NASA Contractor Report 4097

Design Verification of SIFT

Louise Moser, Michael Melliar-Smith,
and Richard Schwartz

CONTRACT NAS1-15528
SEPTEMBER 1987

NASA

NASA Contractor Report 4097

Design Verification of SIFT

Louise Moser, Michael Melliar-Smith,
and Richard Schwartz
SRI International
Menlo Park, California

Prepared for
Langley Research Center
under Contract NAS1-15528

National Aeronautics
and Space Administration

**Scientific and Technical
Information Office**

1987

Contents

1 Introduction 4

2 The Requirements for SIFT 6

3 Substantiating the Reliability of SIFT 8

4 The Specification of Reliability 11

5 Hierarchical Specification and Verification 15

6 An Outline of the Design of SIFT 20

7 The Markov Reliability Model 28

8 An Outline of the Specification Hierarchy 33

9 Fault Model Specification 38

10 Transition Specification 43

11 Input/Output Specification **48**

12 The Replication Specification **54**

13 The Activity Specification **61**

14 PrePost and Imperative Levels **71**

15 Conclusions and Further Work **76**

List of Figures

1 A Hierarchy of Specifications. 16

2 A View of the SIFT Hardware. 21

3 The Broadcasting and Voting of Information in SIFT. . . . 23

4 Information Flow for Error Reporting and Reconfiguration. 25

5 The States of the Markov Model–Projected into the Plane of the First Two Indices, Showing Solid Faults and Reconfiguration but not Transient Faults. 30

6 The Hierarchy of Specifications for the SIFT Design Validation. 34

7 Three tasks in the IO Specification. 50

8 Three Tasks in the Replication Specification. 57

9 The Timing Relationships between Subframes on Asynchronous Processes. 62

10 A Partial View of Three Tasks in the Activity Specification. 64

1

Introduction

A SIFT reliable aircraft control computer system, designed to meet the ultrahigh reliability required for safety critical flight control applications by use of processor replication and voting, was constructed by the Bendix Corporation for SRI, and was delivered to NASA Langley for evaluation in the AIRLAB. To increase our confidence in the reliability projections for SIFT, produced by a Markov reliability model, SRI constructed a formal specification for SIFT, defining the meaning of reliability in the context of flight control. A further series of specifications defined, in increasing detail, the design of SIFT down to pre and post conditions on Pascal code procedures. Mechanically checked mathematical proofs were constructed to demonstrate that the more detailed design specifications for SIFT do indeed imply the formal reliability requirement. An additional specification defined some of the assumptions made about SIFT by the Markov model, and further proofs were constructed to show that these assumptions, as expressed by that specification, did indeed follow from the more detailed design specifications for SIFT. This report provides an outline of the methodology used for this hierarchical specification and proof, and describes the various specifications and proofs performed. An appendix to this report contains the actual specifications and proofs themselves, but in a form that is not suitable for casual perusal.

1. INTRODUCTION

Rather than provide a comprehensive description of the SIFT system and the algorithms used to achieve the desired fault tolerance, this report describes the process of refining the high-level specifications of SIFT down to the implementation level and of justifying the refinement of the specifications by mathematical proof. A more detailed description of SIFT can be found in [1] and [2]. A description of the SIFT executive appears in [3]. The SIFT hardware is documented in [4], [5] and [6]. The fault tolerance algorithms employed are defined in [2] and [7].

The specifications and proofs described here were performed using the Enhanced HDM Specification and Verification System, constructed by SRI for the National Computer Security Center. This report should be read in conjunction with the Enhanced HDM User Manual [13] and the Revised Special Language Definition [14].

2

The Requirements for SIFT

The SIFT computer system has been designed to meet the requirements for future passenger aircraft control. Such aircraft must be designed to use significantly less fuel than current aircraft. Many design innovations are expected to assist in achieving the desired fuel economy, innovations in materials, structures, aerodynamics, engines, and almost every other aspect of aircraft design. Several of these innovations will require computer control of the flight of the aircraft, particularly to maintain the stability of the aircraft and to reduce the stresses in the structures of the aircraft. This computer control will be essential at all times to ensure the safety of flight. Existing aircraft use computers for various purposes, but never to perform flight safety critical functions, and thus do not have to meet the very demanding reliability requirements that apply to safety critical components of the aircraft.

The reliability requirements for a safety critical flight control computer, as proposed by FAA and NASA, allows a probability of life threatening failure no greater than 10^{-9} during a 10 hour flight. This is equivalent to a mean time between failures of about one million years of operation, assuming maintenance after each 10 hour flight. The requirement allows higher rates for less critical failures, but the difficulty of assessing all the consequences of failures in computer systems has led us to regard any deviation from the "correct" output as a failure of the system. Not only must the SIFT computer system be designed to meet this reliability requirement, but the design and design methodology must be such that the reliability estimates made for the system can be substantiated.

3

Substantiating the Reliability of SIFT

The extreme reliability required of SIFT imposes a very difficult task of justifying the achievement of that level of reliability. At the required reliability rate, mere observation, even of a large number of systems, will be ineffective. Furthermore, a SIFT system must be able to recover successfully from several mission 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 be convincing that SIFT meets its reliability requirements.

Our belief 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 examined 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. An outline of that reliability analysis and of its results are given in [1].

The validity of the Markov analysis depends also on the assumptions 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 many states with complex transitions between them, and the correspondence must be maintained even when one or more of the processors has suffered a fault. Because even a very small defect in the correspondence could allow failures at an unacceptable rate, the validation of the correspondence must be performed with as much rigor as we can achieve, the rigor of formal mathematical proof.

The increase in our confidence as to the reliability of SIFT results from the careful identification in the specifications of the assumptions on which that reliability rests, and on our confidence that those assumtions are indeed valid for the actual SIFT implementation, and on our confidence in the validity of the logical deduction leading from those assumtions to the reliability of SIFT. Absolute certainty is not possible, but it our belief that the methodology described here, carefully followed, can indeed significantly reduce our concerns that errors in the design or overlooked assumptions will reduce the level of reliability actually achieved to the point of significant hazard to our systems.

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

- How does one define the criteria so that they are sufficient to ensure the reliable operation of the system?

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

4

The Specification of Reliability

The ability to define formally the requirement for reliability 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 reliable operation of the system do indeed reflect the intended behavior.

Unfortunately, there is no formal or mechanical means by which it is possible to check that these top-level specifications correctly state the property required of the system, the property of reliability in the case of SIFT. That they express the intuitive intent of the system designer must, in the end, be determined by human inspection. The rigorous mathematical verification of correspondence between specification and implementation is meaningful only to the extent that we believe that the specifications do indeed characterize the behavior that we require.

