbl ~= primary_table
             => iscell (get (x, primary_table)) AND (atomp (x) OR iscell (x)))
        END_FORALL END_FORALL;

  MAPPINGS

    ucell (ulist_ptr x1, x2, x) :
         EXISTS table tbli :
               get (x2, tbli, primary_table) AND get (x1, x, tbli) END_EXISTS
           AND cell (x1, x2, x);

END_MAP


MODULE search_space

  PARAMETERS INTEGER max_search_size;

  ASSERTIONS max_search_size > 0;

  FUNCTIONS

    VFUN search_read (INTEGER arg) -> INTEGER v;

    OFUN search_write (INTEGER arg, val);
        ASSERTIONS arg >= 0 AND arg < max_search_size;
        EFFECTS
          FORALL INTEGER i :
            'search_read(i) =
              IF i = arg THEN val ELSE search_read(i) END_IF END_FORALL;

    OVFUN search_readop (INTEGER arg) -> INTEGER v;
        ASSERTIONS arg >= 0 AND arg < max_search_size;
        EFFECTS
           v = search_read(arg);

END_MODULE
```

```
MODULE list_space

  PARAMETERS INTEGER max_list_size;

  ASSERTIONS max_list_size > 0;

  FUNCTIONS

    VFUN list_read (INTEGER arg) -> INTEGER v;

    OFUN list_write (INTEGER arg, val);
        ASSERTIONS arg >= 0 AND arg < max_list_size;
        EFFECTS
          FORALL INTEGER i :
            'list_read(i) =
              IF i = arg THEN val ELSE list_read(i) END_IF END_FORALL;

    OVFUN readop (INTEGER arg) -> INTEGER v;
        ASSERTIONS arg >= 0 AND arg < max_list_size;
        EFFECTS
           v = list_read(arg);

END_MODULE


MAP search TO search_space

  ASSERTIONS

    search_read (0) >= 0;
    search_read (0) <= max_tables;
    table_size * max_tables + 1 <= max_search_size

  MAPPINGS

     get (INTEGER key, val; table tbl) :
        IF tbl >= 0 AND tbl < read (0, search_space) THEN
           EXISTS INTEGER ptr :
                ptr >= tbl * (table_size * 2 + 1) + 2 and
                ptr <= 2 * read (tbl * (table_size * 2 + 1)) + 1
                       + tbl * (table_size * 2 + 1) + 2
             => read (ptr, search_space) = key AND
                read (ptr + 1, search_space) = val END_EXISTS
        ELSE FALSE END_IF;
     primary_table = 0;
     tablep (table tbl) : 0 <= tbl AND tbl < read (0, search_space);

END_MAP


MAP list TO array_mod;

   DEFINITIONS

     BOOLEAN evenp (INTEGER x) IS
        x = (x/2) * 2;
   ASSERTIONS
        evenp (read (0, list_space));
        read (0, list_space) >= 100;
        max_list_ptrs < maxarraysize;

   MAPPINGS

     cell (list_ptr x1, x2, x) :
        IF x >= read (0, list_space) OR x < 100 OR NOT evenp (x) THEN FALSE
        ELSE x1 = read (x, list_space) AND
             x2 = read (x+1, list_space) END_IF;
     list_ptr : INTEGER;

END_MAP
```

```
const max_array_size = 511;
      max_list_ptrs = 510;
      primary_table = 0;
      table_size = 25;
      max_tables = 10;

type max_array_index = 0 .. max_array_size;
     array_type = array [max_array_index] of integer;
     list_ptr = integer;
     table = integer;
     ptr = integer;
     ulist_ptr = integer;

var exc : integer;
    list_space : array_type;
    search_space : array_type;

procedure write (arg, val : integer; var a : array_type);
begin
  a[arg] := val
end;

function readop (arg : integer; var a : array_type) : integer;
begin
  readop := a[arg]
end;

procedure lcons (x1, x2 : list_ptr; var x : list_ptr);
begin
  x := readop(0, list_space);
  if (x + 2) >= max_list_ptrs then exc := 1
  else begin
        write (x, x1, list_space);
        write (x+1, x2, list_space);
        write (0, x+2, list_space)
       end
end;

procedure lcar (x : list_ptr; var x1 : list_ptr);
begin
  x1 := readop (x, list_space)
end;

procedure lcdr (x : list_ptr; var x2 : list_ptr);
begin
  x2 := readop (x + 1, list_space)
end;

function consp (x : list_ptr) : boolean;
begin
  consp := (x >= 100) and (x < readop (0, list_space)) and ((x mod 2) = 0)
end;

procedure newtable (var tbl : table);
var n : integer;
begin
  n := readop (0, search_space);
  if n >= max_tables then exc := 1
  else begin
```

```
        tbl := n;
        write (tbl * (table_size * 2 + 1) + 1, 0, search_space);
        write (0, n+1, search_space)
       end
end;

procedure save (key : integer; value : ptr; tbl : table);
var n, table_start, offset : integer;
begin
  table_start := tbl * (table_size * 2 + 1) + 1;
  n := readop (table_start, search_space);
  if n >= table_size then exc := 1
  else begin
        offset := n * 2 + 1;
        write (table_start + offset, key, search_space);
        write (table_start + offset + 1, value, search_space);
        write (table_start, n+1, search_space)
       end
end;

procedure getop (key : integer; tbl : table; var value : ptr);
var table_start, i, max : integer; v : ptr;
begin
  exc := 0;
  table_start := tbl * (table_size * 2 + 1) + 1;
  i := table_start + 1;
  max := table_start + 1 + readop (table_start, search_space) * 2;
  while key <> readop (i, search_space) and (i < max) do i := i + 2;
  if i >= max then exc := 1
    else value := readop (i+1, search_space)
end;

procedure ucons (x1, x2 : ulist_ptr; var x : ulist_ptr);
var cdr_table : ulist_ptr; new_t : table; cons_cell : list_ptr;
begin
  getop (x2, primary_table, cdr_table);
  if exc = 1 then
    begin
      newtable (new_t);
      if exc = 0 then
        begin
          save (x2, new_t, primary_table);
          if exc = 0 then
            begin
              lcons(x1, x2, cons_cell);
              if exc = 0 then
                begin
                  save (x1, cons_cell, new_t);
                  x := cons_cell
                end
            end
        end
    end
  else
    begin
      getop (x1, cdr_table, cons_cell);
      if exc = 0 then x := cons_cell
      else begin
        lcons (x1, x2, cons_cell);
        if exc = 0 then save (x1, cons_cell, cdr_table);
        x := cons_cell
        end
    end
end;

procedure ucar (x : ulist_ptr; var x1 : ulist_ptr);
begin
  lcar (x, x1)
end;

procedure ucdr (x : ulist_ptr; var x2 : ulist_ptr);
begin
  lcdr (x, x2)
end;

function uconsp (x : ulist_ptr) : boolean;
begin
  uconsp := consp (x)
end;
```

```
MODULE table

  PARAMETERS INTEGER max_entries;

  ASSERTIONS max_entries >= 0 AND max_entries <= maxint;

  INVARIANTS num_of_entries() >= 0 AND num_of_entries() <= max_entries;
             FORALL INTEGER k : not in_table(k) => assoc(k) = 0 END_FORALL;

  FUNCTIONS

    VFUN in_table (INTEGER key) -> BOOLEAN v;
        INITIALLY v = FALSE;

    VFUN assoc (INTEGER key) -> INTEGER v;
        INITIALLY v = 0;

    VFUN num_of_entries () -> INTEGER v;
        INITIALLY v = 0;

    OFUN insert_table (INTEGER key, val);
          EXCEPTIONS
             bad_key : key = 0;
             table_full : num_of_entries() = max_entries;
          EFFECTS
             FORALL INTEGER i :
               'assoc(i) = IF i = key THEN val ELSE assoc(i) END_IF END_FORALL;
             FORALL INTEGER j :
               'in_table(j) = IF j = key THEN TRUE ELSE in_table(j)
                                   END_IF END_FORALL;
             'num_of_entries() = num_of_entries() + 1;

    OVFUN table_op (INTEGER key) -> INTEGER val;

        $( Perform a table lookup, using the key )

          EXCEPTIONS $( The key not being in the table means an exception )
             bad_key : key = 0;
             key_not_there : not in_table(key);
          EFFECTS
             val = assoc(key);

    OFUN delete_table (INTEGER key);
          EFFECTS
             FORALL INTEGER i :
                'in_table(i) =
                    IF i = key THEN FALSE ELSE in_table(i) END_IF END_FORALL;
             FORALL INTEGER i :
                'assoc(i) = IF i = key THEN 0 ELSE assoc(i) END_IF END_FORALL;
             'num_of_entries() =
                IF in_table(key) THEN num_of_entries()-1
                                 ELSE num_of_entries() END_IF;

    OFUN clear_table ();
          EFFECTS
             FORALL INTEGER i: 'in_table (i) = FALSE END_FORALL;
             FORALL INTEGER i: 'assoc (i) = 0 END_FORALL;
             'num_of_entries() = 0;
```

```
END_MODULE

MAP table TO array_mod;

  PARAMETERS INTEGER lookup (INTEGER key);

  DEFINITIONS
    BOOLEAN evenp (INTEGER x) IS
        EXISTS INTEGER y : y*2 = x END_EXISTS;

    BOOLEAN in_array (INTEGER i) IS i >=0 AND i <= Maxarraysize;

    BOOLEAN in_table_def (INTEGER key) IS
        key ~= 0 AND
        EXISTS INTEGER i :
           in_array(i) AND evenp(i) AND read(i) = key END_EXISTS;

  ASSERTIONS
    Maxarraysize = max_entries*2;
    FORALL INTEGER k :
        FORALL INTEGER i :
             in_array(i) AND evenp(i) AND read(i) = k AND k ~= 0
          => lookup(k) = read(i+1) END_FORALL END_FORALL;

  MAPPINGS
    in_table (INTEGER key) : in_table_def(key);
    assoc (INTEGER key) : IF in_table_def(key) THEN lookup(key) ELSE 0 END_IF;
    num_of_entries() :
        CARDINALITY({INTEGER i | evenp(i) AND in_array(i) AND read(i) ~= 0});

END_MAP

MODULE array_mod

  PARAMETERS INTEGER Maxarraysize;

  FUNCTIONS

    VFUN read (INTEGER arg) -> INTEGER v;

    OFUN writeop (INTEGER arg, val);
        ASSERTIONS
          arg >= 0 AND arg < Maxarraysize;
        EFFECTS
          FORALL INTEGER j :
                'read (j) = IF j = arg THEN val ELSE read(j) END_IF END_FORALL;

    OVFUN readop (INTEGER arg) -> INTEGER v;
        ASSERTIONS
          arg >= 0 AND arg < Maxarraysize;
        EFFECTS
          v = read(arg);

END_MODULE
```

```
CONST maxarraysize = 100;
      max_entries = 50;

TYPE exc_kinds = (normal_return, bad_key, table_full, key_not_there);

VAR exc : exc_kinds;

FUNCTION readop (arg:INTEGER) : INTEGER;
BEGIN END;

PROCEDURE writeop (arg, val : INTEGER);
BEGIN END;

PROCEDURE insert_table (key, val : INTEGER);
  VAR index, temp, blank_pos : INTEGER;
  BEGIN
    IF key = 0 THEN exc := bad_key
    ELSE BEGIN
      index := 0;
      blank_pos := -1;
      REPEAT
        temp := readop(index);
        IF temp = 0 then blank_pos := index;
        index := index + 2
      UNTIL (temp = key) OR (index >= Maxarraysize)
        ASSERT temp = read(index-2) AND index > 1
               AND FORALL INTEGER i :
                      in_array(i) AND i <= index-2 AND evenp(i)
                   => read(i) ~= key END_FORALL
               AND IF EXISTS INTEGER j : j >= 0 AND j <= index-2
                          AND evenp(j) AND read(j) = 0 END_EXISTS
                   THEN read(blank_pos) = 0 AND evenp(blank_pos)
                        AND blank_pos >= 0 AND blank_pos <= index-2
                   ELSE blank_pos = -1 END_IF
        DECREASING maxarraysize - index;
      exc := normal_return;
      IF temp = key THEN writeop (index-1, val)
      ELSE IF blank_pos >= 0 THEN
        BEGIN
          writeop (blank_pos, key);
          writeop (blank_pos+1, val)
        END ELSE exc := table_full
    END;
  END;

PROCEDURE table_op (key : INTEGER; VAR val : INTEGER);
  VAR index : INTEGER;
  BEGIN
    IF key = 0 THEN exc := bad_key
    ELSE BEGIN
           index := 0;
           REPEAT
               val := readop (index);
               index := index + 2;
             UNTIL (val = key) OR (index >= maxarraysize)
             ASSERT val = read(index-2) AND index > 1
               AND FORALL INTEGER i : i >= 0 AND i <= index-2
                          => key ~= read(i) END_FORALL
             DECREASING maxarraysize - index;
          IF val = key THEN
             BEGIN
                 val := readop(index-1);
                 exc := normal_return
             END
          ELSE exc := key_not_there
        END
  END;

PROCEDURE delete_table (key : INTEGER);
  VAR index,temp : INTEGER;
  BEGIN
  IF key <> 0 THEN
  BEGIN
    index := 0;
    REPEAT
        temp := readop(index);
        index := index + 2
      UNTIL (temp = key) OR (index > maxarraysize)
      ASSERT key ~= 0 AND
             FORALL INTEGER i :
                        (i >= 0 AND i < index AND evenp(i) => key ~= read(i))
             END_FORALL
      DECREASING maxarraysize - index
    END;
    IF temp = key THEN writeop(index-2, 0)
  END;

PROCEDURE TABLE_INIT;
VAR n : INTEGER;
BEGIN
  FOR n := 0 TO Maxarraysize
      ASSERT FORALL INTEGER i : i >= 0 AND i <= n => read(i) = 0 END_FORALL
      DO writeop(n, 0)
END;
```

CHAPTER 14

VERIFICATION OF SIFT CODE

The SIFT Code Specifications

1. Introduction

The specification of SIFT consists of two parts, the specifications of the SIFT models and the specifications of the SIFT PASCAL program which actually implements the SIFT system. The code specifications are the last of a hierarchy of models describing the operation of the SIFT system and hence are related to the SIFT models as well as the PASCAL program. These specifications serve to link the SIFT models to the running program.

Due to the need to prove the consistency between the PASCAL program and the code specifications, the specifications are very large and detailed and closely follow the form and organization of the PASCAL code. In addition to describing each of the components of the SIFT code, the code specifications describe the assumptions of the upper SIFT models which are required to actually prove that the code will work as specified. These constraints are imposed primarily on the schedule tables.

This document assumes an understanding of the motivation and basic algorithms of SIFT. An acquaintance with the Hierarchical Development Methodology (HDM) and SPECIAL is helpful but not required. This SIFT specification was written in a variant of SPECIAL developed as part of a PASCAL code verification system. The specification is written as a series of paragraphs with names such as TYPES and PARAMETERS. Comments are enclosed in "$(...)" and serve only as informal descriptions of the formal specification. The data objects of SIFT are called VFUNs in the specification but otherwise are identical to the variables of the SIFT program.

The code specification is not a complete description of the SIFT program. The SIFT system consists of a number of processors all working concurrently in approximate synchronization. This specification considers only one of those processors and contains no description of how the processors communicate or how they maintain synchronization. Any part of the SIFT program which explicitly involves the passage of time is outside the realm of this specification.

The next sections will refer in detail to the SPECIAL specification of SIFT.

2. The Type Declarations

The TYPES paragraph at the top of the specification declare the data types used in the SIFT implementation as well as some types used purely for specification purposes. These data types are very similar in meaning to data types used in most programming languages such as PASCAL.

The data types are broken into sections for each major data component of SIFT. The first data component is the schedule table whose data type is called SCHED_ARRAY. It is an array of arrays which has a component for each processor, each configuration, and each subframe. The second component is the datafile through which communication between processors is done. The datafile type is called the DATAFILE_ARRAY and has components for each taskname and each element of the results produced by each task. The POLL_ARRAY type describes an entry for each configuration, each processor, and each task and contains a boolean value indicating whether that task is run by that processor in that configuration. The ERROR_ARRAY type describes an array containing the error count for each processor. The INPUT_ARRAY has an element array for each task.

Some data types are used for specification purposes and for internal processing in the implementation. SET_OF_INT describes a set of integers in the usual mathematical sense of a set. TASK_ARRAY is an array of integers indexed by task name. BOOL_ARRAY is an array of boolean values.

3. The Parameter Declarations

The parameters of the specification correspond to the constant values of the implementation. The actual values of these objects are not specified but instead sufficient constraints are placed on the possible values so that the proof will succeed. The parameters and their meaning are given below:

frame_size	The number of subframes in a frame
max_processors	The maximum number of processors in any configuration
my_processor	The physical number of this processor
max_activities	The maximum number of activities allowed in any subframe
max_elems	The maximum number of values any task can produce
max_tasks	The maximum number of tasks in the system
bottom_val	The special value returned when a task does not run and when no majority is found in voting
err_threshold	The number of error reports before a processor is considered faulty
vote,dummy_vote, execute	The possible activities of a subframe
reconfig	The name of the reconfiguration task
global_exec	The name of the global executive task
error_report	The name of the error reporting task
null_task	The name of the null or maintenance task
sched_table	The schedule table for each processor, configuration, and subframe, giving a set of activities to perform
poll	Determines whether a processor in a given configuration runs a particular task
inputs	Gives the names of the tasks which will produce results used by a particular task
i_c	Indicates which tasks are interactive consistency tasks
result_size	Gives the number of values returned by each task
error_i_c_tasks	The names of the error reporting interactive consistency tasks

4. The Definitions

Definitions are used as a handy way to name a concept in the specification so that it can be more easily referred to. They are equivalent to pure mathematical functions in that they take an sequence of arguments and produce a determined result. The only definition in the specifications defines the concept of a majority of a set of values. The values being voted on are found in the datafile corresponding to each processor which ran the task as indicated by the POLL. The definition the set of all processors which ran the task and compares that to the set of all processors which agree on the resulting value. If the second set is larger than half the first set, a majority value has been found.

5. The Parameter Invariants

This section of the specifications contain the constraints on the parameter values mentioned in the above section on parameters. Some of the constraints seem very obvious, but they must be made explicit in order for the mechanical verification effort to succeed. Each constraint is described below.

1) frame_size, max_processors, max_activities, max_elems, and max_tasks should all be positive values.

2) 1 <= my_processor <= max_processors
The processor number of this processor must be a legitimate processor number.

3) The activities vote, dummy_vote, and execute should be positive numbers and each should be different.

4) The tasks reconfig, global_exec, null_task, the error reporting tasks, and the error reporting interactive consistency tasks should all be different tasks, that is not equal to each other.

5) An execute which uses values must follow the vote or dummy_vote on those values.

6) There is never scheduled both a vote and a dummy_vote on the results of a task during a subframe.

7) The results of an execute are not voted on during the same subframe they are produced.

8) Reconfiguration is always the last thing done in the subframe. This is because reconfiguration completely changes the current schedule so it is not possible to continue the old schedule.

9) No vote is scheduled during the same subframe as an error report. That is because a vote might change the error count which the error reporting tasks broadcasts.

10) The result of an execute is only voted on once.

11) No task is executed more than once in any subframe.

12) The only activities scheduled are vote, dummy_vote, and schedule.

13) No other activities are scheduled after the null task is started to fill out the subframe.

14) Only one processor runs an interactive consistency task but three processors run all other tasks.

15) The global executive only takes as input itself (the previous execution) and the error reporting interactive consistency tasks.

16) The error reporting interactive consistency tasks broadcast their inputs.

17) The global executive considers a processor to be no longer working if it was not previous working or if a majority of the other processors have declared it to be bad.

6. The Specification Functions

The specification functions serve two purposes. They define the names and types of the variables which make up the state of the system and they define the operations which modify the state during execution. The state variables correspond to the global PASCAL variables of the SIFT implementation and the operations of the specification correspond to the FUNCTIONs and PROCEDUREs of the implementation. The variables are described first followed by the operations.

1) Subframe contains the current subframe number which ranges from 0 to one less than the maximum number of subframes.

2) Config contains the current configuration. This is the number of processors which are currently considered to be working.

3) Input is an array of values waiting to be input to particular tasks. They are the result of voting on the outputs of other tasks.

4) Datafile is the broadcast area where results of tasks are placed and automatically broadcast.

5) Errors is the number of errors counted for each processor due to non-agreement in voting.

6) Real_to_virt is a mapping from a real physical processor number to the processor number used in the schedule tables for this configuration.

7) Virt_to_real is a reverse mapping of real_to_virt.

The main procedure of the implementation is Dispatcher which is invoked at the start of each subframe due to a clock interrupt and which in turn invokes each of the activities of that subframe. The specifications of the dispatcher describe how each of the state variables described above is changed according to the schedule tables. The change to each state variable is described in sequence:

1) The subframe number is incremented by one indicating that another subframe has elapsed. If the end of the frame is reached, the subframe number is reset to zero.

2) The config is only changed if a reconfiguration was done during this subframe. Its new value corresponds to the number of processors reported working by the global executive.

3) Input is changed to reflect the updated values found by voting on values produced by tasks in other subframes. For each entry in the input table, if no vote was done in this subframe on the value, then the value remains unchanged. If there was a vote, the new input value corresponds to the majority value computed by the vote.

4) The entries in the datafile are updated when tasks broadcast their results. If a task did not run during this subframe then its old values from the previous subframe remain. If the task did run, its output values are placed in the datafile.

5) If any voting found less than total agreement, then the error counts were incremented by the number of non-agreements found.

6) If a reconfiguration was done, the variables real_to_virt and virt_to_real

were correctly updated to reflect the new processor configuration.

The dispatcher calls a variety of other routines to perform the activities. The Vote_activity routine does a vote by collecting the values to be voted on and calling the three way voter VOTE3. Dummy_vote just replaces the old values by the special value. The global executive counts up the error reports and decides which processors are running. The reconfiguration task uses the results of the global executive to reconfigure the system. Since the actual application tasks running on the SIFT computer are not known for this proof exercise, a general routine representing all of the possible application tasks is included.

SUBSECTION 14.1

PRE/POST SPECIFICATION IN SPECIAL

```
MODULE sift

$( The TYPES)

  TYPES
        $( currently the activity kinds are vote, dummy_vote, and execute )

    activity_kinds : INTEGER;

        $( the task kinds are the names of the various tasks.  The special
           ones are global_exec, reconfig, and error_report )

    task_kinds : INTEGER;

        $( This is an array of processor numbers used by the 3-way voter, etc )

    proc_array : ARRAY of INTEGER;

$( --------------------------------------------------------------------- )
        $( The schedule table type definition )

        $( The array of schedules is a configuration schedule for each possible
           processor )

    sched_array : ARRAY of processor_array;

    activity_record : RECORD (activity_kinds activity;
                              task_kinds taskname;
                              INTEGER elem);

        $( This is a sequence of activities for a given subframe. )

    subframe_array : ARRAY of activity_record;

        $( This is a sequence of subframe actions for each configuration. )

    config_array : ARRAY of subframe_array;

        $( This is a set of configurations for each processor )

    processor_array : ARRAY of config_array;

$( --------------------------------------------------------------------- )

        $( A datafile contains an array of task data spaces for each
           process which exists. )

    datafile_array : ARRAY of taskname_array;

        $( An input/output for a process is an array of data )

    elem_array : ARRAY of integer;

        $( Each task has an array of data of its own )

    taskname_array : ARRAY of elem_array;

$( --------------------------------------------------------------------- )
```

```
        $( For each configuration, indicate whether a processor executes
           a task )

    poll_array : ARRAY of poll_proc_array;

        $( Each task either is executed or not {true or false} )

    poll_task_array : ARRAY of BOOLEAN;

        $( For each processor, indicate whether the various tasks are
           executed or not )

    poll_proc_array : ARRAY of poll_task_array;

$( ----------------------------------------------------------------------- )

        $( The number of errors seen from each processor. )

    error_array : ARRAY of INTEGER;

        $( For each task, the element array which is input to that task )

    input_array : ARRAY of elem_array;

        $( A set of integer, for reasoning about properties of things )

    set_of_int : SETOF INTEGER;

        $( An array of integer, one for each task )

    task_array : ARRAY of INTEGER;

        $( An array of boolean )

    bool_array : ARRAY of BOOLEAN;

$(
The PARAMETERS )


PARAMETERS INTEGER frame_size, max_processors, my_processor,
                   max_activities, max_elems, max_tasks, bottom_val,
                   err_threshold, vote, dummy_vote, execute, reconfig,
                   global_exec, error_report, null_task;

      $( The sched table is a sequence of schedules for each configuration.
         It is of the form:
              sched_table [proc_num][configuration][subframe][activity_num]
       and gives a record of activities to do.  Given a processor
       number, and a configuration number, and a subframe number,
       then there are a sequence of activities to do, each one
       described by its ACTIVITY field. The activities are currently
       VOTE, DUMMY_VOTE, and EXECUTE. For votes, the taskname field is
       the task to vote on and the element number is the element to
       vote on.  For dummy_votes, the entire element sequence of the
       taskname is set to bottom.  For executes, the taskname is invoked. )

      sched_array sched_table;

      $( The poll tells whether a processor ran a task in a given
         configuration.  It is referenced as:
              poll [configuration][processor][taskname] )

      poll_array poll;

      $( The inputs tell which tasks have produced input for a
         particular task.  It is indexed by the task to run and
         the number of the task which is input {from 1 to n}. )

      taskname_array inputs;

      $( This indicates which tasks are interactive consistency tasks. )

      bool_array i_c;

      $( This returns the number of elements output by each task )

      task_array result_size;

      $( The error report tasks are a group of tasks {one for each
         processor} which do the error reporting.  They are indicated
         here. )

      proc_array error_report_tasks;

      $( The error_i_c_tasks are the interactive consistency tasks
         which broadcast around the error reports.  There are three
         of these tasks with the specified task numbers. )

      proc_array error_i_c_tasks;


$(
The DEFINITIONS )

 DEFINITIONS

   BOOLEAN is_in_majority (INTEGER c, t, e, val) IS

       $( given a configuration c, a taskname t, and an element number e,
          return true if val is in the majority of the outputs of all of
          the processors which produced output according to POLL.  If there
          is no majority, then val must be the default value. )

      IF EXISTS INTEGER maj_val :
           CARDINALITY ({ INTEGER q | q >= 1 AND q <= max_processors
                             AND poll[c][real_to_virt()[q]][t]
                             AND maj_val = datafile()[q][t][e]}) * 2
              > CARDINALITY ({ INTEGER p | p >= 1 AND p <= max_processors
                                    AND poll[c][p][t]}) END_EXISTS
      THEN FORALL INTEGER maj :
             CARDINALITY ({ INTEGER q | q >= 1 AND q <= max_processors
                             AND poll[c][real_to_virt()[q]][t]
                             AND maj = datafile()[q][t][e]}) * 2
              > CARDINALITY ({ INTEGER p | p >= 1 AND p <= max_processors
                                    AND poll[c][p][t]})
           => val = maj END_FORALL
      ELSE val = bottom_val END_IF;


$(
The PARAMETER INVARIANTS )

 PARAMETER_INVARIANTS

       $( Constraints on the simple parameters.  Various constants have
          appropriate values.)

   frame_size > 0;  max_processors > 0;
   my_processor >= 1 AND my_processor <= max_processors;
   max_activities > 0; max_elems > 0; max_tasks > 0;
   vote > 0 AND dummy_vote > 0 AND execute > 0;
   vote ~= dummy_vote AND vote ~= execute AND dummy_vote ~= execute;
   FORALL INTEGER i : FORALL INTEGER j :
       reconfig ~= global_exec AND
       reconfig ~= null_task AND
       null_task ~= global_exec AND
       error_report_tasks[i] ~= reconfig AND
       error_report_tasks[i] ~= global_exec AND
       error_report_tasks[i] ~= null_task AND
       error_report_tasks[i] ~= error_i_c_tasks[j] AND
       error_i_c_tasks[i] ~= reconfig AND
       error_i_c_tasks[i] ~= null_task AND
       error_i_c_tasks[i] ~= global_exec END_FORALL END_FORALL;

       $( Constraints on the schedule table and associated data structures )

   FORALL INTEGER p : FORALL INTEGER c : FORALL INTEGER sf :
       FORALL INTEGER j1 : FORALL INTEGER j2 : FORALL INTEGER i :

               $( any execute which needs the results of a vote or
                  dummy_vote must follow the vote )

             sched_table[p][c][sf][j1].activity = execute
        AND (   sched_table[p][c][sf][j2].activity = vote
              OR sched_table[p][c][sf][j2].activity = dummy_vote)
        AND inputs[sched_table[p][c][sf][j1].taskname][i]
                = sched_table[p][c][sf][j2].taskname
       => j1 > j2
     END_FORALL END_FORALL END_FORALL END_FORALL END_FORALL END_FORALL;

   FORALL INTEGER p : FORALL INTEGER c : FORALL INTEGER sf :
       FORALL INTEGER j1 : FORALL INTEGER j2 :

               $( there does not exist a vote and a dummy_vote on the
                  same task during a subframe. )

        NOT (    sched_table[p][c][sf][j1].activity = vote
              AND sched_table[p][c][sf][j2].activity = dummy_vote
              AND sched_table[p][c][sf][j1].taskname =
                   sched_table[p][c][sf][j2].taskname)
       END_FORALL END_FORALL END_FORALL END_FORALL END_FORALL;

   FORALL INTEGER p : FORALL INTEGER c : FORALL INTEGER sf :
       FORALL INTEGER j1 : FORALL INTEGER j2 :

               $( there does not exist a vote on the results of an execute
                       in the same subframe )
```

```
            $( There do not exist two executes on
               the same task in the same subframe )

              sched_table[p][c][sf][i].activity = execute
          AND sched_table[p][c][sf][j].activity = execute
          AND i ~= j
       =>    sched_table[p][c][sf][i].taskname
          ~= sched_table[p][c][sf][j].taskname
 END_FORALL END_FORALL END_FORALL END_FORALL END_FORALL;

FORALL INTEGER p : FORALL INTEGER c : FORALL INTEGER sf :
    FORALL INTEGER j :

            $( all activities are either vote, dummy_vote, or execute. )

       sched_table[p][c][sf][j].activity = vote
    OR sched_table[p][c][sf][j].activity = dummy_vote
    OR sched_table[p][c][sf][j].activity = execute
    OR sched_table[p][c][sf][j].activity = 0
 END_FORALL END_FORALL END_FORALL END_FORALL;

FORALL INTEGER p : FORALL INTEGER c : FORALL INTEGER sf :
    FORALL INTEGER j : FORALL INTEGER i :

            $( zero fill in sched table )

           sched_table[p][c][sf][j].activity = 0 AND i > j
        => sched_table[p][c][sf][i].activity = 0
    END_FORALL END_FORALL END_FORALL END_FORALL END_FORALL;

FORALL INTEGER c : FORALL INTEGER ti :

            $( The number of processors running a particular task
                    is 1 for interactive consistency tasks or 3 otherwise )

    CARDINALITY({ INTEGER p | p >= 1 AND p <= max_processors
                                    AND poll[c][p][ti]})
        = IF i_c[ti] THEN 1 ELSE 3 END_IF
 END_FORALL END_FORALL;

            $( The inputs to the global executive are itself and the
               error interactive consistency tasks. )

    inputs[global_exec][1] = global_exec
AND FORALL INTEGER i : (i >= 1 AND i <= max_processors
                      => inputs[global_exec][i+1] = error_i_c_tasks[i])
               AND (i < 1 OR i > max_processors
                     => inputs[global_exec][i] = null_task)
     END_FORALL;

    $( these are the constraints on the output of various tasks )

FORALL input_array inp : FORALL INTEGER i :
    task_results (error_i_c_tasks[i], inp) = inp[1]
END_FORALL END_FORALL;

        $( returns 1 {not-working} if it was previously not working
           or if a majority of those working consider it bad. )
       NOT (    sched_table[p][c][sf][j1].activity = execute
            AND sched_table[p][c][sf][j2].activity = vote
            AND sched_table[p][c][sf][j1].taskname =
                    sched_table[p][c][sf][j2].taskname)
      END_FORALL END_FORALL END_FORALL END_FORALL END_FORALL;

FORALL INTEGER p : FORALL INTEGER c : FORALL INTEGER sf :
    FORALL INTEGER j1 : FORALL INTEGER j2 :

            $( Any reconfiguration done must be at the end of a subframe )

           sched_table[p][c][sf][j1].activity = execute
       AND sched_table[p][c][sf][j1].taskname = reconfig
       AND (   sched_table[p][c][sf][j2].activity = execute
            OR sched_table[p][c][sf][j2].activity = vote)
     => j1 > j2 END_FORALL END_FORALL END_FORALL END_FORALL END_FORALL;

FORALL INTEGER p : FORALL INTEGER c : FORALL INTEGER sf :

            $( no vote and error_report allowed in the same subframe. )

           EXISTS INTEGER j :
               sched_table[p][c][sf][j].activity = vote END_EXISTS
        => NOT EXISTS INTEGER i :
                      sched_table[p][c][sf][i].activity = execute
                  AND sched_table[p][c][sf][i].taskname
                            = error_report_tasks[p] END_EXISTS
 END_FORALL END_FORALL END_FORALL;

FORALL INTEGER p : FORALL INTEGER c : FORALL INTEGER sf :
    FORALL INTEGER i : FORALL INTEGER j :

            $( There do not exist two votes on the same element of
               the same task in the same subframe )

              sched_table[p][c][sf][i].activity = vote
          AND sched_table[p][c][sf][j].activity = vote
          AND i ~= j
       =>    sched_table[p][c][sf][i].taskname
                 ~= sched_table[p][c][sf][j].taskname
          OR sched_table[p][c][sf][i].elem
                 ~= sched_table[p][c][sf][j].elem
 END_FORALL END_FORALL END_FORALL END_FORALL END_FORALL;

FORALL INTEGER p : FORALL INTEGER c : FORALL INTEGER sf :
    FORALL INTEGER i : FORALL INTEGER j :

            $( There do not exist two dummy_votes on the same element of
               the same task in the same subframe )

              sched_table[p][c][sf][i].activity = dummy_vote
          AND sched_table[p][c][sf][j].activity = dummy_vote
          AND i ~= j
       => sched_table[p][c][sf][i].taskname
              ~= sched_table[p][c][sf][j].taskname
 END_FORALL END_FORALL END_FORALL END_FORALL END_FORALL;

FORALL INTEGER p : FORALL INTEGER c : FORALL INTEGER sf :
    FORALL INTEGER i : FORALL INTEGER j :
```

```
    FORALL input_array inp : FORALL INTEGER p :
      IF   inp[1][p] = 1   $( = input[global_exec][p] )
        OR CARDINALITY({INTEGER q1 | inp[1][q1] = 0
                                  AND p ~= q1 AND inp[q1+1][p] = 1}) * 2
                                         $( = input[error_i_c_tasks[q1]] )
             > CARDINALITY ({INTEGER q2 | inp[1][q2] = 0 AND p ~= q2})
      THEN task_results (global_exec, inp)[p] = 1
      ELSE task_results (global_exec, inp)[p] = 0 END_IF
     END_FORALL END_FORALL;

  FUNCTIONS

$(
The State Functions )

        $( The subframe count.  Used to index into various tables )

   VFUN subframe() -> INTEGER s;

        $( The current configuration {ie, the number of processors
           currently assumed to be working}. )

   VFUN config() -> INTEGER c;

        $( This is the input values for a task.  It is referenced as:
                input [taskname][element] )

   VFUN input() -> input_array value;

        $( The datafile is the broadcast area.  It is referenced as:
                datafile [processor][taskname][element] )

   VFUN datafile() -> datafile_array value;

        $( The errors accumulated for each processor, indexed by processor )

   VFUN errors() -> error_array v;

        $( Given a real processor number, this returns the processor number
           used in the various tables for this configuration. )

   VFUN real_to_virt() -> proc_array v;

        $( Given a processor number in the tables, this maps to the current
           real processor which is associated with it )

   VFUN virt_to_real() -> proc_array v;

$(
The Operations )

   OFUN dispatcher ();

      $( The dispatcher is invoked each subframe.  It bumps the subframe
         number and does each of the activities for that subframe. )

     ASSERTIONS

      subframe() < frame_size and subframe() >= 0;
      FORALL INTEGER c : FORALL INTEGER ti :
          CARDINALITY({ INTEGER p | p >= 1 AND p <= max_processors
                                       AND poll[c][real_to_virt()[p]][ti]})
              = IF i_c[ti] THEN 1 ELSE 3 END_IF
       END_FORALL END_FORALL;

     EFFECTS

      $( changes to subframe )

      'subframe() = (subframe + 1) MOD frame_size;

      $( poll set still has 1 or 3 )

      FORALL INTEGER c : FORALL INTEGER ti :
          CARDINALITY({ INTEGER p | p >= 1 AND p <= max_processors
                                       AND poll[c][real_to_virt()[p]][ti]})
              = IF i_c[ti] THEN 1 ELSE 3 END_IF
       END_FORALL END_FORALL;

      $( Changes to INPUT:
         For all tasks ti and for all data elements of the task,
         if this element of this task was voted on then the input now
         has the majority value, else if it was dummy voted it has
         bottom as value, else it hasn't changed. )

              $( Frame axiom: if no vote or dummy vote, nothing changed )

      FORALL INTEGER ti : FORALL INTEGER ei :
          NOT EXISTS INTEGER j : j >= 1 AND j <= max_activities
               AND ti = sched_table[real_to_virt()[my_processor]][config()]
                                    [subframe()][j].taskname
               AND (    (    sched_table[real_to_virt()[my_processor]]
                                        [config()][subframe()][j].activity
                                   = vote
                        AND ei = sched_table[real_to_virt()[my_processor]]
                                             [config()][subframe()][j].elem)
                    OR sched_table[real_to_virt()[my_processor]][config()]
                                   [subframe()][j].activity
                          = dummy_vote) END_EXISTS
         => 'input()[ti][ei] = input()[ti][ei]
      END_FORALL END_FORALL;

              $( Vote activity )

      FORALL INTEGER ti : FORALL INTEGER ei :
              EXISTS INTEGER j : j >= 1 AND j <= max_activities
               AND sched_table[real_to_virt()[my_processor]][config()]
                               [subframe()][j].activity = vote
```

```
           AND ti = sched_table[real_to_virt()[my_processor]][config()]
                               [subframe()][j].taskname
           AND ei = sched_table[real_to_virt()[my_processor]][config()]
                               [subframe()][j].elem END_EXISTS
           => is_in_majority (config(), ti, ei, 'input()[ti][ei])
  END_FORALL END_FORALL;

          $( Dummy vote activity )

  FORALL INTEGER ti : FORALL INTEGER ei :
          EXISTS INTEGER j : j >= 1 AND j <= max_activities
                  AND sched_table[real_to_virt()[my_processor]][config()]
                                 [subframe()][j].activity
                                   = dummy_vote
                  AND ti = sched_table[real_to_virt()[my_processor]]
                                      [config()][subframe()][j].taskname
             END_EXISTS AND ei >= 1 AND ei <= result_size[ti]
           => 'input()[ti][ei] = bottom_val
  END_FORALL END_FORALL;

  $( Changes to ERRORS:
     For all processors p, for every non interactive consistency
     vote that p was involved in for which p was not in the majority,
     the error count for p goes up by one. )

  FORALL INTEGER p :
     'errors()[p] = errors()[p]
         + CARDINALITY ({INTEGER j | j >= 1 AND j <= max_activities
          AND sched_table[real_to_virt()[my_processor]][config()]
                         [subframe()][j].activity = vote
          AND NOT i_c[sched_table[real_to_virt()[my_processor]][config()]
                                 [subframe()][j].taskname]
          AND poll[config()][real_to_virt()[p]]
                  [sched_table[real_to_virt()[my_processor]][config()]
                              [subframe()][j].taskname]
          AND NOT is_in_majority
                   (config(),
                    sched_table[real_to_virt()[my_processor]][config()]
                               [subframe()][j].taskname,
                    sched_table[real_to_virt()[my_processor]][config()]
                               [subframe()][j].elem,
                    datafile()[p]
                          [sched_table[real_to_virt()[my_processor]]
                                      [config()][subframe()][j].taskname]
                          [sched_table[real_to_virt()[my_processor]]
                                      [config()][subframe()][j].elem]})
  END_FORALL;

  FORALL INTEGER j : FORALL INTEGER p :
      j >= 1 AND j <= max_activities
      AND sched_table[real_to_virt()[my_processor]][config()]
                     [subframe()][j].activity = execute
      AND sched_table[real_to_virt()[my_processor]]
                     [config()][subframe()][j].taskname
                  = error_report_tasks[my_processor]
             => 'errors()[p] = 0 END_FORALL END_FORALL;