Formal specifications are hard to read and understand, and it is not at all easy to consider all of the details and interactions that ensure our intent in every case. The larger and more complex the system, the more acute the problem becomes. Were we to provide very detailed specifications that reflect closely the actual construction and behavior of the implementation, our task of verifying the correctness of the implementation would be greatly simplified. But experience has shown that such specifications are just as large, just as complex, just as difficult to understand, and just as liable to error, as the implementation itself. If we begin with specifications that are too detailed for validation by human inspection, the considerable effort required for formal verification would, and should, add little to our confidence in the system.

If we are to obtain a specification that can be understood and believed, it is essential that the specification should be brief and simple. Specifications that are only a page or two in length can be read and understood, even by people that are not computer scientists. Such specifications cannot state every detail of the behavior of the system; only the most critical properties we require can be included. In the case of SIFT, these top-level formal specifications define only what we mean by reliable operation and leave almost all other aspects of SIFT to be determined by subsequent stages of design.

For SIFT, the top-level specification consists of two parts linked together by a predicate **system_safe** which indicates that the replication of each of the tasks is sufficient for the voting to be able to mask the effects of the faults present in the system. The two parts are

- A discrete Markov reliability model, with the numbers of working and faulty processors defining the states and the fault and reconfiguration rates defining the transitions. The model also enumerates the states in which **system_safe** is assumed to be true.
- A specification for the behavior required of the system when **system_safe** is true, asserting that the system will perform the desired flight control function by applying the desired computation to the inputs to generate the outputs.

The methodology for achieving a high confidence in the reliability estimates for SIFT now consists of five parts.

- First, the Markov reliability model, based on appropriate estimates for the failure rates of individual components, must be used to compute the probability that **system_safe** remains true for the entire duration of the flight, confirming that this probability is sufficiently high for our purpose.

- The specification of the behavior of the system while **system_safe** is true must be validated by human inspection as being sufficient to ensure the reliable performance of the flight control function that we require. This specification describes the essential behavior we need from the system. Since there are many possible behaviors that we might conceivably need, there is no way, other than human inspection, to determine that this particular behavior is indeed what is required.

- Next, it is necessary to demonstrate by formal verification that, so long as **system_safe** is true, the design of the system, as expressed by the more detailed design specifications, does indeed satisfy the behavioral specification and thus performs the desired flight control function, even though one or more processors may be faulty.

- Further, we must demonstrate that the Markov analysis computes an upper bound on the probability that **system_safe** becomes false. This involves showing that the states of the model are states of the actual system, and that the transitions of the system, resulting from faults and reconfiguration actions, are accurately reflected by the transitions of the model.

- Last, we must achieve as much confidence as possible that the physical implementation precisely corresponds to the design stated in the specifications, and that the assumptions about the implementation expressed by the specifications are reasonable.

The form of expression of the Markov model does not lend itself to direct manipulation by the Enhanced HDM specification and verification system used for this validation. It is therefore not possible to validate the Markov model within the Enhanced HDM system. Rather, an additional set of top-level specifications were prepared that represent the assumptions made by the Markov model about the states and transitions of the actual system. There is no mechanical check of the adequacy of these specifications and, in the current state of the art, human inspection is necessary to establish that the specifications capture the assumptions essential to the validity of the Markov model. The demonstration that the Markov analysis computes an upper bound on the probability of failure thus requires

- Confirmation by human inspection that the additional set of transition specifications adequately state the requirements of the Markov model.
- Validation by human inspection of the transition rates of the model, since the more formal specifications do not yet model the rates at which faults and reconfiguration actions take place.
- Formal verification that the actual system satisfies the set of transition specifications.

5

Hierarchical Specification and Verification

To obtain adequate confidence in the correctness of the top-level specifications, it is necessary to construct very high-level specifications that abstract from many of the details of the system, since only very high level specifications can be succinct enough to be validated by human inspection. The high level of these specifications increases the difficulty of formally verifying that the actual system does indeed satisfy these specifications. Because the top-level specifications address only particular critical aspects of the system and do not describe much of the system's mechanism at all, there may be little in common between them. It is often not obvious how the specifications constrain the results of computations about which they say nothing directly, and it may be quite hard to reconcile the broad requirements of the very high-level specifications with the detailed data structures and transformations of the programs for the actual system.

In the construction of a formal mathematical argument demonstrating that the detailed implementation satisfies a very abstract specification, it has been found necessary to construct a hierarchy of intermediate specifications, as illustrated in Fig. 1.

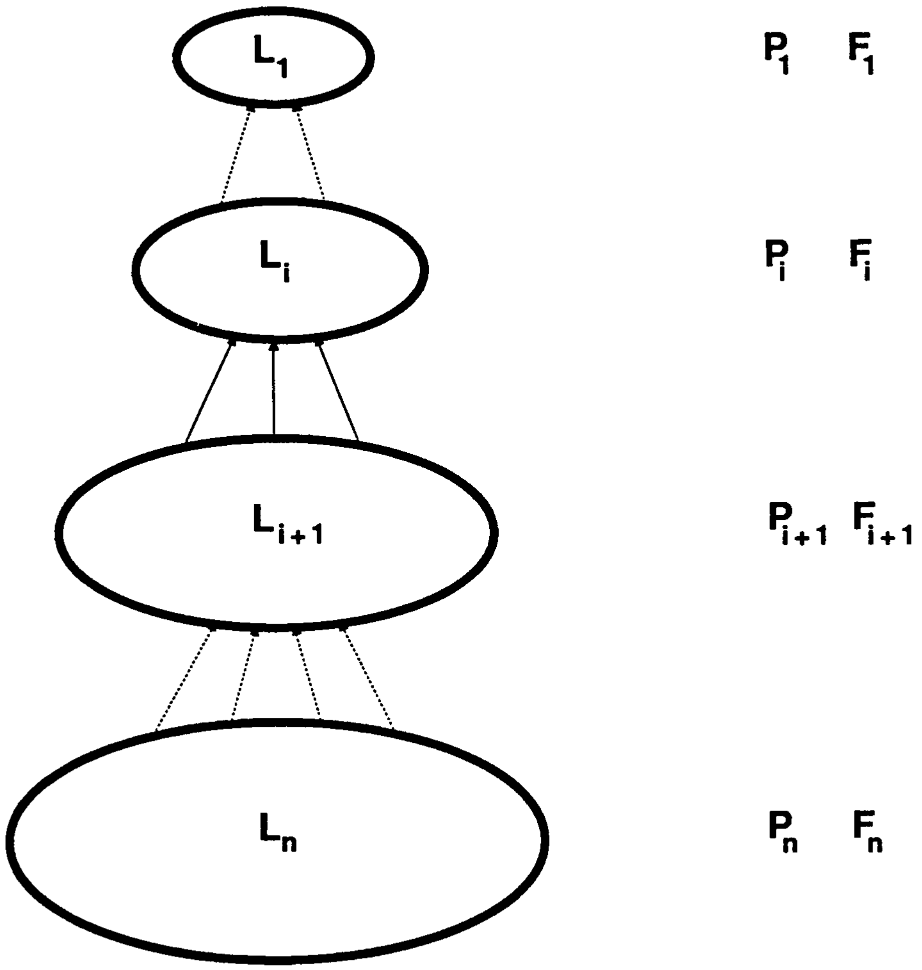

Figure 1: A Hierarchy of Specifications.