$( Changes to DATAFILE:
  For each task that is executed, the datafile contains the
  output for that task. )

          $( Frame axiom: if no execute is done on a task, then
             its datafile area stays the same. )

 FORALL INTEGER p : FORALL INTEGER ti : FORALL INTEGER ei :
    NOT (    p = my_processor
         AND EXISTS INTEGER j : j >= 1 AND j <= max_activities AND
             sched_table[real_to_virt()[my_processor]][config()]
                        [subframe()][j].activity = execute
            AND sched_table[real_to_virt()[my_processor]][config()]
                           [subframe()][j].taskname = ti
                  END_EXISTS)
           => 'datafile()[p][ti][ei] = datafile()[p][ti][ei]
   END_FORALL END_FORALL END_FORALL;

          $( Execute activity )

 FORALL INTEGER ti : FORALL INTEGER ei : FORALL INPUT_ARRAY inp :
         EXISTS INTEGER j : j >= 1 AND j <= max_activities AND
             sched_table[real_to_virt()[my_processor]][config()]
                        [subframe()][j].activity = execute
            AND sched_table[real_to_virt()[my_processor]][config()]
                           [subframe()][j].taskname = ti
            AND ti ~= reconfig
            AND ti ~= error_report_tasks[my_processor]
          END_EXISTS
     AND FORALL INTEGER taski : FORALL INTEGER j : FORALL INTEGER elemi :
                   j >= 1 AND j <= result_size[taski]
               AND inputs[ti][j] = taski AND taski ~= null_task
            => inp[j][elemi] = 'input()[taski][elemi]
          END_FORALL END_FORALL END_FORALL
     => 'datafile()[my_processor][ti][ei] = task_results (ti, inp)[ei]
  END_FORALL END_FORALL END_FORALL;

 FORALL INTEGER ei : FORALL INTEGER j :
           j >= 1 AND j <= max_activities
       AND sched_table[real_to_virt()[my_processor]][config()]
                      [subframe()][j].activity = execute
       AND sched_table[real_to_virt()[my_processor]]
                      [config()][subframe()][j].taskname
                  = error_report_tasks[my_processor]
     => 'datafile()[my_processor][error_report_tasks[my_processor]][ei]
              = IF errors()[ei] > err_threshold THEN 1 ELSE 0 END_IF
  END_FORALL END_FORALL;

$( Changes to CONFIG, REAL_TO_VIRT, and VIRT_TO_REAL:
  These are only changed by reconfig.  Config is set to the number of
  processors which are currently working, as reported by the global_exec
  task.  Real_to_virt is set so that the nth processor is mapped to
  the mth working processor.  Virt_to_real is set so that the nth
  working processor is mapped to the mth processor. )

          $( Frame axiom: if there is no reconfiguration, then the
             reconfiguration data stays the same. )

     NOT EXISTS INTEGER j : j >= 1 AND j <= max_activities
           AND sched_table[real_to_virt()[my_processor]][config()]
                          [subframe()][j].activity = execute
            AND sched_table[real_to_virt()[my_processor]][config()]
                           [subframe()][j].taskname = reconfig
          END_EXISTS
        => 'config() = config() AND 'real_to_virt() = real_to_virt() AND
           'virt_to_real() = virt_to_real();

           $( reconfiguration activity )

      EXISTS INTEGER j : j >= 1 AND j <= max_activities
           AND sched_table[real_to_virt()[my_processor]][config()]
                          [subframe()][j].activity = execute
           AND sched_table[real_to_virt()[my_processor]][config()]
                          [subframe()][j].taskname = reconfig
          END_EXISTS
       => FORALL INTEGER x :
                'config() = CARDINALITY ({INTEGER p1 |
                                           input()[global_exec][p1] = 0})
            AND 'real_to_virt()[x] = CARDINALITY ({INTEGER p2 |
                           p2 <= x AND input()[global_exec][p2] = 0})
            AND 'virt_to_real()[x]
                    = x + CARDINALITY ({INTEGER p3 |
                               p3 <= x AND input()[global_exec][p3] = 1})
          END_FORALL;


OFUN vote_activity (INTEGER c, t, e);
  ASSERTIONS
   FORALL INTEGER c : FORALL INTEGER ti :
       CARDINALITY({ INTEGER p | p >= 1 AND p <= max_processors
                                   AND poll[c][real_to_virt()[p]][ti]})
           = IF i_c[ti] THEN 1 ELSE 3 END_IF
    END_FORALL END_FORALL;

  EFFECTS
   is_in_majority (c, t, e, 'input()[t][e]);
   FORALL INTEGER ti : FORALL INTEGER ei :
        ti ~= t OR ei ~= e => 'input()[ti][ei] = input()[ti][ei]
      END_FORALL END_FORALL;
   FORALL INTEGER q :
      IF poll[c][real_to_virt()[q]][t] AND NOT i_c[t] THEN
        IF is_in_majority (c, t, e, datafile()[q][t][e])
           THEN 'errors()[q] = errors()[q]
           ELSE 'errors()[q] = errors()[q] + 1 END_IF
      ELSE 'errors()[q] = errors()[q] END_IF END_FORALL;

OVFUN vote3 (INTEGER t, e; proc_array p) -> INTEGER result;
  ASSERTIONS
       p[1] ~= p[2] AND p[1] ~= p[3] AND p[2] ~= p[3]
   AND p[1] >= 1 AND p[1] <= max_processors
   AND p[2] >= 1 AND p[2] <= max_processors
   AND p[3] >= 1 AND p[3] <= max_processors;
  EFFECTS
   IF EXISTS INTEGER maj_val :
         CARDINALITY ({INTEGER q1 | q1 >= 1 AND q1 <= max_processors
                                   AND datafile()[q1][t][e] = maj_val
                                   AND (q1 = p[1] OR q1 = p[2] OR
                                        q1 = p[3])}) > 1 END_EXISTS
   THEN FORALL INTEGER maj :
            CARDINALITY ({INTEGER q1 | q1 >= 1 AND q1 <= max_processors
                                     AND datafile()[q1][t][e] = maj
                                     AND (q1 = p[1] OR q1 = p[2] OR
                                          q1 = p[3]})) > 1
          =>      result = maj
             AND FORALL INTEGER j :
                     IF j ~= p[1] AND j ~= p[2] AND j ~= p[3]
                     THEN 'errors()[j] = errors()[j]
                     ELSE IF datafile()[j][t][e] = maj
                          THEN 'errors()[j] = errors()[j]
                          ELSE 'errors()[j] = errors()[j] + 1 END_IF END_IF
               END_FORALL
          END_FORALL
    ELSE      result = bottom_val
         AND FORALL INTEGER j :
                 IF j = p[1] OR j = p[2] OR j = p[3]
                 THEN 'errors()[j] = errors()[j] + 1
                 ELSE 'errors()[j] = errors()[j] END_IF
          END_FORALL END_IF;

 OFUN dummy_vote_activity (INTEGER c, t);
   EFFECTS
    FORALL INTEGER ti : FORALL INTEGER ei :
      'input()[ti][ei] =
            IF ti = t AND ei >= 1 AND ei <= result_size[t]
            THEN bottom_val ELSE input()[ti][ei] END_IF
    END_FORALL END_FORALL;

OFUN gexectask ();
  EFFECTS
    FORALL INTEGER t : FORALL INTEGER e : FORALL INTEGER p :
           IF p = my_processor AND t = global_exec THEN
              IF    inp[global_exec][p] = 1
                 OR CARDINALITY ({INTEGER q1 | inp[global_exec][q1] = 0
                      AND p ~= q1 AND inp[error_i_c_tasks[q1]][p] = 1}) * 2
                    > CARDINALITY ({INTEGER q2 | inp[global_exec][q2] = 0
                                                    AND p ~= q2})
              THEN 'datafile()[p][t][e] = 1
              ELSE 'datafile()[p][t][e] = 0 END_IF
           ELSE 'datafile()[p][t][e] = datafile()[p][t][e] END_IF
    END_FORALL END_FORALL END_FORALL;

OFUN errtask ();
  EFFECTS
    FORALL INTEGER p : FORALL INTEGER t : FORALL INTEGER e :
       'datafile()[p][t][e] =
          IF t = error_report AND p = my_processor THEN
            IF errors()[e] > err_threshold THEN 1 ELSE 0 END_IF
          ELSE datafile()[p][t][e] END_IF END_FORALL END_FORALL
      AND (p >= 1 AND p <= max_processors => 'errors()[p] = 0) END_FORALL;

OFUN recftask ();
  EFFECTS
    'config() = CARDINALITY ({INTEGER p1 | input()[global_exec][p1] = 0});
    FORALL INTEGER x2 :
       'real_to_virt()[x2] = CARDINALITY ({INTEGER p2 |
                                p2 <= x2 AND input()[global_exec][p2] = 0})
     END_FORALL;
    FORALL INTEGER x3 :
       'virt_to_real()[x3] = x3 + CARDINALITY ({INTEGER p3 |
                                         p3 <= x3 AND input()[global_exec][p3] = 1})
       END_FORALL;

    OVFUN do_err_ic (INTEGER ti) -> BOOLEAN task_done;
      EFFECTS
        IF EXISTS INTEGER i : error_i_c_tasks[i] = ti END_EXISTS
        THEN FORALL INTEGER p : FORALL INTEGER taski : FORALL INTEGER ei :
                 'datafile()[p][taski][ei] =
                     IF p = my_processor AND ti = taski
                       THEN input()[inputs[ti][1]][ei]
                       ELSE datafile()[p][taski][ei] END_IF
               END_FORALL END_FORALL END_FORALL
             AND task_done = TRUE
        ELSE     task_done = FALSE
             and 'datafile() = datafile() END_IF;

    OFUN general_task (INTEGER ti);
      EFFECTS
        FORALL INTEGER p : FORALL INTEGER taski : FORALL INTEGER ei :
            IF p = my_processor AND taski = ti THEN
               FORALL input_array inp :
                  FORALL INTEGER input_task : FORALL INTEGER j :
                       FORALL INTEGER elemi :
                             j >= 1 AND j <= result_size[input_task]
                         AND inputs[ti][j] = input_task
                         AND input_task ~= null_task
                      => inp[j][elemi] = 'input()[input_task][elemi]
                    END_FORALL END_FORALL END_FORALL
                => 'datafile()[p][ti][ei] = task_results (ti, inp)[ei]
               END_FORALL
           ELSE 'datafile()[p][taski][ei] = datafile()[p][taski][ei] END_IF
          END_FORALL END_FORALL END_FORALL;
       'input() = input();

END_MODULE
```

SUBSECTION 14.2

SIFT CODE

```
const
  frame_size = 64;
  max_processors = 7;
  my_processor = 0;
  max_activities = 10;
  max_elems = 100;
  max_tasks = 10;
  bottom_val = 0;
  err_threshold = 2;
  vote = 1;
  dummy_vote = 2;
  execute = 3;
  null_task = 0;
  reconfig = 1;
  global_exec = 2;
  error_report = 3;

type
  activity_kinds = integer;
  activity_range = 1..max_activities;
  task_kinds = 1..max_tasks;
  subframe_range = 1..max_subframe;
  elem_range = 1..max_elems;
  config_range = 1..max_processors;
  processors = 0..max_processors;
  proc_array = array [processors] of integer;
  activity_record = record activity : activity_kinds;
                           taskname : task_kinds;
                           elem : elem_range end;
  subframe_array = array [activity_range] of activity_record;
  config_array = array [subframe_range] of subframe_array;
  processor_array = array [processors] of config_array;
  sched_array = array [processors] of processor_array;
  elem_array = array [elem_range] of integer;
  taskname_array = array [task_kinds] of elem_array;
  datafile_array = array [processors] of taskname_array;
  poll_task_array = array [task_kinds] of boolean;
  poll_proc_array = array [processors] of poll_task_array;
  poll_array = array [config_range] of poll_proc_array;
  error_array = array [processors] of integer;
  input_array = array [task_kinds] of elem_array;
  task_array = array [task_kinds] of integer;
  bool_array = array [processors] of boolean;

var sched_table : sched_array;
    poll : poll_array;
    inputs : taskname_array;
    i_c : bool_array;
    result_size : task_array;
    task_results : elem_array;
    error_report_tasks : proc_array;
    error_i_c_tasks : proc_array;

    subframe, config : integer;
    input : input_array;
    datafile : datafile_array;
    errors : error_array;
    real_to_virt, virt_to_real : proc_array;
    exc : integer;

procedure dispatcher;
var i : integer;
    activity_num : activity_range;
    ti : task_kinds;
    task_done : boolean;
begin
  activity_num := 1;
  while      (sched_table[real_to_virt[my_processor]][config]
                        [subframe][activity_num].activity <> 0)
       and not (     (sched_table[real_to_virt[my_processor]][config]
                              [subframe][activity_num].activity = execute)
                and (sched_table[real_to_virt[my_processor]][config][subframe]
                              [activity_num].taskname = reconfig))
        and (activity_num <= max_activities)
      assert

        activity_num >= 1 and

       (* poll set still same *)

       forall integer c : forall integer ti :
           cardinality({ integer p | p >= 1 and p <= max_processors
                                         and poll[c][real_to_virt()[p]][ti]})
               = if i_c[ti] then 1 else 3 end_if
        end_forall end_forall and

       (* Changes to INPUT:
          For all tasks ti and for all data elements of the task,
          if this element of this task was voted on then the input now
          has the majority value, else if it was dummy voted it has
          bottom as value, else it hasn't changed. *)

                (* Frame axiom: if no vote or dummy vote, nothing changed *)

       forall integer ti : forall integer ei :
           not exists integer j : j >= 1 and j < activity_num
                and ti = sched_table[real_to_virt()[my_processor]][config()]
                                    [subframe()][j].taskname
                and (    (    sched_table[real_to_virt()[my_processor]]
                                        [config()][subframe()][j].activity
                                    = vote
                          and ei = sched_table[real_to_virt()[my_processor]]
                                             [config()][subframe()][j].elem)
                     or sched_table[real_to_virt()[my_processor]][config()]
                                  [subframe()][j].activity
                           = dummy_vote) end_exists
          => 'input()[ti][ei] = input()[ti][ei]
       end_forall end_forall and

                (* Vote activity *)

       forall integer ti : forall integer ei :
               exists integer j : j >= 1 and j < activity_num
                and sched_table[real_to_virt()[my_processor]][config()]
                               [subframe()][j].activity = vote
                and ti = sched_table[real_to_virt()[my_processor]][config()]
                                    [subframe()][j].taskname
           and ei = sched_table[real_to_virt()[my_processor]][config()]
                               [subframe()][j].elem end_exists
           => is_in_majority (config(), ti, ei, 'input()[ti][ei])
  end_forall end_forall and

          (* Dummy vote activity *)

  forall integer ti : forall integer ei :
          exists integer j : j >= 1 and j < activity_num
                  and sched_table[real_to_virt()[my_processor]][config()]
                                 [subframe()][j].activity
                                   = dummy_vote
                  and ti = sched_table[real_to_virt()[my_processor]]
                                      [config()][subframe()][j].taskname
             end_exists and ei >= 1 and ei <= result_size[ti]
           => 'input()[ti][ei] = bottom_val
  end_forall end_forall and

  (* Changes to ERRORS:
     For all processors p, for every non interactive consistency
     vote that p was involved in for which p was not in the majority,
     the error count for p goes up by one. *)

  forall integer p :
     'errors()[p] = errors()[p]
         + cardinality ({integer j | j >= 1 and j < activity_num
          and sched_table[real_to_virt()[my_processor]][config()]
                         [subframe()][j].activity = vote
          and not i_c[sched_table[real_to_virt()[my_processor]][config()]
                                 [subframe()][j].taskname]
          and poll[config()][real_to_virt()[p]]
                  [sched_table[real_to_virt()[my_processor]][config()]
                              [subframe()][j].taskname]
          and not is_in_majority
                   (config(),
                    sched_table[real_to_virt()[my_processor]][config()]
                               [subframe()][j].taskname,
                    sched_table[real_to_virt()[my_processor]][config()]
                               [subframe()][j].elem,
                    datafile()[p]
                          [sched_table[real_to_virt()[my_processor]]
                                      [config()][subframe()][j].taskname]
                          [sched_table[real_to_virt()[my_processor]]
                                      [config()][subframe()][j].elem])})
  end_forall and

  forall integer j : forall integer p :
      j >= 1 and j < activity_num
      and sched_table[real_to_virt()[my_processor]][config()]
                     [subframe()][j].activity = execute
      and sched_table[real_to_virt()[my_processor]]
                     [config()][subframe()][j].taskname
                  = error_report_tasks[my_processor]
             => 'errors()[p] = 0 end_forall end_forall and

(* Changes to DATAFILE:
  For each task that is executed, the datafile contains the
  output for that task. *)
```

```
           (* Frame axiom: if no execute is done on a task, then
              its datafile area stays the same. *)

  forall integer p : forall integer ti : forall integer ei :
     not (    p = my_processor
          and exists integer j : j >= 1 and j < activity_num and
              sched_table[real_to_virt()[my_processor]][config()]
                         [subframe()][j].activity = execute
             and sched_table[real_to_virt()[my_processor]][config()]
                            [subframe()][j].taskname = ti
                   end_exists)
            => 'datafile()[p][ti][ei] = datafile()[p][ti][ei]
    end_forall end_forall end_forall and

           (* Execute activity *)

  forall integer ti : forall integer ei : forall input_array inp :
          exists integer j : j >= 1 and j < activity_num and
              sched_table[real_to_virt()[my_processor]][config()]
                         [subframe()][j].activity = execute
             and sched_table[real_to_virt()[my_processor]][config()]
                            [subframe()][j].taskname = ti
             and ti ~= reconfig
             and ti ~= error_report_tasks[my_processor]
           end_exists
      and forall integer taski : forall integer j : forall integer elemi :
                    j >= 1 and j <= result_size[taski]
                and inputs[ti][j] = taski and taski ~= null_task
             => inp[j][elemi] = 'input()[taski][elemi]
           end_forall end_forall end_forall
      => 'datafile()[my_processor][ti][ei] = task_results (ti, inp)[ei]
   end_forall end_forall end_forall and

  forall integer ei : forall integer j :
            j >= 1 and j < activity_num
        and sched_table[real_to_virt()[my_processor]][config()]
                       [subframe()][j].activity = execute
        and sched_table[real_to_virt()[my_processor]]
                       [config()][subframe()][j].taskname
                   = error_report_tasks[my_processor]
      => 'datafile()[my_processor][error_report_tasks[my_processor]][ei]
               = if errors()[ei] > err_threshold then 1 else 0 end_if
   end_forall end_forall

  counting max_activities - activity_num
do begin
      case sched_table[real_to_virt[my_processor]][config]
                      [subframe][activity_num].activity of
          vote : vote_activity
                    (config,
                     sched_table[real_to_virt[my_processor]][config]
                                [subframe][activity_num].taskname,
                     sched_table[real_to_virt[my_processor]][config]
                                [subframe][activity_num].elem);
          dummy_vote :
             dummy_vote_activity
                   (config,
                    sched_table[real_to_virt[my_processor]][config]
                               [subframe][activity_num].taskname);
              execute :
                 begin
                   ti := sched_table[real_to_virt[my_processor]][config]
                                    [subframe][activity_num].taskname;
                   if ti = global_exec then gexectask
                   else if ti = error_report_tasks[my_processor] then errtask
                   else begin
                            do_err_ic (ti, task_done);
                            if not task_done then general_task (ti)
                        end
                 end
          end;
          activity_num := activity_num + 1
       end;

  if     (sched_table[real_to_virt[my_processor]][config]
                    [subframe][activity_num].activity = execute)
     and (sched_table[real_to_virt[my_processor]][config]
                    [subframe][activity_num].taskname = reconfig)
     then recftask;

  if subframe = frame_size-1 then subframe := 0
  else subframe := subframe + 1;

end;

procedure vote_activity (c, t, e : integer);
var i, q, voted : integer;
    p : proc_array;
begin
  if i_c[t] then
      begin
         q := 1;
         while not poll[c][real_to_virt[q]][t]
               assert i_c[t] and q >= 1 and q <= max_processors and
                      forall integer q1 : q1 >= 1 and q1 < q
                             => not poll[c][real_to_virt()[q1]][t] end_forall
               counting max_processors - q
           do q := q + 1;
         input[t][e] := datafile[q][t][e]
      end
  else begin
          i := 0;
          for q := 1 to max_processors
              assert     not i_c[t] and i >= 0
                     and   cardinality ({integer x | x >= 1 and x <= q
                                       and poll[c][real_to_virt()[x]][t]})
                         = cardinality ({integer y | exists integer k :
                                          k >= 1 and k <= i and p[k] = y
                                          end_exists})
                     and cardinality ({integer y | exists integer k :
                                          k >= 1 and k <= i and p[k] = y
                                        end_exists}) = i
                     and forall integer k :
                               k >= 1 and k <= i
                            => p[k] >= 1 and p[k] <= q end_forall
                     and forall integer k1 : forall integer k2 :
                              k1 >= 1 and k1 <= i and k2 >= 1 and k2 <= i
                          and k1 ~= k2 => p[k1] ~= p[k2] end_forall end_forall
```

```
|              do if poll[c][real_to_virt[q]][t] then
|                  begin
|                      i := i + 1;
|                      p[i] := q
|                  end;
|           vote3 (t, e, p, voted);
|          input[t][e] := voted
|       end;
|end;

|procedure vote3(t, e : integer; p : proc_array; var result:integer);
|var v1, v2, v3 : integer;
|begin
|  v1 := datafile[p[1]][t][e];
|  v2 := datafile[p[2]][t][e];
|  v3 := datafile[p[3]][t][e];
|  if v1=v2 then
|    begin
|      if v1<>v3 then errors[p[3]] := errors[p[3]]+1;
|      result:=v1
|    end
|  else if v1=v3 then
|      begin
|       errors[p[2]] := errors[p[2]]+1;
|       result:=v1
|      end
|  else if v2=v3 then
|      begin
|       errors[p[1]] := errors[p[1]]+1;
|       result:=v2
|      end
|  else
|      begin
|       errors[p[1]] := errors[p[1]]+1;
|       errors[p[2]] := errors[p[2]]+1;
|       errors[p[3]] := errors[p[3]]+1
|      end
|end;

|procedure dummy_vote_activity (c, t : integer);
|var res : integer;
|begin
|  for res := 1 to result_size[t]
|        assert forall integer ti : forall integer ei :
|                 'input()[ti][ei] =
|                     if ti = t and ei >= 1 and ei <= res then bottom_val
|                     else input()[ti][ei] end_if end_forall end_forall
|       do input[t][res] := bottom_val
|end;

procedure gexectask;
var q, errcount : integer;
    proc_votes, bad_proc_votes : proc_array;
begin
  for q := 0 to max_processors
    do begin
         proc_votes[q] := 0;
         for p := 0 to max_processors
          do if (q <> p) and (input[global_exec][p] = 0) then
```

```
              begin
                  proc_votes[q] := proc_votes[q] + 1;
                  if input[error_i_c_tasks[p]][q] = 1 then
                      bad_proc_votes[q] := bad_proc_votes[q] + 1
              end;
       end;
  for q := 0 to max_processors
    do if bad_proc_votes[q] * 2 > proc_votes[q]
        then datafile[my_processor][global_exec][q] := 1
        else datafile[my_processor][global_exec][q] := 0
end;

procedure errtask;
var p : integer;
begin
  for p := 0 to max_processors
    do begin
        if errors[p] > err_threshold then
                datafile[my_processor][error_report[my_processor]][p] := 1
           else datafile[my_processor][error_report[my_processor]][p] := 0;
        errors[p] := 0
       end
end;

procedure do_err_ic (ti : integer; var did_it : boolean);
var i : integer;
begin
  i := 1;
  did_it := true;
  while (i < max_activities) and (ti <> error_i_c_tasks[i]) do i := i + 1;
  if ti = error_i_c_tasks[i] then
      datafile[my_processor][error_i_c_tasks[err_no]][1] :=
            input[error_report_tasks[i]][1]
  else did_it := false
end;

procedure recftask;
var i : integer;
begin
  config := 0;
  for i := 1 to max_processors do
      if input[global_exec][ i] = 0 then
          begin
              config := config + 1;
              real_to_virt[i] := config;
              virt_to_real[config] := i
          end
end;

procedure general_task (ti : integer);
begin end;
```

CHAPTER 15

SAMPLE VERIFICATION CONDITIONS AND PROOFS

1. The Proof of the SIFT Implementation

The SIFT operating system was implemented in PASCAL and specified in SPECIAL. The SPECIAL specification is the lowest level description of the hierarchy of models which describe SIFT. This lowest level describes the implementation by specifying the effect of each PASCAL routine (FUNCTION or PROCEDURE).

The SIFT specifications consist primarily of constraints on the schedule table and descriptions of the changes each routine makes to the global variables. The proof proceeds by proving that each routine is consistent with its specification. Each of these proofs is separate from the others and the order of the proofs is unimportant. A routine is proved by generating a set of verification conditions from the PASCAL code and the SPECIAL specifications. Each verification condition is a set of assertions derived from a particular path in the routine. A verification condition (VC) is generated for each possible path through the routine.

These VCs are formulae of first order predicate calculus with equality. They must be proven to be true in the mathematical sense in order to assure that every possible path through the routine accomplishes the change of state required by the specifications. These VCs can be proven by hand but they are very large and detailed and difficult to work with. The Shostak Theorem Prover (STP) [1] is used to prove these formulae. The context of the VCs must be given to STP in the form of declarations of variables and functions referenced. Each assertion of the path is asserted to STP and the requirements of the end of the path are then proven.

An example VC is given at the end of this section. Not all of the declarations needed are shown. The axioms of the form SET.FN.AXIOM describe the intensional set constructors used in the specifications. Sets are constructed to define the concept of a majority vote. The axioms of the form VC4.H are the hypotheses of VC number 4 and come from the path which is being proven. The final formula VC4.C9 is the ninth conjunct of the conclusion of the path. This path is part of the dispatcher procedure and is the path from the top of the while loop through the vote activity back to the top of the loop. It is necessary to prove that the vote was done properly. The effects of the procedure call to the VOTE_ACTIVITY can be assumed in the proof of this path since the called procedure will be proven separately. The major difficulty in this proof is showing that the vote activity does not interfere with the other activities of the subframe. The constraints on the schedule table are used to assure this.

The SIFT implementation and specifications are organized in a hierarchy of FUNCTION and PROCEDURE calls. The topmost routine is the DISPATCHER. This procedure is called at the beginning of each subframe and it directs the execution of each activity of that subframe as directed by the schedule table. Each activity in the schedule table for the current configuration, processor number, and subframe is dispatched as a call to another PASCAL routine. The vote is done by the VOTE_ACTIVITY, the dummy vote is done by the DUMMY_VOTE_ACTIVITY, the global executive task is GEXECTASK, the error reporting task is ERRTASK, the interactive consistency error reporting is done by do_err_ic, and the general application tasks are done by GENERAL_TASK. Other routines such as VOTE3 (the three-way voter) are used to accomplish these tasks.

Each of these routines are specified separately in the SIFT code specification. The proof proceeds by taking each routine and proving that it is consistent with its specifications. The lowest level routines do not call any other routines and consist only of PASCAL statements with known effects. The verification condition generator analyzes these statements to prove that they modify the global variables as specified. The next higher routines can then assume that any call to these lowest level routines will have the right effect since that will have been proven. The DISPATCHER at the highest level brings together the combined effect of all the routine calls done during the processing of a single subframe and assumes during its proof that each called routine performs as specified. This proves that one subframe on one processor is correctly processed. The proof that all of the subframes on all of the processors are correctly combined is done in the proof of the higher models of SIFT.

The proof of SIFT consists of a large complicated set formulae each proven to be true using a mechanical theorem prover. This process is long and arduous and requires a great deal of work and interaction with the designers, specifiers, and implementors. When a formula fails to be proven by the theorem prover it is necessary to inspect the formula to decide if it is too complicated for the theorem prover or if it is untrue. Most proof failures are caused by some missing information in the specifications which are necessary for the proof. Often some constraint on the schedule table was not fully or correctly specified.

References

[1] R. Shostak, R. Schwartz and P.M. Melliar-Smith.
STP: A Mechanized Logic for Specification and Verification.
1981.
Submitted to Working Conference on the Formal Description of Programming Concepts.

```
SET.FN.AXIOM.7: axiom
        ∀ P#13, TI#40, C#14:
           P#13 ∈ SET.FN#7(TI#40, C#14)
                ≡
           P#13 ≥ 1
           ∧ P#13 ≤ MAX_PROCESSORS()
           ∧ POLL()[TI#40, REAL_TO_VIRT(STATE.SIFT())[P#13], C#14]

SET.FN.AXIOM.9: axiom
        ∀ Q#1, MAJ_VAL#2, STATE.SIFT#3, E#1, T#1, C#17:
           Q#1 ∈ SET.FN#9(MAJ_VAL#2, STATE.SIFT#3, E#1, T#1, C#17)
                ≡
           Q#1 ≥ 1
           ∧ Q#1 ≤ MAX_PROCESSORS()
           ∧ POLL()[T#1, REAL_TO_VIRT(STATE.SIFT#3)[Q#1], C#17]
           ∧ MAJ_VAL#2 = DATAFILE(STATE.SIFT#3)[E#1, T#1, Q#1]

SET.FN.AXIOM.10: axiom
        ∀ P#17, T#2, C#18:
           P#17 ∈ SET.FN#10(T#2, C#18)
                ≡
           P#17 ≥ 1 ∧ P#17 ≤ MAX_PROCESSORS() ∧ POLL()[T#2, P#17, C#18]

SET.FN.AXIOM.11: axiom
        ∀ Q#2, MAJ#2, STATE.SIFT#4, E#2, T#3, C#19:
           Q#2 ∈ SET.FN#11(MAJ#2, STATE.SIFT#4, E#2, T#3, C#19)
                ≡
           Q#2 ≥ 1
           ∧ Q#2 ≤ MAX_PROCESSORS()
           ∧ POLL()[T#3, REAL_TO_VIRT(STATE.SIFT#4)[Q#2], C#19]
           ∧ MAJ#2 = DATAFILE(STATE.SIFT#4)[E#2, T#3, Q#2]

SET.FN.AXIOM.12: axiom
        ∀ P#18, T#4, C#20:
           P#18 ∈ SET.FN#12(T#4, C#20)
                ≡
           P#18 ≥ 1 ∧ P#18 ≤ MAX_PROCESSORS() ∧ POLL()[T#4, P#18, C#20]

VC4.H1: axiom
        ACTIVITY_NUM() ≥ 1

VC4.H2: axiom
        ∀ C#13, TI#39:
           CARDINALITY(SET.FN#7(TI#39, C#13))
              = if I_C()[TI#39] then 1 else 3 end if

VC4.H3: axiom
        ∀ TI#41, EI#28:
           ¬(∃ J#32:
                J#32 ≥ 1
                ∧ J#32 < ACTIVITY_NUM()
                ∧ TI#41
                     = SCHED_TABLE()
                         [J#32, SUBFRAME(STATE.SIFT()),
                          CONFIG(STATE.SIFT()),
                          REAL_TO_VIRT(STATE.SIFT())[MY_PROCESSOR()]]
                         .TASKNAME
                ∧ ((SCHED_TABLE()
                      [J#32, SUBFRAME(STATE.SIFT()),
                       CONFIG(STATE.SIFT()),
                       REAL_TO_VIRT(STATE.SIFT())[MY_PROCESSOR()]]
                      .ACTIVITY
                       = VOTE()
                    ∧ EI#28
                         = SCHED_TABLE()
                             [J#32, SUBFRAME(STATE.SIFT()),
                              CONFIG(STATE.SIFT()),
                              REAL_TO_VIRT(STATE.SIFT())
                                [MY_PROCESSOR()]]
                             .ELEM)
                   ∨ SCHED_TABLE()
                       [J#32, SUBFRAME(STATE.SIFT()),
                        CONFIG(STATE.SIFT()),
                        REAL_TO_VIRT(STATE.SIFT())[MY_PROCESSOR()]]
                       .ACTIVITY
                        = DUMMY_VOTE()))
                ⊃
           INPUT(NEXT(STATE.SIFT()))[EI#28, TI#41]
              = INPUT(STATE.SIFT())[EI#28, TI#41]

VC4.H4: axiom
        ∀ TI#42, EI#29:
           (∃ J#33:
               J#33 ≥ 1
               ∧ J#33 < ACTIVITY_NUM()
               ∧ SCHED_TABLE()
                   [J#33, SUBFRAME(STATE.SIFT()), CONFIG(STATE.SIFT()),
                    REAL_TO_VIRT(STATE.SIFT())[MY_PROCESSOR()]]
                   .ACTIVITY
                    = VOTE()
               ∧ TI#42
                    = SCHED_TABLE()
                        [J#33, SUBFRAME(STATE.SIFT()),
                         CONFIG(STATE.SIFT()),
                         REAL_TO_VIRT(STATE.SIFT())[MY_PROCESSOR()]]
                        .TASKNAME
               ∧ EI#29
                    = SCHED_TABLE()
                        [J#33, SUBFRAME(STATE.SIFT()),
                         CONFIG(STATE.SIFT()),
                         REAL_TO_VIRT(STATE.SIFT())[MY_PROCESSOR()]]
                        .ELEM)
                ⊃
           if ∃ MAJ_VAL#8:
                 CARDINALITY(SET.FN#9(MAJ_VAL#8, STATE.SIFT(), EI#29,
                                      TI#42, CONFIG(STATE.SIFT())))
                 *2
                    > CARDINALITY(SET.FN#10(TI#42,
                                            CONFIG(STATE.SIFT())))
              then
                 ∀ MAJ#8:
                    CARDINALITY(SET.FN#11(MAJ#8, STATE.SIFT(), EI#29,
                                          TI#42, CONFIG(STATE.SIFT())))
                    *2
                       > CARDINALITY(SET.FN#12(TI#42,
                                               CONFIG(STATE.SIFT())))
                        ⊃
                    INPUT(NEXT(STATE.SIFT()))[EI#29, TI#42] = MAJ#8
               else
                  INPUT(NEXT(STATE.SIFT()))[EI#29, TI#42] = BOTTOM_VAL()
            end if

VC4.H5: axiom
        ∀ TI#43, EI#30:
            (∃ J#34:
                J#34 ≥ 1
                ∧ J#34 < ACTIVITY_NUM()
                ∧ SCHED_TABLE()
                    [J#34, SUBFRAME(STATE.SIFT()), CONFIG(STATE.SIFT()),
                     REAL_TO_VIRT(STATE.SIFT())[MY_PROCESSOR()]]
                    .ACTIVITY
                     = DUMMY_VOTE()
                ∧ TI#43
                     = SCHED_TABLE()
                         [J#34, SUBFRAME(STATE.SIFT()),
                          CONFIG(STATE.SIFT()),
                          REAL_TO_VIRT(STATE.SIFT())[MY_PROCESSOR()]]
                         .TASKNAME)
            ∧ EI#30 ≥ 1
            ∧ EI#30 ≤ RESULT_SIZE()[TI#43]
                ⊃
            INPUT(NEXT(STATE.SIFT()))[EI#30, TI#43] = BOTTOM_VAL()

VC4.H6: axiom
        ∀ P#14, TI#44, EI#31:
           ¬(P#14 = MY_PROCESSOR()
             ∧ (∃ J#35:
                   J#35 ≥ 1
                   ∧ J#35 < ACTIVITY_NUM()
                   ∧ SCHED_TABLE()
                       [J#35, SUBFRAME(STATE.SIFT()),
                        CONFIG(STATE.SIFT()),
                        REAL_TO_VIRT(STATE.SIFT())[MY_PROCESSOR()]]
                       .ACTIVITY
                        = EXECUTE()
                   ∧ SCHED_TABLE()
                       [J#35, SUBFRAME(STATE.SIFT()),
                        CONFIG(STATE.SIFT()),
                        REAL_TO_VIRT(STATE.SIFT())[MY_PROCESSOR()]]
                       .TASKNAME
                        = TI#44))
                ⊃
           DATAFILE(NEXT(STATE.SIFT()))[EI#31, TI#44, P#14]
              = DATAFILE(STATE.SIFT())[EI#31, TI#44, P#14]

VC4.H7: axiom
        ∀ TI#45, EI#32, INP#7:
           (∃ J#36:
               J#36 ≥ 1
               ∧ J#36 < ACTIVITY_NUM()
               ∧ SCHED_TABLE()
                   [J#36, SUBFRAME(STATE.SIFT()), CONFIG(STATE.SIFT()),
                    REAL_TO_VIRT(STATE.SIFT())[MY_PROCESSOR()]]
                   .ACTIVITY
                    = EXECUTE()
               ∧ SCHED_TABLE()
                   [J#36, SUBFRAME(STATE.SIFT()), CONFIG(STATE.SIFT()),
                    REAL_TO_VIRT(STATE.SIFT())[MY_PROCESSOR()]]
                   .TASKNAME
                    = TI#45)
           ∧ (∀ TASKI#7, J#37, ELEMI#7:
                 J#37 ≥ 1
                 ∧ J#37 ≤ RESULT_SIZE()[TASKI#7]
                 ∧ INPUTS()[J#37, TI#45] = TASKI#7
                 ∧ ¬(TASKI#7 = NULL_TASK())
                      ⊃
                 INP#7[ELEMI#7, J#37]
                    = INPUT(NEXT(STATE.SIFT()))[ELEMI#7, TASKI#7])
                ⊃
           DATAFILE(NEXT(STATE.SIFT()))[EI#32, TI#45, MY_PROCESSOR()]
              = TASK_RESULTS(TI#45, INP#7)[EI#32]

VC4.H8: axiom
        SUBFRAME(NEXT(STATE.SIFT())) = SUBFRAME(STATE.SIFT())

VC4.H9: axiom
        CONFIG(NEXT(STATE.SIFT())) = CONFIG(STATE.SIFT())

VC4.H10: axiom
        ERRORS(NEXT(STATE.SIFT())) = ERRORS(STATE.SIFT())

VC4.H11: axiom
        REAL_TO_VIRT(NEXT(STATE.SIFT())) = REAL_TO_VIRT(STATE.SIFT())

VC4.H12: axiom
        VIRT_TO_REAL(NEXT(STATE.SIFT())) = VIRT_TO_REAL(STATE.SIFT())

VC4.H13: axiom
        ¬(SCHED_TABLE()
            [ACTIVITY_NUM(), SUBFRAME(NEXT(STATE.SIFT())),
             CONFIG(NEXT(STATE.SIFT())),
             REAL_TO_VIRT(NEXT(STATE.SIFT()))[MY_PROCESSOR()]]
            .ACTIVITY
             = 0)

VC4.H14: axiom
        ACTIVITY_NUM() ≤ MAX_ACTIVITIES()

VC4.H15: axiom
        SCHED_TABLE()
          [ACTIVITY_NUM(), SUBFRAME(NEXT(STATE.SIFT())),
           CONFIG(NEXT(STATE.SIFT())),
           REAL_TO_VIRT(NEXT(STATE.SIFT()))[MY_PROCESSOR()]]
          .ACTIVITY
           = VOTE()