5. *HIERARCHICAL SPECIFICATION AND VERIFICATION*

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 system is given by a set of axioms, characterizing those properties of the system appropriate for that level of system abstraction. The specification for level i is constructed as a completely self-contained description of the system, completely describing the system at that level of abstraction. Thus, every level of our hierarchy describes the same system, at different levels of detail. We refer to this hierarchy of specifications as a *vertical* hierarchy.

Specification and programming languages often provide a different type of hierarchy, which we call a ***horizontal hierarchy.*** This horizontal hierarchy is formed when specification modules defining more complex concepts are defined in terms of existing simpler specification modules. Note that, in a horizontal specification hierarchy, successive levels of the hierarchy describe different concepts, whereas all the levels of a vertical hierarchy describe the same concept but at differing levels of abstraction and detail. It is of course quite permissible for the specifications at one level of a vertical hierarchy to be constructed as a horizontal hierarchy, and indeed the specifications for SIFT are so constructed.

Our objective in the formal verification is to show that the implementation satisfies the top-level specification or, in other words, that the properties required by the top-level specification follow as a logical consequence of the implementation. In practice, we perform this verification one level at a time.

The specification at level i is given in terms of sets of predicates P_i and functions F_i, together with a set of axioms that define their properties. The specification at level $i + 1$ is similarly defined. Since these two specifications are given at different levels of abstraction, some of the predicates and functions in terms of which they are expressed will differ. But the more abstract specification at level i is an abstraction of the more detailed specification at level $i + 1$ and thus every predicate P_i and function F_i is an abstraction (possibly complex) of corresponding predicates and functions at level $i + 1$. To establish this correspondence, we construct a mapping between the two levels that defines each primitive function and predicate of the higher level L_i in terms of the functions and predicates of the lower level L_{i+1}. 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 satisfies the mapping axioms.

The verification of satisfaction now requires that each axiom at level L_i, its predicates and functions mapped into those of level L_{i+1}, follows as a logical consequence of the axioms of level L_{i+1}, i.e., that it can be proved as a theorem from those axioms. Thus, under this mapping, any property that can be proved to follow from the specification L_i can also be proved to follow from the lower-level specification L_{i+1}. By demonstrating this correspondence between each successive pair of 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 specification hierarchy for SIFT, 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.

6

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 [8], 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.

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 Fig. 2. The BDX930 is a 16 bit processor specifically designed for military and aircraft use, with an instruction set reminiscent of, but not compatible with, Data General

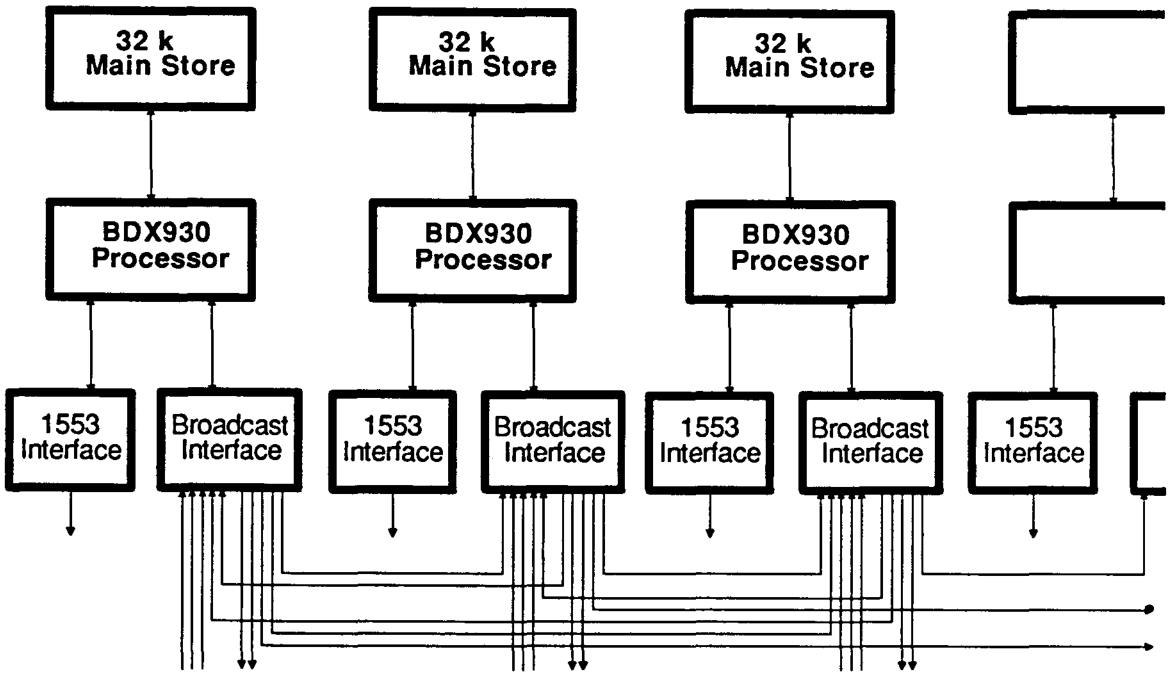

Figure 2: A View of the SIFT Hardware.

computers and a speed of less than one million instructions per second. Each BDX930 processor has its own 32K word main store, which cannot 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 2000 hours.

The processors communicate with each other through the broadcast interface, which contains the drivers and receivers for the star-connected broadcast cables and a 1024 word area of storage called the data file. The broadcast interface operates autonomously from the BDX930 processor, and is designed so that if all processors broadcast simultaneously, the broadcast receivers will still be fast enough to receive and store all the information broadcast. The data file is divided into eight regions, one of which, called the broadcast buffer, is used to hold information to be broadcast, while the other seven regions, collectively known as the input buffer, are used for the storage of information received from up to seven other processors. Thus, if a faulty processor broadcasts garbage, that garbage will all be placed in a specific region of the input buffer of every other processor's data file, where it can be ignored and where it cannot damage sound information being broadcast by other processors.

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. Fig. 3 shows a task *b*, replicated on three processors, with its output being used by a task *a*, of which only one replication is shown. As task *b* generates its outputs during execution, it invokes an executive function which copies the results into the output region of the data file, and broadcasts them to all the other processors. At each of the processors, the various replications of the results of task *b* are received in the regions of the data file corresponding to input from the various processors executing task *b*. In each of the processors, the three versions of the results from task *b* 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

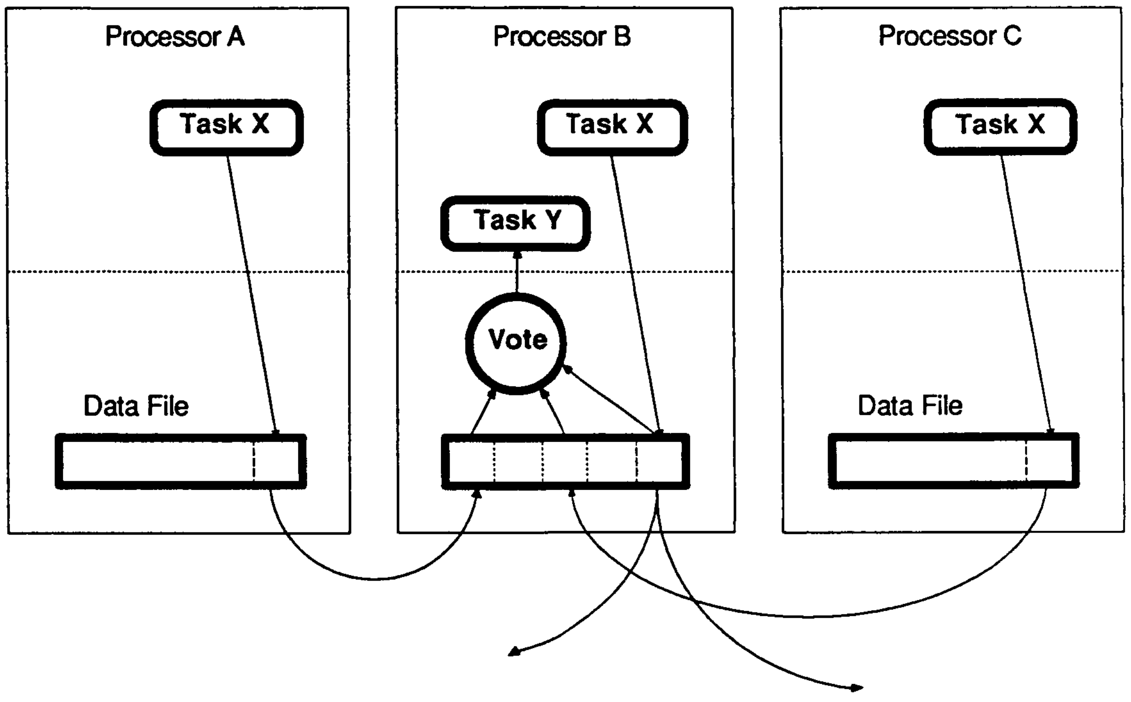

Figure 3: The Broadcasting and Voting of Information in SIFT.

needs to use the results of task *b*. All results broadcast are voted in every processor, even though possibly no task on that processor will use the voted value. Since voting takes time, the various words that are components of the result of a task are voted independently.

The voting software notes any discrepancies among the values on which it votes. A task error reporter, run periodically on every processor, generates a synopsis of the errors detected on that processor and broadcasts the synopsis, as is shown in Fig. 4. The global executive 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 local executive 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 to their own processor alone. Care is taken in the design to ensure that errors in the results of an error reporter or a local executive can damage only its own processor.

The schedule for SIFT is designed so that different combinations of tasks can be executed on different processors, and replicated tasks can be executed at different times on different processors. The schedule is organized into equal subframes, which would typically be 1 or 2 ms long, and are triggered by interrupts from each processor's clock system, the only interrupts in the SIFT system. The sequence of activities to be performed by a processor within a subframe is determined by a schedule table, which

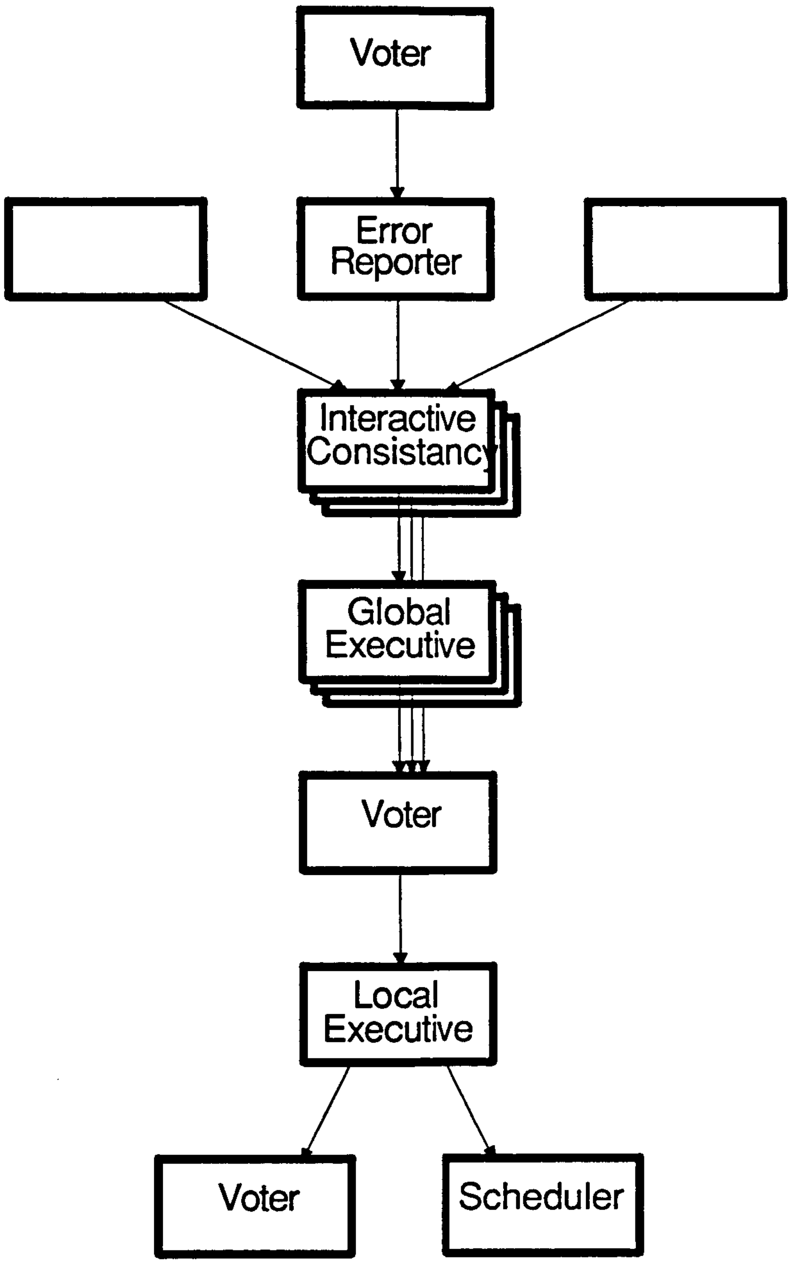

Figure 4: Information Flow for Error Reporting and Reconfiguration.