VC4.H16: axiom
        if ∃ MAJ_VAL#9:
              CARDINALITY(SET.FN#9(MAJ_VAL#9, NEXT(STATE.SIFT()),
                                   SCHED_TABLE()
                                     [ACTIVITY_NUM(),
                                      SUBFRAME(NEXT(STATE.SIFT())),
                                      CONFIG(NEXT(STATE.SIFT())),
                                      REAL_TO_VIRT(NEXT(STATE.SIFT()))
                                        [MY_PROCESSOR()]]
                                     .ELEM,
                        SCHED_TABLE()
                          [ACTIVITY_NUM(),
                           SUBFRAME(NEXT(STATE.SIFT())),
                           CONFIG(NEXT(STATE.SIFT())),
                           REAL_TO_VIRT(NEXT(STATE.SIFT()))
                             [MY_PROCESSOR()]]
                          .TASKNAME,
                        CONFIG(NEXT(STATE.SIFT()))))
   *2
      > CARDINALITY(SET.FN#10(SCHED_TABLE()
                                [ACTIVITY_NUM(),
                                 SUBFRAME(NEXT(STATE.SIFT()))
                                 ,
                                 CONFIG(NEXT(STATE.SIFT())),
                                 REAL_TO_VIRT
                                  (NEXT(STATE.SIFT()))
                                    [MY_PROCESSOR()]]
                                .TASKNAME,
                              CONFIG(NEXT(STATE.SIFT()))))
then
   ∀ MAJ#9:
      CARDINALITY(SET.FN#11(MAJ#9, NEXT(STATE.SIFT()),
                            SCHED_TABLE()
                              [ACTIVITY_NUM(),
                               SUBFRAME(NEXT(STATE.SIFT())),
                               CONFIG(NEXT(STATE.SIFT())),
                               REAL_TO_VIRT
                                (NEXT(STATE.SIFT()))
                                  [MY_PROCESSOR()]]
                              .ELEM,
                            SCHED_TABLE()
                              [ACTIVITY_NUM(),
                               SUBFRAME(NEXT(STATE.SIFT())),
                               CONFIG(NEXT(STATE.SIFT())),
                               REAL_TO_VIRT
                                (NEXT(STATE.SIFT()))
                                  [MY_PROCESSOR()]]
                              .TASKNAME,
                            CONFIG(NEXT(STATE.SIFT()))))
      *2
         > CARDINALITY(SET.FN#12(SCHED_TABLE()
                                   [ACTIVITY_NUM(),
                                    SUBFRAME
                                     (NEXT(STATE.SIFT())),
                                    CONFIG
                                     (NEXT(STATE.SIFT())),
                                    REAL_TO_VIRT
                                     (NEXT(STATE.SIFT()))
                                       [MY_PROCESSOR()]]
                                   .TASKNAME,
                                 CONFIG(NEXT(STATE.SIFT()))))
           ⊃
      INPUT(NEXT(NEXT(STATE.SIFT())))
        [SCHED_TABLE()
           [ACTIVITY_NUM(), SUBFRAME(NEXT(STATE.SIFT())),
            CONFIG(NEXT(STATE.SIFT())),
            REAL_TO_VIRT(NEXT(STATE.SIFT()))[MY_PROCESSOR()]]
           .ELEM,
         SCHED_TABLE()
```

```
                      [ACTIVITY_NUM(), SUBFRAME(NEXT(STATE.SIFT())),
                       CONFIG(NEXT(STATE.SIFT())),
                       REAL_TO_VIRT(NEXT(STATE.SIFT()))[MY_PROCESSOR()]]
                      .TASKNAME]
                    = MAJ#9
           else
              INPUT(NEXT(NEXT(STATE.SIFT())))
                [SCHED_TABLE()
                   [ACTIVITY_NUM(), SUBFRAME(NEXT(STATE.SIFT())),
                    CONFIG(NEXT(STATE.SIFT())),
                    REAL_TO_VIRT(NEXT(STATE.SIFT()))[MY_PROCESSOR()]]
                   .ELEM,
                 SCHED_TABLE()
                   [ACTIVITY_NUM(), SUBFRAME(NEXT(STATE.SIFT())),
                    CONFIG(NEXT(STATE.SIFT())),
                    REAL_TO_VIRT(NEXT(STATE.SIFT()))[MY_PROCESSOR()]]
                   .TASKNAME]
                 = BOTTOM_VAL()
        end if

VC4.H17: axiom
        ∀ TI#46, EI#33:
           ¬(TI#46
                = SCHED_TABLE()
                    [ACTIVITY_NUM(), SUBFRAME(NEXT(STATE.SIFT())),
                     CONFIG(NEXT(STATE.SIFT())),
                     REAL_TO_VIRT(NEXT(STATE.SIFT()))[MY_PROCESSOR()]]
                    .TASKNAME)
           ∨ ¬(EI#33
                  = SCHED_TABLE()
                      [ACTIVITY_NUM(), SUBFRAME(NEXT(STATE.SIFT())),
                       CONFIG(NEXT(STATE.SIFT())),
                       REAL_TO_VIRT(NEXT(STATE.SIFT()))[MY_PROCESSOR()]]
                      .ELEM)
                ⊃
           INPUT(NEXT(NEXT(STATE.SIFT())))[EI#33, TI#46]
              = INPUT(NEXT(STATE.SIFT()))[EI#33, TI#46]

VC4.H18: axiom
        SUBFRAME(NEXT(NEXT(STATE.SIFT())))
           = SUBFRAME(NEXT(STATE.SIFT()))

VC4.H19: axiom
        CONFIG(NEXT(NEXT(STATE.SIFT()))) = CONFIG(NEXT(STATE.SIFT()))

VC4.H20: axiom
        DATAFILE(NEXT(NEXT(STATE.SIFT())))
           = DATAFILE(NEXT(STATE.SIFT()))

VC4.H21: axiom
        ERRORS(NEXT(NEXT(STATE.SIFT()))) = ERRORS(NEXT(STATE.SIFT()))

VC4.H22: axiom
        REAL_TO_VIRT(NEXT(NEXT(STATE.SIFT())))
           = REAL_TO_VIRT(NEXT(STATE.SIFT()))

VC4.H23: axiom
        VIRT_TO_REAL(NEXT(NEXT(STATE.SIFT())))
           = VIRT_TO_REAL(NEXT(STATE.SIFT()))
```

```
VC4.C9: formula
        ∀ TI#50, EI#35:
           (∃ J#39:
               J#39 ≥ 1
               ∧ J#39 < ACTIVITY_NUM()+1
               ∧ SCHED_TABLE()
                   [J#39, SUBFRAME(STATE.SIFT()), CONFIG(STATE.SIFT()),
                    REAL_TO_VIRT(STATE.SIFT())[MY_PROCESSOR()]]
                   .ACTIVITY
                    = VOTE()
               ∧ TI#50
                    = SCHED_TABLE()
                        [J#39, SUBFRAME(STATE.SIFT()),
                         CONFIG(STATE.SIFT()),
                         REAL_TO_VIRT(STATE.SIFT())[MY_PROCESSOR()]]
                        .TASKNAME
               ∧ EI#35
                    = SCHED_TABLE()
                        [J#39, SUBFRAME(STATE.SIFT()),
                         CONFIG(STATE.SIFT()),
                         REAL_TO_VIRT(STATE.SIFT())[MY_PROCESSOR()]]
                        .ELEM)
                ⊃
           if ∃ MAJ_VAL#10:
                 CARDINALITY(SET.FN#9(MAJ_VAL#10, STATE.SIFT(), EI#35,
                                      TI#50, CONFIG(STATE.SIFT())))
                 *2
                    > CARDINALITY(SET.FN#10(TI#50,
                                            CONFIG(STATE.SIFT())))
              then
                 ∀ MAJ#10:
                    CARDINALITY(SET.FN#11(MAJ#10, STATE.SIFT(), EI#35,
                                          TI#50, CONFIG(STATE.SIFT())))
                    *2
                       > CARDINALITY(SET.FN#12(TI#50,
                                               CONFIG(STATE.SIFT())))
                       ⊃
                    INPUT(NEXT(NEXT(STATE.SIFT())))[EI#35, TI#50]
                       = MAJ#10
              else
                 INPUT(NEXT(NEXT(STATE.SIFT())))[EI#35, TI#50]
                    = BOTTOM_VAL()
           end if
```

CHAPTER 16

FORMAL DEFINITION OF HDM METHODOLOGY

A Formal Semantics for the SRI Hierarchical Program Design Methodology

Robert S. Boyer and J S. Moore

ABSTRACT

This document is intended for the reader with modest abilities in understanding formal logical talk, an intuitive understanding of the SRI Methodology, and a large measure of patience. We present a formal statement of what it means to use (a subset of) the methodology. In particular, we formally define what it means to say that some specified module exists and what it means to say that another module is correctly implemented on top of it. We pay no attention to motivation, either of the methodology or of our formal development of it. Instead, we concentrate entirely upon mathematical succinctness and precision. We conclude with a discussion of how to use certain INTERLISP programs which implement our formal definitions. Among these are a program which we allege generates Floyd-like verification conditions sufficient to imply the correctness of a module implementation.

ACKNOWLEDGMENTS

The fundamental methodology ideas here are those of Parnas and Robinson. We have only explicitly articulated a few things that were previously implicit and formalized much of what has long been intuitively but precisely talked about in the Computer Science Laboratory.

Thanks to W.W. Bledsoe, K. Levitt, L. Robinson, and J. Spitzen for hearing a version of the following details. They receive all blame for any remaining errors.

I SOME CONVENTIONS

We assume the existence of the non-negative integers, finite sequences, pairs, and sets. We assume that no integer is a sequence or a pair, and that no sequence is a pair. We enumerate the members of a sequence starting from 1.

When we write the double quotation mark followed by a sequence of ASCII characters not including the double quotation mark followed by another double quotation mark, we are refering to the sequence whose members are the ASCII numbers of the characters between the double quotation marks.

II FORMULAS

A *word* is a non-empty sequence of integers that are members of "0123456789.-*'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".

An *identifier* is a word one of whose characters is not a member of "0123456789".

A *function symbol* is an identifier.

F is a *formula* if F is a member of (the intersection of all sets S containing the words and such that any sequence whose first member is a function symbol and whose other members are in S is a member of S).

All but the first member of s is defined (when s is a finite non-empty sequence) to be the sequence whose length is one less than the length of s and whose i^{th} member (for i from 1 to (the length of s)-1) is the $i+1^{th}$ member of s.

The *arguments* of a formula not a word is all but the first member of the formula.

A *parameter sequence* is "NIL" or a sequence of distinct identifiers.

A *parameter sequence** is "NIL" or a sequence of distinct identifiers none of which are "STATE", "NEWSTATE", "NEWVECT", nor contain "'" nor "*".

ID(x) is (if x is "NIL" then the empty sequence, otherwise x).

The *variables* of a formula is defined recursively as:

> If the formula is a word, then if the formula is an identifier, then the set whose only member is the identifier, otherwise the empty set,
>
> Otherwise, the union of the variables of the members of the arguments of the formula.

The *result of substituting* a[1] *for* b[1], a[2] *for* b[2], ..., *and* a[n] *for* b[n] *in* the formula c is defined recursively as:

> If for some i c is b[i], then a[the first i such that b[i] is c],
>
> Otherwise, if c is a word then c,
>
> Otherwise, the sequence whose i^{th} element is the result of

substituting a[1] for b[1], a[2] for b[2], ..., and a[n] for b[n] in the i^{th} member of c (for i from 1 to the length of c).

We will also use the phrase *result of replacing* b[1] *by* a[1], ... *and* b[n] *by* a[n] to mean the result of substituting a[1] for b[1], ... and a[n] for b[n].

S *is a presentation of* the formula F is defined recursively as

If F is a word, then S is F,

Otherwise, S is the concatenation in order of the members of a sequence of odd length whose first member is "(", whose second member is the function symbol of F, whose last member is ")", whose $2*i^{th}$ member is a presentation of the i^{th} member of F and whose $2*i-1^{th}$ member is a non-empty sequence of spaces and carriage-returns, for i from 2 to the length of F.

It is a theorem that any sequence is a presentation of at most one formula.

The *formula of* a sequence is the unique formula (if there is one) such that the sequence is a presentation of the formula.

The *application* of f to a sequence s is the sequence whose length is one greater than that of s, whose first element is f, and whose $i+1^{st}$ element is the i^{th} element of s, for i from 1 to the length of s.

The *application** of f to x[1], ..., and x[n] is the sequence of length n+1 whose first member is f and whose $i+1^{st}$ member is x[i], for i from 1 to n.

If FS is a sequence of formulas, we define the *conjunction* of FS recursively as:

If FS is empty then the formula of "(TRUE)",

Otherwise the application* of "AND" to the first member of FS and (the conjunction of all but the first member of FS).

If FS is a sequence of formulas, we define the *disjunction* of FS recursively as:

If FS is empty then the formula of "(FALSE)",

Otherwise the application* of "OR" to the first member of FS and (the disjunction of all but the first member of FS).

The *implication* of formulas F1 and F2 is the application* of "IMPLIES" to F1 and F2.

The *equation* of E1 and E2 is the application* of "EQUAL" to E1 and E2.

III INTERPRETATIONS AND MODELS

I is an *interpretation of* the set S (where S is a set of pairs consisting of a function symbol and a non-negative integer) if and only if I is a pair consisting of a set D, called the domain of I, and a function G whose domain is S, and for each pair <fn, ar> in S, G(<fn,ar>) is a function on D^{ar} to D.

F1 *is a subformula* of F2 is defined recursively as

If F1 is F2 then true,

Otherwise, if F2 is a word, then false,

Otherwise, F1 is a subformula of some member of the arguments of F2.

The *arity* of a formula not a word is the length of the formula minus 1.

The *arity set* of a formula F is the set of pairs <fn, ar> such that some sub-formula of F not a word has function symbol fn and arity ar.

E is an *environment* if and only if E is a function whose domain is the set of identifiers.

If I is an interpretation whose domain includes the integers and words, E is an environment whose range is a subset of the domain of I, and the domain of the second member of I is a superset of the arity set of a formula F, then the *meaning of F with respect to* I *and* E is defined recursively as

If F is an identifier, then E(F)

Otherwise, if F is a word, then the non-negative integer represented by F in decimal,

Otherwise, if F can be obtained by replacing "X" by a word L in the formula of "(QUOTE X)", then if L is an identifier, then L, otherwise the non-negative integer represented by L in decimal.

Otherwise, if F is the application* of fn to x[1], x[2], ... x[n], then G(<fn, n>)(v[1],v[2],...,v[n]), where v[i] is the meaning of x[i] with respect to I and E.

Let TRUE and FALSE be distinct objects not integers, pairs, or sequences.

The *standard domain* is the smallest set S containing TRUE, FALSE, the words, the non-negative integers, and the pair of each element in S.

I is a *standard interpretation* if and only if the domain of I is a superset of the standard domain

and

G(<"IF",3>)(x, y, z) is y if x is not FALSE and z otherwise.

G(<"EQUAL",2>)(x, y) is TRUE if x is y and FALSE otherwise.

G(<"TRUE",0>)() is TRUE

G(<"FALSE",0>)() is FALSE

G(<"NUMBERP",1>)(x) is TRUE if x is a non-negative integer and FALSE otherwise.

G(<"SUB1", 1>)(x) is x-1 if x is a positive integer and 0 otherwise.

G(<"ADD1",1>)(x) is x+1 if x is a non-negative integer and 2 otherwise.

G(<"LITATOM",1>)(x) is TRUE if x is an identifier and FALSE otherwise.

G(<"LISTP", 1>)(x) is TRUE if x is a pair and FALSE otherwise,

G(<"CONS", 2>)(x, y) is the pair (x, y),

G(<"CAR", 1>)(x) is u if x is the pair (u, v) for any u and v and "NIL" otherwise.

G(<"CDR", 1>)(x) is v if x is the pair (u, v) for any u and v and "NIL" otherwise.

I *is a model of FS* provided that I is a standard interpretation, FS is a set of formulas, the domain of the second member of I is the union of the arity-sets of the members of FS together with the set {<"IF",3>,<"EQUAL",2>,<"TRUE",0>,<"FALSE",0>,<"NUMBERP",1>,<"ADD1",1><"SUB1",1>, <"LITATOM",1>,<"LISTP",1>,<"CONS",2>,<"CAR",1>,<"CDR",1>}, and for every environment E whose range is a subset of the domain of I, the meaning of each member of FS with respect to I and E is not FALSE.

TH *follows* from FS provided that

TH is a formula,

FS is a set of formulas, and

every model of FS is a model of FS union singleton TH.

D *is an admissible definitional extension* of FS provided that

D results from substituting a function symbol, fn-symbol, for "fn-symbol" a parameter sequence, param-seq, for "param-seq" and a formula, body, for "body" in the formula of "(DEFN1 fn-symbol param-seq body)", and

for each model I of FS there exists a model I' of FS union singleton (the result of substituting (the application of fn-symbol to ID(param-seq)) for "LHS", and body for "body" in the formula of "(EQUAL LHS body)") such that the domain of I is the domain of I' and the second member of I is a subset of the second member of I'.

An instruction is an application* of "ASSUME" to a formula or the application* of "DEFN1" to a function symbol, a parameter sequence, and a formula.

The theory resulting from the sequence of instructions instrs is defined recursively as

If instrs is empty, then the empty set

Otherwise, if the last instruction in instrs is the application* of "ASSUME" to a formula fm, then the set whose members are fm and the members of the theory resulting from all but the last member of instrs

Otherwise, if the last instruction in instrs is the application* of "DEFN1" to fn-symbol, param-seq, and body, and that instruction is a definition admissible to the theory resulting from all but the last member of instrs, then the set resulting from adding (the application* of "EQUAL" to (the application of fn-symbol to ID(param-seq)) and body) to the theory resulting from all but the last member of instrs.

Otherwise the theory resulting from all but the last member of instrs.

We say that (=> instrs fm) when fm follows from the theory resulting from instrs.

IV MODULES

Let the reader beware that we will sometimes present definitions in the opposite order from that in which they might normally be presented. Our purpose in doing this is to avoid beginning with many apparently silly subsidiary concepts.

A module is the application* of an identifier, (called the module.name), to (the application of "VFNS" to a sequence of vfn.specifications (see below)) and (the application of "OVFNS" to a sequence of ovfn.specifications (see below)).

A vfn.specification is an application* of an identifier (called the vfn.name) to a parameter sequence* (called the vfn.parameter.sequence) and (an application* of "ASSERTION" to a formula (called the vfn.assertion)).

An ovfn.specification is an application* of an identifier (called the ovfn.name) to a parameter sequence* (called the ovfn.parameter.sequence), (an application* of "ASSERTION" to a formula (called the ovfn.assertion)), (an application* of "VALUE" to a formula (called the ovfn.value)), and (an application* of "EXCEPTIONS" to an exception.sequence (called the ovfn.exception.sequence)(see below)).

An exception.sequence is a sequence of formulas (including the identifier "RESOURCE.ERROR", which will have a special role).

The type.function.name of a module is the identifier resulting from concatenating the module.name with ".TYPE.FN". The *invariant.function.name* of a module is the identifier resulting from concatenating the module.name with ".INVRNT". The *initial.state.name* of a module is the identifier resulting from the concatenation of "INITIAL.", the module.name, and ".STATE". The invoke.function.name of a module is the concatenation of "INVOKE." and the module.name.

The *ovfn.correctness.assertion1* of the sequence of exceptions exceptions, the non-negative integer n, the ovfn.specification ovfnspec and the module module is defined recursively as

If exceptions is empty, then the conjunction of the ovfn.assertion of ovfnspec, the equation of the formula of "(ANSWER NEWVECT)" with the ovfn.value of ovfnspec, the formula of "(EQUAL (RETURN.NUMBER NEWVECT) 0)", and the application* of the type.function.name of module to the formula of "(ANSWER NEWVECT)",

Otherwise, if the first member of exceptions is the formula of "RESOURCE.ERROR" then the disjunction of (the conjunction of the null.assertion (see below) of module with the equation of the formula of "(RETURN.NUMBER NEWVECT)" and the shortest string of decimal digits whose meaning is n) and the ovfn.correctness.assertion1 of all but the first member of exceptions, n+1, ovfnspec, and module,

Otherwise, the application* of "IF" to the first member of exceptions, (the conjunction of the null.assertion of the module and (the equation of the formula of "(RETURN.NUMBER NEWVECT)" with the shortest string of decimal digits whose meaning is n)), and the ovfn.correctness.assertion1 of all but the first member of exceptions, n+1, ovfnspec, and module.

The *ovfn.correctness.assertion* of an ovfn.specification ovfnspec and a module module is the implication of

the conjunction of

the equation of "NEWVECT" and the application of the ovfn.name of ovfnspec to the concatenation of the ovfn.parameter.sequence of ovfnspec with the sequence whose only member is "STATE",

the formula of "(EQUAL NEWSTATE (STATE NEWVECT))", and the conjunction of the sequence of application*s of the type.function.name of module to each of the identifiers in the ovfn.parameter.sequence of ovfnspec

and the conjunction of

the application* of the invariant.function.name of module to "NEWSTATE" and

the ovfn.correctness.assertion1 of the exceptions of ovfnspec, 1, ovfnspec, and module.

The *module.correctness.assertion* of a module module is the conjunction of the concatenation of the sequence, in order, of the vfn.assertions of the vfn.specifications of module and

the sequence whose two members are

> the application* of the invariant.function.name of module to the (application of the initial.state.name of module to the empty sequence)) and
>
> the conjunction of the sequence in order of the ovfn.correctness.assertions of each ovfn.specification of module with respect to module.

The *null.assertion* of a module module is the conjunction of the sequence, in order, for each vfn.specification vs of module the equation of (the application of the vfn.name of vs to (the concatenation of the renamed vfn.parameter.sequence (see below) of vs to the sequence whose only member is "STATE")) with (the application of the vfn.name of vs (to the the concatenation of the renamed vfn.parameter.sequence of vs to the sequence whose only member is "NEWSTATE")).

The *renamed vfn.parameter.sequence of* vs is a sequence of identifiers obtained, in order, from vs by concatenating each identifier in vs with the sequence whose only member is "'".

V MACHINES

The *invoke function definition* of a module module is the result of replacing in the formula of "(DEFN1 FN (RHS ENVIRONMENT STATE) BODY)" "FN" with the invoke.function.name of module and "BODY" with the invoke.code of the ovfn.specifications of module, where the *invoke.code* of a sequence of specifications s is defined recursively as:

> If s is empty then the formula of "(MAKE.VECT 0 0 0 0)",
>
> Otherwise, the result of substituting in the formula of "(IF (EQUAL (CAR RHS) NAME) CALL REST)" the application* of "QUOTE" to the ovfn.name of the first member of s for "NAME", the calling pattern (see below) of the first member of s for "CALL", and the invoke.code of all but the first member of s for "REST".

The *calling pattern* of an ovfn.specification ovfnspec is the application of the ovfn.name of ovfnspec to (the concatenation of the (RHS-arguments (see below) of the length of the ovfn.parameter.sequence of ovfnspec) with the sequence whose only member is "STATE").

The *RHS-arguments* of n is the sequence of length n containing, in the i^{th} position, the application* of "CAR" to the i^{th} CDR of RHS (see below).

The i^{th} *CDR of RHS*, where i a non-negative integer, is "RHS" if i is 0, and is the application* of "CDR" to the i-1st CDR of RHS otherwise.

The *definition of the run1 function of* the module module is the result of substituting the invoke.function.name of module for "INVOKE" and (the concatenation "RUN1." with the module.name of module) for "RUN1" in the formula of

```
"(DEFN1 RUN1 (CODE LOC ENVIRONMENT STATE)

(IF

  (EQUAL (CAAR LOC)

         (QUOTE GO))

  (RUN1 CODE (GOTO (CADAR LOC)

                   CODE)

        ENVIRONMENT STATE)

  (IF

    (EQUAL (CAAR LOC)

           (QUOTE SWITCH))

    (RUN1 CODE (GOTO (IF (VALUE (QUOTE AC1)

                                ENVIRONMENT)

                         (CADAR LOC)

                         (CADDAR LOC))

                     CODE)

          ENVIRONMENT STATE)

    (IF

      (EQUAL (CAAR LOC)

             (QUOTE IMMEDIATE))

      (RUN1 CODE (CDR LOC)

            (BIND (QUOTE AC1)

                  (CADAR LOC)

                  ENVIRONMENT)

            STATE)

      (IF

        (EQUAL (CAAR LOC)

               (QUOTE ASSIGN))

        (IF
(LISTP (CADDAR LOC))
(RUN1
  CODE
  (NTH
    LOC
    (ADD1 (RETURN.NUMBER
            (INVOKE (CADDAR LOC)
                    ENVIRONMENT STATE)))
    )
  (IF
    (EQUAL
      0
      (RETURN.NUMBER
        (INVOKE (CADDAR LOC)
                ENVIRONMENT STATE)))
    (BIND (CADAR LOC)
          (ANSWER
            (INVOKE (CADDAR LOC)
                    ENVIRONMENT STATE))
          ENVIRONMENT)
    ENVIRONMENT)
  (STATE (INVOKE (CADDAR LOC)
                 ENVIRONMENT STATE)))
(RUN1 CODE (CDR LOC)
      (BIND (CADAR LOC)
            (VALUE (CADDAR LOC)
                   ENVIRONMENT)
            ENVIRONMENT)
      STATE))
```

```
(IF (EQUAL (CAAR LOC)
           (QUOTE ANSWER))
    (MAKE.VECT (VALUE (QUOTE AC1)
                      ENVIRONMENT)
               ENVIRONMENT STATE 0)
    (IF (EQUAL (CAAR LOC)
               (QUOTE EXCEPTION))
        (MAKE.VECT 0 ENVIRONMENT STATE
                   (CADAR LOC))
        (RUN1 CODE (CDR LOC)
              ENVIRONMENT STATE)))))))".
```

The *run.function.name* of a module is the concatenation of "RUN." and the module.name of the module.

The *definition of the run function of* the module module is the result of substituting the run.function.name of module for "RUN" and the concatenation of "RUN1." with the module.name of module for "RUN1" in the formula of

```
"(DEFN1 RUN (CODE ENVIRONMENT STATE)
   (RUN1 CODE CODE ENVIRONMENT STATE))"
```

VI IMPLEMENTATIONS

An *implementation* of a module upper.module on a module lower.module is a sequence consisting (in order) of an ovfn.implementation of upper.module (see below), a vfn.implementation of upper.module (see below), the application* of "INITIALIZATION" to a formula (called the initialization.prog), the application* of "TYPE" to a formula (called the type.formula), and the application* of "INVRNT" to a formula (called the invrnt.formula).

An *ovfn.implementation* of upper.module is a sequence beginning with "OVFNS" which contains for each ovfn.specification ov of upper module exactly one application* of the ovfn.name of ov to a formula (called the ovfn.implementation.formula).

A *vfn.implementation* of upper.module is a sequence beginning with "VFNS" which contains for each vfn.specification vfnspec of upper.module exactly one application* of the vfn.name to a formula (called the vfn.implementation.formula).

The *implementation definitions of* an implementation implementation, a module upper.module, and a module lower.module is the sequence of formulas consisting of the result of substituting the invrnt.formula of the implementation for "fm" and the invariant.function.name of the upper module for "name" in the formula of "(DEFN1 name (STATE) fm)",

the result of substituting the type.function.name of the upper.module for "name" and the type.formula of the implementation for "fm" in the formula of "(DEFN1 name (OBJ) fm)",

for each vfn.specification vs of the upper.module, the application* of "DEFN1" to the vfn.name of vs, the concatenation of ID(the vfn.parameter.sequence of vs) to the sequence whose only member is "STATE", and the vfn.implementation.formula of the vfn.name of vs in the vfn.implementations of implementation, and

the invoke function definition of lower.module,

the run1 function definition of lower.module,

the run function definition of lower.module,

the application* of "DEFN1" to the initial.state.name of upper.module, "NIL", and (the application* of the run.function.name of lower.module to the initialization.prog of implementation, the formula of "(EMPTY.ENVIRONMENT)", and (the application of the initial.state.name of lower.module to the empty.sequence))).

for each ovfn.specification ov of the upper.module, the application* of "DEFN1" to the ovfn.name of ov, the concatenation of ID(the ovfn.parameter.sequence of ov) to the sequence whose only member is "STATE", and (the application* of the run.function.name of lower.module to the ovfn.implementation.formula of the ovfn.name of ov in implementation, the initial environment of ID(the ovfn.parameter.sequence of ov), and "STATE").

The _initial environment_ of a parameter sequence l is defined recursively as:

If l is empty, then the formula of "(EMPTY.ENVIRONMENT)",

Otherwise, the result of replacing "NAME" with the first of l and "REST" with the initial environment of all but the first of l in the formula of "(BIND (QUOTE NAME) NAME REST)".

VII WHAT IT MEANS TO USE THE METHODOLOGY

To use the methodology upon upper.module, lower.module, and implementation _starting from_ the sequence of instructions instrs1 _and ending with_ the sequence of instructions instrs2

is to establish (=> hyps (the module.correctness.assertion of upper.module)) where hyps is the concatenation of the auxiliary definitions (see below),

instrs1,

the implementation definitions of implementation, upper.module, and lower.module,

the application* of "ASSUME" to the module.correctness.assertion of lower.module, and

instrs2.

The *auxiliary definitions* are the formulas of the following strings:

```
"(DEFN1 AND (P Q) (IF P Q (FALSE)))",

"(DEFN1 OR (P Q) (IF P (TRUE) Q))",

"(DEFN1 IMPLIES (P Q) (IF P Q (TRUE)))",

"(DEFN1 MAKE.VECT (X Y Z U) (CONS X (CONS Y (CONS Z

     (CONS U  (QUOTE NIL))))))",

"(DEFN1 RETURN.NUMBER (X) (CAR (CDR (CDR (CDR X)))))",

"(DEFN1 STATE (X) (CAR (CDR (CDR X))))",

"(DEFN1 ENVIRONMENT (X) (CAR (CDR X)))",

"(DEFN1 ANSWER (X) (CAR X))",

"(DEFN1 EMPTY.ENVIRONMENT () (QUOTE NIL))",

"(DEFN1 BIND (X Y Z) (CONS (CONS X Y) Z))",

"(DEFN1 GOTO (X L)

  (IF (LISTP L)

      (IF (EQUAL (CAR L) X)

          L

          (GOTO X (CDR L)))

      (QUOTE NIL)))",

"(DEFN1 NTH (L I)

  (IF (LISTP L)

      (IF (EQUAL I 0)

          L

          (NTH (CDR L) (SUB1 I)))

      (QUOTE NIL)))", and
```

```
"(DEFN1 VALUE (X Y) (IF (LISTP Y)
                        (IF (EQUAL X (CAR (CAR Y)))
                            (CDR (CAR Y))
                            (VALUE X (CDR Y)))
                        0))".
```

Appendix A
USING <THOR>RUN

This appendix discusses the use of the verification condition generator and semantics generators for the subset of the methodology just described. We here, and in the next appendix, return to the ordinary mathematical and programming style of requiring the reader to understand what we mean rather than what we say.

The file <THOR>RUN is Interlisp-10 code. There are in that file three important functions, USE.THE.METHODOLOGY, USE.FLOYD, and VCG.COMPILE. We will discuss the use of these functions but not their implementation.

1. USE.THE.METHODOLOGY

The function USE.THE.METHODOLOGY takes as input three arguments, an UPPER.MODULE, a LOWER.MODULE, and an IMPLEMENTATION. The form of these arguments should be obvious to the experienced Interlisp user from the foregoing discussion of the methodology. (We could of course spell things out precisely here but that would require linking up with the Interlisp virtual machine specification.) The result of calling USE.THE.METHODOLOGY is a formula of the form (=> hyp concl) that is the same as the formula described in the previous section except that instrs1, instrs2, and the auxiliary definitions are omitted. This statement is correct with the following reservations:

> The type.function.definition of an implementation is coerced to include as admissible objects (TRUE), (FALSE), (UNDEF), 0, and all numbers between 0 and (MAX.NO).
>
> An omitted ovfn.value is taken to be 0.
>
> An omitted ovfn.assertion is taken to be the null.assertion.
>
> An omitted ovfn.exception.sequence is taken to be the empty sequence.
>
> An omitted vfn.assertion is taken to be (TRUE).
>
> The definition of RUN is enhanced as if the lower.module included among its ovfn.specifications the following:

```
(TRUE NIL (VALUE (TRUE)))

(FALSE NIL (VALUE (FALSE)))

(UNDEF NIL (VALUE (UNDEF)))

(EQUAL (X Y)

       (VALUE (EQUAL X Y)))

(ZEROP (X)

       (VALUE (EQUAL X 0)))

(GREATERP (X Y)
          (VALUE (GREATERP X Y)))
(LESSP (X Y)
       (VALUE (LESSP X Y)))
(ADD1 (X)
      (EXCEPTIONS (GREATERP (ADD1 X)
                            (MAX.NUMBER)))
      (VALUE (ADD1 X)))
(PLUS (X Y)
      (EXCEPTIONS (GREATERP (PLUS X Y)
                            (MAX.NUMBER)))
      (VALUE (PLUS X Y)))
```

2. USE.FLOYD

USE.FLOYD takes the same arguments as does USE.THE.METHODOLOGY and it returns a list of things that are either definitions or =>'s. It is here alleged that if one takes the theory resulting from instrs1, the definitions in the output, and instrs2 and in that theory checks that A follows from B when (=> A B) occurs in the output, then one has correctly used the methodology, subject to the reservations made in the subsection on USE.THE.METHODOLOGY.

In order to obtain from USE.FLOYD some =>'s that are provable, one must exercise considerable ingenuity in inventing what are called Floyd assertions and certain numeric formulas called "clocks" here. USE.FLOYD insists that every program has a formula of the form (ASSERT fm cl) cutting every loop. Variables in these formulas are taken to refer to the current value of the variable if it is anywhere used as a program variable in the program. Otherwise, the reference is taken to be universally quantified. The term (START) is taken to be a reference to the state at the beginning of the execution of the program, and the term (X*) is taken to be a reference to the value of the variable X at the beginning of the execution of the program.

The formula fm in an assertion is used in the usual way as a Floyd-assertion. The formula cl is used as a numerically valued expression and the verification condition generator produces formulas requiring that these expressions get smaller on each path. If proved, these relations insure that the program terminates.

3. VCG.COMPILE

Because it is painful for some people to write in the assembly language that RUN works on, both USE.THE.METHODOLOGY and USE.FLOYD call the function VCG.COMPILE to translate all implementation programs into the assembly language used. VCG.COMPILE is given the program and the lower module as arguments. For convenience we have defined VCG.COMPILE to translate a LISP-like language, containing COND, PROG, CLISP, etc. The user is invited to invent his own language.

It should be observed that VCG.COMPILE is essentially a semantics for one's programming language. However, it should be realized that the intent is to use the "assembly" code to get a computer running. It would certainly be easier to write a correct assembler and loader for RUN than to prove the correctness of the implementation of one's programming language. The beauty of this approach is that it does not matter if there are "mistakes" in one's compiler. Those "mistakes" cannot caůse incorrectly generated code to be proven correct when it is not.

Appendix B
QUANTIFICATION

The reader will have noted the omission of quantified expressions from our formalization of the methodology. We here explain how one can easily obtain the effects he wants.

Suppose that one wished to write a specification in which some assertion included a subformula (SOME X (P X Y)). Then let him simply invent a new function, say FOO, and conjoin to his assertion (IMPLIES (P X Y) (P (FOO Y) Y)) and let him use (P (FOO Y) Y) where he wanted to use the SOME.

Since ALL can be defined in terms of SOME, the reader can see how to handle ALL.

The construct LET can be handled by a similar hack. Suppose that one wished to use the construct (LET X (P X Y) (Q X Y)). Then, let him just add to his assertion, for some new FOO, the conjunct (IMPLIES (P X Y) (P (FOO Y) Y)), and let him use (IF (P (FOO Y) Y) (Q (FOO Y) Y) (UNDEF)) where he intended to use the LET expression.

It should be observed that using these "Skolem" functions explicitly will have a salient effect upon one's attempts at proving the correctness of his implementations. After all, when one uses a LET expression in a specification, he had better have in mind an explicit object in his implementation.

One more observation is required. If the quantified expression is used in the VALUE part of an ovfn, then the conjunct ought to be conjoined to the assertion.

Appendix C
A MISCELLANY

1. Where Derived V-functions went

The reader may have noticed that we have omitted derived V-functions from our subset of the methodology. In fact, all the known uses for derived V-functions can be had by simply putting the definitions at the appropriate places in the process of using the methodology. The derived V-functions of the lower.module ought to occur simply as definitions in instrsl. The derived V-functions of the upper.module ought to occur right after the definitions of the V-functions in the implementation definitions.

2. Naming clashes

If the user insists upon the previous practice of having some of his V-functions be also OV-functions, he is going to suffer the consequence of have the implementation definitions contain two definitions of the same concept that are radically different.

3. Interfaces

The user familiar with the methodology may ask where the interfaces, those combinations of modules that previously were the main constituents of hierarchies, have gone. It is said that interfaces were a convenience, and that their exact semantics could have always been obtained by writing an appropriately large module. In the interest of semantic simplicity, then, we have imposed this burden of writing upon the user. It ought not to be hard actually to write the program that will merge modules together. It is undoubtedly safer to take this course than to meddle with constructing a new methodology semantics and a new verification condition generator.

4. Deus Ex Machina

Lovers of abstract machine interpreters will note with regret that we have entirely evaded the issue of where RUN is actually implemented. That is, we nowhere permit the user actually to use the methodology upon the god that actually steps through one's programs. Furthermore, we have not dealt with that special moment when the god of the lower machine suddenly becomes the god of the higher machine, presumably by the process of refusing to honor future requests for lower level operations (except as they arise in the implementations of higher level machines). The reason for these omissions is that we do not know yet how to do these things precisely, and consequently it is inconceivable that we could have formalized these matters. It is also gravely doubtful whether the known ideas (particularly the Floyd(Hoare) ideas) are at all adequate to the task of practically proving correctness in such delicate matters, even if we had a clear idea of what correctness meant. The problem with the Floyd approach is that it, too, assumes the existence of the deus ex machina: it assumes that the program is a fixed entity that does not itself change (much less change itself) in the execution of the program. The fundamental idea of the stored program computer is utterly inconsistent with that assumption.

5. Objects Upstairs and Down

It has long been discussed whether objects can exist on the upper level machine that do not exist on the lower level machine (e.g., capabilities on an integer machine). We have taken the position that there is no way this can come about. Note, in RUN1, that the objects in the upper level machine are passed, by INVOKE, to the ov-functions of the lower level machine, implicitly requiring that they be objects down there. (No proof of the correctness of an implementation of upper.module on lower.module could appeal to the correctness assertion for the lower module without establishing that the objects passed to the ov-functions were objects in the lower machine.) This implicit requirement in USE.THE.METHODOLOGY is made explicit in the output of USE.FLOYD, where we generate a condition requiring that the type function of the lower machine include that of the upper machine.

6. On Compilers

In writing a compiler for our language (VCG.COMPILE, cf. above), we inadvertently adopted the LISP style of retaining the association between variables and identifiers in the compiled code. That is, if X had a certain value at a certain moment in the higher level version of a program, then X had the same value at a related moment in the lower program. The Algol style is, of course, completely to abandon any particular connection of this sort. The writer of any compiler for the RUN package should beware: the invariants and clocks in ASSERT statements are only useful if they refer to variables in a sensible way. If you rename the program variables, then you will need to rename the variables in the assertions.

CHAPTER 17

AN INITIAL APPROACH TO VERIFYING A SCHEDULER – WRITTEN IN ASSEMBLY LANGUAGE

On Why It Is Impossible to Prove That the BDX90 Dispatcher Implements a Time-Sharing System

Robert S. Boyer and J S. Moore

The SIFT system, as coded by Chuck Weinstock, is all written in Pascal except for about a page of machine code. The reason that machine code is used at all is that the SIFT system implements a small time-sharing system in which Pascal programs for separate application tasks are executed according to a schedule with real-time constraints. The Pascal language has no provision for handling the notion of an "interrupt" such as the B930 clock interrupt. The Pascal language also lacks the notion of running a Pascal subroutine for a given amount of time, suspending it, saving away the suspension, and later activating the suspension. Machine code was used to overcome these inadequacies of Pascal (and most other higher order languages). Code which handles clock interrupts and suspends processes is called a dispatcher.

The BDX930 SIFT dispatcher consists of the following 14 BDX930 instructions.