is selected from several such tables by the configuration broadcast by the global executive. Within a subframe, the schedule can require a sequence of votes and task executions, within the time constraint imposed by the length of the subframe. A task execution can use results voted earlier in the same subframe. However, the voting of results broadcast earlier in the same subframe is prohibited and must await the subframe after that in which the last of the three replications is broadcast. This design decision was made to avoid a complex synchronous proof that the result will have been received before it is voted. The overhead associated with the handling of the clock interrupt, together with the control exercised over the skew between clocks, is sufficient to ensure that results broadcast by a processor in one subframe can safely be voted at any time during the next subframe by any other processor.

Many of the flight control tasks require the same iteration rate, typically 10–20 iterations per second, but other tasks can be run less frequently. This interval, within which these important flight control functions run, is known as the system frame. Tasks, such as those of the global and local executives, are also run within the same frame. Other tasks, such as navigation tasks, could typically be run more slowly, but for this proof the simplifying assumption that all tasks run at the same rate was made. The execution window for each task is the interval of time within which the task must be executed and its results voted. The stability of the control laws mechanized by the flight control programs depends on avoiding long transport delays between the reading of sensor values and the commanding of actuator positions, and thus, for faster flight control tasks, the execution window may be only a few subframes. Slower tasks are less demanding and the execution window for them may extend over much of the frame.

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. The result obtained from an unreplicated source may be erroneous, which the tasks using that value must be able to recognize and accommodate, using data from other sources and application dependent algorithms. The faulty source might even broadcast different values to different processors, thus causing replicated tasks on those processors to obtain different results. The majority-voting algorithms cannot mask errors where all of the replications obtain different results, and this possibility must be prevented. In SIFT, a mechanism called interactive consistency [7] 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.

7

The Markov Reliability Model

The design of SIFT aims to ensure that errors generated by faults are masked by majority voting. Provided that every computation is performed by many correctly working processors and only a few failing processors, so that the correct results are in a majority, this technique is effective and SIFT will operate reliably even in the presence of faults. SIFT can fail in either of two ways:

- So many processors have failed that there are too few processors left in the configuration to ensure that tasks are executed by correctly working processors. This situation is known as *exhaustion of spares.*
- Several processors fail in rapid succession, so that there is not enough time for the reconfiguration algorithms to remove the first failing processor from the configuration before further processor failures occur. Until reconfiguration is completed, some tasks may be executed by more faulty processors than working processors, resulting in a possibility of failure. This situation is known as *coincident faults.*

In both of these cases the cause of failure is one or more tasks being executed on a set of processors in which correctly working processors are not in a majority.

The objective of the Markov reliability analysis is to compute the probability that such a situation never occurs during a flight of appropriate duration, thus ensuring that majority voting will suffice to ensure the reliable operation despite the failure of individual processors. Note that coincident faults or exhaustion of spares do not automatically cause SIFT to fail; the particular allocations of tasks to processors, or the nature of the faults, may be such that SIFT can continue to operate, but the possibility of failure is present. The reliability analysis takes the conservative view that if SIFT reaches a state in which failure is possible then SIFT has failed.

The analysis of the reliability of SIFT is based on a discrete Markov model, described in more detail in [5]. The analysis is absolutely dependent on the assumption that the faults are statistically independent and are well modeled by a Poisson distribution. The hardware design of SIFT aims to support this assumption. In the model, the states of the system are represented by a three dimensional matrix, with transitions between states representing faults and recovery actions. The indices of the three dimensions of the model are

- The number of processors that have been removed from the configuration by reconfiguration
- The number of processors that have developed solid faults
- The number of processors that have developed transient faults.

Fig. 5 shows an example of the projection of the model into the plane of the first two indices, showing the effects of only solid faults and reconfiguration. The initial state of the system is the 0,0 state at top left. The incidence of solid faults is represented by horizontal transitions to the right, while the effect of successful recognition of the fault and reconfiguration of the system to remove that processor from the configuration is represented

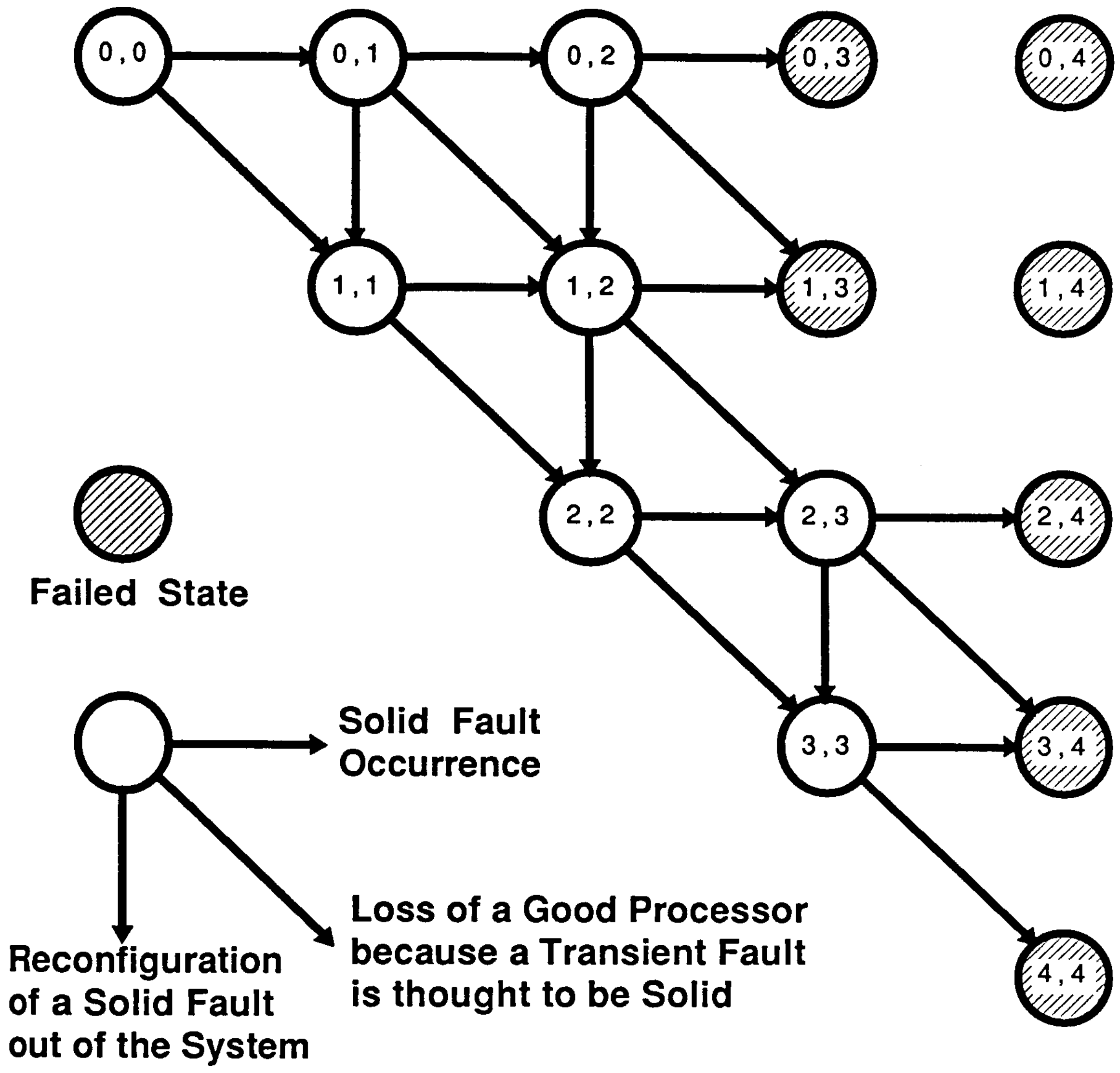
0,0
0,1
0,2
0,3
0,4
1,1
1,2
1,3
1,4
2,2
2,3
2,4
3,3
3,4
4,4
Failed State
Solid Fault Occurrence
Loss of a Good Processor because a Transient Fault is thought to be Solid
Reconfiguration of a Solid Fault out of the System