```
CINT    PUSHF   15              save the flags
        PUSHM   1,13            Save registers
        PUSHM   0,0             and the resume address
        LOAD    0,ACLK          indicate a clock tick
SCHG    TRA     1,15            save the current stack pointer
        LDM     15,15,STACK     point at the "exec" stack
        PUSHM   0,1             set function code and resume stack
        JSS*    ASCHE           call the scheduler which is a pascal function
        TRA     15,12           that returns the new tasks r15 value.
        POPM    0,0             restore the resume PC to R0
        POPM    1,13            restore some registers.
        POPF    15              and the flags
        CONT    ES              allow interrupts
        RET     0               and go resume this routine
```

When the current task is interrupted by a clock interrupt before normal termination, control is transferred to CINT by the clock interrupt mechanism. The code at CINT pushes onto the task's Pascal stack the current flags, registers, and pc, and sets a flag in register 0 to indicate that the task was interrupted prematurely. Control then reaches SCHG.

On the other hand, when the current task terminates normally, code not shown here does the following: the clock interrupt mechanism is disabled, the necessary reinitialization information is saved on its Pascal stack, register 0 is set to indicate that the task was terminated normally, and control is transferred to SCHG.

In either case, the dispatcher then saves the task's stack pointer, reinstates a stack pointer used exclusively by the dispatcher and scheduler and jumps off to the Pascal code for the scheduler. The scheduler stores in the task table the task's stack and saved state information. The scheduler returns to the dispatcher the stack and state information for the next task to be run. The dispatcher then reinstates the flags, registers, and pc for that task, enables interrupts, and returns to the task.

Thus, the dispatcher and scheduler apparently implement a time-sharing system in which each user task is running on a "virtual" BDX930. We set out to try to prove that the 14 lines of code above correctly implemented those virtual BDX930s.

A reasonably simple specification of the time-sharing system goes something like this: The real BDX930 is supporting n vitual BDX930s, each devoted to a different user task. The virtual BDX930s are identical to the real BDX930 except for the absence of clock interrupts and certain parts of memory. A "mapping function" can be defined that maps the state of the real machine into an n-tuple of states of the virtual machines. When an instruction is executed on the real machine either the n-tuple of virtual states is unchanged or else one of the virtual states is advanced by one instruction and the remaining states are unchanged.

To capture the semantics of the instruction set, we encoded in our logic a recursive function that describes the state changes induced by each BDX930 instruction. Thirty pages are required to describe the top level driver and the state changes induced by each instruction (in terms of certain still undefined bit-level functions such as the 8-bit signed addition function). We encountered difficulty getting the mechanical theorem-prover to process such a large definition. However, the system was improved and the function was eventually admitted. We still anticipate great difficulty proving anything about the function because of its large size. However, the problems that have stopped us have nothing to do with mechanical proof; instead they are in formalizing a suitable specification.

We discuss three problems below: specifying the interrupt mechanism on the BDX930, specifying the mapping function, the specifying the restrictions on user tasks.

Interrupts: Clock interrupts on the BDX930 occur at a specified interval. But it is difficult to get precise statements regarding how long any given instruction will take. The situation is complicated by the fact that some cycles are stolen to service writes to the data file by concurrent processors, thus introducing a true nondeterminacy in precise timings. The best one can expect is to get some kind of interval indicating how fast or slow each instruction is. For these reasons we abandoned the idea of trying to model precisely the clock interrupt mechanism.

In our model of the BDX930, an interrupt can occur at any time while interrupts are enabled. One must state explicitly where interrupts are assumed not to occur. This exposes a problem in the dispatcher above. If control reaches the dispatcher because of the clock interrupt mechanism, the dispatcher and the scheduler are executed with clock interrupts enabled. A clock interrupt during either of these processes causes chaos. In our model, we must assume explicitly that no clock interrupt occurs during this processing. To prove that no interrupts can occur, one must determine the maximum time it takes to execute the dispatcher and scheduler. To do this one must (a) have a precise specification of the times taken by individual BDX930 operations and (b) treat the scheduler as BDX930 code rather than Pascal. This particular problem could be avoided if the dispatcher always disabled interrupts on entry. However, the complete lack of constraint on interrupts in the current model is unsettling and unrealistic.

Mapping Function: To determine the state of each virtual machine the mapping function must consider each task and determine the state of the task. Consider how one might determine the contents of the thirteen saved registers in each task. The registers of a suspended task are stored in positions 2 through 14 of the stack saved for the task in the task table. The registers of the active task are somewhat more difficult to ascertain. If the program counter (pc) is in the code for the active task, the virtual registers are in the corresponding actual registers. But if the pc is in the dispatcher or scheduler, the virtual registers may be in any number of places. For example, if the instruction at CINT has just been executed, they are in the actual registers. But if the instruction just after CINT has just been executed, the virtual registers are in positions 1 through 13 of the stack in register 15. And if the second instruction after CINT has just been executed, they are in positions 2 through 14 of that stack.

In general, to recover the state of the active task, it is necessary to consider (while defining the mapping function) each instruction in the dispatcher and scheduler. Furthermore, it is necessary to treat the scheduler as BDX930 code rather than as Pascal code, since otherwise one cannot trace where in the real machine the components of the state are being kept while in transit to the task table.

Restrictions on User Tasks: Two restrictions on user tasks are necessary if the dispatcher is to implement the kind of time-sharing system described.

The first concerns the size of the Pascal stack for each task. Recall that the state of an interrupted process is saved by pushing the flags, 13 registers, and the pc on the stack. If there is insufficient room on the stack, instructions or data (possibly from another task) are overwritten. Thus, one restriction on the user tasks is that they never come within 15 words of exhausting the allocated stack space. But the stack is used primarily to store temporaries and subroutine links and its management is entirely under the control of the Pascal compiler. One cannot determine whether a given Pascal program satisfies this restriction unless one looks at the code generated by a given compiler. Note that in general it is impossible to verify with a static analysis that a given user task -- even displayed as BDX930 code -- does not use too much of the stack, since depth of recursion and other runtime considerations influence stack use.

The second restriction is more subtle. In its attempt to save the state of an interrupted process, the dispatcher saves only the flags, registers, and pc. It is assumed that all other parts of the state of the task are private to the task itself and will not be affected by the execution of other tasks. In particular, tasks may not share variables that are read and written. At first sight one may conclude that this assumption can be checked by confirming that the Pascal code for a set of tasks share no variables. However, such a check is insufficient. Again, the compiler must be considered. Suppose that the compiler uses certain memory locations as temporaries. Then those temporaries must be saved by the dispatcher too. But if user tasks are considered to be unrestricted BDX930 code, the check becomes even more difficult because it is not possible to determine with a static analysis what memory locations are read and written. It is also necessary to require of user tasks that they not use the clock interrupt mechanism and not overwrite the area of the BDX930 dedicated to the operating system. Specifying the requirements on user programs requires a rigorous formal understanding of the BDX930, the Pascal compiler, and the linking loader. Thus an attempt to verify the few lines of machine code in SIFT lead to the requirement that we have formal specifications for several huge objects which have never yet been adequately formalized.

This concludes our discussion of difficulties encountered while trying to formalize the simple time-sharing system sketched above. However, the worst is yet to come. The simple model sketched is inadequate for SIFT because tasks are supposed to share data.

It is common in SIFT for task A to compute a result and put it somewhere for a later task, B, to read. For example, many tasks share parts of the datafile with the prevote task. But this suddenly introduces the notion of time. In the simple model, tasks A and B each run on their own processor and do not interract. If each task is to be repeated indefinitely then each processor endlessly iterates its own task. There is no sense in which the iterations of A are synchronized with those of B. Under the current SIFT scheme however, the dispatcher is used to "time share" tasks that share data, but the schedules tables are arranged so that the iterations of A do not overlap those of B.

If one attempts to patch things up while preserving the notion that A and B are running on independent virtual BDX930s, one is forced to introduce the notion of communicating virtual machines -- an idea somewhat more complicated than the truth. We now question the utility of the abstraction of virtual machines. Indeed, the whole idea that the dispatcher is implementing a time-sharing system comes into question since a major use of it is to orchestrate fixed sequences of subroutine calls.

The time-sharing/virtual machine idea is completely destroyed by the reconfiguration task. This tasks redefines the task table. Thus, after termination of the reconfiguration task, the tasks run by the dispatcher have no relation to those run before reconfiguration. It is impossible to view the dispatcher as a time-sharing system implementing virtual BDX930s running concurrently when one "process" can wipe out the others.

In our view, it is a mistake to think of the dispatcher in abstract terms. It seems to be just a program running on a von Neumann machine. By carefully arranging certain tables you can program the machine to execute a few instructions from here and then a few instructions from there, almost as though you had two different machines. By cleverly arranging those tables you can make one piece of code share data with another, almost as though your machines were communicating. By being still more clever you can synchronize them to the point that the two programs appear to be running sequentially on just one machine. Indeed, by carelessly arranging those same tables you can cause arbitrary chaos. But the moral is clear: you are programming a single machine and not a set of virtual machines.

We think that things might be a lot more clear if schedules were not encoded as tables that were interpretted by the scheduler and dispatcher but were merely Pascal programs that iteratively called fixed sequences of subroutines.

Research Topics Worthy of Consideration

Abstract Interpreters:

Interrupts:

Real time -- Newtonian time -- clock synchronization.

CHAPTER 18

FORMAL DEFINITION OF BDX930 INSTRUCTION SET

Defining of the BDX930 Assembly Language

R. Boyer and J Moore

Introduction

We started with a shell definition for the B930 state which had 16 components, 12 of which were numerically typed. When we ran that through the <BOYER> theorem prover it blew up on the CONS.EQUAL axiom because it tried to normalize the IFs on the rhs (right hand sides) of the equality.

We changed DEFN0 and MAKE.REWRITE.RULES so that the bodies of defns and the rhs's of rewrites were not altered by the system.

Because we anticipate being wiped out by all the nonrec defns in the B930, we considered editing REWRITE.FNCALL to make it open up nonrec fns more carefully than now. It has been suggested that we adopt some draconian restriction (e.g., open only if no IFs are introduced) just to force us to think about more reasonable restrictions. That would make us fail to prove (AND P Q) -> (AND Q P) by simplification alone. (We could actually prove it, by virtue of the expansion of ANDs in hypotheses if we recognized (AND P Q) as an AND.) Another idea was similar to W. Bledsoe's pairings (as we have always imagined it) namely keep track of what tests the fns were interested in and have some high level procedure split on the most important ones to force certain fns to open together. In the end we decided not to touch REWRITE.FNCALL for now but it is lying in wait for the B930 to come along.

A BOOT.STRAP failed because DIFFERENCE was no longer numerically typed (and so RECURSION.BY.DIFFERENCE was rejected as an induction lemma) because it returns I after testing not (ZEROP I).

We considered several alternatives. One was the idea of a "type set lemma", e.g., (NOT (ZEROP X)) -> (NUMBERP X), which could be implemented by generalizing RECOGNIZER.ALIST to two alists, one for use when the recognizer is assumed true and one for when it is assumed false. E.g., NLISTP would be bound the type bits for LISTP on the false alist and bound to the complement on the true alist. ZEROP would be on the false one, bound to numbers. This would probably not slow down TYPE.SET. We would have to write some code to recognize type set lemmas and produce the bit patterns from the recognizer proposition.

A second alternative was to define a new class of functions called "defined recognizers" which were boolean valued nonrec fns and to open them up all the time (in preprocessing) and to normalize. That is the approach we took. We changed PUT.TYPE.PRESCRIPTION to so preprocess sdefns before guessing the type. But we left the unpreprocessed sdefn as the one used in theorem proving.

We then decided to address a problem Elspas raised, namely that (EQUAL NIL NIL) is proved rather circuitously by expanding the abbreviation PACK.EQUAL to the equality of the unpacks and (using CONS.EQUAL under the rule in CLAUSIFY.INPUT that says you can split a conjunction at the top level) rewrote that to the conjoined equalities of the ascii codes. When we tried to reproduce the silly proof with the just modified theorem prover it was not so silly because CONS.EQUAL wasn't used. The reason was that in the modified tp the rhs of CONS.EQUAL is (AND & &) instead of (IF & & F) and so is not recognized as a conjunction!

We are reluctant to let this state of affairs persist because we are basically happy with the current preprocessing of large vcs and don't want to inadvertantly change that preprocessing. We thought perhaps we should always expand ANDs (and other boot strap propositional fns) and normalize. But that fails drastically on a 16 component shell.

On the subject of abbreviations, we were troubled by the fact that PACK.EQUAL is an abbreviation but ADD1.EQUAL is not. Thank goodness for that (and the fact that ADD1.EQUAL is not a conjunction), since otherwise (EQUAL 1000 1000) would fail due to the problem noted by Elspas. But it seems odd that two schematically related rewrite rules are not treated equally.

Finally, Elspas's problem would never have arisen had the preprocessing put expressions in reduced form. One suggestion is to make CONS.TERM always apply 1fns, instead of just doing it for shells. If we did that, one might argue that we ought to review the uses of FCONS.TERM to determine whether they should be replaced by CONS.TERMs to enforce a new invariant on terms, namely that they are always in reduced form.

Another subject we have discussed is that we should make nonrecursive functions and unconditional rewrite rules behave identically. This idea was suggested after considering further how we might handle a 16 component shell with type restrictions. E.g., we could make CONS.EQUAL rewrite (EQUAL (CONS X Y) (CONS U V)) to the conjunction of things like

(EQUAL (CAR (CONS X Y)) (CAR (CONS U V))).

And then treat the rewrite rule (CAR (CONS X Y)) = (IF type X dv) in the way we treat nonrec defns, namely not apply such rewrites when they introduce too many IFs or otherwise blow us up.

If we decided to make unconditional rewrites and nonrec fns behave identically we could do it by eliminating nonrec defns altogether and just storing them as rewrite rules.

Another idea on the subject of large shells is that we could rewrite the equality of two conses to the equality of some coercions, eg. (FIX X) = (FIX U), instead of naked IFs, and then let the nonrec fn handler worry about the explosion.

Still another suggestion was to eliminate shells other than the boot strap ones and force people to use lists the way mathematicians do. That will probably require some better handling of nonrec fns than we have now since abbreviations become quite useful. There is also the worry that things will get troublesome the way they did in our efforts at the University of Edinburgh when we couldn't distinguish lists from numbers.

BDX30 Assembly Language Definition

```
(SETQ XXX '(
(ADD.SHELL STATE NIL STATEP
   ((MEM (NONE.OF) ZERO)
    (PC (NONE.OF) ZERO)
    (ACS (NONE.OF) ZERO)
    (OV (NONE.OF) ZERO)
    (IE (NONE.OF) ZERO)
    (IR (NONE.OF) ZERO)
    (F1 (NONE.OF) ZERO)
    (F2 (NONE.OF) ZERO)
    (EXT1 (NONE.OF) ZERO)
    (EXT2 (NONE.OF) ZERO)
    (EXT3 (NONE.OF) ZERO)
    (HALT (NONE.OF) FALSE)
    (ERROR (NONE.OF) FALSE)
    (CONTROL.PANELP (NONE.OF) FALSE)
    (INDIRECT.CNT (NONE.OF) ZERO)
    (EXECR.CNT (NONE.OF) ZERO)))

(DEFN FETCH (MEM LOC)
    (IF (NLISTP MEM) 0
        (IF (EQUAL (CAAR MEM) LOC) (CDAR MEM) (FETCH (CDR MEM) LOC))))
(DEFN PUT (LOC VAL MEM) (CONS (CONS LOC VAL) MEM))

(DEFN SET.MEM (LOC VAL ST)
      (STATE (PUT LOC VAL (MEM ST))
             (PC ST)
             (ACS ST)
             (OV ST)
             (IE ST)
             (IR ST)
             (F1 ST)
             (F2 ST)
             (EXT1 ST)
             (EXT2 ST)
             (EXT3 ST)
             (HALT ST)
             (ERROR ST)
             (CONTROL.PANELP ST)
             (INDIRECT.CNT ST)
             (EXECR.CNT ST)))

(DEFN SET.PC (PC ST)
      (STATE (MEM ST)
             PC
             (ACS ST)
             (OV ST)
             (IE ST)
             (IR ST)
             (F1 ST)
             (F2 ST)
             (EXT1 ST)
             (EXT2 ST)
             (EXT3 ST)
             (HALT ST)
             (ERROR ST)
             (CONTROL.PANELP ST)
             (INDIRECT.CNT ST)
             (EXECR.CNT ST)))
```

```
(DEFN SET.AC (AC VAL ST)
      (STATE (MEM ST)
             (PC ST)
             (PUT AC VAL (ACS ST))
             (OV ST)
             (IE ST)
             (IR ST)
             (F1 ST)
             (F2 ST)
             (EXT1 ST)
             (EXT2 ST)
             (EXT3 ST)
             (HALT ST)
             (ERROR ST)
             (CONTROL.PANELP ST)
             (INDIRECT.CNT ST)
             (EXECR.CNT ST)))

(DEFN SET.AC.&.AC+1 (AC PAIR ST)
      (STATE (MEM ST)
             (PC ST)
             (PUT AC (CAR PAIR) (PUT (ADD1 AC) (CDR PAIR)  (ACS ST)))
             (OV ST)
             (IE ST)
             (IR ST)
             (F1 ST)
             (F2 ST)
             (EXT1 ST)
             (EXT2 ST)
             (EXT3 ST)
             (HALT ST)
             (ERROR ST)
             (CONTROL.PANELP ST)
             (INDIRECT.CNT ST)
             (EXECR.CNT ST)))

(DEFN SET.OV (VAL ST)
      (STATE (MEM ST)
             (PC ST)
             (ACS ST)
             VAL
             (IE ST)
             (IR ST)
             (F1 ST)
             (F2 ST)
             (EXT1 ST)
             (EXT2 ST)
             (EXT3 ST)
             (HALT ST)
             (ERROR ST)
             (CONTROL.PANELP ST)
             (INDIRECT.CNT ST)
             (EXECR.CNT ST)))

(DEFN SET.IE (VAL ST)
      (STATE (MEM ST)
             (PC ST)
             (ACS ST)
```

```
           (OV ST)
           VAL
           (IR ST)
           (F1 ST)
           (F2 ST)
           (EXT1 ST)
           (EXT2 ST)
           (EXT3 ST)
           (HALT ST)
           (ERROR ST)
           (CONTROL.PANELP ST)
           (INDIRECT.CNT ST)
           (EXECR.CNT ST)))


(DEFN SET.F1 (VAL ST)
      (STATE (MEM ST)
             (PC ST)
             (ACS ST)
             (OV ST)
             (IE ST)
             (IR ST)
             VAL
             (F2 ST)
             (EXT1 ST)
             (EXT2 ST)
             (EXT3 ST)
             (HALT ST)
             (ERROR ST)
             (CONTROL.PANELP ST)
             (INDIRECT.CNT ST)
             (EXECR.CNT ST)))
(DEFN SET.F2 (VAL ST)
      (STATE (MEM ST)
             (PC ST)
             (ACS ST)
             (OV ST)
             (IE ST)
             (IR ST)
             (F1 ST)
             VAL
             (EXT1 ST)
             (EXT2 ST)
             (EXT3 ST)
             (HALT ST)
             (ERROR ST)
             (CONTROL.PANELP ST)
             (INDIRECT.CNT ST)
             (EXECR.CNT ST)))
(DEFN SET.HALT (VAL ST)
      (STATE (MEM ST)
             (PC ST)
             (ACS ST)
             (OV ST)
             (IE ST)
             (IR ST)
             (F1 ST)
             (F2 ST)
             (EXT1 ST)
             (EXT2 ST)
             (EXT3 ST)
             VAL
             (ERROR ST)
             (CONTROL.PANELP ST)
             (INDIRECT.CNT ST)
             (EXECR.CNT ST)))
(DEFN SET.ERROR (VAL ST)
      (STATE (MEM ST)
             (PC ST)
             (ACS ST)
             (OV ST)
             (IE ST)
             (IR ST)
             (F1 ST)
             (F2 ST)
             (EXT1 ST)
             (EXT2 ST)
             (EXT3 ST)
             (HALT ST)
             VAL
             (CONTROL.PANELP ST)
             (INDIRECT.CNT ST)
             (EXECR.CNT ST)))



(DEFN EXPT (I J)
      (IF (ZEROP J) 1 (TIMES I (EXPT I (SUB1 J)))))

(DEFN FIELD (WRD HI LO) (REMAINDER (QUOTIENT WRD (EXPT 2 LO))
                                   (EXPT 2 (ADD1 (DIFFERENCE HI LO)))))



                        (* to be defined to return the integer
                          represented by bits HI through LO
                          inclusive in the binary representation
                          of the integer WRD.))

(DEFN AM (WRD) (FIELD WRD 11 10))
(DEFN IBIT (WRD) (FIELD WRD 15 15))
(DEFN D (WRD) (FIELD WRD 7 0))
(DEFN OP1 (WRD) (FIELD WRD 14 12))
(DEFN OP2 (WRD)(FIELD WRD 11 8))
(DEFN A (WRD) (FIELD WRD 7 4))
(DEFN B (WRD) (FIELD WRD 3 0))
(DEFN AC (WRD) (FETCH WRD 9 8))
(DEFN OP3 (WRD) (FIELD WRD 7 4))
(DEFN DELTA (WRD) (FETCH WRD 3 0))

(DEFN TURN.OFF.HI.BIT (WRD) (FIELD WRD 14 0))

(DCL B930.ADD.8BIT (WRD DISPL)
     (* adds the 16 bit quantity WRD to the 8 bit
        signed quantity DISPL and produces a new 16 bit
        quantity.  This fn is always used to
        construct an address -- e.g., a pc or
        stack pointer or effective address.
        We are not sure what happens if WRD represents
        a negative quantity.  Also, we assume such
        arithmetic isn't senstive to the F1 flg.  Also
        we don't know what happens when an overflow occurs.))

(DCL B930.ADD.4BIT (WRD DELTA)
     (* adds 16 bits to 4 bit signed quantity to
        produce new 16 bit thing.  The comments about
        B930.ADD.8BIT apply here too.))

(DEFN PC+1 (ST) (B930.ADD.8BIT (PC ST) 1))

(DEFN TRACE.INDIRECT.CHAIN (WRD MEM CNT)
 (IF (ZEROP CNT)
     0
     (IF (EQUAL (IBIT WRD) 1)
         (TRACE.INDIRECT.CHAIN (FETCH MEM (TURN.OFF.HI.BIT WRD))
                               MEM (SUB1 CNT))
         WRD))
 (* We suppose you turn the high bit off before you treat WRD as
    an address. The ISP doesn't.))




(DEFN MAR (WRD ST)
 (TRACE.INDIRECT.CHAIN
  (IF (EQUAL (AM WRD) 0)
      (D WRD)
  (IF (EQUAL (AM WRD) 1)
      (B930.ADD.8BIT (PC ST) (D WRD))
  (IF (EQUAL (AM WRD) 2)
      (B930.ADD.8BIT (FETCH (ACS ST) 0)
                    (D WRD))
      (B930.ADD.8BIT (FETCH (ACS ST) 1)
                     (D WRD)))))
  (MEM ST) (INDIRECT.CNT ST))
 (* The ISP seems to just add the displacement when we think it
    ought to use the 8-bit add and permit negative displacements.
    The ISP indicates that the indirect bit is to be interpreted
    as here, i.e., once calculate an address from D etc and then
    chain through the indirect pointers.  But the programmers manual
    has evidence that the MAR calculation is more akin to the PDP-10
    style where one recomputes the effective address recursively.
    Every place we call MAR on WRD2 of a double word instr, we pass
    an ST whose PC points to the first of the two words.  Should it
    pass BUMP.PC of ST instead?  The ISP indicates yes with its
    GROUP command, but the manual indicates no under the discussion
    of JSS.))
(DCL B930.ADDR (WRD1 WRD2 F1) (* Returns the 16 bit number the
                                B930 would if asked to add
                                WRD1 and WRD2 with F1 set to 1 or 0.))
(DCL B930.SUBR (WRD1 WRD2 F1))

(DCL B930.ADDR.OV (WRD1 WRD2 F1) (* Returns value of OV flag after
                                      the appropriate B930 add.  What
                                      is the parity of the bit?  Sometimes
                                      -- e.g. in DECEQ -- we set the
```

```
                                       OV flg to 0 meaning NO OVERFLOW. Is
                                       that right?))
(DCL B930.SUBR.OV (WRD1 WRD2 F1))



(DEFN JU (WRD1 WRD2 ST) (SET.PC (MAR WRD1 ST) ST))
(DEFN JSA0 (WRD1 WRD2 ST)
          (SET.PC (MAR WRD1 ST)
                  (SET.AC 0 (PC+1 ST) ST))
          (* We compute MAR w.r.t. the unmodified state.  But the
             manual and the ISP imply that MAR is computed after
             smashing ac 0; We did it this way just because it
             seemed more likely.))
(DEFN JSA1 (WRD1 WRD2 ST)
      (SET.PC (MAR WRD1 ST)
              (SET.AC 1 (PC+1 ST) ST))
      (* See JSA0))
(DEFN JMA0  (WRD1 WRD2 ST)
 (SET.PC (MAR WRD1 ST)
         (SET.AC 0 (PC+1 ST)
                 (SET.MEM (B930.ADD.8BIT (FETCH (ACS ST) 15) 1)
                          (FETCH (ACS ST) 0)
                          (SET.AC 15 (B930.ADD.8BIT (FETCH (ACS ST) 15) 1)
                                  ST))))
      (* we are unsure of whether we are to compute
         MAR of the original ST as here or of
         state after the modifications below.  Note that the treatment
         of indirect address chains is affected.))
(DEFN ADD  (WRD1 WRD2 ST)
 (BUMP.PC (SET.AC (AC WRD1)
                  (B930.ADDR (FETCH (ACS ST) (AC WRD1))
                            (FETCH (MEM ST) (MAR WRD1 ST))
                            (F1 ST))
                  (SET.OV (B930.ADDR.OV
                           (FETCH (ACS ST) (AC WRD1))
                           (FETCH (MEM ST) (MAR WRD1 ST))
                           (F1 ST))
                          ST))))
(DEFN SUB  (WRD1 WRD2 ST)
 (BUMP.PC (SET.AC (AC WRD1)
                  (B930.SUBR (FETCH (ACS ST) (AC WRD1))
                            (FETCH (MEM ST) (MAR WRD1 ST))
                            (F1 ST))
                  (SET.OV (B930.SUBR.OV
                           (FETCH (ACS ST) (AC WRD1))
                           (FETCH (MEM ST) (MAR WRD1 ST))
                           (F1 ST))
                          ST))))
(DEFN CMP  (WRD1 WRD2 ST)
 (SET.PC
  (B930.ADD.8BIT (PC ST)
                 (IF (B930.LESSP (FETCH (ACS ST)
                                        (AC WRD1))
                                 (FETCH (MEM ST)
                                        (MAR WRD1 ST)))
                     3
                     (IF (B930.EQP (FETCH (ACS ST) (AC WRD1))
                                   (FETCH (MEM ST) (MAR WRD1 ST)))
                          1
                          2)))
  (SET.OV (B930.CMP.OV WRD1 WRD2 ST) ST)))
(DEFN LOAD  (WRD1 WRD2 ST)
         (BUMP.PC (SET.AC (AC WRD1)
                          (FETCH (MEM ST) (MAR WRD1 ST))
                          ST)))
(DEFN STO  (WRD1 WRD2 ST)
         (BUMP.PC (SET.MEM (MAR WRD1 ST)
                           (FETCH (ACS ST) (AC WRD1))
                           ST)))
(DEFN TRA/NOP (WRD1 WRD2 ST)
         (BUMP.PC (SET.AC (A WRD1)
                          (FETCH (ACS ST) (B WRD1))
                          ST)))
(DEFN DECEQ  (WRD1 WRD2 ST)
 (IF (EQUAL (FETCH (ACS ST) (A WRD1)) 1)
     (SET.PC
      (B930.ADD.4BIT (PC+1 ST)
                     (B WRD1))
      (SET.AC (A WRD1) 0
              (SET.OV 0 ST)))
     (BUMP.PC (SET.AC (A WRD1)
                      (B930.SUBR (FETCH (ACS ST) (A WRD1))
                                 1
                                 (F1 ST))
                      (SET.OV (B930.SUBR.OV (FETCH (ACS ST)
                                                   (A WRD1))
                                            1
                                            (F1 ST))
                              ST)))))
(DEFN LCM  (WRD1 WRD2 ST)
         (BUMP.PC (SET.AC (A WRD1)
                          (B930.LCM (FETCH (ACS ST)
                                                   (B WRD1)))
                          ST)))

(DEFN RLS  (WRD1 WRD2 ST)
         (BUMP.PC (SET.AC (A WRD1)
                          (B930.RLS (FETCH (ACS ST) (A WRD1))
                                    (B WRD1))
                          ST)))
(DEFN B930.CONT (OLD FN)
 (IF (EQUAL FN 0) OLD
     (IF (EQUAL FN 1) 0
         (IF (EQUAL FN 2) 1 (IF (EQUAL OLD 0) 1 0)))))

(DEFN CONT  (WRD1 WRD2 ST)
 (BUMP.PC
  (SET.F1 (B930.CONT (F1 ST) (FIELD WRD1 5 4))
          (SET.F2 (B930.CONT (F2 ST) (FIELD WRD1 7 6))
                  (SET.IE (B930.CONT (IE ST) (FIELD WRD1 3 2))
                          (SET.OV (B930.CONT (OV ST) (FIELD WRD1 1 0)) ST))))))


(DEFN DECNE  (WRD1 WRD2 ST)
 (IF (NOT (EQUAL (FETCH (ACS ST) (A WRD1)) 1))
     (SET.PC
      (B930.ADD.4BIT (PC+1 ST)
                     (B WRD1))
      (SET.AC (A WRD1)
              (B930.SUBR (FETCH (ACS ST) (A WRD1))
                        1
                        (F1 ST))
              (SET.OV (B930.SUBR.OV (FETCH (ACS ST)
                                          (A WRD1))
                                   1
                                   (F1 ST))
                      ST)))
     (BUMP.PC (SET.AC (A WRD1) 0
                      (SET.OV 0 ST)))))
(DEFN ANDOP (WRD1 WRD2 ST)
 (BUMP.PC (SET.AC (A WRD1)
                  (B930.AND (FETCH (ACS ST) (A WRD1))
                            (FETCH (ACS ST) (B WRD1)))
                  ST)))
(DEFN RLL  (WRD1 WRD2 ST)
         (IF (NOT (EVEN (A WRD1)))
             (SET.ERROR T ST)
             (BUMP.PC
              (SET.AC.&.AC+1 (A WRD1)
                             (B930.RLL (FETCH (ACS ST)
                                              (A WRD1))
                                       (FETCH (ACS ST)
                                              (ADD1 (A WRD1)))
                                       (B WRD1))
                             ST))))
(DEFN ADDR  (WRD1 WRD2 ST)
 (BUMP.PC
  (SET.AC (A WRD1)
          (B930.ADDR (FETCH (ACS ST) (A WRD1))
                    (FETCH (ACS ST) (B WRD1))
                    (F1 ST))
          (SET.OV (B930.ADDR.OV
                   (FETCH (ACS ST) (A WRD1))
                   (FETCH (ACS ST) (B WRD1))
                   (F1 ST))
                  ST))))
(DEFN IR/CLA (WRD1 WRD2 ST)
 (IF (EQUAL (A WRD1) (B WRD1))
     (BUMP.PC (SET.AC (A WRD1) 0 ST))
     (BUMP.PC
      (SET.AC (A WRD1)
              (FETCH (ACS ST) (B WRD1))
              (SET.AC (B WRD1)
                      (FETCH (ACS ST) (A WRD1))
                      ST))))
  (* The ISP uses arithmetic to achieve this effect.  The manual
     does not mention the use of arithmetic.  Neither document
     suggests that OV gets set.)
)

(DEFN OROP (WRD1 WRD2 ST)
 (BUMP.PC (SET.AC (A WRD1)
                  (B930.OR (FETCH (ACS ST) (A WRD1))
                           (FETCH (ACS ST) (B WRD1)))
                  ST)))
```

```
(DCL B930.MPY (A B) (* Returns the double word value of A times B.
                       The manual and the ISP both suggest that something
                       weird happens at -1 but nobody seems to know what
                       it is.  The manual suggests that something weird
                       happens with the most negative number.  Finally,
                       the result is apparently left shifted one.))
(DEFN MPY  (WRD1 WRD2 ST)
 (IF (OR (NOT (EVEN (A WRD1)))
         (EQUAL (A WRD1) (B WRD1))
         (EQUAL (ADD1 (A WRD1)) (B WRD1)))
     (SET.ERROR T ST)
     (BUMP.PC
      (SET.AC.&.AC+1 (A WRD1)
                     (B930.MPY (FETCH (ACS ST) (A WRD1))
                               (FETCH (ACS ST) (B WRD1)))
                     ST))))
(DEFN CLAO/SUBR (WRD1 WRD2 ST)
 (BUMP.PC
  (SET.AC (A WRD1)
          (B930.SUBR (FETCH (ACS ST) (A WRD1))
                     (FETCH (ACS ST) (B WRD1))
                     (F1 ST))
          (SET.OV (B930.SUBR.OV
                   (FETCH (ACS ST) (A WRD1))
                   (FETCH (ACS ST) (B WRD1))
                   (F1 ST))
                  ST)))
          (* The ISP says OV gets zeroed but we think not.))

(DEFN ACM  (WRD1 WRD2 ST)
 (BUMP.PC
  (SET.AC (A WRD1)
          (B930.SUBR 0
                     (FETCH (ACS ST) (B WRD1))
                     (F1 ST))
          (SET.OV (B930.SUBR.OV
                   0
                   (FETCH (ACS ST) (B WRD1))
                   (F1 ST))
                  ST))))
(DEFN CMPR  (WRD1 WRD2 ST)
 (SET.PC (B930.ADD.8BIT (PC ST)
                        (IF (B930.LESSP (FETCH (ACS ST)
                                               (A WRD1))
                                        (FETCH (ACS ST)
                                               (B WRD1)))
                            3
                            (IF (B930.EQP (FETCH (ACS ST) (A WRD1))
                                          (FETCH (ACS ST) (B WRD1)))
                                1
                                2)))
         (SET.OV (B930.CMP.OV WRD1 WRD2 ST) ST)))
(DEFN DIV  (WRD1 WRD2 ST)
         (IF (OR (NOT (EVEN (A WRD1)))
                 (EQUAL (A WRD1) (B WRD1))
                 (EQUAL (ADD1 (A WRD1)) (B WRD1)))
             (SET.ERROR T ST)
             (BUMP.PC
              (SET.AC.&.AC+1 (A WRD1)
                              (B930.DIV (FETCH (ACS ST) (A WRD1))
                                        (FETCH (ACS ST) (ADD1 (A WRD1)))
                                        (FETCH (ACS ST) (B WRD1)))
                              (SET.OV
                               (B930.DIV.OV
                                (FETCH (ACS ST) (A WRD1))
                                (FETCH (ACS ST) (ADD1 (A WRD1)))
                                (FETCH (ACS ST) (B WRD1)))
                               ST))))
         (* The ISP contains an error in that SP[A] instead of SP[A]@SP[A+1] is
            compared with SP[B])
)
(DEFN SLSA  (WRD1 WRD2 ST)
 (BUMP.PC (SET.AC (A WRD1)
                  (B930.SLSA (FETCH (ACS ST) (A WRD1))
                             (B WRD1))
                  (SET.OV (B930.SLSA.OV (FETCH (ACS ST)(A WRD1))
                                        (B WRD1))
                          ST))))
(DEFN SLLA  (WRD1 WRD2 ST)
 (IF (NOT (EVEN (A WRD1)))
     (SET.ERROR T ST)
     (BUMP.PC
      (SET.AC.&.AC+1 (A WRD1)
                     (B930.SLLA (FETCH (ACS ST)
                                       (A WRD1))
                                (FETCH (ACS ST)
                                       (ADD1 (A WRD1)))
                                (B WRD1))
                     (SET.OV
                      (B930.SLLA (FETCH (ACS ST)
                                        (A WRD1))
                                 (FETCH (ACS ST)
                                        (ADD1 (A WRD1)))
                                 (B WRD1))
                      ST)))))
(DEFN SKGT  (WRD1 WRD2 ST)
 (IF (B930.LESSP 0 (FETCH (ACS ST) (A WRD1)))
     (SET.PC
      (B930.ADD.4BIT (PC+1 ST)
                     (B WRD1))
      ST)
     (BUMP.PC ST)))
(DEFN SKLT  (WRD1 WRD2 ST)
 (IF (B930.LESSP (FETCH (ACS ST) (A WRD1)) 0)
     (SET.PC
      (B930.ADD.4BIT (PC+1 ST)
                     (B WRD1))
      ST)
     (BUMP.PC ST)))
(DEFN SLSL  (WRD1 WRD2 ST)
 (BUMP.PC (SET.AC (A WRD1)
                  (B930.SLSL (FETCH (ACS ST) (A WRD1))
                             (B WRD1))
                  ST)))
(DEFN SLLL  (WRD1 WRD2 ST)
 (IF (NOT (EVEN (A WRD1)))
      (SET.ERROR T ST)
     (BUMP.PC
      (SET.AC.&.AC+1 (A WRD1)
                     (B930.SLLL (FETCH (ACS ST)
                                       (A WRD1))
                                (FETCH (ACS ST)
                                       (ADD1 (A WRD1)))
                                (B WRD1))
                     ST))))
(DEFN SKGE  (WRD1 WRD2 ST)
 (IF (NOT (B930.LESSP (FETCH (ACS ST) (A WRD1)) 0))
     (SET.PC
      (B930.ADD.4BIT (PC+1 ST)
                     (B WRD1))
      ST)
     (BUMP.PC ST)))
(DEFN SKLE  (WRD1 WRD2 ST)
 (IF (NOT (B930.LESSP 0 (FETCH (ACS ST) (A WRD1))))
     (SET.PC
      (B930.ADD.4BIT (PC+1 ST)
                     (B WRD1))
      ST)
     (BUMP.PC ST)))
(DEFN SRSA  (WRD1 WRD2 ST)
 (BUMP.PC (SET.AC (A WRD1)
                  (B930.SRSA (FETCH (ACS ST) (A WRD1))
                             (B WRD1))
                  ST)))
(DEFN SRLA  (WRD1 WRD2 ST)
  (IF (NOT (EVEN (A WRD1)))
      (SET.ERROR T ST)
      (BUMP.PC
       (SET.AC.&.AC+1 (A WRD1)
                      (B930.SRLA (FETCH (ACS ST)
                                        (A WRD1))
                                 (FETCH (ACS ST)
                                        (ADD1 (A WRD1)))
                                 (B WRD1))
                      ST))))
(DEFN SKEQ  (WRD1 WRD2 ST)
 (IF (B930.EQP 0 (FETCH (ACS ST) (A WRD1)))
     (SET.PC
      (B930.ADD.4BIT (PC+1 ST)
                     (B WRD1))
      ST)
     (BUMP.PC ST)))
(DEFN SRSL  (WRD1 WRD2 ST)
 (BUMP.PC (SET.AC (A WRD1)
                  (B930.SRSL (FETCH (ACS ST) (A WRD1))
                             (B WRD1))
                  ST)))
(DEFN SRLL  (WRD1 WRD2 ST)
 (IF (NOT (EVEN (A WRD1)))
     (SET.ERROR T ST)
     (BUMP.PC
      (SET.AC.&.AC+1 (A WRD1)
                     (B930.SRLL (FETCH (ACS ST)
                                       (A WRD1))
                                (FETCH (ACS ST)
                                       (ADD1 (A WRD1)))
                                (B WRD1))
```

```
                    ST))))
(DEFN SKNE  (WRD1 WRD2 ST)
 (IF (NOT (B930.EQP 0 (FETCH (ACS ST) (A WRD1))))
     (SET.PC
      (B930.ADD.4BIT (PC+1 ST) (B WRD1))
      ST)
     (BUMP.PC ST)))

(DEFN IAR  (WRD1 WRD2 ST)
 (BUMP.PC (SET.AC (A WRD1)
                  (B930.ADDR (FETCH (ACS ST) (A WRD1))
                             (IF (LESSP (B WRD1) 8)
                                 (B WRD1)
                                 (DIFFERENCE
                                  (PLUS (EXPT 2 15)
                                        (B WRD1))
                                  (EXPT 2 3)))
                             (F1 ST))
                  (SET.OV (B930.ADDR.OV (FETCH (ACS ST) (A WRD1))
                                        (IF (LESSP (B WRD1) 8)
                                            (B WRD1)
                                            (DIFFERENCE
                                             (PLUS (EXPT 2 15)
                                                   (B WRD1))
                                             (EXPT 2 3)))
                                        (F1 ST))))))
```

```
(DEFN SFE1 (WRD1 WRD2 ST)
      (IF (EQUAL (EXT1 ST) 0)
          (SET.PC (B930.ADD.4BIT (PC+1 ST)
                                 (B WRD1))
                  ST)
          (BUMP.PC ST)) (* We have assumed that TRUE means 1 and FALSE 0))

(DEFN SFE2 (WRD1 WRD2 ST)
      (IF (EQUAL (EXT2 ST) 0)
          (SET.PC (B930.ADD.4BIT (PC+1 ST)
                                 (B WRD1))
                  ST)
          (BUMP.PC ST)))

(DEFN SFE3 (WRD1 WRD2 ST)
      (IF (EQUAL (EXT3 ST) 0)
          (SET.PC (B930.ADD.4BIT (PC+1 ST)
                                 (B WRD1))
                  ST)
          (BUMP.PC ST)))

(DEFN SRIE (WRD1 WRD2 ST)
      (IF (EQUAL (IE ST) 0)
          (SET.PC (B930.ADD.4BIT (PC+1 ST)
                                 (B WRD1))
                  ST)
          (BUMP.PC ST))
```

```
     (* The ISP calls this instr SFIE))

(DEFN SROV (WRD1 WRD2 ST)
     (IF (EQUAL (OV ST) 0)
         (SET.PC (B930.ADD.4BIT (PC+1 ST)
                                (B WRD1))
                 ST)
         (BUMP.PC ST)))
(DEFN SFIR (WRD1 WRD2 ST)
     (IF (EQUAL (IR ST) 0)
         (SET.PC (B930.ADD.4BIT (PC+1 ST)
                                (B WRD1))
                 ST)
         (BUMP.PC ST)))
(DEFN SRF1 (WRD1 WRD2 ST)
     (IF (EQUAL (F1 ST) 0)
         (SET.PC (B930.ADD.4BIT (PC+1 ST)
                                (B WRD1))
                 ST)
         (BUMP.PC ST)))
(DEFN SRF2 (WRD1 WRD2 ST)
     (IF (EQUAL (F2 ST) 0)
         (SET.PC (B930.ADD.4BIT (PC+1 ST)
                                (B WRD1))
                 ST)
         (BUMP.PC ST)))

(DEFN STE1 (WRD1 WRD2 ST)
     (IF (NOT(EQUAL (EXT1 ST) 0))
         (SET.PC (B930.ADD.4BIT (PC+1 ST)
                                (B WRD1))
                 ST)
         (BUMP.PC ST)))

(DEFN STE2 (WRD1 WRD2 ST)
     (IF (NOT(EQUAL (EXT2 ST) 0))
         (SET.PC (B930.ADD.4BIT (PC+1 ST)
                                (B WRD1))
                 ST)
         (BUMP.PC ST)))

(DEFN STE3 (WRD1 WRD2 ST)
     (IF (NOT(EQUAL (EXT3 ST) 0))
         (SET.PC (B930.ADD.4BIT (PC+1 ST)
                                (B WRD1))
                 ST)
         (BUMP.PC ST)))

(DEFN SSIE (WRD1 WRD2 ST)
     (IF (NOT(EQUAL (IE ST) 0))
         (SET.PC (B930.ADD.4BIT (PC+1 ST)
                                (B WRD1))
                 ST)
         (BUMP.PC ST)))
(DEFN SSOV (WRD1 WRD2 ST)
     (IF (NOT(EQUAL (OV ST) 0))
         (SET.PC (B930.ADD.4BIT (PC+1 ST)
                                (B WRD1))
                 ST)
```

```
          (BUMP.PC ST)))
(DEFN STIR (WRD1 WRD2 ST)
      (IF (NOT(EQUAL (IR ST) 0))
          (SET.PC (B930.ADD.4BIT (PC+1 ST)
                                 (B WRD1))
                  ST)
          (BUMP.PC ST)))
(DEFN SSF1 (WRD1 WRD2 ST)
      (IF (NOT(EQUAL (F1 ST) 0))
          (SET.PC (B930.ADD.4BIT (PC+1 ST)
                                 (B WRD1))
                  ST)
          (BUMP.PC ST)))
(DEFN SSF2 (WRD1 WRD2 ST)
      (IF (NOT(EQUAL (F2 ST) 0))
          (SET.PC (B930.ADD.4BIT (PC+1 ST)
                                 (B WRD1))
                  ST)
          (BUMP.PC ST)))


(DEFN SET.MULTIPLE.ACS (AC ADDR N ST)
 (IF (ZEROP N)
     ST
     (IF (LESSP AC 16)
         (SET.MULTIPLE.ACS (ADD1 AC)
                       (ADD1 ADDR)
                       (SUB1 N)
                       (SET.AC AC (FETCH (MEM ST) ADDR) ST))
         (SET.ERROR T ST)))
 (* We assume that if asked to smash acs beyond 15 we cause
    an error, but will have modified the preceding acs.
    We don't consider the possibility that ADDR is pushed
    beyond 15 bits.))