Figure 5: The States of the Markov Model–Projected into the Plane of the First Two Indices, Showing Solid Faults and Reconfiguration but not Transient Faults.

by downwards vertical transitions. The shaded states are states in which the system is not safe because the number of failing processors in the configuration could exceed the error-masking ability of the majority-voting algorithm.

The example figure shows that initially three successive faults, represented by three successive transitions to the right, would take SIFT into a failed state. The analysis shows that the probability of this particular failure mode is quite low, because faults occur relatively seldom resulting in low transition rates for the horizontal fault transitions, while reconfiguration is rapid resulting in high rates for the vertical reconfiguration transitions. Consequently, the model predicts a very small probability that, having made the first transition to the right, the system will make two more fault transitions without making any reconfiguration transition.

The third dimension of the model represents the occurrence of transient faults; these are not directly visible in this projection, but the diagonal transitions represent the effects of inability of the global executive to recognize transient faults as transient, resulting in the loss of useable processors from the configuration. If additional transient faults are present, more of the states become unsafe. Note that there are no transitions out of unsafe states; once the system becomes unsafe, we assume conservatively for this reliability analysis that it remains unsafe, even though the actual system may well be able to recover.

The Markov model also represents the occurance and recovery from transient faults, expressed as a third dimension, orthogonal to those presented in Fig. 5. The analysis of transient faults is quite complex and does not have a substantial impact on the specifications and proofs. Consequently, we do not include here discussion of that aspect of the Markov model, but refer the reader to [5] where it is explained in detail.

Rates are assigned to each of these transitions, as explained in more detail in [5]. Starting from an initial probability of 1.0 for the 0,0,0 state and 0.0 for all other states, the discrete Markov analysis computes the probabilities for the various states at the end of the required mission duration. The probabilities for the unsafe states are summed and represent an upper bound on the probability of system failure during the mission, assuming of course that system failure is impossible so long as `system_safe` is true. Examples of the results of such analyses are given in [5].

8

An Outline of the Specification Hierarchy

Fig. 6 shows an outline of the various specifications and analyses that are used in the justification of the reliability of SIFT. Before we describe the individual specifications 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 specifications 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 with 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 specification 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. This specification contains only 8 axioms