(DEFN LDM (WRD1 WRD2 ST)
 (SET.PC (B930.ADD.8BIT (PC ST) 2)
         (SET.MULTIPLE.ACS (A WRD1)
                           (MAR WRD2 ST)
                           (ADD1 (DELTA WRD1))
                           ST))
 (* Is the MAR to be calculated each iteration as in the ISP or just
    once as we have done?  The manual says that B+1 consecutive memory
    words are moved -- which agrees with us.
    Also, when calculating the MAR of the second
    word of an instruction, does the PC point to the first or second
    word?  We assume the second.  What if the effective address is
    eventually bumped beyond the end of memory?))

(DEFN SET.MULTIPLE.MEM (AC ADDR N ST)
 (IF (ZEROP N)
     ST
     (IF (LESSP AC 16)
         (STORE.MULTIPLE.MEM (ADD1 AC)
                       (ADD1 ADDR)
                       (SUB1 N)
                       (SET.MEM ADDR (FETCH (ACS ST) AC) ST))
         (SET.ERROR T ST)))
 (* We assume that if asked to access acs beyond 15 we cause
    an error, but will have modified the preceding acs.
    We don't consider the possibility that ADDR is pushed
    beyond 15 bits.))

(DEFN STM (WRD1 WRD2 ST)
 (SET.PC (B930.ADD.8BIT (PC ST) 2)
         (SET.MULTIPLE.MEM (A WRD1)
                           (MAR WRD2 ST)
                           (ADD1 (DELTA WRD1))
                           ST))
 (* See the questions under LDM))

(DEFN PADDM/POPM/PUSHM (WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD2) 0)
     (PADDM WRD1 WRD2 ST)
     (IF (EQUAL (OP2 WRD2) 1)
         (POPM WRD1 WRD2 ST)
         (IF (EQUAL (OP2 WRD2) 2)
             (PUSHM WRD1 WRD2 ST)
             (SET.ERROR T ST)))))

(DEFN POPM (WRD1 WRD2 ST)
 (IF (OR (LESSP (FETCH (ACS ST) (B WRD2))
                (ADD1 (DELTA WRD1)))
         (LESSP (A WRD2) (DELTA WRD1)))
     (SET.ERROR T ST)
     (SET.PC (B930.ADD.8BIT (PC ST) 2)
             (SET.AC (B WRD2)
                     (DIFFERENCE (FETCH (ACS ST) (B WRD2))
                                 (ADD1 (DELTA WRD1)))
                     (SET.MULTIPLE.ACS (DIFFERENCE (A WRD2) (DELTA WRD1))
                                       (DIFFERENCE (FETCH (ACS ST) (B WRD2))
                                                   (DELTA WRD1))
                                       (ADD1 (DELTA WRD1))
                                       ST))))
 (* We don't know what happens if the initial A is too small to
    be decremented delta+1 times.  We don't know what happens if
    the stack pointer in B is negative or too small to be popped
    delta+1 times.  We cause errors.  We assume the ISP is wrong
    when it says you go back to the stack pointer in B each time
    -- permitting it to be one of the acs smashed -- instead of
    just moving delta+1 consecutive words as stated by the manual.))

(DEFN PUSHM (WRD1 WRD2 ST)
 (IF (OR (NOT (LESSP (IPLUS (FETCH (ACS ST) (B WRD2)) (ADD1 (DELTA WRD1)))
                     (EXPT 2 16)))
         (NOT (LESSP (IPLUS (A WRD2) (DELTA WRD1)) 16)))
     (SET.ERROR T ST)
     (SET.PC (B930.ADD.8BIT (PC ST) 2)
             (SET.AC (B WRD2)
                     (IPLUS (FETCH (ACS ST) (B WRD2)) (ADD1 (DELTA WRD1)))
                     (SET.MULTIPLE.MEM (A WRD2)
                                       (ADD1 (FETCH (ACS ST) (B WRD2)))
                                       (ADD1 (DELTA WRD1))
                                       ST))))
 (* see the comments under POPM))
```

```
(DEFN EXOR (WRD1 WRD2 ST)
      (BUMP.PC (SET.AC (A WRD1)
                       (B930.EXOR (FETCH (ACS ST)
                                         (A WRD1))
                                  (FETCH (ACS ST)
                                         (B WRD1)))
                       ST)))

(DEFN HALT (WRD1 WRD2 ST)
   (IF (CONTROL.PANELP ST)
       (BUMP.PC (SET.HALT T ST))
       (BUMP.PC ST)))

(DEFN RET (WRD1 WRD2 ST)
 (IF (ZEROP (FETCH (ACS ST) 15))
     (SET.ERROR T ST)
     (SET.PC (B930.ADD.8BIT (FETCH (ACS ST) 0) (D WRD1))
             (SET.AC 0 (FETCH (MEM ST) (FETCH (ACS ST) 15))
                     (SET.AC 15 (SUB1 (FETCH (ACS ST) 15))
                             ST)))
 (* We don't know if the new PC is supposed to permit a negative
    displacement or not, i.e., whether we can really use our
    8-bit adder. The ISP is wrong because it doesn't reference
    memory, it just loads ac 15 into ac 0)))

(DEFN JSS (WRD1 WRD2 ST)
 (SET.PC (MAR WRD2 ST)
         (SET.MEM (B930.ADD.8BIT (FETCH (ACS ST) 15) 1)
                  (B930.ADD.8BIT (PC ST) 2)
                  (SET.AC 15
                          (B930.ADD.8BIT (FETCH (ACS ST) 15) 1)
                          ST)))
 (* We don't really know the order of things.  Is the stack
    smashed and the stack pointer bumped before or after the MAR
    calculation?  Another question concerns the right half of
    the first word of the instr.  The manual does not specify
    whether those bits are important or not.  Our dispatcher,
    EXEC17, treats them as though the manual said they were
    don't cares.))


(DEFN RPS (WRD1 WRD2 ST)
 (IF (ZEROP (FETCH (ACS ST) 15))
     (SET.ERROR T ST)
     (SET.PC (B930.ADD.8BIT (FETCH (MEM ST)
                                   (FETCH (ACS ST) 15))
                            (D WRD2))
             (SET.AC 15
                     (SUB1 (FETCH (ACS ST) 15))
                     ST)))
 (* Is the right half of the first word of the instr important?
    Our EXEC17 treates it as don't care.))

(DEFN POPF (WRD1 WRD2 ST)
  (IF (ZEROP (FETCH (ACS ST) (B WRD2)))
      (SET.ERROR T ST)
      (SET.PC (B930.ADD.8BIT (PC ST) 2)
              (SET.SW (FETCH (MEM ST) (FETCH (ACS ST) (B WRD2)))
                      (SET.AC (B WRD2)
                              (SUB1 (FETCH (ACS ST) (B WRD2)))
                              ST))))
  (* What is the difference between the "status word" of the
     programmers manual and the "switch register" of the ISP?
     Is the manual setting of switches and POPF the only
     way of setting SW?))

(DEFN PUSHF (WRD1 WRD2 ST)
 (SET.PC (B930.ADD.8BIT (PC ST) 2)
       (SET.MEM (B930.ADD.8BIT (FETCH (ACS ST) (B WRD2)) 1)
                (SW ST)
                (SET.AC (B WRD2)
                       (B930.ADD.8BIT (FETCH (ACS ST)
                                             (B WRD2))
                                      1)
                       ST)))
 (* See POPF))

(DCL EXECR (WRD1 WRD2 ST)(* We should think carefully about
                            what the PC is set to when the instr
                            is executed.))
(DEFN EXEC07 (WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 13)
     (IF (AND (EQUAL (OP1 WRD2) 1)
              (EQUAL (AC WRD2) 1))
         (LDM WRD1 WRD2 ST)
         (SET.ERROR T ST))
 (IF (EQUAL (OP2 WRD1) 14)
     (IF (AND (EQUAL (OP1 WRD2) 1)
              (EQUAL (AC WRD2) 1))
         (STM WRD1 WRD2 ST)
         (SET.ERROR T ST))
 (IF (EQUAL (OP2 WRD1) 15)
     (IF (AND (EQUAL (A WRD1) 0)
              (EQUAL (IBIT WRD2) 0)
              (EQUAL (OP1 WRD2) 4))
         (PADDM/POPM/PUSHM WRD1 WRD2 ST)
         (SET.ERROR T ST))
     (SET.ERROR T ST))))
 (* The ISP says that any OP2 other than 13, 14, and 15 is a no op;
    we say error.  The programmers manual implies that in addition to
    the conditions on OP2 of WRD1 there are specific bit patterns required
    in WRD2.  We cause errors if these bits are not set correctly.
    EXCEPT, the manual says that A of WRD1 in PUSHM is don't care and
    we require zeroes as in PADDM.))


(DEFN EXEC00 (WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 0)
     (TRA/NOP WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 1)
     (DECEQ WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 2)
     (LCM WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 3)
     (RLS WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 4)
     (CONT WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 5)
     (DECNE WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 6)
     (ANDOP WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 7)
     (RLL WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 8)
     (ADDR WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 9)
     (IR/CLA WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 10)
     (OROP WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 11)
     (MPY WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 12)
     (CLAO/SUBR WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 13)
     (ACM WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 14)
     (CMPR WRD1 WRD2 ST)
     (DIV WRD1 WRD2 ST)))))))))))))))))

(DEFN EXECF (WRD1 WRD2 ST)
 (IF (EQUAL (OP3 WRD1)  0)
     (SFE1 WRD1 WRD2 ST)
 (IF (EQUAL (OP3 WRD1)  1)
     (SFE2 WRD1 WRD2 ST)
 (IF (EQUAL (OP3 WRD1)  2)
     (SFE3 WRD1 WRD2 ST)
 (IF (EQUAL (OP3 WRD1)  3)
     (SRIE WRD1 WRD2 ST)
 (IF (EQUAL (OP3 WRD1) 4)
     (SROV WRD1 WRD2 ST)
 (IF (EQUAL (OP3 WRD1) 5)
     (SFIR WRD1 WRD2 ST)
 (IF (EQUAL (OP3 WRD1) 6)
     (SRF1 WRD1 WRD2 ST)
 (IF (EQUAL (OP3 WRD1) 7)
     (SRF2 WRD1 WRD2 ST)
 (IF (EQUAL (OP3 WRD1) 8)
     (STE1 WRD1 WRD2 ST)
 (IF (EQUAL (OP3 WRD1) 9)
     (STE2 WRD1 WRD2 ST)
 (IF (EQUAL (OP3 WRD1) 10)
     (STE3 WRD1 WRD2 ST)
 (IF (EQUAL (OP3 WRD1) 11)
     (SSIE WRD1 WRD2 ST)
 (IF (EQUAL (OP3 WRD1) 12)
     (SSOV WRD1 WRD2 ST)
 (IF (EQUAL (OP3 WRD1) 13)
     (STIR WRD1 WRD2 ST)
 (IF (EQUAL (OP3 WRD1) 14)
     (SSF1 WRD1 WRD2 ST)
     (SSF2 WRD1 WRD2 ST)))))))))))))))))

(DEFN EXEC10 (WRD1 WRD2 ST)
  (IF (EQUAL (OP2 WRD1) 0)
     (SLSA WRD1 WRD2 ST)
  (IF (EQUAL (OP2 WRD1) 1)
     (SLLA WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 2)
     (SKGT WRD1 WRD2 ST)
  (IF (EQUAL (OP2 WRD1) 3)
     (SKLT WRD1 WRD2 ST)
  (IF (EQUAL (OP2 WRD1) 4)
     (SLSL WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 5)
     (SLLL WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 6)
     (SKGE WRD1 WRD2 ST)
  (IF (EQUAL (OP2 WRD1) 7)
     (SKLE WRD1 WRD2 ST)
  (IF (EQUAL (OP2 WRD1) 8)
      (SRSA WRD1 WRD2 ST)
  (IF (EQUAL (OP2 WRD1) 9)
      (SRLA WRD1 WRD2 ST)
  (IF (EQUAL (OP2 WRD1) 10)
      (SKEQ WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 11)
      (EXECF WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 12)
     (SRSL WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 13)
     (SRLL WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 14)
     (SKNE WRD1 WRD2 ST)
     (IAR WRD1 WRD2 ST)))))))))))))))))

(DEFN EXEC17 (WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 0)
     (DADDR WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 1)
     (DSUBR WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 3)
     (EXOR WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 12)
     (HALT WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 13)
     (RET WRD1 WRD2 ST)
 (IF (EQUAL (OP2 WRD1) 15)
     (IF (EQUAL (OP1 WRD2) 1)
         (IF (EQUAL (AC WRD2) 0)
             (JSS WRD1 WRD2 ST)
         (IF (EQUAL (OP2 WRD2) 2)
             (IF (EQUAL (IBIT WRD2) 0)
                 (RPS WRD1 WRD2 ST)
                 (SET.ERROR T ST))
         (IF (EQUAL (OP2 WRD2) 14)
             (IF (EQUAL (IBIT WRD2) 0)
                 (EXECR WRD1 WRD2 ST)
                 (SET.ERROR T ST))
             (SET.ERROR T ST))))
    (IF (EQUAL (OP1 WRD2) 2)
        (IF (EQUAL (OP2 WRD2) 0)
            (IF (EQUAL (IBIT WRD2) 0)
                (DMPY WRD1 WRD2 ST)
                (SET.ERROR T ST))
        (IF (EQUAL (OP2 WRD2) 1)
            (IF (EQUAL (IBIT WRD2) 0)
                (DACM WRD1 WRD2 ST)
                (SET.ERROR T ST))
            (SET.ERROR T ST)))
    (IF (EQUAL (OP1 WRD2) 4)
        (IF (EQUAL (OP2 WRD2) 5)
            (IF (EQUAL (IBIT WRD2) 0)
                (POPF WRD1 WRD2 ST)
                (SET.ERROR T ST))
            (IF (EQUAL (OP2 WRD2) 6)
                (IF (AND (EQUAL (IBIT WRD2) 0)
                         (EQUAL (A WRD2) 15))
                    (PUSHF WRD1 WRD2 ST)
                    (SET.ERROR T ST))
                (SET.ERROR T ST)))
        (SET.ERROR T ST))))
     (SET.ERROR T ST)))))))

(DEFN EXECUTE (WRD1 WRD2 ST)
 (IF (EQUAL (OP1 WRD1) 0)
  (IF (EQUAL (IBIT WRD1) 0)
      (EXEC00 WRD1 WRD2 ST)
      (EXEC10 WRD1 WRD2 ST))
 (IF (EQUAL (OP1 WRD1) 1)
     (IF (EQUAL (AC WRD1) 0)
         (JU WRD1 WRD2 ST)
     (IF (EQUAL (AC WRD1) 1)
         (JSA0 WRD1 WRD2 ST)
     (IF (EQUAL (AC WRD1) 2)
         (JSA1 WRD1 WRD2 ST)
         (JMA0 WRD1 WRD2 ST))))
 (IF (EQUAL (OP1 WRD1) 2)
     (ADD WRD1 WRD2 ST)
 (IF (EQUAL (OP1 WRD1) 3)
     (SUB WRD1 WRD2 ST)
 (IF (EQUAL (OP1 WRD1) 4)
     (CMP WRD1 WRD2 ST)
 (IF (EQUAL (OP1 WRD1) 5)
     (LOAD WRD1 WRD2 ST)
 (IF (EQUAL (OP1 WRD1) 6)
     (STO WRD1 WRD2 ST)
     (IF (EQUAL (IBIT WRD1) 0)
         (EXEC07 WRD1 WRD2 ST)
         (EXEC17 WRD1 WRD2 ST))))))))))

(DEFN B930 (ST INSTR.CNT INTER.INSTR.LST)
  (IF (ZEROP INSTR.CNT)
      (LIST ST INSTR.CNT)
      (IF (ERROR ST) (LIST ST INSTR.CNT)
          (IF (HALT ST) (LIST ST INSTR.CNT)
              (B930
               (IF (AND (IE ST)
                        (MEMBER INSTR.CNT INTER.INSTR.LST))
                    (EXECUTE (FETCH (MEM ST) 2001Q) (FETCH (MEM ST) 2002Q) ST)
                    (EXECUTE (FETCH (MEM ST) (PC ST))
                             (FETCH (MEM ST) (PC+1 ST))
                             ST))
               (SUB1 INSTR.CNT)
               INTER.INSTR.LST))))
  (* We assume from Chuck's code rather than the programmers manual
    or ISP that interrupts jump to 2001 octal = 2001Q
    For all purposes in the execution of the instruction at 2001,
    the PC points to the instruction we were about to execute.)

)))

STOP
```

CHAPTER 19

VERIFICATION OF NUMERICAL ALGORITHMS

1 Overview of Correctness Proofs

During the past few years there has been very substantial progress in program verification techniques that employ formal methods of program specification together with machine-aided proofs of correctness. All of these methods employ some variation of the inductive-proof schema originally proposed by Floyd [2]. The general approach is to annotate the text of a program with assertions about the relations among program variables at selected points between executable program statements, particularly at the entrance and exit points of program loops. Beginning with an input assertion that describes the domains and initial values of program input variables, every path through the program forms an "assertion chain" that leads to the output assertion. The input and output assertions together constitute a formal specification of the intent of the program. If it can be shown that the program terminates, and that the truth of each assertion in every chain logically implies the truth of its successor then a proof of correctness of the program with respect to its specification has been demonstrated.

As an aid to formulating the intermediate assertions, Hoare [3] has suggested a uniform approach in which each different type of high-level language statement is associated with a "rule" of the form: P{statement}Q, where the informal interpretation is that if P is a true proposition about program values before execution of {statement}, then Q is necessarily true afterwards. Once the high-level language statements have been "axiomatized" in this way, a system called a verification condition generator (VC-generator) can be used to automatically construct the intermediate assertions and to combine the ones in each chain into propositions (theorems) to be proved. The resulting theorems may be proved "by hand" (at the risk of making human errors) or submitted to an automatic deductive system (theorem prover) for verification. Several such theorem provers have been constructed during the past ten years. The most advanced of these is the (SRI) Boyer-Moore system [1], which currently can apply inductive reasoning to sentential logical formulas involving recursive functions, and linear arithmetic in the integer domain.

Most, if not all, of the successful applications of these ideas have been concerned with programs that presume exact representations of all data objects. The published literature on formal program proving topics, which currently comprises about 500 papers and reports, contains only a handful that addresses problems in the numerical algorithm category. The paucity of results in this area can be explained by the fact that in a general setting the problems of formal program specification and error analysis of numerical algorithms appear to be extremely difficult. However, as we shall observe in the following section, there are some classes of nontrivial and useful numerical algorithms for which mechanical verification problems seem to be more tractible. Part of the proposed study would be devoted to finding a good characterization of the types of numerical problems that can be successfully handled by the method explained below. Another part would be concerned with generalizing the technique to cope with a wider variety of algorithmic types.

1 METHOD OF APPROACH

"You must always invert." -- C. G. J. Jacobi.

1.1 The Basic Ideas

Our proposed approach differs from the customary methods of program proof described above in several respects that are believed to be novel in concept, and unique in providing opportunities for exploiting the capabilities of SRI's several mechanical deductive systems. The general strategy is based on the observation that it is usually quite difficult (or impossible) to compose the formal output-specification for a numerical algorithm prior to it's specific implementation in code. Since the output values computed by a program generally will not exactly satisfy any mathematical identities or relationships used in the computation, one does not know in advance precisely what to say in terms of an I/O assertion that will be both correct and acceptable to the program user.

On the other hand, the specification of the "mathematical intent" of a numerical algorithm is usually fairly easy. The computational plan will typically use a combination of mathematical relations that are already very well specified by formal analytic definitions. Moreover, these relations, identities etc., are usually applied in a well-ordered sequence that would survive scrutiny by the so-called "social process" of examination, and consensus on just what mathematical object the program was intended to compute. The latter facts suggest the following strategy for specification and proof, which inverts the usual order of proceedings:

- Defer the construction of a formal program sepecfication with respect to I/O assertions until the correctness of the program with respect to an abstract mathematical model of program intent has been demonstrated.

- From the "partial" program specification in terms of the mathematical description of the object to be computed, prove that an abstract machine (using infinite-precision arithmetic) would compute that object exactly.

- Next, prove that the computational sequences of arithmetic operations that occur in the abstract machine must be precisely the same at every step as those occurring on an actual machine (with finite-precision arithmetic), executing the same program.

- Following this, use a VC-generator that "knows" about the semantics of arithmetic operations (on the actual machine) to annotate the program with assertions that bound (or in some circumstances estimate) the differences between the actual machine state variables and the corresponding ones of the abstract machine.

- Finally, construct the formal program specification by combining the verification conditions into theorems about computational error that can be proved with mechanical assistance.

1.2 The Notion of Separability

Our general approach simplifies the problem of reasoning about the correctness of numerical algorithms by separating the issues of "mathematical correctness" from those of computational-error analysis. The property of mathematical correctness is essentially captured by the notion of executing a program on an abstract machine that carries out perfect (infinite precision) arithmetic. A program is a description of a sequence of arithmetic operations that is alleged to "construct" some mathematical object--for example, the sum of n terms of the Taylor series for sin(x). Proof of mathematical correctness consists of showing by induction, symbolic evaluation, or otherwise that the sequence of operations carried out by the abstract machine produces a result that coincides exactly with some textbook formula, identity, or mathematical definition that is accepted (by human inspection) as a precise specification of the mathematical intent of the program. We believe that existing symbolic evaluation tools can be used to prove this correspondence. However, it may be more convenient to construct our own symbolic evaluators for this purpose.

Having demonstrated mathematical correctness, the next step is to show that the program, when executed on an actual machine of limited precision, will follow exactly the same sequence of computational steps as the abstract machine (for all input data values). This is necessary because otherwise the two machines would not always be computing the same mathematical object. To prove identity of the computational sequences requires some reasoning about branching tests in the program. The property is certainly true if no branching test ever refers to a value that has been computed from imprecisely represented values--and that is quite easy to determine mechanically. It may also be true in other circumstances (such as those occurring in our example below). In such cases we need some additional reasoning to exhibit the correspondence between the computational sequences. Often this will be trivial to demonstrate because the branch test is made on some value (such as an input argument) that necessarily must have the same value on both the abstract and actual machines.

Once the above correspondence has been established, all questions relating to the computational goals of the program (even its mathematical intent!) have been abstracted away. The only thing left is a specification of the domain of input values and a now "meaningless" but definite sequence of arithmetic operations on data. These operations need to be axiomatized-- for example, by replacing the operator + with a Boyer-Moore function-definition that simulates in the discrete-integer domain what happens in some reasonable implementation of finite-precision arithmetic, e.g. the IEEE standard. As a practical matter, such a model of arithmetic is going to be based on some some form of machine-level binary representation, and it will often be convenient to reason about computational error in terms of what happens to the low-order bits of the machine-representation of a value.

Finally, we need to provide assertions about error bounds and prove that the given computational sequence always yields a value representation in that range. A typical assertion might be that an output value v computed on the actual machine will not differ in the first k bits of significance from the v computed on the abstract machine. There are mechanical aids (VC-generators) that can help us formulate such assertions, and automatic deductive systems can help us prove the resulting theorems. An important part of the proposed research would be to find good ways to mechanize the latter steps in the proof process.

Following this plan, we wind up with a program that has been proved correct with respect to a particular specification of output error. The specification may be weaker than is acceptable to a user of the program, or it may be better than he might have expected. In any case it is correct.

1.3 An Annotated Example

The following example is intended to illustrate the above ideas. It is a FORTRAN program to calculate an approximation to the mathematical function exp(Z) for a domain of positive or negative values of Z such that the result would not cause overflow of single-precision values on a machine with real representations equivalent to those of the DEC KL-10. This is not a "toy" program. It is a reasonably fast and relatively accurate algorithm of the sort one might write (as a first attempt) for use on a machine where the system-library did not contain the exponential function. We do not make any claims for optimality--better routines can be written without descending to the machine-code level (indeed, the proof process reveals two possible improvements to the algorithm). Refinements of an algorithm are evident whenever the generation of the I/O specification reveals where the main sources of computational error occur.

The program:

```
      FUNCTION GEXP(Z)

      IF(Z.GE.0.0) GOTO 10
      X = -Z                          !
      GOTO 15                         !    Segment 1
10    X = Z                           !
                                      !
15    IF(X.GT.87.0) X = 87.0

      E = 2.7182818285                !
      ANS = 1.0                       !    Segment 2
      N = X                           !
      X = X-N                         !

20    IF(N.EQ.0) GOTO 30              !
      J = N/2                         !
      IF(N.GT.2*J) ANS = E*ANS        !    Segment 3
      IF(J.GT.0) E = E*E              !
      N = J                           !
      GOTO 20                         !

30    SERIES = 1.0                    !
      DO 40 J = 1,12                  !    Segment 4
40    SERIES = 1.0+X*SERIES/(13-J)    !

                                      !
      ANS = ANS*SERIES                !    Segment 5
      IF(Z.LT.0.0)  ANS = 1.0/ANS     !
      GEXP = ANS

      RETURN
      END
```

Discussion of the program:

First let's walk through the program, just to observe the plan of the computation.

Segment 1 notes whether the input argument of the function is negative and, if so, makes a positive copy of it. We make this code explicit, rather than using the statement

```
        X = ABS(Z)
```

so that correctness of the program will not depend on the correctness of some other function, however trivial.

The statement labeled 15 is a feeble attempt to cope with the possibility of exponent overflow. Since FORTRAN has no explicit means for exception handling, the statement merely insures that we will always return a value that is (approximately) between exp(-87.0) and exp(87.0) both of which are comfortably represented on the DEC KL-10 or any machine that implements the proposed IEEE standard of a single-precision 8-bit exponent. In the rest of the program we ignore the possiblity of underflow on the assumption that underflow defaults to zero without raising an exception. If this were not the case, as in the current implementation of Pascal on the KL-10, then we would have to insert some statements in the above program to check for and deal with this contingency.

Segment 2 assigns to N the integer part of X and adjusts X to be the fractional part of its previous value.

Segment 3 implements the Floyd-King algorithm for raising a value (in this case, E = 2.71828... to an integer exponent). Here the exponent is the integer value N.

Segment 4 evaluates the sum of 13 terms of the Taylor series for exp(X) using a "nested" product expansion, that is, the Horner rule for polynomial evaluation.

Segment 5 forms the product of the approximation to exp(N) from Segment 3 and the value obtained by the computation of the series. It then checks whether the original argument to GEXP(Z) was negative and if so, returns the reciprocal of the previous value.

1.4 Outline of the Proof of Correctness

The plan of the proof follows the scheme described in 1.1. above. To avoid being too tedious, we will merely sketch the main outline of the proof, leaving out a number of small details which would, of course, have to be handled in an actual machine-aided proof of correctness with respect to the final I/O specification.

1.4.1 Mathematical Correctness

First comes the issue of mathematical correctness. A partial specification of mathematical intent for the program (executed on the abstract machine) would state that it is supposed to compute the object represented by:

```
        if Z >= 0.0 then exp(N) * T(13,X) else 1 / (exp(N) * T(13,X))
```

where T(j,x) is the sum of j terms of the Taylor series for exp(x), and

```
N = abs(Z) div 1,     X = abs(Z) mod 1.
```

Note that this specification does NOT say that the abstract machine is supposed to compute exp(Z). Even the abstract machine is computing an approximation to exp(Z), and it is matter of separate analysis and proof to establish that this object would (if computed correctly) have satisfactory accuracy. As a practical matter, it is easy to show in this case that the abstract machine will always compute an approximation to exp(Z) that is more accurate than can be represented in single precision on the DEC KL-10, but it should be emphasized that questions of mathematical intent can and should be treated separately.

Next, does the program actually realize its mathematical intent? A mechanical trace of the program flow-of-control will show that if the Segments 3 and 4 terminate with values for ANS and SERIES, and if the familiar identities,

```
exp(a * b) = exp(a) * exp(b),     exp(-Z) = 1/exp(Z)
```

are assumed, then the program meets the above specification. Segment 3 is one implementation of the Floyd-King algorithm which has an elegant mechanical proof of correctness and termination. Of course this needs to be re-proved in the context of this particular program. Here, we would make use of our currently available FORTRAN VC-generator and the Boyer-Moore system. Segment 4 must terminate because the loop index runs through the literal values 1..12, and the value returned can be shown by symbolic evaluation to coincide exactly with the sum of the first 13 terms of the textbook Taylor series for exp(X).

This disposes of the question of mathematical correctness except for the exceptional case where the input argument exceeds 87.0 in absolute value. This contingency should be treated as part of the final I/O assertion.

1.4.2 Computational Equivalence

Next we must demonstrate that the computational sequences of the real and abstract machines are equivalent (identical).

Segment 1 of the program contains no arithmetic operations on real representations (hereafter called reals) except for the unary minus operation and a test and branch on zero. We will assume that in our implementation of reals, each representable value has a representable negative inverse. This is not true for the KL-10, but is true (or should be) for the IEEE standard. Moreover, the value 0.0 should be exactly (and uniquely) representable in such a standard, so that the abstract and actual machines must take the same branch at the first program statement. It also follows that there is no loss of accuracy in Segment 1. All this is mechanically verifiable.

The statement labled 15 involves a branch on a real value. But that value has not suffered any contamination by arithmetic operations, so this is a test that must lead to the same computational sequence in both the abstract and actual machines (again mechanically checkable).

In Segment 2 we compute an integer N <= 87 and the fractional part of X, both necessarily exact, because N is exactly representable and, therefore, so is the value X - N.

In Segment 3 we use the Floyd-King algorithm to raise an internal representation E (of the "mathematical" e) to the Nth power. The three branching tests (IF statements) in his segment each refer only to integer values computed from N. Since N must have the same value on both machines, the computational sequences are necessarily the same for Segment 3.

Segment 4 contains a DO-loop whose implied branching test does not depend on the values of reals. Indeed, the body of the loop is executed exactly 12 times on either machine, so computational equivalence is obvious.

Finally, Segment 5 makes a branch on the value of the input argument Z, which has never been changed by the program. Consequently, both machines take the same path at this step.

The above reasoning, which for this particular program was all essentially mechanical, establishes that the computational sequences on both the actual and abstract machines are identical. We can now turn to the more interesting part of the exercise, i.e., the error analysis that will form the basis of the I/O specification.

1.4.3 Error Analysis

We have already observed that no program error is generated in Segments 1. and 2., except for the assignment to E. Here E is a real (introduced as a literal program constant in decimal notation) whose internal representation may differ from the abstract machine value in the least significant bit of the machine value. Incidentally, we are aware of some cases in which poor implementation of a conversion algorithm causes the error to be even worse. It is necessary to know the precise semantics of the interpretation of literal program constants before an assertion can be made about the error introduced at this step. We will return to this question in the next section.

The error of Segment 3 arises from two sources--the product E*E and the product E*ANS. According to one reasonable axiomatization of real arithmetic, the relative error of the result of a multiplication is the sum of the relative errors of its components, plus a possible additional error of half the value of the least significant bit of the representation of the product. A Floyd-like assertion to that effect can be constructed by a VC-generator and attached automatically to the appropriate points of the program. If we choose to reason about maximum error bounds in terms of integer units representing a "half-bit" in the least significant place of the representation, then an error assertion of the form

```
        EA = EA + EE + 1
```

could replace the asignment ANS = E*ANS in the program. In this, EA and EE stand for the errors of ANS and E, respectively. Similarly the statement

```
        EE = EE + EE + 1
```

could replace the assignment E = E*E. These statements may be interpreted as either assertions about error bounds, or as symbolically executable state- ments about error values. Note that error values such as EE and EA do not appear as explicit program variables.

We would now like to prove a theorem that bounds the error EA in the value of ANS computed by Segment 3. But what is the theorem? By informal reasoning one can conclude that the assignment E = E*E is executed log2(N) times, and that the assignment ANS = E*ANS is executed a number of times equal to the number of 1's in the binary representation of N. Therefore, it seems that the error is likely to be a discouragingly complicated function of N. To investigate this possibility we first converted the symbolic execution of Segment 3 into an equivalent Boyer-Moore function-definition. This could be done mechanically; ours is a hand translation. The definition looked like this:

```
(DEFN ERROR (N EE EA)
  (IF (ZEROP N) (FIX EA)
    (IF (ODD N)
      (ERROR (QUOTIENT N 2)
          (ADD1 (PLUS`EE EE))
          (ADD1 (PLUS EE EA)))
      (ERROR (QUOTIENT N 2)
          (ADD1 (PLUS EE EE))
          EA))))
```

Next we executed this function on several combinations of values for the input errors of the arguments EE and EA. In the actual program the arguments to ERROR are (N 1 0). After a few such tries, the pattern of output values suggested the following surprising conjecture:

```
ERROR(N EE EA) = EA + N * (EE + 1)
```

which (if true) would be a very simple relationship between the input and output errors of Segment 3 We next asked the Boyer-Moore system to prove the above conjecture. In their syntax this is simply

```
(PROVE.LEMMA FACT.ABOUT.ERROR (REWRITE)
  (EQUAL (ERROR N EE EA)
      (PLUS EA (TIMES N (ADD1 EE)))))
```

The mechanical deductive system easily proved the truth of the above lemma, therefore the truth of our conjecture, and therefore, in the context of the program, that EA = 2*N.

This fact about the error induced by Segment 3 turns out to be much simpler than might have been expected from an inspection of the program. Moreover, we have proved its correctness for any call of GEXP(Z) under the particular axiomatization of error arithmetic modeled above. The analysis of the error induced by Segment 4 of the program follows much the same scheme. We omit the details, since the proof was carried out by hand rather than mechanically. The result is that the computation of the value SERIES cannot be in error by more than 3 "half-bit" units. Here the equivalent mechanicaly proved theorem should show that if the domain of values for X is 0.0..0.5 instead of 0.0..1.0 then the maximum error would be one unit instead of three.

By combining the error analyses of Segments 2, 3, and 4 with thr trivial analysis of Segment 5, one obtains an output assertion that can be stated informally as follows:

If -87.0 <= Z <= 87.0 then the program GEXP(Z) computes the intended mathematical object (described in II-D) with a difference of not more than 2*N + 5 units of "half-bit" significance. Otherwise it computes one of the values GEXP(87.0) or GEXP(-87.0).

We omitted proving the last part of the output specification, but it follows easily from the fact that ANS can be proved to be monotone-increasing in Segment 3, and therefore overflow cannot occur on the actual machine.

1.5 Conclusions

What can be learned from the foregoing example? First, we find that it is possible to prove correctness of a program with respect to a realistic specification when that specification is the last thing that we discover. Second, we observe that the largely mechanical analysis and proof process furnishes a useful guide to the construction of a "better" program implementation of the intended mathematical object. In the case illustrated above, it is obvious that most of the error arises in the generation of integral powers of e. This suggests the use of a small table of accurate values for powers of e, indexed by N. The analysis of the evaluation of SERIES suggests that a further range reduction (so that the value of X lies between 0.0 and 0.5) would be beneficial. The resulting new program would be faster, more accurate, and easier to prove correct.

In summary, we have suggested a method of approach to the proof of correctness of a certain class of numerical algorithms, and tried to indicate by examining a typical example, how mechanical proof aids such as the Boyer-Moore deductive system can be expected to handle many of the details involved in such exercises. Some of the steps in our outline of the proof process have involved human judgments rather than machine deductions, but most of these decisions were about obvious mathematical properties of the function being computed. Consequently, these facts could be introduced to the deductive system as axioms without incurring any human doubt about the validity of the mechanical portion of the proof process.

References

[1] Boyer, R.S. and Moore, J S.
A Computational Logic.
Academic Press, 1979.

[2] Floyd, R.W.
Assigning Meaning to Programs.
In *Proceedings of a Symposium on Applied Mathematics,* pages 19-32. American Mathematical Society, 1967.

[3] Hoare, C.A.R. and Wirth, N.
An Axiomatic Definition of the Programming Language Pascal.
Acta Informatica 2:335-355, 1973.

CHAPTER 20

VERIFICATION OF FLIGHT CONTROL PROGRAMS

The Use of a Formal Simulator to Verify a Simple Real Time Control Program

Robert S. Boyer
Milton W. Green
J Strother Moore

1.1 Abstract

We present an initial and elementary investigation of the formal specification and mechanical verification of programs that interact with environments. We describe a formal, mechanically produced proof that a simple, real time control program keeps a vehicle on a straightline course in a variable crosswind. To formalize the specification we define a mathematical function which models the interaction of the program and its environment. We then state and prove two theorems about this function: the simulated vehicle never gets farther than three units away from the intended course and homes to the course if the wind ever remains steady for at least four sampling intervals.

Key Phrases: autopilot, formal specification, mechanical theorem-proving, modeling, program verification, real time control, simulation.

1.2 Background

Formal computer program verification is a research area in computer science aimed at aiding the production of reliable hardware and software. Formal verification is based on the observation that the properties of a computer program are subject to mathematical proof.

1.2.1 Program Verification

Consider, for example, the following FORTRAN program for computing integer square roots using a special case of Newton's method[1]

[1] V. Kahan, of U.C. Berkeley, reports that the algorithm was in fact advocated by Heron of Alexandria before 400 A.D.