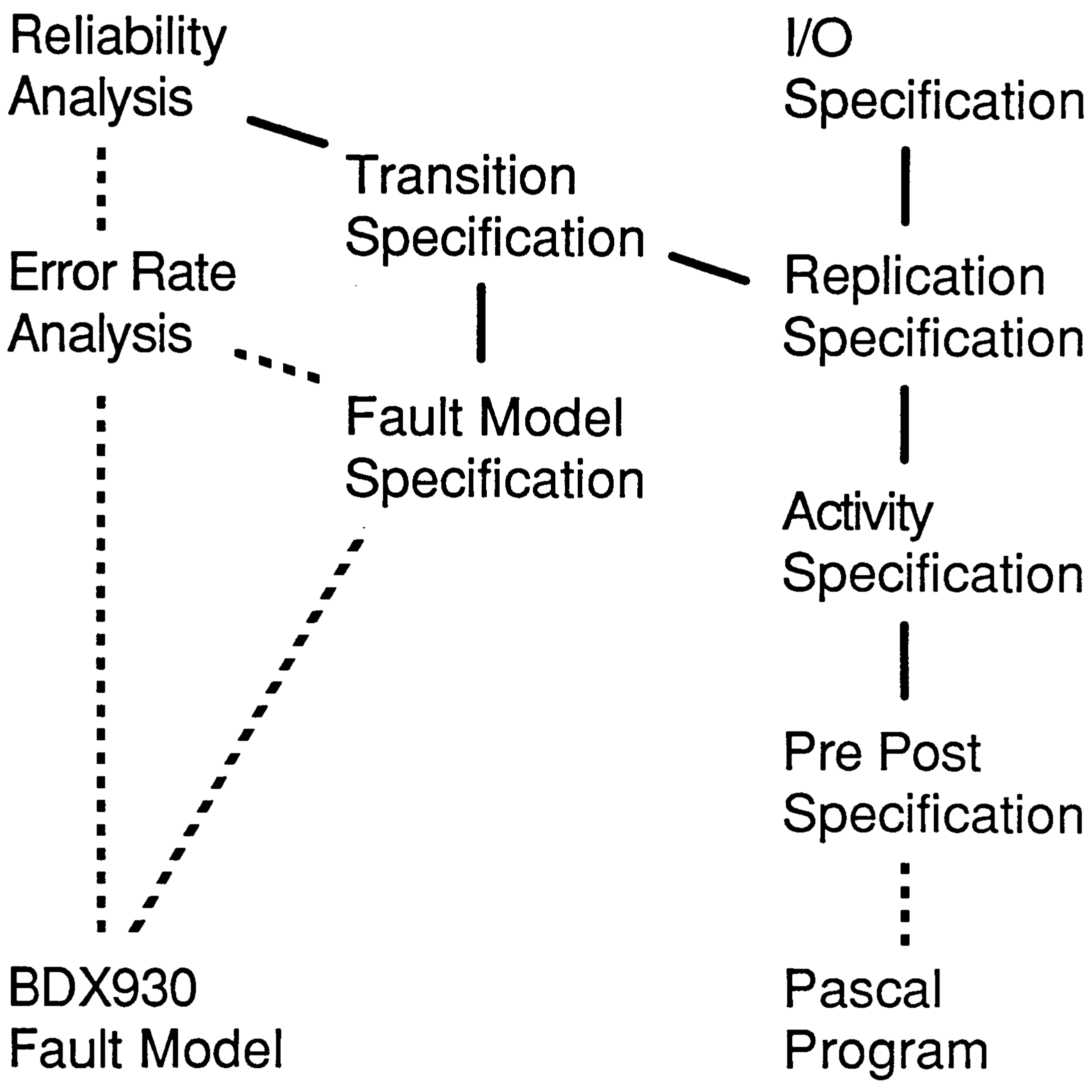

Figure 6: The Hierarchy of Specifications for the SIFT Design Validation.

and is intended to be understandable to an informed aircraft flight control engineer.

The Replication Specification elaborates upon the IO Specification by introducing the concept of processor, and therefore describes the replication of tasks and their allocation of processors, voting on the results of these replicated tasks, and reconfiguration 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 specification 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 to local information only. Each processor has a local clock and a broadcast communication interface and buffers. The asynchrony among processors and its effect upon communication is specified. 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. These specifications are very close to the Pascal programs, and essentially require the programs to "do what they do." In terms of precondition-postcondition pairs, the PrePost Specification facilitates the use of sequential proof techniques to prove properties of the Pascal-based operating system as a sequential program.

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.

8. AN OUTLINE OF THE SPECIFICATION HIERARCHY

The functional behavior described by the I/O Specification is assured only as long as the predicate **system_safe** remains true. The analyses shown on the left of Fig. 6 provide the probability that **system_safe** will remain true and that the desired functional behavior will continue. The BDX930 Fault Model describes the rates of occurrence of various kinds of fault behavior, distinguishing only between faults that cause the same erroneous results to be seen and reported by all other processors, and those that cause different results to be seen, and thus cause conflicting error reports that could confuse the global executive.

The Error Rate Analysis, which is still manual, is used to determine the rates at which faults will cause errors, the rates at which those errors will be detected, the probability that the error reports are clear enough for the global executive to be certain of its diagnosis, and the rates at which the system can be reconfigured in order that the last vestiges of erroneous results can be removed from the system by the majority voting.

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. The analysis computes the probability that **system_safe** remains true for the 10 hour flight duration, as processors become faulty and are reconfigured out of the system. Both the Error Rate Analysis and the Reliability Analysis are Markov models, whose state space must be demonstrated to be an abstraction of the states of the Replication Specification, but whose transition rates are determined by the simpler probabilistic models.

The Transition Specification imposes a set of requirements on the system that are assumed by the Markov model of the Reliability Analysis. The axioms of this specification prohibit some transitions not present in the Markov model, and require others. They also serve to identify the safe states of the model.

The Fault Model Specification describes the various types of faults considered in this proof. The specification distinguishes between solid and transient faults, and between processor and interconnection faults, and defines minimal levels of misbehavior for faulty processors and links that allow the SIFT global executive to recognize them and remove them from the configuration.

9

Fault Model Specification

The Fault Model Specification provides a relatively simplistic definition of the types of faults that might occur in SIFT, a definition that is closely coupled to the capabilities of the SIFT global executive to recognize those faults. More complex global executive algorithms might be coupled with more complex fault models. We give here the statements of the fault model specification, even though understanding some of the details of the specifications requires knowledge of aspects of the replication and activity level specifications described below.

The specification classifies processors as **working**, **hard**, and **transient**, and requires that these be mutually exclusive.

fault1: **axiom**
working_during(p,t) iff not hard(p,t) and not transient(p,t)

fault2: **axiom**
hard(p,t) iff not transient(p,t) and not working_during(p,t)

Hard and transient faults are distinguished in that a hard fault is permanent, but a processor suffering from a transient fault during one error frame is required to be working during the next error frame. The error frame is of the same length as a standard scheduling frame, but is skewed relative to it. Error reports must be collected before the end of a frame to allow time for the global executive to consider them and determine the configuration for the next frame.

```
fault3:  axiom
         t2 ≥ t1 and hard(p,t1)
         implies hard(p,t2)

fault4:  axiom
         is_in_error_frame(i,t1) and is_in_error_frame(i+1,t2)
         and transient(p,t1)
         implies working_during(p,t2)
```

Faults are defined to be either faulty processors or faulty communication links, again mutually exclusive with at most one faulty link per processor.

```
fault5:  axiom
         not working_during(p,t)
         iff faulty_proc(p,t)
            or (exists q: q≠p and faulty_link(p,q,t))

fault6:  axiom
         faulty_proc(p,t)
         iff (forall q: not faulty_link(p,q,t))

fault7:  axiom
         r≠q and faulty_link(p,q,t)
         implies not faulty_link(p,r,t)
```

```
fault8:  axiom
         t1 < t2 and t2 < t1+frame_size
         and faulty_proc(p,t1) and not working_during(p,t2)
         implies faulty_proc(p,t2)

fault9:  axiom
         t1 < t2 and t2 < t1+frame_size
         and faulty_link(p,t1) and not working_during(p,t2)
         implies faulty_link(p,t2)
```

The objective of the design of SIFT is that the specifications define the behavior of working processors, but no assumptions are made about the behavior of faulty processors. This objective aims to reduce the dependence of the reliability on the nature of the faults incurred, particularly against malicious faults.

Our proof can, and indeed does, show that a task that is task_safe obtains correct results without regard to the behavior of any faulty processors present in the configuration. But it is not possible to prove important properties of the reconfiguration behavior, and thus of the transition specification, without making some assumptions about the nature of faults. In particular, a faulty processor that does not generate any erroneous results cannot be detected by the majority-voting algorithms used in SIFT. It also causes no damage until it generates erroneous results. There is a risk that such a fault could lurk undetected in the system until another fault occurs and only then manifest itself, thus exposing the system to much greater risk of failure due to coincident faults. The problem of correlated coincident faults, which violate the assumption of independence, is not addressed in this proof.