```
      INTEGER FUNCTION ISQRT(I)
      IF ((I .LT. 0)) STOP
      IF ((I .GT. 1)) GOTO 100
      ISQRT = I
      RETURN
100   ISQRT = (I / 2)
200   IF (((I / ISQRT) .GE. ISQRT)) RETURN
      ISQRT = ((ISQRT + (I / ISQRT)) / 2)
      GOTO 200
      END
```

It is possible to prove, mathematically, that the program satisfies the following (informally stated) specification:

> If the program is executed on a machine implementing ANSI FORTRAN 66 or 77 [13, 1], and the input to the program is a nonnegative integer representable on the host machine, then the program terminates, causes no arithmetic overflow or other run time error, and the output is the largest integer whose square is less than or equal to the input.

Such program proofs are generally constructed in two steps. In the first step, the code and its mathematical specifications are transformed into a set of formulas to be proved. In the second step the formulas are proved using the usual laws of logic, algebra, number theory, etc. For an introduction to program verification, see [9, 10, 11, 2].

Because the mathematics involved in program verification is often tedious and elementary, mechanical program verification systems have been developed. One such system is described in [6]. That system handles a subset of ANSI FORTRAN 66 and 77 and has verified the above mentioned square root program [8], among others.

To admit mechanical proof, the specifications must be written in a completely formal notation. For example, in the square root example the specification of the program's output is:

$$j^2 \le i < (j+1)^2 \ \& \ 0 \le j,$$

where it is understood that i refers to the value of the FORTRAN variable I on input to ISQRT and j refers to the value returned by ISQRT.

1.2.2 Boebert's Challenge

The square root program is a good example of a programming task in which the specification "obviously" captures the intent of the designer. At issue is whether some algorithm satisfies the specification. However, for some programming tasks it is difficult to find mathematical specifications that obviously capture the designer's intention. Real time control programs are an especially important example of such tasks.

To spur the interest of the program verification research community to consider such specification problems, a version of the following problem was proposed by Earl Boebert.[2]

[2] Honeywell Systems and Research Center, 2600 Ridgway Parkway, Minneapolis, Minnesota 55413

Consider the task of steering a vehicle down a straightline course in a crosswind that varies with time. Let the desired course be down the x-axis of a Cartesian plane (i.e, towards increasing values of x). Suppose the vehicle carries a sensor that, in each sampling interval of time, reads either +1, 0, or -1, according to whether the vehicle is to the left of the course ($y>0$), on the course ($y=0$), or to the right of the course ($y<0$). Suppose also that the vehicle has some actuator that can be used to change the y-component of its velocity under the control of some program reading the sensor. Problem: state formally what it means to keep the vehicle on course and, for some particular control program, prove mechanically that the program satisfies its high level specification.

Observe that the problem necessarily involves a specification of the environment with which the program interacts. Furthermore, unlike the square root example, what is desired is not merely a description of a single input/output interchange between the environment and the program but rather the effects of repeated interchanges over time.

In this paper we describe one solution to Boebert's challenge. Our method involves writing a simulator for the system in formal logic. We present our formal simulator after explaining informally the model and control program we will use.

1.3 The Informal Model

The mechanized logic into which we cast the model provides the integers and other discrete mathematical objects but does not provide the rationals or reals.[3] Thus, we will measure all quantities, e.g., time, wind speed, vehicle position, etc., in unspecified integral units.

We ignore the x-axis and concentrate entirely on the y-axis. For example, we do not consider the x-component of the vehicle's velocity and we ignore any x-component of the wind velocity. Thus, our model more accurately represents a one-dimensional control problem, such as maintaining constant temperature in an environment where the outside temperature varies, or maintaining constant speed, as in an automobile's "cruise control."

We measure the wind speed, w, in terms of the number of units in the y-direction the wind would blow a passive vehicle in one sampling interval. We assume that from one sampling interval to the next w can change by at most one unit. Some such assumption is required since no control mechanism can compensate for an external agent capable of exerting arbitrarily large instantaneous forces. Thus, we assume that the wind speed at time t+1 is the speed at time t plus some increment, dw, that is either -1, 0, or 1.

```
     w(t+1) = w(t) + dw(t+1)
```

where

```
     dw(t+1) = -1, 0, or 1.
```

We permit the wind to build up to arbitrarily high velocities.

[3] This is not a limitation of mechanized logic in general. Several existing mechanical theorem-provers, e.g., those of Bledsoe's school [4, 3], and the MAXSYMA symbolic manipulation system [12], provide analytic capability.

At each sampling interval the control program may increment or decrement the y-component of its velocity (e.g., by turning a rudder or firing a thruster). We let v be the accumulated speed in the y-direction measured as the number of units the vehicle would move in one sampling interval if there were no wind. We make no assumption limiting how fast v may be changed by the control program; our illustrative program changes v by at most ±5 each sampling interval. We permit v to become arbitrarily large.

The y-coordinate of the vehicle at time t+1 is thus its y-coordinate at time t, plus the accumulated v at time t, plus the displacement due to the wind at time t+1:

$$y(t+1) = y(t) + v(t) + w(t+1).$$

The sensor reading at any time is the sign of y, sgn(y). The control program changes v at each sampling interval as a function of the current sensor reading (and perhaps previous readings). Our illustrative control program is a function of the current reading and the previously obtained reading:

$$v(t+1) = v(t) + \mathrm{deltav}(\mathrm{sen1},\mathrm{sen2})$$

where

$$\mathrm{sen1} = \mathrm{sgn}(y(t+1))$$

$$\mathrm{sen2} = \mathrm{sgn}(y(t)),$$

and deltav is the mathematical function specifying the output of the control program.

1.4 The Control Program

It is instructive to consider first the control program with the following specification:

$$\mathrm{deltav}(\mathrm{sen1},\mathrm{sen2}) = -\mathrm{sen1}$$

A steadily increasing wind can blow the vehicle arbitrarily far away from the x-axis. Furthermore, should the wind ever become constant, the vehicle begins to oscillate around the x-axis. See Figure 1.

The control program we consider includes a damping term that also causes the vehicle to resist more strongly any initial push away from the x-axis.

$$\mathrm{deltav}(\mathrm{sen1},\mathrm{sen2}) = -\mathrm{sen1} + 2(\mathrm{sen2}-\mathrm{sen1}).$$

See Figure 2 for an illustration of the behavior of the vehicle under this program.

The following trivial FORTRAN program implements this specification in the following sense. If SEN1 is the current sensor reading, sen1, and the value of the global variable SEN2 is the previous sensor reading, sen2, and sen1 and sen2 are both legal sensor readings, then at the conclusion of the subroutine, the global ANS is set to deltav(sen1,sen2) and the global SEN2 is set to sen1.

```
      SUBROUTINE DELTAV(SEN1)
      INTEGER SEN1, SEN2, ANS
      COMMON /DVBLK/SEN2, ANS
      ANS = ((2 * SEN2) - (3 * SEN1))
      SEN2 = SEN1
      RETURN
      END
```

Proving that the program satisfies its specification is, of course, trivial. At issue is whether the vehicle stays on course.

By observing the behavior of the simulated vehicle under several arbitrarily chosen wind histories we made two conjectures about the behavior of the vehicle:

1. No matter how the wind behaves (within the constraints of the model), the vehicle never strays farther than 3 units away from the x-axis.

2. If the wind ever becomes constant for at least 4 sampling intervals, the vehicle returns to the x-axis and stays there as long as the wind remains constant.

How can we state such specifications in a form that makes them amenable to mechanical proof?

1.5 Formalizing the Model

To state the conjectures formally we must formalize the model of the control program and its environment. We will define this model as a function in the same mechanized mathematical logic used by the FORTRAN verification system [6]. The logic and a mechanical theorem-prover for it are completely described in [5].

The syntax of the logic is akin to that of Church's lambda-calculus. If f is a function in the logic and e1 and e2 are two expressions in the logic, then we write (f e1 e2) to denote the value of f on the two arguments e1 and e2. The more traditional equivalent notation is f(e1,e2). For example, suppose ZPLUS is defined as the usual integer addition function. Then (ZPLUS X Y) is how we write X+Y. Thus, (ZPLUS 3 -10) = -7.[4]

Our formal model is expressed as a recursive function that takes two arguments, a description of the behavior of the wind over some time period and the initial state of the system. The value of the function is the final state of the system after the vehicle has traveled through the given wind under the direction of the control program. Thus, the recursive function may be thought of as a simulation of the model.

Formally, we let states be triples, <w,y,v>, containing the current wind speed, y-position of the vehicle, and accumulated v. The function STATE, of three arguments, is axiomatically defined to return such a triple, and the functions W, Y, and V are defined to return the respective components of such a triple. Thus, the expression (STATE 63 -2 -61) denotes a state in which the wind speed is 63, the y-position of the vehicle is -2, and the accumulated v is -61.

[4] This choice of notation is convenient because most symbols used in program specification are user-defined and do not have commonly accepted names or symbols. Furthermore, the uniformity of the syntax makes mechanical manipulation easier.

```
(W (STATE 63 -2 -61)) = 63
(Y (STATE 63 -2 -61)) = -2
(V (STATE 63 -2 -61)) = -61
```

The function NEXT.STATE is defined to return as its value the next state, given the change in the wind and the current state. The formal definition of NEXT.STATE is:

```
Definition.
(NEXT.STATE DW STATE)
  =
(STATE (ZPLUS (W STATE) DW)
       (ZPLUS (Y STATE) (V STATE) (W STATE) DW)
       (ZPLUS (V STATE)
              (DELTAV (SGN (ZPLUS (Y STATE)
                                  (V STATE)
                                  (W STATE)
                                  DW))
                      (SGN (Y STATE))))).
```

The definition of next state follows immediately from our equations for w(t+1), y(t+1) and v(t+1). The function DELTAV is formally defined as was deltav in our informal model.

The behavior of the wind over n sampling intervals is represented as a sequence of length n. Each element of the sequence is either -1, 0, or 1 and indicates how the wind changes between sampling intervals. Formally, a sequence is either the empty sequence, NIL, or is an ordered pair <hd,tl>, where hd is the first element of the sequence and tl is a sequence containing the remaining elements. Such pairs are returned by the function CONS of two arguments. The functions HD and TL return the respective components of a nonempty sequence, and the function EMPTYP returns true or false according to whether its argument is an empty sequence.

In general we are not interested in wind behaviors other than those permitted by our model. Thus, we define a function that recognizes when an arbitrary sequence consists entirely of -1's, 0's, and 1's.

```
Definition.
(ARBITRARY.WIND LST)
  =
(IF (EMPTY LST)
    T
    (AND (OR (EQUAL (HD LST) -1)
             (EQUAL (HD LST) 0)
             (EQUAL (HD LST) 1))
         (ARBITRARY.WIND (TL LST)))).
```

(ARBITRARY.WIND LST) returns true or false according to whether every element of LST is either -1, 0, or 1. The definition is recursive. The empty sequence has the property. A nonempty sequence has the property provided that (a) the HD of the sequence is -1, 0, or 1, and (b) the TL of the sequence (recursively) has the property.

The recursive function FINAL.STATE takes a description of the wind and an initial state and returns the final state:

```
Definition.
(FINAL.STATE L STATE)
  =
(IF (EMPTY L)
    STATE
    (FINAL.STATE (TL L)
                 (NEXT.STATE (HD L) STATE))).
```

Note that FINAL.STATE is recursively defined and may be thought of as simulating the state changes induced by each change in the wind.

We can now state formally the two properties conjectured earlier.

```
Theorem. VEHICLE.STAYS.WITHIN.3.OF.COURSE:
(IMPLIES (AND (ARBITRARY.WIND LST)
              (EQUAL STATE
                     (FINAL.STATE LST
                                  (STATE 0 0 0))))
         (AND (ZLESSEQP -3 (Y STATE))
              (ZLESSEQP (Y STATE) 3)))
```

This formula may be read as follows. If LST is an arbitrary wind history and STATE is the state of the system after the vehicle has traveled through that wind starting from the initial state <0,0,0>, then the y-coordinate of STATE is between -3 and 3. Put another way, regardless of how the wind behaves, the vehicle is never farther than 3 from the x-axis.

A formal statement of the second conjecture is:

```
Theorem. VEHICLE.GETS.ON.COURSE.IN.STEADY.WIND:
(IMPLIES (AND (ARBITRARY.WIND LST1)
              (STEADY.WIND LST2)
              (ZGREATEREQP (LENGTH LST2) 4)
              (EQUAL STATE
                     (FINAL.STATE (APPEND LST1 LST2)
                                  (STATE 0 0 0))))
         (EQUAL (Y STATE) 0))
```

The function STEADY.WIND recognizes sequences of 0's. The function APPEND is defined to concatenate two sequences. The formula may be read as follows. Suppose LST1 is an arbitrary wind history. Suppose LST2 is a history of 0's at least 4 sampling intervals long. Note that the concatenation of the two histories describes an arbitrary initial wind that eventually becomes constant for at least 4 sampling intervals. Let STATE be the state of the system after the vehicle has traveled through the concatenation of those two wind histories. Then the y-position of the vehicle in that final STATE is 0.

1.6 Proving the Conjectures

The foregoing conjectures can be proved mathematically. Indeed, they have been proved by the mechanical theorem-prover described in [5]. The key to the proof is that the state space of the vehicle can be partitioned into a small finite number of classes. In particular, any state <w,y,v> reachable under the model starting from <0,0,0> can be put into one of the following classes according to y and w+v:

y	w+v
-3	1
-2	1 or 2
-1	2 or 3
0	-1, 0 or 1
1	-2 or -3
2	-1 or -2
3	-1

The automatic theorem-prover is incapable of discovering this fact for itself. Instead, the human user of the theorem-prover may suggest it by defining the function (GOOD.STATEP STATE) to return true or false according to whether STATE is in one of the 13 classes above, and then commanding the theorem-prover to prove the following key lemma:

```
(IMPLIES (AND (GOOD.STATEP STATE)
              (OR (EQUAL DW -1)
                  (EQUAL DW 0)
                  (EQUAL DW +1)))
         (GOOD.STATEP (NEXT.STATE DW STATE))).
```

This theorem establishes that if the current state of the vehicle is one of the "good states" and the wind changes in an acceptable fashion then the next state is a good state. After proving this lemma (by considering the cases and using algebraic simplification) the theorem-prover can establish by induction on the number of sampling intervals that the final state of the vehicle is a good state. From that conclusion it is immediate that the y-position of the vehicle is within ±3 of the x-axis.

The proof of the second theorem is similar. The vehicle is in a good state after LST1 has been processed. But if the vehicle is in a good state and the wind remains steady for four sampling intervals, it is easy to show by cases and algebraic simplification that the vehicle returns to the x-axis with w+v=0. But in this case, it stays on the x-axis as long as w stays constant.

1.7 Comments on the Model

We have proved that the simulated vehicle stays on course under each of the infinite number of different wind histories to which it might be subjected under the model.

Just as the user of a square root or sorting subroutine must look at the specifications to determine whether the subroutine is suitable for his application, so too should the user of this control program. In particular, it is up to the user to determine whether the restrictions on the wind behavior and the model of the environment are sufficiently realistic for his application.

Here are a few of the more obvious oversimplifications:

- Real sensors sometimes give spurious readings due to vibration or other forms of disturbance. The program makes no allowance for such noise.

- No consideration is given to motion or forces in the x- or z-directions. Furthermore, no consideration is given to the orientation of the vehicle with respect to its preferred direction of travel.

- The model of the physics of the vehicle is too simple. The use of discrete measurement is unsatisfying but perhaps justifiable under suitable assumptions about scale. But many physical aspects of real control situations have been ignored: inertia, reaction times of the actuators, response time of the vehicle, maximum permitted g-forces.

Allowance for noise in the sensors can be handled by existing program verification technology. For example, if one provides redundant sensors and employs a signal select algorithm based on software majority voting, DELTAV can be rewritten to use an algorithm such as that verified in [7] to compute the majority sensor reading (if any). The proof that the vehicle stays on course can then be carried over directly if one is willing to assume that at each sampling interval a majority of the sensors agree.

However, the other two unrealistic aspects of our problem are more difficult to handle. While it is easy to define more sophisticated formal simulators it may well be practically impossible to prove interesting properties mechanically. Certainly the proof paradigm used here, depending as it did on the existence of a small partitioning of the state space, will not suffice for more sophisticated models.

1.8 Conclusion

We have illustrated how a formal simulator can be used to specify in a machine readable form the high level intention of a simple real time control program. We have also shown how such a program has been mechanically proved to satisfy its specifications.

Simulation programs are used today to test a variety of applications programs. Among the applications that come to mind are real time control, scheduling, and page fault handling in operating systems. Such simulators suffer the inaccuracy introduced by finite precision arithmetic and resources and in addition offer only the testing of the applications program on a finite number of situations.

Formal simulators are mathematical functions. They need not be realizable on machines and thus need not suffer resource limitations. In addition, formal simulators theoretically permit mechanical analysis of the behavior of the system in an infinite number of possible situations.

References

[1] *American National Standard Programming Language FORTRAN*
American National Standards Institute, Inc., 1430 Broadway, NY 10018, 1978.
Tech. Rept. ANSI X3.9-1978.

[2] Anderson, R. B.
Proving Programs Correct.
John Wiley & Sons, New York, NY, 1979.

[3] Ballantyne, A. M. and Bledsoe, W.W.
Automatic Proofs of Theorems in Analysis Using Non-Standard Techniques.
Technical Report ATP-23, Dept. of Math., University of Texas at Austin, Jul, 1975.

[4] Bledsoe, W.W. and Hines, L.M.
Variable Elimination and Chaining in a Resolution-based Prover for Inequalities.
In W. Bibel and R. Kowalski (editors), *5th Conference on Automated Deduction*, pages 70-87. Springer-Verlag, 1980.

[5] Boyer, R.S. and Moore, J S.
A Computational Logic.
Academic Press, New York, NY, 1979.

[6] Boyer, R.S. and Moore, J S.
A Verification Condition Generator for FORTRAN.
In R.S. Boyer and J S. Moore (editors), *The Correctness Problem in Computer Science*, . Academic Press, London, 1981.

[7] Boyer, R.S. and Moore J S.
MJRTY - A Fast Majority Vote Algorithm.
1981.
SRI International.

[8] Boyer, R.S. and Moore J S.
The Mechanical Verification of a FORTRAN Square Root Program.
SRI International.

[9] Floyd, R.
Assigning Meanings to Programs.
In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics*, pages 19-32. American Mathematical Society, Providence, RI, 1967.

[10] King, J.C.
A Program Verifier.
PhD thesis, Carnegie-Mellon University, 1969.

[11] Manna, Z.
Mathematical Theory of Computation.
McGraw-Hill Book Company, New York, NY, 1974.

[12] Moses.
Algebraic Simplification: A Guide for the Perplexed.
In *2nd Symposium on Symbolic and Algebraic Manipulation.* ACM, 1971.

[13] United States of America Standards Institute.
USA Standard FORTRAN.
Technical Report USAS X3.9-1966, USA Standards Institute, 10 East 40th Street, New York, NY, 1966.

CHAPTER 21

VERIFICATION OF HARDWARE LOGIC

Hierarchical Design and Verification for VLSI

Robert E. Shostak
W. David Elliott
Karl N. Levitt

Dramatic advances in LSI fabrication technology over the last few years have made it possible for the first time to bridge the gap between the high-level computer system architect and the integrated circuit designer. Standardized VLSI system design methodologies, for example, have permitted computer scientists with little or no previous hardware experience to map sophisticated computer architectures directly into silicon. Recognition of the vast potential of VLSI has already prompted researchers in diverse areas of computer science to apply their knowledge and ideas to this filed. The development of structured design disciplines such as the self-timed system (Muller [7], Seitz [12]) and synchronous system concepts is already well under way. The same holds of investigations into ways of exploiting the high levels of concurrency that VLSI makes possible (Foster, Kung [5]). Tools and methodologies that at one time fell exclusively within the province of software engineering are quickly making an impact in VLSI design. Powerful interactive layout systems such as those under development at MIT, Xerox, Cal Tech, and Stanford are exploiting the graphics, editing, and compiler technology that could once be found only in the context of programming environments.

As this trend continues, and as further advances in fabrication technology permit circuits of greater size and complexity, the problems of design cost and reliability attendant to large systems of any kind will become increasingly pressing. For several years, the Computer Science Laboratory at SRI has been concerned with solutions to these problems from the standpoint of large and/or critical software systems. We believe that the fruits of this work can be extended and applied to VLSI.

Our approach is predicated upon two key, mutually-reinforcing concepts: hierarchical design and formal verification.

The hierarchical design methodologies we have developed enable a system architect or design team to decompose a complex design into a formal hierarchy of levels of abstraction. The PSOS (Provably Secure) Operating System (Neumann, et al. [8]), for example, is specified using our methodology as a hierarchy in which successively lower levels of abstraction represent virtual machines that manage system resources successively closer to the hardware base. One could similarly decompose the design of a microprocessor, for instance, into a hierarchy of levels corresponding, say, to the user interface (top level), the register transfer level, the gate level, and the semiconductor level. Naturally, hardware designers have been thinking in terms of such levels of abstraction all along. The point of the design methodology is to formalize this process so that the various levels and their relationships to one another are specified ina clear, precise, and uniform way. The result is a design that is more readily understood, more easily modified, and much more likely to be reliable.

Formal hierarchical design also lends itself to a second key process: verification. What we mean by verification is the formal proof, in a mathematical sense, that a design meets its behavioral specifications. Verification differs from testing in the following important sense: whereas testing demonstrates proper operation of a design for a set of sample inputs, formal verification proves the correctness of the design over the range of all possible inputs. Although verification is generally a time-consuming process requiring the expertise of skilled individuals, its cost is easily justified were reliability is a critical requirement--and especially where thorough testing is difficult, as is often the case for VLSI designs. The techniques used for formal verification are useful even where the proof process is not carried to completion, because they force the designer both to specify and to understand his design clearly and precisely.

The Computer Science Laboratory is known for its state-of-the-art posture with respect to hierarchical design and formal verification. We propose the transfer of this expertise to the VLSI domain in the form of an experimental set of design and verification tools. We envision the eventual incorporation of such tools within the kinds of VLSI design environments that are currently under development elsewhere in the research community.

The following sections of this document describe our specification and verification work in greater detail, and discuss some of the problems and issues to be resolved in their application to VLSI systems.

The Hierarchical Design Methodology (HDM)

As we noted in the last section, SRI's hierarchical design approach factors the design of a system into a partially-ordered hierarchy of levels of abstraction. Each level is again decomposed into a set of modules. In the software development context, each module usually represents and abstract virtual "machine" with its own abstract state an a set of operations that can modify the state. The same would be true of modules at the highter hierarchical levels of a VLSI design. At the lower levels, modules would correspond to cells--PLAs, shifters, random logic, and so on.

The specification of a module always has two components: a description of its structure, and a characterization of its function. At higher levels in the hierarchy, structure is represented as a collection of abstract data structures and operations on these structures. Functional behavior is specified by characterizing the effects of each operation in anon-procedural way.

Figure 1, for example, shows a formal specification of a simple associative memory module. The associative memory maintains a correspondence between keys (represented by integers) and values (represented by items of type valid-entry-value). The single data structure of the module is the VFUN assoc ("VFUN", "OFUN", and OVFUN" are keywords that indicate structures, operations, and value-returning operations, respectively.) The three operations on the structure are read, write, and clear. The read function retrieves from assoc the value corresponding to a given key; the write function causes a given value to be associated with a given key, and the clear function disassociates all keys from their values.

It is important to note that the effects of each operation are specified completely non-procedurally: they describe what happens, but not how.

At the lower levels of a VLSI design, modules would describe cells, and at still lower levels, individual devices. Figure 2 for example, gives a simple specification of a VLSI barrel shifter. The lowest level of an NMOS design might contain modules explicitly describing the geometry of the poly, diffusion, and metal layers.

Once the various levels of a design have been identified and the modules at each level have been defined, the designer must "sew" the various levels together. More specifically, he must show how each level of abstraction is implemented in terms of the levels beneath it in the hierarchy. This is done by giving, for each module, a formal mapping from the structures of that module to the structures of modules at lower levels. Similarly, the operations of each module are mapped to operations on the corresponding structures in the implementing modules.

Figure 3 illustrates the idea with the implementation of the barrel shifter using arrays of FETS and buses. Note that the implementation formally specifies the connections among the devices of the implementation.

A key aspect of the hierarchical design methodology is the specification language used. The value of specification languages for hardware design has, of course, long been established. Literally dozens of hardware design languages (HDLS) have been developed over the last decade for applications ranging from documentation to simulation. None of them, however, are well suited for hierarchical design.

The deficiencies fall into anumber of categories. First, existing HDLs are adequate only for one or two levels of description. ISP, for example, lends itself well to description at the level of register transfer, but is less well suited to for description at the gate level, and totally inadequate for the topological level of specification required by VLSI design. VLSI layout description languages, of course, such as the Caltech intermediate Form, are well suited for describing graphic items (mask features) but are not at all appropriate for higher-level description. Among the most important benefits of hierarchical design is uniformity of expression from the highest to the lowest levels of design. In order to make this possible, the specification language must have the flexibility to be useful across levels.

While it is presently unrealistic to expect a single description language to handle layout specifications and high-level architectural features with equal ease, we feel that a single specification language can be used to span most of the distance between these levels. The keys to the needed flexibility are extensibility and abstraction. These capabilities are almost universally absent from existing HDLs. Owing to the lack of adequate abstraction facilities, the hardware designer cannot specify the intended function of a design independently from its implementation. The ability to separate specification from implementation lies at the very heart of the hierarchical design philosophy. Without adequate extension features, moreover, the designer lacks the power of expression needed to define new constructs.

Another deficiency of existing HDLs is their primitive capabilities for describing timing properties, even though timing permeates many aspects of hardware design. The designer must ensure that

```
MODULE AssocMemory
          $( This simple associative memory maintains a
           correspondence between keys and values. Entries are
           stored using "write" and retrieved using "read" with
           respect to a key. The use of parameters allows the
           specification to handle memory overflow.)


    TYPES

valid_entry_value: { INTEGER i | i >= 0 AND i < field_size };


    PARAMETERS

INTEGER field_size;
INTEGER max_entries;


    FUNCTIONS


VFUN assoc(INTEGER key) -> valid_entry_value v;
    HIDDEN;
    INITIALLY
       v = ?;

OVFUN read(valid_entry_value key) -> valid_entry_value v;
    EXCEPTIONS
       no_entry: assoc(key) = ?;
    EFFECTS
       v = assoc(key);

OFUN write(valid_entry_value key, v);
    EXCEPTIONS
       overflow: assoc(key) = ?
                AND CARDINALITY({ INTEGER i | assoc(i) ~= ? })
                   >= max_entries;
    EFFECTS
       FORALL INTEGER i:
          'assoc(i) =(IF i = key THEN v ELSE assoc(i));

OFUN clear();
    EFFECTS
       FORALL INTEGER i: 'assoc(i) = ?;

END_MODULE
```

Figure 1: Specification of an Associative Memory

```
MODULE BarrelShifter
      $( This barrel shifter cell rotates input signal bus inbus
         right by a number of positions specificied by the control
         bus controlbus and outputs to output bus outbus.
         For proper operation, exactly one control bus input must
         be high; the jth line is brought high to cause a rotate
         of j position. )$

     PARAMETERS

INTEGER n      ;bus width

    INPUTS/OUTPUTS

INPUT  inbus  ARRAY(0:n-1) OF SIGNAL-SOURCE
INPUT  controlbus  ARRAY(0:n-1) OF SIGNAL-SOURCE
OUTPUT outbus  ARRAY(0:n-1) OF SIGNAL-OUTPUT

    I/O-EXCEPTIONS

NOT THEREEXISTS INTEGER i 0 <= i <= n-1 SUCH THAT
      controlbus(i) = HIGH AND
      FORALL INTEGER j 0 <= j <= n-1
           j NOTEQUAL i IMPLIES controlbus(j) = LOW

   EFFECTS

FORALL INTEGER i 0 <= i <= n-1
       controlbus(i) = HIGH IMPLIES
          FORALL INTEGER j 0 <= j <= n-1
                  inbus(j) = outbus (MOD(n,j+i))
```

Figure 2. Barrel Shifter Specification

```
IMPLEMENTATION-OF-MODULE BarrelShifter
     $(Implementation specification of barrel shifter module
       by means of an array of mosfets (See Mead and Conway[ 22 ],
       p. 159))

STRUCTURES

fet-array  ARRAY(0:n-1, 0:n-1) OF mosfet
vertical-bus  ARRAY(0:n-1) OF bus
diagonal-bus  ARRAY(0:2n-2) OF bus


CONNECTIONS


FORALL INTEGER i 0 <= i <= n-1 CONNECTED(inbus(i),vertical-bus(i))

FORALL INTEGER i,j 0 <= i,j <= n-1
               CONNECTED(vertical-bus(i),fet-array(i,j).source)

FORALL INTEGER i,j 0 <= i,j <= n-1
               CONNECTED(outbus(j),fet-array(i,j).drain)

FORALL INTEGER j 0 <= j <= n-1
               CONNECTED(controlbus(j),fet-array(0,j).gate)

FORALL INTEGER k 0 <= k <= 2(n-1)      ;counts diagonals
         FORALL INTEGER i,j 0 <= i,j <= n-1
                k = i+(n-1)-j IMPLIES
                   CONNECTED(diagonal-bus(k),fet.array(i,j).gate)

FORALL INTEGER k 0 <= k <= n-2
                   CONNECTED(Fet-array(n-1,j).gate, control-bus(j+1))
```

Figure 3. Implementation of Barrel Shifter Using Mosfet Gate Logic

a system meets its functional specifications without timing problems such as critical hazards and oscillations. He must therefore be able to express time-related phenomena in a natural way.

The SPECIAL specification language (Robinson, et al. [10]) we have used in connection with our hierarchical software development methodology provides a starting point for the development of a suitable VLSI description language. Its abstraction and extension capabilities were specifically developed for the purpose of supporting hierarchical design. A great deal more work needs to be done, however, to provide the kind of capabilities necessary to support a useful VLSI design facility.

Verification

For the last several years SRI has been studying program verification as a means of reducing the possibility of errors in programs. We have found that the reliability of programs that have been proved correct using mechanical tools far surpasses that of programs that have been "debugged" using conventional hit-or-miss techniques such as testing. By "correct", we mean that whenever a proven program is invoked on input data satisfying some precise mathematical specification, that program produces output data that satisfies some other precise mathematical description. By "using mechanical tools" we mean that the proof of correctness is checked by computer.

SRI's work in this area has resulted in a number of program verification environments, including systems for proving FORTRAN (Boyer, Moore [2]) and JOVIAL (Elspas et al. [3]) programs, as well as a system for proving the correctness of designs specified using our hierarchical development methodology (Robinson [10]).

The benefits to be gained by bringing verification methods to VLSI design are substantial. The transfer of this technology would not only make possible a degree of reliability not previously enjoyed by hardware designs, but could also decrease the number of costly iterations needed to finalize a design. We believe that these benefits can be achieved by modifying and extending existing methods for verifying programs.

The various techniques for verifying sequential (Floyd [4]) and concurrent (Owicki [9], lamport [6]) programs all fundamentally depend on the method of loop invariants that was popularized by R. Floyd. The method of loop invariants entails the association of a mathematical formula, or assertion, with certain strategic points in a program. Each assertion characterizes a relationship among the variables of the program that must hold true whenever the point in the program with which the assertion is associated is reached. The input assertion associated with the program's entry point specifies properties that input data are assumed to satisfy. (In the case of a real square root program, for example, the input assertion might require the input datum to be non-negative.) The output assertion specifies the relation that the program outputs should bear to the inputs. Finally, a number of intermediate assertions, called loop assertions, are used to capture the relationship among variables at intermediate points of program execution. These assertions are said to be invariants because they must hold whenever program flow reaches the points with which they are associated.

The verification process is concerned with showing that if the program is invoked with inputs satisfying the input assertion, the resulting outputs must satisfy the output assertion. The process involves three steps, the first two of which can be carried out automatically, and the third of which at least semi-automatically.

The first step is to dissect the program into a set of straight-line paths each of which begins and terminates at an assertion point. The paths collectively account for all possible flows of control from one assertion point to another. The second step entails the development, for each path, of a mathematical formula called a verification condition. The verification condition for a given path is logically valid if and only if the truth of the assertion at the head of the path prior to the path's execution suffices to guarantee the truth of the assertion at the end of the path immediately after its execution. The third, and most difficult step in the process is the mathematical proof of the validity of each verification condition. This step is carried out either automatically or semi-automatically using a mechanical theorem-prover.

Our proposed approach to the verification of hardware behavior depends upon the adaptation of the assertional method just described to circuit graphs. As one might expect, the proposed method also entails annotation of the circuit to be verified with assertions. Unlike program invariants, however, the assertions characterize the signals with which they are associated as a function of time. The input and output assertions, taken together, thus define the transfer predicate of the circuit. The transfer predicate at once characterizes the functional and timing behavior of the circuit.

As we have said, the first step in the program verification process after assertion placement is path generation. The adaptation of this process to circuit verification is complicated by the fact that individual circuit elements may, unlike program elements, possess more than one input port. It therefore becomes necessary to develop trees rather than simple paths.

The next step after tree formation is the generation from the trees of the verification conditions themselves. The approach taken here is similar in spirit to the corresponding step in program verification but requires modeling of the semantics of circuit elements rather than program statements. In the hierarchical design context, the "circuit elements" may in fact be complex aggregates whose semantics are specified by the user as part of the design at a given level.

The last step is once again that of proving the verification conditions using a mechanical theorem-prover. The theorem-prover must be capable of treating the kinds of constructs that occur in the verification conditions; these are likely to involve quantifiers and real variables that represent time.

The appendix exemplifies the entire process with an outline of the proof of correctness of a simple but nontrivial circuit for comparing the frequencies of two asynchronous spare waves.

While the example focuses on sequential circuits, the underlying assertional method is applicable at all levels of a hierarchical design, from the user interface all the way down to the layout. What differs from level to level is the semantics of the primitive constructs, and the nature of the properties that must be established. One level of abstraction, for example, may be concerned with signaling among a set of self-timed (Seitz [12]) elements. The specification of semantics at this level might entail the use of a sequence logic, or perhaps temporal logic (Schwartz, Melliar-Smith [11]). At lower levels of abstraction, paradigms able to express quasi-analog behavior are appropriate.

In some cases, specialized techniques developed for handling certain kinds of programs may be directly applicable. The communication among components in speed-independent circuits, for example, is similar to that occurring in communication networks, wherein nodes send and receive messages according to well-defined handshaking protocols. In other cases, such as the specification of timing properties, rather little existing program verification knowledge can be borrowed, but classical engineering techniques can be formalized and incorporated. In the case of synchronous circuits, for example, one must show that the delay in signal transmission through the combinatorial part of circuit is less than the clock period. Traditionally, circuit designers have determined whether timing problems can occur through the use of race detection algorithms or simply by computing delays along worst-case communication paths. We believe that this kind of analysis can easily be mechanized.

As we mentioned earlier, the most difficult stage in the verification process is the automatic or semi-automatic proof of the verification conditions. SRI has developed a number of powerful mechanical theorem provers in connection with program verification. The Boyer-Moore prover (Boyer, Moore [1]), with its sophisticated induction facilities, is capable of proving verification conditions arising from extremely complex designs. A more recent theorem-prover (Shostak [13]) based on fast decision procedures is optimized for formulas involving real variable, and will soon have quantificational capabilities. This combination of abilities is particularly well suited for verification conditions that are likely to arise from hardware designs. In any case, a great deal more investigation of the theorem-proving aspects of the verification process will be necessary to apply this technique to its full potential.

References

[1] Boyer, R.S. and Moore, J S.
A Computational Logic.
Academic Press, New York, 1979.

[2] Boyer, R.S. and Moore J S.
A Fast Majority Vote Algorithm.
1981.
ARI International, Computer Science Laboratory.

[3] Elspas, B. et al.
A JOVIAL Verifier.
Interim, SRI International, Jul, 1978.

[4] Floyd, R.W.
Assigning Meanings to Programs.
In *Mathematical Aspects of Computer Science, Proceedings of the Symposium of Applied Mathematics*, pages 19-32. American Mathematical Society, Providence, Rhode Island, 1967.
Vol. 19.

[5] Foster, M.J. and Kung, H.T.
Design on Special-Purpose VLSI Chips: Example and Opinions.
Computer Science Research Review 8(35), 1979.
Carnegie-Mellon University.

[6] Lamport, L.
Proving the Correctness of Multiprocess Programs.
IEEE Transactions on Software Engineering SE-3(2):125-143, Mar, 1977.

[7] Muller, D.E.
Asynchronous Logics and Application to Information Processing.
In *Switching Theory in Space Technology*, . Stanford University Press, 1963.

[8] Neumann, P. G., et al.
A Provably Secure Operating System: The System, Its Applications, and Proof.
Technical Report, SRI International, May, 1980.

[9] Owicki, S.S.
Axiomatic Proof Techniques for Parallel Programs.
PhD thesis, Cornell University, Aug, 1975.

[10] Robinson, L.
The HDM Handbook, Volumes I-III.
Technical Report, SRI International, Jun, 1979.

[11] Schwartz, R.L. and Melliar-Smith, P.M.
Temporal Logic Specification of Distributed Systems.
In *Proceedings of the Second International Conference on Distributed Systems*, pages 1981. IEEE, Paris, France, 1981.

[12] Seitz, C.L.
System Timing.
In *Introduction to VLSI Systems*, chapter 7. Addison Wesley, 1980.

[13] Shostak, R.E.
A Fast Simplifier for Quantifier-Free Logic.
Paper in progress.

APPENDIX: PROOF OF CORRECTNESS OF A FREQUENCY COMPARATOR

This appendix demonstrates the feasibility of applying software verification techniques to digital circuitry. A nontrivial circuit for comparing the frequencies of two asynchronous squarewaves is proved correct using an adaptation of Floyd's method of program verification.

Circuit Description

The frequency comparator shown in Figure 1, compares two squarewaves with frequencies K and K'. The circuit consists of two five-flip-flop ring counters, the outputs of which are fed through a combinatorial network to a latching circuit. (Although the circuit elements could be realized in a variety of technologies, TTL has been chosen so we can specify the function of each device unambiguously. The flip-flops, in particular, are assumed to be D-type; data is clocked in only on the rising edge of a clock input.) At time t=0, ring flops F_0 and $F'_0$are assumed to be set, and all other flops are assumed to be reset. As time proceeds, the two input squarewaves clock their corresponding ring counters, causing each "1" to race around its ring. As soon as one of the two 1's attains a two-flop lead over the other (modulo 5), the output of one of the AND gates in the combinatorial network rises, clocking either output latch F_{OUT} or F'_{OUT}. If, for example, input K has higher frequency than input K', the nonprimed ring counter will win the race, causing F_{OUT} to be clocked high. The D inputs of the two output flops are cross-coupled so that the first flop clocked will remain set indefinitely, locking the other one out.

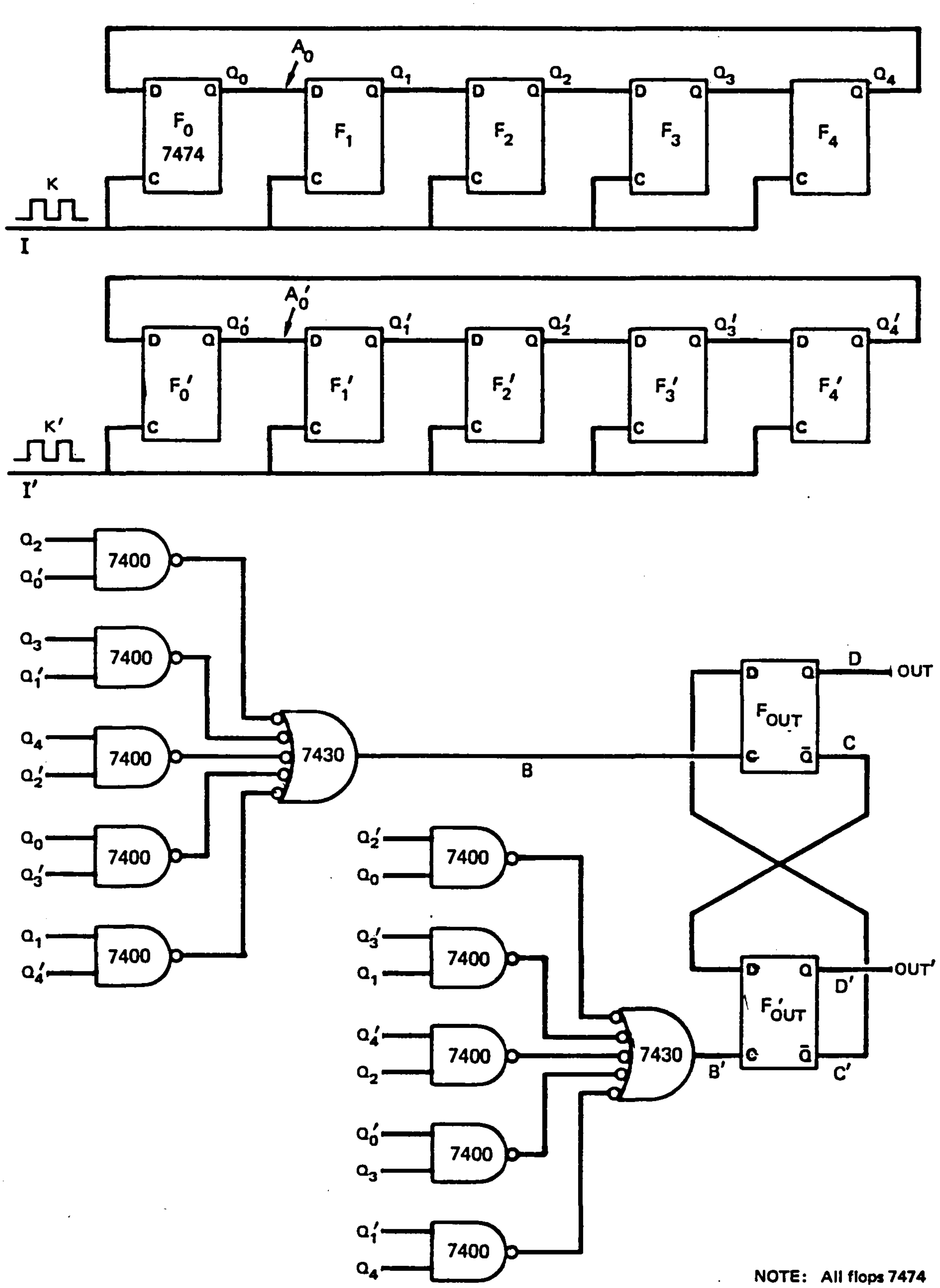
Note that the comparator produces an output only if the two inputs are of unequal frequency. The time required to produce a result, moreover, varies depending on the phasing and frequency of the inputs. Note also that the correctness of the circuit is not at all trivial; a four element ring, for example, would not work.

Method of Proof

Our approach to proving this property adapts Floyd's method for program verification, also known as the method of loop invariants. The basic idea is to associate an assertion with each input and output of the circuit and at certain internal points. Each assertion is a predicate that characterizes the signal at that point as a function of time. Input assertions specify assumptions about circuit inputs, while output assertions specify the intended behavior of the outputs. Internal assertions that facilitate the proof are placed so that each feedback loop in the circuit is cut. Assertion point A_0 in Figure 1, for example, cuts the loop formed by the top ring counter. Assertion point C cuts the cross-coupling loop formed by $F_{OUT}$and F'_{OUT}. For convenience of proof, additional assertions are associated with certain other strategic points in the circuit to be verified (such as point B in the Figure 1).

The proof process proceeds in much the same way as for programs. As we shall see, however, verification conditions are developed from trees embedded in the circuit graph rather than merely from paths. The verification conditions can then be proved. Our proof here is manual, although we believe that the proofs will be mechanized.

Assertions



NOTE: All flops 7474

FIGURE 1 SCHEMATIC OF FREQUENCY COMPARATOR

It is notationally convenient to introduce two special types of assertions: a rising edge predicate R_t that is true for those times t when a rising edge occurs, and a level predicate L_t giving the logic level (true or false) of a given signal at time t. Rise and fall times will be assumed to be negligible in our model, so that rising edges can be specified to occur at points rather than intervals of time. Note that since level predicates completely characterize a signal, a level predicate for a given signal provides all the information that a rising edge predicate would. As in program verification, however, assertions need (and sometimes must) only capture incomplete information about the point in the program or circuit with which they are associated.

The assertions associated with the assertion points indicated in Figure 1 are listed in Table 1. Since the circuit is completely symmetrical with respect to its primed and nonprimed elements, the table lists only those assertions associated with the nonprimed signals; the others are duals.

As the ring counters in the example are sensitive only to rising edges at their clock inputs, the input assertions I and I' are rising edge assertions. The rising edges of the inputs are determined completely by their periods (P,P') and phases ($\emptyset$,$\emptyset$'). The phase gives the time of the first rising edge after t=0. The rising edges of input K thus occur at $t=\emptyset$, $\emptyset+P$, $\emptyset+2P$, etc. The rising edge predicate R_t for K can therefore be written

$$R_t \equiv [t]_p = \emptyset$$

where $[t]_p$ (read "t mod P") denotes the smallest non-negative quantity that differs from t by an integral multiple of P.

The output assertion D is formed using a level predicate. It states that if K is of higher frequency than K' (i.e., $P < P'$) then at the same time u smaller than

$$\tau = \frac{2PP' + P'\emptyset - P\emptyset'}{P' - P}$$

the F_{OUT} output will rise and remain high indefinitely. D also states that if $P' \leq P$, F_{OUT} will remain low for all time $t \geq 0$.

The loop assertion A_0 associated with the Q output of F_0 captures the ring behavior. It is a level predicate that asserts that Q_0 will be high every five pulses of the input clock K.

Loop assertion C associated with the Q output of F_{OUT} is understood merely by noting that this output must be complementary to the Q output.

Assertion B is neither an input nor output assertion, nor does it cut any loop in the circuit. It is not strictly required by the proof methodology, but has been included to abstract the circuit behavior at the point at which it occurs and hence to simplify proof of the verification conditions. The assertion states that if $P < P'$, the OR-gate output will rise by $t=\tau$, but will remain low up until $t=\tau$ otherwise.

In addition to input, output, and loop assertions, it is necessary to provide an assertion at each point in the circuit for which an initial state (state at t=0) must be specified. Initial state

$I:\ R_t \equiv ([t]_p = \emptyset)$

$D:\quad (P < P' \longrightarrow \exists u\ (0 \leq u \leq \tau)\ L_t \equiv t \geq u)$

$\wedge\quad (P \geq P' \longrightarrow \forall t \geq 0\ \neg L_t)$

$A_0:\ L_t \equiv (\emptyset - P \leq [t]_{5P} < \emptyset)$

$C:\ (P < P' \longrightarrow \forall\ t \geq \tau \neg L_t)$

$(P \geq P' \longrightarrow \forall\ t \geq 0\ \ L_t)$

$B:\ (P < P') \longrightarrow \exists u\ (0 \leq u \leq \tau)\ \forall t\ 0 \leq t \leq u\ R(t) \equiv t = u$

$\wedge\ (P \geq P' \longrightarrow \forall t\ 0 \leq t \leq \tau' \neg L_t$

$A_i:\ L_t \equiv (\emptyset - (1-i)P \leq [t]_{5P} < \emptyset + iP)$

where $\tau = \dfrac{2PP' + P'\emptyset - P\emptyset'}{P' - P}$

TABLE 1. ASSERTIONS FOR EXAMPLE

information is most frequently necessary at the outputs of devices such as flip-flops and counters that have explicit storage. Owing to the presence of loops, however, state may exist implicitly at arbitrary points in the circuit.

In our example, however, initial state information is necessary only for the outputs of the flip-flops. Flops F_0 and F_{OUT} (and their primed counterparts) already have assertions that specify t=0 behavior, so only the outputs of F_1 through F_4 need to be annotated. The appropriate asserts (A_i, i=1 to 4) are shown in Table 1. Note that these assertions are analogues to A_0.

Verification Condition Generation and Proof

The first step in verification condition generation for programs is path analysis, which involves unfolding the flow chart of the program into straight-line paths between assertions. The paths account for all possible flows of control from one assertion point to another (possibly the same) assertion point. The adaptation of the method to circuit verification is somewhat more complicated, since individual circuit elements may, unlike program elements, possess more than one input port. It therefore becomes necessary to develop trees rather than simple paths. A tree is constructed for each assertion point P other than input assertion points. The tree is rooted at P and is developed by tracing backwards from P through the circuit graph, splitting whenever a circuit element has more than one input, and terminating whenever another assertion point (possibly P) is encountered. (Our model assumes that devices whose outputs are wire-or'ed together be considered as a single device.)

The tree corresponding to assertion point A_0, for example, is shown in Figure 2. Note that the only non-trivial trees generated for our example are those for B and B'.

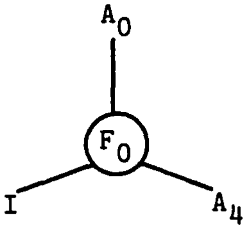


Figure 2. Graph for A_0

The next step after tree formation is the generation of the verification conditions themselves. Generation is carried out by a symbolic evaluation process beginning with the assertions at the leaves of each tree and proceeding forward toward the root. Each node is processed only after all of the nodes below it have been processed. Processing consists of the application of a Hoare-like rule or some other characterization of the transfer function of the processed node element to produce a post-condition assertion. The verification condition yielded by the tree consists of the implication whose hypothesis is the assertion resulting from the symbolic evaluation, and whose conclusion is the root assertion.

The question of exactly how the semantics of each circuit element are encoded is a difficult one and should be a subject of investigation. For our example, however, it is easy to verify by inspection and informal reasoning that the verification conditions arising from all assertion points other than B (and B') will be trivial, given any reasonable formal model of the D flip-flop. We will therefore confine ourselves, for the remainder of this discussion, to the verification condition for B.

In as much as the only circuit elements in the tree for B are logic gates, the verification condition itself is easily generated:

$$V_B \equiv (H \supset B)$$

where $H \equiv A'_0 \wedge A_2 \vee A'_1 \wedge A_3 \vee A'_2 \wedge A_4 \vee A'_3 \wedge A_0 \vee A'_4 \wedge A_1$

The proof of V_B is more difficult, and requires some lemmas. We first introduce a useful definition:

Defn. Let $C(t) = \left\lfloor \frac{t-\emptyset}{P} \right\rfloor$

Intuitively, C(t) counts the number of rising edges (beginning at t=0) of input waveform K.

Lemma 1 $P < P' \rightarrow \tau > 0$

Pf. τ can be written, $\tau =$

$$\frac{\frac{2+\emptyset}{P} - \frac{\emptyset'}{P'}}{\frac{1}{P} - \frac{1}{P'}}$$

Since P, P', $\emptyset$, $\emptyset'$ are all non-negative and since $\emptyset < P'$, the numerator is positive. Then since $P < P'$ implies that the denominator is positive, $\tau > 0$.

Lemma 2. $C(\tau) = C'(\tau) + 2$

Pf. Straightforward from the defn. of τ

Lemma 3 $P \geq P' \rightarrow \forall t \; 0 \leq t \leq \tau' \; C(t) \leq C'(t) + 1$

Pf. also follows from the defn of τ.

Lemmas 2 and 3 immediately give rise to the following corollaries:

Corollary 4. $[C(\tau) - C'(\tau)]_5 = 2$

Corollary 5. $P \geq P' \rightarrow \forall t \; 0 \leq t \leq \tau'$

$$[C'(t) - C(t)]_5\emptyset = 2$$

The proof of V_B requires one other lemma:

Lemma 6. For $0 \leq i \leq 4$, A_i can be written

$$L_t \equiv ([C(t)]_5 = [1-i]_5)$$

Pf. We must show that

$$\emptyset - (1-i)P \leq [t]_{5P} < \emptyset + iP \quad \text{iff}$$

$$[\lfloor \frac{t-\emptyset}{P} \rfloor]_5 = [i-1]_5.$$

The left hand expression is true iff ∃ an integer k s.t.

$$\emptyset - (1-i)P \leq t-5Pk < \emptyset + iP$$

$$\Longleftrightarrow \quad 5k+i-1 \leq \frac{t-\emptyset}{P} < i + 5k$$

$$\Longleftrightarrow \quad \left\lfloor \frac{t-\emptyset}{P} \right\rfloor = 5k+i-1$$

$$\Longleftrightarrow \quad \left[\left\lfloor \frac{t-\emptyset}{P} \right\rfloor\right]_5 = [i-1]_5$$

$$\Longleftrightarrow \quad [C(t)]_5 = [i-1]_5$$

Q.E.D.

Now, it follows from Lemma 6 and Corollary 4 that the hypothesis H of V_B implies $P < P' \rightarrow L_\tau$. Then from Lemma 1, and from the fact that $H \rightarrow \neg L_0$, we have

$$H \longrightarrow [\ P < P' \rightsquigarrow \exists u\ 0 \leq u \leq \tau\, R_u]$$

Also, from Corollary 5 and Lemma 6 we have that

$$H \longrightarrow [P \geq P' \supset \forall t\ 0 \leq t \leq \tau'\ \neg L_t]$$

H thus implies B, completing the proof.

CHAPTER 22

THE BOYER-MOORE THEOREM PROVER AND ITS APPLICATION TO PROGRAM VERIFICATION

Three Lectures on Theorem-Proving and Program Verification

J Strother Moore

1.1 History

How does one prove theorems? How can we build a machine to prove theorems?

Because mechanical theorem-proving has its roots in mathematics, and because mathematicians and philosophers have long asked the questions above, it is difficult to put a date on when mechanical theorem-proving was born. For example, the idea of mechanical proof, in the sense that we think of it today, would not have surprised Leibniz (1646-1716) who, on the one hand perfected and presented to the Royal Society, London, a mechanical binary adder (also capable of multiplication, division and square root computations) and on the other hand believed that all reasoning (moral and otherwise) could be reduced to an "algebra of thought."

The early 20th century saw the development of formal axiomatic systems characterized by a set of "well-formed formulae," a set of "axioms" and a set of "inference rules" with which one may deduce "theorems" from the axioms and previously deduced theorems. A "proof" of some formula p is just a finite sequence of formulae, the last of which is p and each of which is either an axiom or is derived from the preceding formulae by a rule of inference.

For example, here is a proof of the formula (-A v A) in the logic of Russell and Whitehead from Principia Mathematica.

Proof of (-A v A).

1. (Q -> R) -> ((P v Q) -> (P v R))   Axiom 4

2. (Q -> A) -> ((-A v Q) -> (-A v A))   Subst into 1

3. (Q -> A) -> ((A -> Q) -> (-A v A))   Def of "->"

4. ((A v A) -> A) -> ((A -> (A v A)) -> (-A v A)) Subst into 3

5. (P v P) -> P   Axiom 1

6. (A v A) -> A   Subst into 5

7. (A -> (A v A)) -> (-A v A)   M.P. 4 and 6

8. Q -> (P v Q)   Axiom 2

9. A -> (A v A)   Subst into 8

10.(-A v A) M.P. 7 and 9

It is easy to determine whether a sequence of formulae is a proof; theorem-proving is the art of discovering a proof -- if any -- for a given formulae.

The 1920's and 1930's saw the careful study of formal axiomatic systems, primarily to clarify the then extensive debates between the various schools of thought on how the newly uncovered paradoxes in the foundations of mathematics might be remedied. Hilbert proposed to formalize classical mathematics (e.g., arithmetic) in logic and undertake the proof of its consistency via constructive means. Starting in the 1920's, this program was undertaken by Hilbert, Ackermann, von Neumann, Herbrand and others. Among the interesting results proved by this school was Herbrand's Theorem (1930), which is a constructive version of a theorem proved earlier by Skolem [30] that suggests a mechanical means for finding a proof when it exists.

In 1931, Goedel showed that there exist formal sentences of arithmetic that are true (in the intended interpretation) but unproveable. Furthermore, he showed that if arithmetic is consistent then its consistency cannot be proved in arithmetic. In a certain sense, this undermined Hilbert's program. However, thanks in large part to Hilbert and his school, the formal study of formal proofs had been born.

During this same period, Church, Turing, and Goedel (the latter following a suggestion by Herbrand) developed what turned out to be the equivalent notions of lambda-definable, Turing computable, and general recursive functions. These developments led, in 1936 and 1937, to the demonstrations that there were no decision procedures for arithmetic or first order predicate calculus. It is perhaps ironic that the concepts that eliminated the hope that perfect theorem-provers could be built simultaneously formed part of the theoretical foundations for the development of the device that makes imperfect theorem-provers realizable and perhaps practical.

The first heuristic mechanical theorem-prover physically realized was the Logic Theory Machine, programmed in the mid-50's by Newell, Shaw, and Simon. The Logic Theory Machine constructed proofs in the propositional calculus using the axioms and rules of inference of Principia Mathematica. A succinct description of the Logic Theory Machine and its capabilities is provided in Computers and Thought [11].

The Logic Theory Machine attacked its problems in much the same way a human might attack them, when limited to the axioms, rules of inference and previously proved theorems of Principia Mathematica. The program contained four "methods" or "heuristics" for decomposing the given problem into "simpler" subproblems (e.g., instances of the axioms). An executive routine selected the methods to be tried and the subproblems to be worked on.

It should be observed that the authors of this early program were not so much concerned with answering the question "Is this propositional formula a theorem?" as they were with the question "How does one go about solving hard problems?" Judged solely by its ability to answer the former question, the Logic Theory Machine was not impressive. It was able to prove only 38 of the 52 propositional theorems in Chapter 2 of Principia. By contrast, Wang's algorithm [32], published in January, 1960 and based on the "semantic" idea of attempting to construct an assignment for falsifying a formula, was able to announce the validity of all of the propositional theorems in Principia.

But the Logic Theory Machine was a significant contribution to the infant field of "artificial intelligence" and was the first program to confront the hard problem that is *unavoidable* in the nonpropositional case: how does one choose which of many alternatives to pursue? The Logic Theory Machine inspired several other early AI programs, notably Gelernter's Geometry Theorem Proving Machine and Slagle's Symbolic Automatic Integrator for elementary calculus problems. (Both Gelernter's and Slagle's programs are landmarks of AI and mechanical theorem-proving and are described in *Computers and Thought*, [11].

However, many researchers were more interested in the "ends" than the "means" and launched a no-holds barred attack on the problem of building a program to determine if a propositional formula, and more generally, a first order formula, was a theorem or, equivalently (thanks to Goedel), valid.

In the early years -- the late 50's and early 60's -- the field was dominated by logicians who pursued quite different approaches to the theorem-proving problem. Among the early researchers were Wang, Gilmore, Davis and Putnam, and Prawitz.

Then, in 1965, J. A. Robinson published the paper "A Machine-Oriented Logic Based on the Resolution Principle" [26]. The resolution principle's simplicity and elegance made it a very attractive mechanism.

Suppose we wished to prove the following theorem of first order predicate calculus:

```
[A X A Y  P(X,Y) -> Q(Y)

      &

 A X E Y  P(X,Y)

      &

 A X  Q(X) -> Q(G(X))]

->

 E X  Q(G(G(X)))
```

To apply resolution we actually work on the negation of the problem and attempt to derive a contradiction. The negation of the formula above is that the first three hypotheses are true and the conclusion is false. Then we put the conjecture into conjunctive normal form, using Skolem functions to eliminate the existential quantifiers. The result is the following conjunction of disjunctions:

```
-P(X,Y) v Q(Y)

     &

P(Z,F(Z))
```

&

-Q(U) v Q(G(U))

&

-Q(G(G(V)))

Finally, Robinson writes this as a set of clauses. A clause is set of literals, each literal being an atom or negated atom.

{-P(X,Y) Q(Y)}

{P(Z,F(Z))}

{-Q(U) Q(G(U))}

{-Q(G(G(V)))}

Having distilled the problem down to this simple but universal notation we can now apply the "resolution principle": Consider any two clauses in the set, rename their variables so they have no variable in common, and then consider each literal of one clause against each literal of the other. If the two literals have opposite signs and there exists a substitution that makes their atoms identical, instantiate both clauses with the most general such substitution, delete the two (now complementary) literals from the two instantiated clauses and union the two resulting sets together. The resulting clause is a "resolvent" of the two parent clauses and should be added to the set of clauses. Repeat this process indefinitely. Should the empty clause ever be formed, the original set of clauses was unsatisfiable -- i.e., the original quantified formula is a theorem.

Perhaps more important than resolution itself was Robinson's "unification algorithm" which is a way to determine either the most general substitution that makes two terms identical or that no such substitution exists. For example, the unification algorithm determines that P(X,F(X)) and P(A(),Z) are unified by replacing X by A() and Z by F(A()), while P(X,X) and P(Y,F(Y)) have no common instance.

Here is a resolution proof of the example theorem above:

1. {-P(X,Y) Q(Y)}
2. {P(Z,F(Z))}
3. {-Q(U) Q(G(U))}
4. {-Q(G(G(V)))}
5. {Q(F(Z))} resolving 1 x 2

6. {Q(G(F(Z)))} resolving 3 x 5

7. {Q(G(G(F(Z))))} resolving 3 x 6

8. {} resolving 4 x 7

Despite its simplicity, resolution is a sound and complete inference procedure for first order predicate calculus[1]. For more details the reader should see [9, 21, 27]

Note how easily a resolution theorem-prover can be implemented. Clauses may be represented as lists of literals. The basic operation on a Resolution Logic Machine is:

1. Choose a clause to factor or two clauses to resolve upon.

2. Form all possible factors or resolvents and add them to the set of clauses.

3. If any clause is empty, report that the original set was unsatisfiable.

4. Otherwise, repeat from step 1.

As one might gather from the above description, the only difficult problem is the choice of which two clauses to use as parents in any given round. This is called the search strategy and is the hard problem confronting the serious implementor of a resolution theorem-proving.

There are two classic search strageties. One, called breadth first, constructs all the resolvents from among the initial set S before adding them to S to form the new set S', and then iterates on the set S'. Thus, the so-called "search tree" -- the tree of all possible resolvents -- is grown in horizontal layers. The other common variation is called depth first, in which one prefers as a parent the most recently produced clause. In a depth first search, long branches of the search tree are grown first.

It is fair to say that very few resolution theorem-provers use either search strategy in the rigid way they are defined above. It is also fair to say that resolution is not the only part of theorem-proving concerned with search strategy. The consideration of search strategy dominates the implementation of a resolution theorem-prover largely because resolution has distilled the theorem-proving process down to where there is very little else to do. But every theorem-proving machine (for sufficiently rich logics) stands or falls on its ability to make the right choices at the right time.

To the criticism that resolution was "unnatural" (to many people) the response was similar to Minsky's later defense of the attempt to build an intelligent machine [paraphrase]: If you wanted to build a machine that flies, would you cover it with feathers? If you wanted to build a machine that thinks, would you use meat? During the late 60's the vast majority of published work on mechanical theorem-proving was resolution based.

[1] For completeness one must include an additional rule called "factoring" with which one can instantiate a clause so as to cause two literals in it to become identical.

But saying that the vast majority of the published work in the 60's was resolution based is not to say that all the resolution researchers were working on the same idea. The very simplicity of resolution encouraged its elaboration. Resolution was restricted, refined, and extended. There was (in no particular order) unit resolution, hyperresolution, linear resolution, and paramodulation. There was linear paramodulation and hyperparamodulation. There was E-resolution, OL-resolution, P1-resolution, SL-resolution, V-resolution, and P-hyperparamodulation.

In short, the late 60's were an exciting time in the history of mechanical theorem-proving. There were three (causally related) reasons:

1. technological improvements brought a tremendous increase in the computer power available,

2. the economy boomed and made money available for computer science research in previously unheard of quantities -- much of it funnelled through the Advanced Research Projects Agency (ARPA) of the U.S. Defense Department, and

3. Artificial Intelligence emerged as an endeavor that captured the imaginations of many researchers (and funding agencies) and, theoretically at least, theorem-proving could solve many of the hard problems in AI. For example, several typical AI problems such as natural language understanding, robotics problem solving, and question answering systems could be cast in the framework of first order predicate calculus problems and solved with sufficient theorem-proving power.

While resolution theorem-proving did not directly receive very much of the money channeled to AI, it benefited greatly from the availability of computer power and the interest in mechanical problem solving generated by AI.

Of course, not all researchers pursued resolution, even in its heyday. Most nonresolution work was directed down branches of mathematics not easily cast into the predicate calculus. The interested reader should see, for example, the work of Bledsoe and Gilbert [2] on set theory and Bledsoe, Boyer and Henneman [4] on proofs of limit theorems in real analysis. During this same time, the field of "symbolic manipulation" matured to the point where programs were able to aid physicists and engineers in algebraic simplification and integral calculus. See the review by Moses [24].

In the mid-70's the excitement declined because researchers began to realize that the paradigm established by Robinson -- formulate a restriction of resolution and prove that it is complete -- produced a plethora of theoretical papers but very few successful mechanical theorem-provers.

Many people attributed this disparity to the "unnaturalness" of resolution and began to pursue new directions. At about the same time, new AI programming languages began to catch on (e.g., PLANNER). For a while in the early 70's controversy raged between those on opposite sides of the question: "Is it better to use 'declarative' or 'procedural' encodings of knowledge?" This controversy has since died out, partially because PLANNER and its descendents did not really solve the hard problems and partially because people like Kowalski and Hayes successfully argued that predicate calculus could be used as a programming language and made to perform as well (or badly) as "conventional" languages like PLANNER.

In my view, the disparity between the number of publications and the number of successful implementations was due to inadequate attention to search strategy. While the search strategy problem was certainly recognized by all, it was more or less left to the "hackers" who put together theorem-provers. It is certainly safe to say that most researchers hoped that victory would be achieved without the invention of messy, ad hoc heuristics. That hope has waned considerably since the early 70's.

During the 70's theorem-proving research was supported mainly by the emerging fields of programming language design and program verification. The main application in programming language design has been the implementation of efficient "interpreters" (i.e., theorem-provers) for nondeterministic predicate calculus programs. The interested reader should see Kowalski's article "Predicate Calculus as a Programming Language" [18], and the work on implementing such a language by Colmerauer and Roussel of the University of Marseille [10, 28], and Warren at the University of Edinburgh [33]. It is interesting to note that in this application search strategy is often less important than in general purpose theorem-proving because the user of the theorem-prover can often constrain the search space by appropriately formulating his "programs."

The theorem-proving research supported by program verification has been both more and less traditional -- more traditional in the sense that the goal is to mechanize mathematics and less traditional in the sense that the approaches used are often radically different from those suggested by resolution. The basic idea -- as will be elaborated in the third lecture -- is that it is easy to transform the question "Is this program correct?" into the question "Are these formulae theorems?" The formulae are then submitted to a mechanical theorem-prover for proof. A theorem-prover for program verification must be good at deriving theorems from a large data base containing facts that may be instantiated and chained together -- just as the AI applications demanded. But, in addition, program verification added some new demands:

1. The proofs of the conjectures produced by program verifiers require induction. Why? Because those conjectures usually involve inductively constructed mathematical objects (e.g., integers, sequences, trees) and inductively defined concepts (e.g., addition, permutation, fringe).

2. Program verification has caused the construction of new logical theories in which the semantics of programs are expressed.

3. Program verification aims at putting the theorem-prover in the hands of a "user" who is considered willing to help the theorem-prover but who is not logically infallible. For example, to specify his program the user may need to define previously unstudied mathematical concepts (e.g., majority vote). The addition of axioms purported to describe the properties of such concepts must not be taken lightly. Experience has shown that users are notoriously bad at getting the details right when dealing with concepts outside of their traditional training -- and the accidental production of an inconsistent set of axioms may lead to "proofs" of incorrect programs whose specifications do not even involve those axioms. On the other hand, experience has shown that many users have excellent intuitions about why things are true and can be of great help in guiding the system to a proof.

Because of these demands theorem-proving research in the 70's has branched out considerably.

Let me merely list some of the main themes of theorem-proving in the 70's:

1. The construction of proof checkers and interactive theorem-provers. See for example the FOL system of Weyhrauch [34] or Jutting's description of the use of the AUTOMATH system to proof check all of Landau's text on the development of elementary mathematics from Peano axioms to the reals [17].

2. The construction of theorem-provers for decidable theories, such as Presburger arithmetic and "data structures." See for example the work of Bledsoe [3], Shostak [29], Oppen [25].

3. The construction of theorem-provers or proof-checkers for logics other than first-order predicate calculus. For example, our work [6] is based on a quantifier free logic with recursive functions and induction. The Edinburgh LCF system [14] is based on Scott's logic and Litvintchouk and Pratt's system is based on modal logic [20].

4. The application of rewrite rules to simplify formulas and the study of the theoretical properties of such "rewrite systems." See the survey paper by Huet and Oppen [16].

5. The study of "metatheoretic extensibility" -- the use of a theorem-prover to prove the correctness of extensions. See below and [7].

6. The further study of resolution and proof procedures suggested by resolution. See for example the proceedings of the latest Workshop on Automatic Deduction or Kowalski's "connection graph" proof procedure suggested by the failure modes of resolution [19].

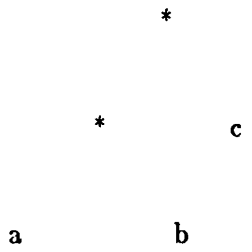
Rather than try to summarize each of these fields I will, in my next lecture, acquaint you with how one state-of-the-art theorem-prover works and what are the current limits of its abilities.

1.2 The Boyer-Moore Theorem-Prover

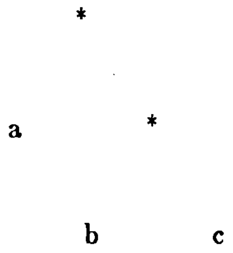
For the past nine years Bob Boyer and I have been developing an automatic theorem-prover capable of constructing inductive proofs. The development of the theorem-prover is being sponsored by NSF Grant MCS-7904081 and ONR Contract N00014-75-C-0816. The theorem-prover deals with a quantifier free first order logic. In addition to modus ponens, instantiation, and substitution of equals for equals, the logic provides for the axiomatic introduction of new "types" of inductively constructed objects (e.g., integers, sequences, graphs) the definition of new mathematical functions (e.g., prime, permutation, path), and proof by induction on well-founded relations.

The addition of definitional equations purporting to define new functions raises a difficult problem: how can we insure that the new axiom actually defines a function? In our logic we require that for each new definition there exist a "measure" of the arguments of the function and a well-founded relation such that in every "recursive call" in the body, the measure of the arguments to the call is strictly smaller than the measure of the input arguments. This condition, together with some trivial syntactic requirements, is sufficient to insure that the new axiom satisfied by one and only one function.

For example, consider the idea of computing the "fringe" of a binary tree. One way to do it is to consider the successive CDR's of the tree and repeatedly transform subtrees of the form:



into the form:



until a is an atom. Using a LISP-like syntax we express this function as:

```
(NORMTREE X)
  =
(IF (LISTP X)
   (IF (LISTP (CAR X))
      (NORMTREE (CONS (CAAR X)
                 (CONS (CDAR X) (CDR X))))
      (CONS (CAR X) (NORMTREE (CDR X))))
   (CONS X NIL)).
```

What measure is going down here? Our system is not capable of discovering (on its own) such a measure. However, if the user of our system defines the function:

```
(MS X)
  =
(IF (LISTP X)
   (TIMES (SQUARE (MS (CAR X))) (MS (CDR X)))
   1),
```

which is accepted because the size of the argument gets smaller in each call, then the system can prove that (MS X) decreases in both of the recursive calls of NORMTREE in the definition of NORMTREE. Thus, after the introduction of MS and the proof of the two lemmas establishing that it decreases, NORMTREE is accepted by our system as a true definitional equation.

The theorem-prover itself consists of an ad hoc collection of heuristic proof techniques. The two most important ones are simplification and the invention of "appropriate" induction arguments. The system also contains heuristics for eliminating "undesirable" expressions (e.g., X-Y can be eliminated by replacing X with I+Y), the use of equality, generalization, and the elimination of irrelevance.

The simplification routine is driven by conditional rewrite rules derived from axioms, recursive definitions, and previously proved theorems. The system contains fairly sophisticated search strategic heuristics for controlling the expansion of definitions, backwards chaining to establish hypotheses of rewrite rules, permutative rewrites, etc.

The induction routine attempts to find an induction argument that is "appropriate" for the conjecture being proved. Roughly speaking, it attempts to find an n-way case split and some induction hypotheses such that when certain of the recursive functions in the induction conclusion of a given case are expanded, the resulting recursive calls are involved in the hypotheses for that case. To find -- and justify -- the induction argument, the induction routine analyzes the measures and well-founded relations justifying the recursive functions in the conjecture. We have found that the direct analysis of these measures and well-founded relations is simpler than the analysis of the recursive functions themselves and permits the system more often to piece together induction arguments "appropriate" for several functions in the conjecture. The reason for this is that the function definitions frequently contain tests that are irrelevant to the recursions and these tests obscure the correct choice of induction cases.

To illustrate how our system proves theorems, let us consider proving that NORMTREE computes the fringe as defined (in the more traditional way) by the recursive function FLATTEN:

```
(FLATTEN X)
  =
(IF (LISTP X)
   (APPEND (FLATTEN (CAR X))
         (FLATTEN (CDR X)))
   (CONS X NIL)),
```

where APPEND concatentates two lists:

```
(APPEND X Y)
  =
(IF (LISTP X)
   (CONS (CAR X)
        (APPEND (CDR X) Y))
   Y).
```

We will prove:

```
(EQUAL (NORMTREE X) (FLATTEN X)).
```

The proof may be briefly sketched as follows: We induct on X, using the measure and well-founded relation justifying NORMTREE, we simplify, using the axioms defining lists and "opening up" certain recursive functions "calls", we rerepresent the variables in the problem to simplify the conjecture, use our induction hypothesis (and throw it away to generalize our goal), replace certain terms by variables, restricted to the range of the terms replaced, to generalize the problem still more, and then perform a second induction. Below is the complete proof, as discovered and described in English by our system:

```
_PROVE.LEMMA(CORRECTNESS.OF.NORMTREE
             (REWRITE)
             (EQUAL (NORMTREE X) (FLATTEN X)))

  Give the conjecture the name *1.
```

We will appeal to induction. Two inductions are suggested by terms in the conjecture, both of which are unflawed. So we will choose the one suggested by the largest number of nonprimitive recursive functions. We will induct according to the following scheme:

```
  (AND (IMPLIES (NOT (LISTP X)) (p X))
       (IMPLIES (AND (LISTP X)
                     (LISTP (CAR X))
                     (p (CONS (CAAR X)
                              (CONS (CDAR X) (CDR X)))))
                (p X))
       (IMPLIES (AND (LISTP X)
                     (NOT (LISTP (CAR X)))
                     (p (CDR X)))
                (p X))).
```

The inequalities MS.DECREASES.UNDER.CDR and MS.DECREASES.UNDER.ROTATION establish that the measure (MS X) decreases according to the well-founded function LESSP in each induction step of the scheme. The above induction scheme generates the following three new conjectures:

```
Case 1. (IMPLIES (NOT (LISTP X))
                 (EQUAL (NORMTREE X) (FLATTEN X))),

  which we simplify, unfolding the definitions of NORMTREE
  and FLATTEN, to:

      (TRUE).

Case 2. (IMPLIES
          (AND
            (LISTP X)
            (LISTP (CAR X))
```

```
 (EQUAL (NORMTREE (CONS (CAAR X)
              (CONS (CDAR X) (CDR X))))
     (FLATTEN (CONS (CAAR X)
             (CONS (CDAR X) (CDR X))))))
(EQUAL (NORMTREE X) (FLATTEN X))),
```

which simplifies, applying CDR.CONS and CAR.CONS, and opening up FLATTEN and NORMTREE, to:

```
(IMPLIES
 (AND
  (LISTP X)
  (LISTP (CAR X))
  (EQUAL
     (NORMTREE (CONS (CAAR X)
              (CONS (CDAR X) (CDR X))))
     (APPEND (FLATTEN (CAAR X))
         (FLATTEN (CONS (CDAR X) (CDR X))))))
 (EQUAL (NORMTREE.(CONS (CAAR X)
              (CONS (CDAR X) (CDR X))))
     (APPEND (FLATTEN (CAR X))
         (FLATTEN (CDR X))))).
```

This simplifies again, applying CDR.CONS and CAR.CONS, and opening up the definition of FLATTEN, to:

```
(IMPLIES
 (AND
   (LISTP X)
   (LISTP (CAR X))
   (EQUAL (NORMTREE (CONS (CAAR X)
              (CONS (CDAR X) (CDR X))))
       (APPEND (FLATTEN (CAAR X))
           (APPEND (FLATTEN (CDAR X))
               (FLATTEN (CDR X))))))
 (EQUAL (NORMTREE (CONS (CAAR X)
              (CONS (CDAR X) (CDR X))))
     (APPEND (APPEND (FLATTEN (CAAR X))
             (FLATTEN (CDAR X)))
          (FLATTEN (CDR X))))).
```

Applying the lemma CAR/CDR.ELIM, we now replace X by (CONS Z V) to eliminate (CAR X) and (CDR X) and Z by (CONS W D) to eliminate (CAR Z) and (CDR Z). This generates:

```
(IMPLIES
  (EQUAL (NORMTREE (CONS W (CONS D V)))
            (APPEND (FLATTEN W)
                    (APPEND (FLATTEN D) (FLATTEN V))))
     (EQUAL (NORMTREE (CONS W (CONS D V)))
            (APPEND (APPEND (FLATTEN W) (FLATTEN D))
                    (FLATTEN V)))).
```

We now use the above equality hypothesis by substituting:

```
     (APPEND (FLATTEN W)
             (APPEND (FLATTEN D) (FLATTEN V)))
```

for (NORMTREE (CONS W (CONS D V))) and throwing away the equality. The result is:

```
     (EQUAL (APPEND (FLATTEN W)
                    (APPEND (FLATTEN D) (FLATTEN V)))
            (APPEND (APPEND (FLATTEN W) (FLATTEN D))
                    (FLATTEN V))),
```

which we generalize by replacing (FLATTEN V) by Y, (FLATTEN D) by A, and (FLATTEN W) by U. We restrict the new variables by appealing to the type restriction lemma noted when FLATTEN was introduced. This produces:

```
     (IMPLIES (AND (LISTP Y) (LISTP A) (LISTP U))
              (EQUAL (APPEND U (APPEND A Y))
                     (APPEND (APPEND U A) Y))),
```

which we will name *1.1.

```
Case 3. (IMPLIES (AND (LISTP X)
                      (NOT (LISTP (CAR X)))
                      (EQUAL (NORMTREE (CDR X))
                             (FLATTEN (CDR X))))
                 (EQUAL (NORMTREE X) (FLATTEN X))),
```

which we simplify, expanding the definitions of NORMTREE and FLATTEN, to:

```
     (IMPLIES (AND (LISTP X)
                   (NOT (LISTP (CAR X)))
                   (EQUAL (NORMTREE (CDR X))
                          (FLATTEN (CDR X))))
              (EQUAL (CONS (CAR X) (NORMTREE (CDR X)))
                     (APPEND (FLATTEN (CAR X))
                             (FLATTEN (CDR X))))).
```

This simplifies again, applying CDR.CONS, CAR.CONS, and CONS.EQUAL, and opening up the functions FLATTEN and APPEND, to:

```
      (TRUE).
```

So let us turn our attention to:

```
(IMPLIES (AND (LISTP Y) (LISTP A) (LISTP U))
         (EQUAL (APPEND U (APPEND A Y))
                (APPEND (APPEND U A) Y))),
```

which we named *1.1 above. We will appeal to induction. Three inductions are suggested by terms in the conjecture. They merge into two likely candidate inductions. However, only one is unflawed. We will induct according to the following scheme:

```
(AND (IMPLIES (NOT (LISTP U)) (p U A Y))
     (IMPLIES (AND (LISTP U) (p (CDR U) A Y))
              (p U A Y))).
```

The inequality CDR.LESSP establishes that the measure (COUNT U) decreases according to the well-founded function LESSP in the induction step of the scheme. The above induction scheme produces two new goals:

```
Case 1. (IMPLIES (AND (NOT (LISTP (CDR U)))
                      (LISTP Y)
                      (LISTP A)
                      (LISTP U))
                 (EQUAL (APPEND U (APPEND A Y))
                        (APPEND (APPEND U A) Y))).
```

This simplifies, applying CDR.CONS, CAR.CONS, and CONS.EQUAL, and expanding the definition of APPEND, to:

```
(IMPLIES (AND (NOT (LISTP (CDR U)))
              (LISTP Y)
              (LISTP A)
              (LISTP U))
         (EQUAL (APPEND (CDR U) (APPEND A Y))
                (APPEND (APPEND (CDR U) A) Y))),
```

which again simplifies, opening up the definition of APPEND, to:

```
      (TRUE).
```

```
Case 2. (IMPLIES (AND (EQUAL (APPEND (CDR U) (APPEND A Y))
                             (APPEND (APPEND (CDR U) A) Y))
                      (LISTP Y)
                      (LISTP A)
                      (LISTP U))
```

```
            (EQUAL (APPEND U (APPEND A Y))
                   (APPEND (APPEND U A) Y))),

  which simplifies, applying CDR.CONS, CAR.CONS, and
  CONS.EQUAL, and opening up the function APPEND, to:

        (TRUE).


That finishes the proof of *1.1, which, consequently, finishes the proof of *1.  Q.E.D.
        Load average during proof:  1.865178
        Elapsed time:  14.509 seconds
        CPU time (devoted to theorem proving):  7.727 seconds
        IO time:  3.385 seconds
        CONSes consumed:  11520
```

In the proof above the system "discovers" the lemma that APPEND is associative and proves it by the second induction.

The theorem-prover is automatic in the sense that once it begins a proof the user contributes nothing. However, it is interactive in the sense that the user can improve the theorem-prover's behavior by "teaching" it important relationships and rewrite rules. This "teaching" (which might be more appropriately called "memorization by rote") is accomplished by instructing the theorem-prover to prove lemmas that "inform" it of new conditional rewrite rules, useful measures for the justification of recursions and inductions, etc. For example, had the user previously instructed the system prove the associativity of APPEND the system would have used that fact early in the proof above, leading to a substanially simpler proof.

The user of our system does not have to be trusted. That is, as long as he confines himself to the "rules of the game" (i.e., defining new types and functions and proving new lemmas), the theorem-prover is entirely responsibible for the validity of any conjecture it claims is a theorem.

While the user who abides by the rules need not be trusted, an intelligent and well-trained user is indispensable in the proof of difficult theorems because the theorem-prover requires so much carefully prepared groundwork in the form of previously proved lemmas. Much of our research is aimed at reducing some of this burden on the user. However, even at the current rudimentary stage of the system's development, we have found that we (as human users) are quite good at the task required of us (i.e., the strategic planning of proofs encoded in the statement of key lemmas) and are relatively weak at the tasks already performed by the system (the consideration of countless nitty gritty details).

The system has been used to prove the correctness of a wide variety of programs including:

1. a "toy" expression compiler,

2. a recursive descent parser (the theorem-prover established the required relationship between "printing" and "reading"),

3. the totality, soundness, and completeness of a decision procedure for propositional calculus,

4. the soundness of an arithmetic simplifier now in routine use in the system,

5. the termination of the TAK function over the positive and negative integers (using a lexicographic measure corresponding to "less than" in omega^^3^), and

6. several working FORTRAN programs including the correctness of the fastest known string searching algorithm.

The FORTRAN programs were coded in a subset of both ANSI FORTRAN 66 and ANSI FORTRAN 77 and the verification conditions included the consideration of aliasing, side-effects via labeled COMMON, arithmetic overflow, array bounds violations, undefined variables, and termination. I will discuss this aspect of our work in the third lecture.

The most difficult theorem proved to date is the existence and uniqueness of prime factorizations, which was derived entirely from Peano's axioms. While this theorem is not often involved in the correctness proofs of real programs (encryption algorithms excepted), the system's ability to prove it from the ground up is indicative of the theorem-prover's power.

All of the theorems cited above were proved by the same version of the theorem-prover from the same initial set of axioms. The axioms are those defining TRUE, FALSE, IF, and EQUAL, plus the Peano-like axiomatization of the "data types" involved.

Given that the system has some "learning" (or "rote memorization") ability, the question arises: "Is it possible to teach the system new proof techniques that were not anticipated by the designers of the theorem-prover?" Of course, we wish to preserve the soundness of the system, i.e., it should not be possible for the user to render the system unsound by teaching it faulty "proof" techniques.

Since our system is oriented towards proving properties of programs, an obvious approach is for the user to write a new theorem-proving routine to be added to the system, and then have the trusted version of the system prove the new extension correct before encorporating it. Can a system which is inherently inadequate (after all, it is in need of extension) be expected to prove the correctness of a useful extension? We have investigated this problem and believe the answer, for our system, is "yes."

One experiment we performed involved the addition of a simple cancellation routine. Suppose I, J, K, and L are nonnegative integers. Then it is easy to prove that I+J=I+K iff J=K. This is the traditional statement of the cancellation law for addition. But note that this rule cannot be applied to L+J=K+(I+L), because the common term, L, does not occur as the first addend. While we could prove many different versions of the cancellation law, no finite number of rewrite rules can capture the underlying idea: you can cancel any term occurring as an addend on both sides of an equality. How can we teach our system this idea?

We can proceed as follows. Define the function CANCEL on list expressions that, when given an expression representing an equation between two PLUS-trees, returns a new expression with all the common addends deleted, and when given any other expression returns the input expression. To cancel common addends CANCEL computes the fringe of the two PLUS-trees, intersects them, subtracts the intersection from each fringe, and then reconstitutes the remaining lists of terms as right-associated PLUS-trees and equates them. One must be careful to keep in mind that the fringes are bags, not sets, and that duplications have significance (e.g., if A occurs twice on one side and only once on the other, one A can be cancelled).

Once CANCEL has been defined it can be used as a new proof technique provided we can prove the following "metatheorem": Suppose X is a list structure representing a term in our logic and MEANING is the function that assigns meanings to terms, given an assignment of values to variables. Then we wish to prove that under all assignments the MEANING of X is equal to the MEANING of (CANCEL X) and (CANCEL X) represents a term. That is, we wish to prove:

```
(IMPLIES (FORMP X)
       (AND (EQUAL (MEANING X A)
                 (MEANING (CANCEL X) A))
           (FORMP (CANCEL X)))).
```

This theorem can be proved by the current system, after the user has had the system prove the rudiments of "bag theory" (e.g., that the difference between two bags is a subbag of the first) and the fundamental relationships induced by MEANING between bag operations and arithmetic (e.g., if Y is a subbag of X then the MEANING of the PLUS-tree formed from the bag difference of X and Y is the arithmetic difference of the MEANINGs of the PLUS-trees formed from X and Y individually).

After proving the correctness of CANCEL, the system is justified in using CANCEL to perform arbitrarily deep cancellations, an ability it did not have before or during the correctness proofs.

Except for the work on "metatheorems" all of the work described here is described in complete detail in our book, [6]. The book describes our formal theory (assuming only that the reader is familiar with propositional calculus and equality) and all of the proof techniques used by our program. The techniques are demonstrated in many substantial examples worked by the program. The techniques are described in sufficient detail to permit a student to use our techniques to discover proofs as well as to program a computer to reproduce our results. The work on metatheorems mentioned here is described in complete detail in [7].

1.3 Program Verification

There are many ways to reduce the question "Is this program correct?" to the question "Are these formulae theorems?" Once such a reduction has occurred, it would be convenient if a mechanical theorem-prover were capable of proving the necessary theorems. In this talk I will illustrate how we are using the theorem-prover described above to address such questions. I will use three different methods of giving meaning to programs: the functional method, the interpreter method, and the inductive assertion method. Each will be illustrated by the same "toy" problem. Then I will describe a verification condition generator for ANSI FORTRAN that handles such real problems as aliasing, global COMMON, and arithmetic overflow. The talk will conclude with a description of the verification of a FORTRAN version of the fastest known string searching algorithm.

1.3.1 A Toy Example

Let us consider a simple assembly language program to sum the numbers from 1 to I:

```
0    MOVEI AC, 0        ;set AC to 0
1    SKIPNE I           ;skip next if I not 0
2    STOP               ;stop
3    ADD AC, I          ;set AC to AC+I
4    SUBI I, 1          ;set I to I-1
5    JUMP 1             ;jump to instruction 1
```

We wish to prove that when this program is executed the final value of AC is (i*i+1)/2, where i is the initial value of I.

We will consider three different methods of attaching semantics to this program. It is advantageous in all three cases to first introduce the recursive function that sums the integers from I to M:

```
(SIGMA I M)
  =
(IF (LESSP I M)
   (PLUS M (SIGMA I (SUB1 M)))
   0).
```

For example, (SIGMA 3 7) is 7+6+5+4+3. It is also worthwhile proving the general result that (SIGMA 0 I) is (I+(I+1))/2. This is proved by the theorem-prover described above, using induction on I. Having proved this lemma, it is now sufficient to establish that our 6-line assembly program computes (SIGMA 0 I).

1.3.2 The Functional Method

The first method we will consider, often called the "functional" or "McCarthy" method [22], is to view one's program as a mathematical function from input states to output states and to prove the correctness of the resulting function. To transform the program above into this paradigm we first construct the function, FN, of I and AC, that embodies the loop through instruction 1, and think of the entire program as being functionally equivalent to (FN I 0).

```
(FN I AC)
  =
(IF (ZEROP I)
   AC
   (FN (DIFFERENCE I 1)
      (PLUS AC I))).
```

As McCarthy notes, this transformation from a program to a recursive equation can be carried out by a machine which embodies the semantics of the programming language. (For example, see the application described in [23].)

The conjecture we wish to prove is that (FN I 0) is equal (SIGMA 0 I). As often is the case, it is easier to prove the following more general fact about FN:

```
(EQUAL (FN I AC)
     (PLUS AC (SIGMA 0 I))),
```

for all numeric AC. The proof of this generalization is straightforward by induction on I and can be constructed by the theorem-prover described above.

1.3.3 The Interpreter Method

A second method for formalizing the properties of a program is to specify formally an interpreter for the programming language. This is akin to "denotational semantics" [13].

In this case we must specify the "hardware" that runs our 6-line program. We will do so by writing a recursive function, EXEC, that takes three arguments: the program counter, pc; a memory, mem, mapping integer addresses to their values; and a clock, clk, that ticks once every time we execute a jump instruction. The clock is used to make EXEC a total recursive function. EXEC is an accurrate if somewhat simple formalization of the idea of a stored program computer. Each instruction is a list encoding an "opcode" and some arguments and will occupy one location in memory. In our example, the program will be loaded into memory locations 0 through 5, we will use locations 6 and 7 for the variables I and AC.

EXEC operates as follows. If the clock is 0, EXEC returns an error signal. Otherwise, EXEC fetches the contents of location pc in mem and decodes it as an instruction, obtaining an "opcode," op, and two operands arg1 and arg2. If op is STOP, EXEC returns the final memory configuration. Otherwise, EXEC determines new values for pc, mem, and clk based on op and the operands and recurses on those new values. For example, if op is JUMP, EXEC recurses, replacing pc by arg1 and decrementing clk. If op is ADD, EXEC recurses replacing pc by pc+1 and mem by

```
(SET arg1
    (PLUS (GET arg1 mem) (GET arg2 mem))
    mem).
```

That is, the "new" memory is that obtained by adding the contents of arg1 and arg2 in the old memory and then setting the contents of arg1 to that sum. (GET and SET are defined functions that operate on finite sequences denoting the contents of successive memory locations.) The other opcodes used in our program are handled similarly.

Once EXEC is defined we can state the correctness of our program as the following conjecture:

```
(IMPLIES (AND (EQUAL MEM
                (APPEND '((MOVEI 7 0)
                          (SKIPNE 6)
```

```
           (STOP)
           (ADD 7 6)
           (SUBI 6 1)
           (JUMP 1))
          REST))
  (EQUAL I (GET 6 MEM))
  (LESSP I CLK))
(EQUAL (GET 7 (EXEC 0 MEM CLK))
   (SIGMA 0 I)))
```

This formula says: If locations 0-5 of MEM contain the program in question and if I is the contents of location 6 in MEM and is less than CLK, then the value of location 7 in the memory obtained by executing the program starting at pc 0 in MEM with CLK is (SIGMA 0 I).

This conjecture can be proved by the theorem-prover described above. The proof requires that the system first prove a lemma similar in spirit to an "inductive assertion": provided there is sufficient time on the clock, EXEC computes AC+(SIGMA 0 I) if started at location 1 (instead of location 0) in our program.

1.3.4 The Inductive Assertion Approach

We now move on to an illustration of the "inductive assertion" or "Floyd/Hoare" method [12, 15]. The basic idea is to decorate one's program with assertions that purport to describe the state of the machine at certain points in the computation and then to generate a set of formulas, called "verification conditions" that establish that each assertion holds each time it is encountered. The verification condition generator ("vcg") is an encoding of the semantics of the programming language.

The annotation of our example program above is as follows. Suppose K is the initial value of I. The "input assertion" is T; that is, we put no constraints on I initially. The "output assertion," at the STOP instruction at location 2, is that AC is equal to (SIGMA 0 K). The "loop invariant," at the SKIPNE instruction at location 1, is that AC is equal to (SIGMA I K) and I <$M-1_ K. By exploring the paths through the program (using some formal specification of the effects of each instruction) we generate three "verification conditions" to prove:

The loop assertion is true when first encountered:

```
(AND (EQUAL 0 (SIGMA K K))
   (LESSEQP K K)).
```

The loop assertion remains true as we go around the loop:

```
(IMPLIES (AND (NOT (ZEROP I))
            (EQUAL AC (SIGMA I K))
            (LESSEQP I K))
       (AND (EQUAL (PLUS AC I)
                   (SIGMA (SUB1 I) K))
            (LESSEQP (SUB1 I) K))).
```

The loop assertion implies the output assertion when the loop stops:

```
  (IMPLIES (AND (ZEROP I)
                (EQUAL AC (SIGMA I K))
                (LESSEQP I K))
           (EQUAL AC (SIGMA 0 K))).
```

These formulas can be proved by the theorem-prover described above.

1.3.5 Comparisons

The three program verification methods sketched are more striking in their similarities than in their differences.

First, it should be noted that the introduction of SIGMA simplifies the conjectures produced by all three methods. A more commonly used specification style -- at least when the inductive assertion method is chosen -- is to restrict oneself to the "primitives" such as addition, multiplication, and division already built into the system. This makes the verification conditions more difficult to prove because one is simultaneously grappling with the fundamental mathematical fact that (SIGMA 0 I) is (I*(I+1))/2 and with a particular algorithm for computing (SIGMA 0 I).

Second, all three methods require some creative step beyond the mere specification of the input/output relation. In the functional method, this creative step was the generalization of:

```
  (EQUAL (FN I 0) (SIGMA 0 I))
```

to

```
  (EQUAL (FN I AC) (PLUS AC (SIGMA 0 I))).
```

In the interpreter method, the creative step was the statement of the lemma that when EXEC starts executing at location 1 and runs to normal completion, the answer is (PLUS AC (SIGMA 0 I)). In the inductive assertion method, the creative step was the invention of the loop invariant that AC is (SIGMA I K). Note the subtle difference between the latter two invariants: while an inductive assertion generally states an invariant that holds between two successive arrivals at some program point, the interpreter-style lemma states an invariant that holds between arrival at the point and the end of the computation.

It should be noted that with the functional and interpreter methods the creativity occurred after the problem had been cast mathematically and while a proof was being sought. That is, the creative steps were just generalizations in the mathematical sense: given to prove p we decided to prove q, where q implies p. In the inductive assertion method, we were obliged to think about q before the problem could be stated without reference to the program text. Thus, when the former methods are applied, the creative aspect of the problem is just a theorem-proving problem.

On this particularly simple problem, the theorems required by the functional method are the easiest to prove, with the inductive assertion method second and the interpreter method a distant third. Of course, nothing in general should be inferred from this ranking.

For example, applying the functional method to messier programs -- especially programs manipulating large global data structures -- often produces unmanageably large recursive equations; in such cases the inductive assertion method can often be used to segment the program and isolate side-effects.

On the other hand, the interpreter method has an elegance the other two lack because the program was proved correct with respect to a formal programming language semantics expressed entirely within the logic itself rather than in some extralogical axioms or vcg programs. Furthermore, the interpreter method as it was applied here dealt with a problem neither of the other two methods could possibly handle: the instructions were being fetched from a memory that was being modified by the execution of the program. While the program does not happen to modify itself, consideration of that possibility vastly complicates the proof. When the interpreter method is formalized so that the program is in "read only" memory (i.e., a memory not changed by any instruction) the interpreter-based proofs are comparable to the inductive assertion style proofs.

1.3.6 Toys v. Reality

The preceding sketches were meant to summarize several different approaches to program verification and to illustrate the role of theorem-proving in each of them. However, all three sketches dealt with toy problems. They do not describe real programming languages. They ignore many difficult problems of programming language design (e.g., global data structures, subroutine calls, aliasing). They ignore many difficult problems of programming language implementation (e.g., arithmetic overflow, nontermination, undefined variables). In short, the toy problems discussed here bear about as much resemblence to real programming problems as E=Mc^^2^ does to a nuclear power plant. Rather than attempt to describe how these problems can be dealt with I will simply "advertise" and illustrate how we have dealt with them in the context of one real programming language.

We have implemented a verification condition generator for a subset both of FORTRAN 66 [31] and FORTRAN 77 [1]. While constraints are placed on the language that are not found in the ANSI specifications, our language is a true subset in the sense that a processor that correctly implements either FORTRAN correctly implements our language. The development of the FORTRAN verification condition generator was supported by ONR Contract N00014-75-C-0816.

Unusual features of our system -- aside from our choice of FORTRAN and our use of a quantifier free specification language -- include a syntax checker that enforces all our syntactic restrictions on the language, the thorough analysis of aliasing, the generation of verification conditions to prove termination, and the generation of verification conditions to ensure against such run-time errors as array-bound violations and arithmetic overflow.

Although our syntax checker and verification condition generator handle programs involving finite precision REAL arithmetic, we have not yet formalized the semantics of those operations and hence cannot mechanically verify programs that operate on REALs.

We define our subset precisely in [8] and specify the verification conditions we generate. The following description of our work is extremely informal.

The input to our verification condition generator must include not only the subprogram (function or subroutine) to be verified, but also all subprograms referenced somehow by the candidate subprogram. Each referenced subprogram must have been previously specified and verified.

The FORTRAN statements in our subset are:

Arithmetic assignment
Logical assignment
GO TO assignment
Unconditional GO TO
Assigned GO TO
Computed GO TO
Arithmetic IF
CALL
RETURN
CONTINUE
STOP
PAUSE
Logical IF
DO
DIMENSION
COMMON
INTEGER
REAL
DOUBLE PRECISION
COMPLEX
LOGICAL
EXTERNAL
Statement function
FUNCTION
SUBROUTINE
END

Our subset does not include the following FORTRAN 77 statements:

BACKSPACE
BLOCK DATA
Block IF
CHARACTER
Character assignment
CLOSE
DATA
ELSE
ELSEIF
ENDFILE
ENDIF
ENTRY
EQUIVALENCE
FORMAT
IMPLICIT
INQUIRE
INTRINSIC
OPEN
PARAMETER
PRINT
PROGRAM
READ
REWIND
SAVE
WRITE

For those statements in our subset we enforce all of the restrictions of both FORTRAN 66 and 77; furthermore, we enforce some additional restrictions. Some of our restrictions are:

> Every expression using infix operators must be fully parenthesized. For example, either (A + (B + C)) or ((A + B) + C) must be written instead of A + B + C. The precise order of combination affects the analysis of overflow.
>
> Subroutines and functions may not be passed as arguments to subprograms.
>
> In a CALL statement or function reference, if the actual argument is an array, then the corresponding argument must be an array of the same number of dimensions.
>
> Function subprograms may not side-effect their arguments or anything in COMMON.
>
> No call of a subroutine may pass an entity to a subroutine that might violate the strict aliasing restrictions of FORTRAN. For example, if a subroutine has two arguments and possibly smashes the first, then that subroutine may not be called with the same array passed in both arguments nor may an array in common be passed as the first argument if the subroutine "knows" about the common block, even via subprograms.

While some of our restrictions may appear radical to those unfamiliar with the details of the FORTRAN specifications, many of the most severe (e.g., prohibition of side-effects in FUNCTIONs and aliasing in SUBROUTINEs) are in fact closely related to restrictions in both the 1966 and 1977 specifications. Many of the restrictions in the ANSI specifications were motivated by the desire to encourage the implementation of correct optimizing compilers and -- while the restrictions are not as elegantly stated as they might have been -- it could be argued that FORTRAN 66 was several years ahead of its time. In [8] we compare our restrictions to those of the ANSI specifications. All of our restrictions are enforced by the system in the sense that programs violating these restrictions are rejected by the verification condition generator.

We make the following claim about our system. If a FORTRAN subprogram is accepted by our syntax checker, the verification conditions are proved, and the program can be loaded onto a FORTRAN processor that meets the ANSI specification of FORTRAN and satisfies certain parameterized constraints on the accuracy of arithmetic, then any invocation of the program in an environment satisfying the input condition of the program will terminate without run-time errors and produce an environment satisfying the output condition of the program.

We have used the theorem-prover to prove the verification conditions produced for several working FORTRAN programs, including a FORTRAN implementation of the Boyer-Moore fast string searching algorithm, and several subprograms performing "big number" arithmetic operations on arrays of integers regarded as numbers in a large base (e.g., 2^{18}).

1.3.7 A Real Example

In a 1977 Communications of the ACM article [5], we described an algorithm for finding the first occurrence of one character string, PAT, in another, STR. The algorithm is currently the fastest known way to solve this problem on the average. Our algorithm has two unusual properties. First, in verifying that PAT does not occur within the first i characters of STR the algorithm will typically fetch and look at fewer than i characters. Second, as PAT gets longer the algorithm speeds up. That is, the algorithm typically spends less time to find long patterns than short ones. In this section we will briefly describe the verification of a FORTRAN version of the algorithm. A more complete description may be found in [8].

The whole idea behind the algorithm is illustrated by the following example. Suppose we are trying to find PAT in STR and, having scanned some initial part of STR and failed to find PAT, are now ready to ask whether PAT occurs at the position marked by the arrow below:

```
PAT:       EXAMPLE
STR: LET_US_CONSIDER_A_SIMPLE_EXAMPLE
          ^
```

Instead of focusing on the left-hand end of the pattern (i.e., on the "E" indicated by the arrow) the algorithm considers the right-hand end of the pattern. In particular, the algorithm fetches the "I" in the word "SIMPLE." Since "I" does not occur in PAT, the algorithm can slide the pattern down by seven (the length of PAT) without missing a possible match. Afterwards, it focuses on the end of the pattern again, as marked by the arrow below.

```
PAT:          EXAMPLE
STR: LET_US_CONSIDER_A_SIMPLE_EXAMPLE
                ^
```

In general, as the next step would suggest, the algorithm slides PAT down by the number of characters that separate the end of the pattern from the last occurrence in PAT of the character, c, just fetched from STR (or the length of PAT if c does not occur in PAT). In the configuration above, PAT would be moved forward by five characters, so as to align the "X" in PAT with the just fetched "X" in STR.

If the algorithm finds that the character just fetched from STR matches the corresponding character of PAT, it moves the arrow backwards and repeats the process until it either finds a mismatch and can slide PAT forward, or matches all the characters of PAT.

The algorithm must be able to determine efficiently for any character c, the distance from the last occurrence of c in PAT to the right-hand end of PAT. But since there are only a finite number of characters in the alphabet we can preprocess PAT and set up a table that answers this question in a single array access.

The reader is referred to [5] for a thorough description of an improved version of the algorithm that can be implemented so as to search for PAT through i characters of STR and execute less than i machine instructions, on the average. In addition, [5] contains a statistical analysis of the average case behavior of the algorithm and discusses several implementation questions.

A FORTRAN version of the algorithm is exhibited below. The subroutine FSRCH is the search algorithm itself; it takes five arguments, PAT, STR, PATLEN, STRLEN, and X. PAT and STR are one-dimensional adjustable arrays of length PATLEN and STRLEN respectively. X is the dummy argument into which the answer is smashed. The answer is either the index into STR at which the winning match is found, or else it is STRLEN+1 indicating no match exists.

FSRCH starts by CALLing the subroutine SETUP, which preprocesses PAT and smashes the COMMON array DELTA1. DELTA1 has one entry for each character code in the alphabet. SETUP executes in time linear in PATLEN. It initializes DELTA1 as though no character occurred in PAT and then sweeps PAT once, from left to right, filling in the correct value of DELTA1 for each character occurrence, as though that occurrence were the last occurrence of the character in PAT. Thus, if the same character occurs several times in PAT (as "E" does in "EXAMPLE") then its DELTA1 entry is smashed several times and the last value is the correct one.

```
      SUBROUTINE FSRCH(PAT, STR, PATLEN, STRLEN, X)
      INTEGER DELTA1
      INTEGER PATLEN
      INTEGER STRLEN
      INTEGER PAT
      INTEGER STR
      INTEGER I
      INTEGER J
      INTEGER C
      INTEGER NEXTI
      INTEGER X
      INTEGER MAX0
      DIMENSION DELTA1(128)
      DIMENSION PAT(PATLEN)
      DIMENSION STR(STRLEN)
      COMMON /BLK/DELTA1
      CALL SETUP(PAT, PATLEN)
      I = PATLEN
200    CONTINUE
      IF ((I.GT.STRLEN)) GO TO 500
      J = PATLEN
      NEXTI = (1+I)
300    CONTINUE
      C = STR(I)
      IF ((C.NE.PAT(J))) GO TO 400
      IF ((J.EQ.1)) GO TO 600
      J = (J-1)
      I = (I-1)
      GO TO 300
400    I = MAX0((I+DELTA1(C)), NEXTI)
      GO TO 200
500    X = (STRLEN+1)
      RETURN
600    X = I
      RETURN
      END
      SUBROUTINE SETUP(A, MAX)
      INTEGER DELTA1
      INTEGER A
      INTEGER MAX
      INTEGER I
      INTEGER C
      DIMENSION DELTA1(128)
      DIMENSION A(MAX)
      COMMON /BLK/DELTA1
      DO 50 I=1, 128
      DELTA1(I) = MAX
50    CONTINUE
      DO 100 I=1, MAX
      C = A(I)
      DELTA1(C) = (MAX-I)
100   CONTINUE
      RETURN
      END
```

To specify the input and output assertions FSRCH we must introduce the mathematical concepts of (a) a sequence being a "character string" on a given sized alphabet, (b) the initial segments of two strings "matching," and (c) the leftmost match of PAT in STR. Below we give the definitions of these mathematical functions.

```
Definition.
(STRINGP A I SIZE)
  =
(IF (ZEROP I)
   T
   (AND (NUMBERP (ELT1 A I))
        (NOT (EQUAL (ELT1 A I) 0))
        (NOT (LESSP SIZE (ELT1 A I)))
        (STRINGP A (SUB1 I) SIZE)))
```

```
Definition.
(MATCH PAT J PATLEN STR I STRLEN)
  =
(IF (LESSP PATLEN J)
   T
   (IF (LESSP STRLEN I)
      F
      (AND (EQUAL (ELT1 PAT J) (ELT1 STR I))
           (MATCH PAT
                  (ADD1 J)
                  PATLEN STR
                  (ADD1 I)
                  STRLEN))))
```

Definition.

```
(SEARCH PAT STR PATLEN STRLEN I)
   =
(IF (LESSP STRLEN I)
   (ADD1 STRLEN)
   (IF (MATCH PAT 1 PATLEN STR I STRLEN)
      I
      (SEARCH PAT STR PATLEN STRLEN
            (ADD1 I)))))
```

For example, (MATCH PAT J PATLEN STR I STRLEN) determines whether the characters of PAT in positions J through PATLEN are equal to the corresponding characters of STR starting at position I and not exceeding STRLEN. MATCH is recursive. That is, provided J $\leq$ PATLEN and I $\leq$ STRLEN, MATCH checks that the Jth character of PAT is equal to the Ith character of STR and, if so, requires that there be a MATCH starting at positions I+1 and J+1. The recursive function SEARCH is the mathematical expression of the naive string searching algorithm. (SEARCH PAT STR PATLEN STRLEN I) asks, for each position in STR between I and STRLEN, whether a MATCH with PAT occurs at that position.

The input specification for FSRCH includes the assertion that PAT and STR are both strings on the alphabet from 1 to 128. The output assertion for FSRCH is that whenever it exits, X is set to (SEARCH PAT STR PATLEN STRLEN 1). The loop invariants for FSRCH are just expressions in terms of MATCH and SEARCH, asserting that (at label 200) the winning occurrence of PAT in STR has not yet been found and (at label 300) that a partial match has been established between the terminal substring of PAT and part of STR. The verification condition generator produces some 50 theorems that must be proved to establish that these assertions hold, that both SETUP and FSRCH terminate, and that no runtime errors occur. For example, the statement, at location 400 in FSRCH:

```
I = MAX0((I+DELTA1(C)),NEXTI)
```

requires that we prove (1) C is defined, (2) C is a legal index into DELTA1, (3) DELTA1(C) is defined, (4) I is defined, (5) I+DELTA1(C) does not cause an overflow, and (6) NEXTI is defined. The proof that I+DELTA1(C) does not cause an overflow requires that we put an additional input assertion on FSRCH, namely that the sum of lengths of PAT and STR be expressible on our machine.

References

[1] American National Standards Institute, Inc.
American National Standard Programming Language FORTRAN.
American National Standards Institute, Inc., 1430 Broadway, New York, NY, 1978.

[2] Bledsoe, W.W.
Splitting and Reduction Heuristics in Automatic Theorem-Proving.
Artificial Intelligence (3):27-60, 1972.

[3] Bledsoe.
A New Method for Proving Certain Presburger Formulas.
In *Advance Papers, 4th Int. Joint Conf. on Artificial Intelligence,* pages 15-21. Conference, Tibilisi, Georgia, USSR, 1975.

[4] Bledsoe, W.W., Boyer, R.S., and Henneman, W.H.
Computer Proofs of Limit Theorems.
Artificial Intelligence (3):27-60, 1972.

[5] Boyer, R.S. and Moore, J S.
A Fast String Searching Algorithm.
Communications of the Association for Computer Machinery 20(10):762-772, 1977.

[6] Boyer, R.S. and Moore, J S.
A Computational Logic.
Academic Press, New York, 1979.

[7] Boyer, R.S. and Moore, J S.
Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures.
Technical Report CSL Technical Report 108, SRI International, Menlo Park, CA, 1979.

[8] Boyer, R.S. and Moore, J S.
A Verification Condition Generator for FORTRAN.
Technical Report CSL Technical Report 103, SRI International, Menlo Park, CA, 1980.

[9] Chang, C. and Lee, R.C.T.
Symbolic Logic and Mechanical Theorem Proving.
Academic Press, 1973.

[10] Colmerauer, A., et al.
Un Systeme de Communication Homme-Machine en Francais.
Technical Report, Groupe de Researche en Intelligence Artificielle, Universite d'Aix-Marseille, Luminy, 1972.

[11] Feigenbaum, E. and Feldman, J.
Computers and Thought.
McGraw-Hill Book Company, 1963.

[12] Floyd, R.
Assigning Meanings to Programs.
In *Mathematical Aspects of Computer Science, Proceedings Symposium on Applied Mathematics*, pages 19-32. American Mathematical Society, Providence Rhode Island, 1967.

[13] Gordon, M.
The Denotational Description of Programming Languages.
Springer-Verlag, New York, 1979.

[14] Gordon, M., et al.
Edinburgh LCF.
Technical Report CSR-11-77, Edinburgh University, 1977.

[15] Hoare, C.
An Axiomatic Basis for Computer Programming.
Communications Association on Computer Machinery 12(10):576-583, 1969.

[16] Huet, G. and Oppen, D.C.
Equations and Rewrite Rules, A Survey.
Technical Report, SRI International, Menlo Park, CA, 1981.

[17] Jutting, L.S.
Checking Landau's 'Grundlagen' in the AUTOMATH System.
PhD thesis, Eindhoven University of Technology, 1976.

[18] Kowalski, R.
Predicate Calculus as a Programming Language.
North-Holland, 1974.

[19] Kowalski, R.
A Proof Procedure Using Connection Graphs.
Journal for the Association of Computer Machinery 22:572-595, 1975.

[20] Litvintchouk, S. and Pratt, V.
A Proof-Checker for Dynamic Logic.
In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence.* Carnegie-Mellon University, Pittsburgh, PA, 1977.

[21] Loveland, D.W.
Automated Theorem Proving: A Logical Basis.
North-Holland, 1978.

[22] McCarthy, J.
Recursive Functions of Symbolic Expressions and Their Computation by Machine.
Communications of Association for Computer Machinery 3(4):184-195, 1960.

[23] Moore, J S.
Introducing Iteration into the Pure LISP Theorem Prover.
IEEE Transactions on Software Engineering 1(3):328-338, 1975.

[24] Moses, J.
Algebraic Simplification: A Guide for the Perplexed.
In S.R. Petrick (editor), *Proceedings ACM 2nd Symposium on Symbolic and Algebraic Manipulation.* ACM, 1971.

[25] Oppen, D.
Reasoning About Recursively Defined Data Structures.
Technical Report CS Report STAN-CS-78-678, Stanford University, 1978.

[26] Robinson, J.A.
A Machine-Oriented Logic Based on the Resolution Principle.
Journal for the Association of Computer Machinery 12:23-41, Jan, 1965.

[27] Robinson, J.A.
Logic: Form and Function.
North-Holland, New York, 1979.

[28] Roussel.
PROLOG: Manuel de reference et d'utilisation.
Technical Report, Groupe de Researche en Intelligence Artificielle, Universite d'Aix-Marseille, Luminy, 1975.

[29] Shostak.
Deciding Linear Inequalities by Computing Loop Residues.
Technical Report, SRI International, Menlo Park, 1978.

[30] Skolem.
On Mathematical Logic.
In J. van Heijenoort (editor), *From Frege to Goedel*, . Harvard University Press, Cambridge, MA, 1967.

[31] United States of America Standards Institute.
USA Standard FORTRAN.
USA, 10 East 40th Street, New York, 1966.

[32] Wang, H.
Towards Mechanical Mathematics.
IBM Journal for Research Development (4):2-22, Jan, 1960.

[33] Warren, D.
Implementing PROLOG, a Language for Programming in Logic.
Technical Report, University of Edinburgh, 1976.

[34] Weyhrauch.
A User's Manual for FOL.
Technical Report STAN-CS-77-432, Stanford University, 1977.

CHAPTER 23

CONCLUSIONS

1. Conclusions and Recommendations

Our general conclusions are the following:

1. This effort and others (University of Texas, Stanford University, University of Southern California - Information Sciences Institute, Compion Company, and the System Development Corporation) show that the mechanized verification of complex systems is now feasible, although still difficult.

2. We are quite pleased with the progress in verification from 5 years ago when most of the effort was on verifying algorithms (albeit complex ones).

3. We are also quite pleased that comparatively simple requirements statements can be formulated for complex statements; when one removes the implementation detail from a description of a system what is left can be quite simple.

4. In the verification of SIFT, most of the effort was absorbed in creating the models; once created, the verification of the models was not difficult using the STP Theorem Prover.

5. The creation of models is not a routine activity. It is likely that inexperienced users or those not skilled in mathematical logic will have to go through many iterations of model creation/proof before succeeding in a verification of a complex system. Frustration might force such users to give up before achieving success. Several improvements below should help alleviate this problem, including a more intuitive specification language, a library of previously created models, and tools that are more helpful in the design/verification process.

6. The tools we developed, although still relatively primitive and experimental, were useful; indeed, a completely manual proof of SIFT would have been too tedious and error-prone.

7. The techniques for verification of numerical algorithms and hardware are still very tentative. A major effort, perhaps of the magnitude we carried out for SIFT, is required to fully understand the problems and to develop the tools and techniques to make the verification process feasible.

Our recommendations for future investigation are the following:

1. The verification of SIFT should be completed, including the reconfiguration design (and code), the synchronization and interactive consistency programs, and the actual running code of SIFT. By having the running code verified, it will be possible for demonstrate that a verified system can withstand the careful testing that skeptics of verification would wish to carry out. Under a continuation of the effort reported here, SRI is moving towards a complete verification of SIFT.

2. A single specification language should be developed; in this project we suffered through a succession of three specification languages, each with its advantages for particular situations, but none ideal. Under contract to the DoD Computer Security Center, we are currently developing such a language, the novel features of which are:

- Semantics that can be completely defined in terms of the STP Theorem Prover logic

- Support for the creation and use of new types

- *Parameterized* types, to allow specifications to apply in many situations

- Specifications that can apply to a single operation or to a sequence of operations

- Specifications of temporal behavior. Current techniques require time to directly expressed as functions. A more implicit approach, as provided by the newly emerging Temporal and Interval Logics, give promise of leading to more concise specifications of real time systems.

- Incorporation of Hoare logic directly in the language. This feature will allow the user to direct the verification of programs without having to confront verification conditions. It has been our experience that errors in programs are difficult to analyze when presented with reference to verification conditions. Moreover, it is difficult for a user to create lemmas that would help verify verification conditions; such lemmas are more easily formulated with reference to the code and specifications -- as facilitated by the inclusion of Hoare logic in the specification language.

3. Better engineered tools are essential. These tools should support an incremental approach to verification, since that is the only way a complex system can be verified. In this incremental paradigm, the user should be able to compose and verify incomplete programs, models and specifications; to change and reverify components -- the reverification being only for those components impacted by the change; and to ask for the status of the verification at any time.

4. The verification of other *real* systems should be attempted; only by gaining experience in verifying such systems can the verification community fully understand what is required for particular applications and the user community appreciate the utility of the verification technique There is considerable effort being devoted to the design and verification of operating systems that are secure, where *security* means that the operating system prevents data flows in violation of various security models. Despite some limitations, the most popular security model is that concerned with *multilevel security* -- the allowed information flows are governed by the security level (unclassified, confidential, etc.) of the system users and their documents. Other applications worthy of verification are those typically associated with life-critical functions or where errors can have a severe cost penalty, e.g., power plant control systems, mass transit control systems, flight-critical aircraft control systems, or electronic funds transfer systems.

5. As experience is accumulated in real system verification, specifications and models should be published and made available -- perhaps over the Arpanet. Having such documentation should make it easier for the less experienced users to verify their systems.

1. Report No.
NASA Contractor Report 166008

2. Government Accession No.

3. Recipient's Catalog No.

4. Title and Subtitle
Investigation, Development and Evaluation of Performance Proving for Fault-Tolerant Computers

5. Report Date

6. Performing Organization Code

7. Author(s) Karl N. Levitt, P.M. Melliar-Smith, Richard Schwartz, Robert E. Shostak, Dwight Hare, Robert Boyer, J S. Moore, Milton Green, W. David Elliott

8. Performing Organization Report No.
SRI Project 7821

9. Performing Organization Name and Address
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025

10. Work Unit No.

11. Contract or Grant No.
NAS1-15528

12. Sponsoring Agency Name and Address
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665

13. Type of Report and Period Covered
Interim Report

14. Sponsoring Agency Code

15. Supplementary Notes

16. Abstract

Formal Verification is a technique for demonstrating by mathematical reasoning that a system satisfies its requirement. By a *system* we mean the computer hardware and the collection of programs that run on the hardware. A *requirement* is a description of the function to be carried out by the system. The requirement indicates the system's response to all sequences of inputs that could be applied to the system. If the verification is successfully carried out, the system is, in principle, guaranteed to be *correct*; no further validation (e.g., testing) should be required. In practice, testing will be required to discharge assumptions that underly the verification, e.g., does the formally stated requirement satisfy the intents of the client, and is the hardware implementation correct.

This report is concerned with the Formal Verification of computer systems. In the course of carrying out the work reported herein we developed several methodologies for verifying systems, developed computer-based tools that assist users in verifying their systems, and have applied these tools to verifying in part the SIFT ultrareliable aircraft computer. Our approach to verifying SIFT consists of two steps: (1) *Design Verification* in which specifications for each of SIFT's top-level subprograms is shown to be a refinement of a sequence of high-level models, the topmost being an intuitive model of the goals of SIFT, and (2) *Code Verification* in which the subprograms' specifications are shown to be consistent with the Pascal implementation. Using design verification tools, the heart of which is the STP Theorem Prover, we verified that tasks processed by SIFT will yield correct values within predetemined deadlines as long as enough working processors remain to assure that voting masks failures. Current work is considering the verification of other properties of SIFT (e.g., *reconfiguration* -- faulty processors are ignored by good processors), and continuing the verification of the SIFT implementation.

Other work reported herein is concerned with techniques for the verification of (1) the precision of numerical algorithms, (2) control systems, e.g., where the requirement is a control law for a flight control system, and (3) hardware logic.

17. Key Words (Suggested by Author(s))
System verification, design proof, code proof, fault-tolerant computers, flight control system verification, hardware logic verification.

18. Distribution Statement
Unclassified--Unlimited

19. Security Classif. (of this report)
Unclassified

20. Security Classif. (of this page)
Unclassified

21. No. of Pages
597

22. Price

For sale by the National Technical Information Service, Springfield, Virginia 22161

End of Document