To allow us to justify the Transition Specification, the Fault Model Specification imposes assumptions on the nature of processor and link faults. A relatively complex axiom requires that a faulty processor produces at least one wrong result at least once per error frame and broadcasts that wrong result to every other processor, where a wrong result is one that differs from the result obtained by a safe processor. This allows the voting algorithms in every safe processor to detect the error and the global executive to diagnose it. The axiom is complex because the property to be described is best defined in terms of the relatively detailed Activity Specification rather than the simpler Replication Specification.

```
fault10:  axiom
          faulty_proc(p,t)
          and is_in_error_frame(i,t)
          and member(q,safe(ending(of(i-1,global_exec))))
          and data_member(p,config(ending(of(i-1,global_exec)),q))
          implies
          (exists k: is_in_error_frame(i,beggining(of(j,k)))
                     and on_during(k,j)
                     and (forall q,r:
                          (exists t2,y,activ2:
                           (member(q,safe_for(of(j,k)))
                            and member(r,safe_for(of(j,k))))
                            implies
                            (* q votes k *)
                            beginning(of(j,k)) ≤ t2
                            and t2 < ending(of(j,k))
                            and seq_member(activ2,ached(config(t2,q),t2,q))
                            and action(activ2) = vote
                            and task_action(activ2) = k
                            and elem_action(activ2) = y
                            and 1 ≤ y and y ≤ result_size(k)
                            (* p and r both voted by q *)
                            and member(p,pollby_for(q,k,t2))
                            and member(r,pollby_for(q,k,t2))
                            and seq_elem(datafilein_for_on(q,k,p,start(t2,q)),y)
                                ≠ seq_elem(datafilein_for_on(q,k,r,start(t2,q)),y)
```

9. *FAULT MODEL SPECIFICATION*

SIFT is reconfigured by removing entire processors from the configuration, and there is no mechanism to avoid using a faulty link while continuing to use both of the processors it connects. Thus reconfiguration to eliminate a faulty link necessitates removal of one or other of those two processors. For a non-malicious faulty link, there is no reason to select either one of these two processors in preference to the other. But there is a risk that a malicious fault might be able to take advantage of reconfiguration for faulty links to eliminate other processors systematicly from the configuration. To reduce this risk, in SIFT faulty links are associated with the processor receiving information over that communication link, since it appears that the probability of a malicious fault in the relatively complex reception mechanism is much higher than in the very simple transmission mechanism. We assume that a processor affected by a link fault has only one faulty link. The processor is assumed to process information correctly and to receive information correctly on its other links, but to receive only garbled information over the faulty link.

```
fault10:  axiom
          faulty_link(p,r,t1)
          and working_during(q,t1)
          and seq_member(activ,sched(config(t1,q),t1,q))
          and action(activ) = execute
          and task_action(activ) = k
          and r ≠ q
          implies
          datafilein_for_on(p,k,q,finish(t1,q)+broadcast_delay)
          ≠ datafilein_for_on(p,k,r,finish(t1,r)+broadcast_delay)
```

The version of the global executive analyzed by this proof does not contain algorithms that can handle malicious faults at the transmission end of links (a relatively low probability type of fault). More recent versions of the global executive do contain such algorithms, but the proofs undertaken do not address their validity and the Fault Model Specification imposes the assumption that such faults do not occur.

10

Transition Specification

The Transition Specification must express the constraints on the system that are assumed by the Markov reliability analysis. These constraints should be conservative, so that the Markov analysis does not deem the reliability to be better than it really is. The constraints affect both the states and the transitions of the Markov model, as follows:

- The model must not assume a state to be safe when it is not.
- The model must not ignore unfavorable transitions, nor assume favorable ones that do not occur.

These constraints are defined in terms of **config_set**(t) which defines the set of processors comprising the current configuration at time t, and of a predicate **broken**(p, t) which determines whether a processor p is inoperable at time t. The predicate **transient**(p, t) from the fault model is used to distinguish processors that are only transiently faulty, leaving **broken**(p, t) for processors whose disability is more permanent. The predicate **broken**(p, t) is not equivalent to the predicate **solid**(p, t) of the fault model. As will be seen below, a processor becomes unusable if, due to a transient fault, it looses synchronization with the other processors or it no longer has an appropriate value for the current configuration. The set of processors denoted by **safe**(t) comprises those that are neither **broken** nor **transient** at time t.

Transient faults are important in determining safe states but have little effect on the reconfiguration strategy. Consequently, the description below concentrates on solid faults and on the projection of the Markov model shown in Fig. 5.

The axioms that constrain the transitions must ensure that the Markov model does not make optimistic assumptions. The model contains no transitions to the left or upwards, which is expressed by the following axioms:

rep_err3: axiom
t2 $\geq$ t1 implies subset(config_set(t2),config_set(t1))

rep_err5: axiom
t2 $\geq$ t1 and broken(p,t1) implies broken(p,t2)

These axioms state that once a processor has been removed from the configuration it is never readmitted and once a processor is broken it remains broken.

Transitions to the right in Fig. 5 (faults) must not be underestimated, but there is little that we can do to constrain the occurrence of faults. It is, however, possible for the reconfiguration algorithms to make the situation worse by throwing away working processors. This possibility is addressed by the following axioms:

rep_err4: axiom
not member(p,config_set(t1)) implies broken(p,t1)

```
rep_err1:   axiom
            all_safe_history(start_up,i+1)
            and i > start_up
            and (forall t1: is_in_error_frame(i,t1)
                            implies member(p,safe(t1)))
            and (forall t2: is_in_error_frame(i-1,t2)
                            implies member(p,safe(t2)))
            and ending(of(start_up,global_exec)) ≤ t3
            and t3 ≤ ending(of(i+1,global_exec))-1
            implies member(p,config_set(t3))
```

The first of these states that if a processor is removed from the configuration then it becomes broken. If the processor is already broken when the reconfiguration algorithms cause it to be removed, this axiom is of little significance. But if the algorithms erroneously remove from the configuration a working processor, this axiom implies that the transition in the Markov model is not just vertically downwards but rather diagonally to the right.

The second axiom, **rep_err1**, requires that a processor remain in the configuration so long as it is safe. The predicate **all_safe_history**(start_up, $i+1$) requires that **system_safe** be true throughout every frame from the initial start up to iteration $i+1$. The processor must be safe for two successive iterations to guarantee its continued retention in the configuration, for it is possible for a faulty-vote action in one frame to be manifest only in the results of calculations performed during the next frame. In practice, the reconfiguration algorithms of the global executive should be capable of rather better discrimination against transient faults than is implied by this axiom. A more precise statement would however be quite complex, and the axiom as stated is conservative.

The Markov model assumes that faulty processors will be removed from the configuration, as required by:

```
rep_err2:  axiom
           all_safe_history(start_up,i+2)
           and i > start_up
           and (exists t1: is_in_error_frame(i,t1)
                           and broken(p,t1)
           and t2 ≥ ending(of(i+1,global_exec))
           implies not member(p,config_set(t2))
```

Here, again, we must wait until the second frame to ensure removal of the processor from the configuration.

That these axioms adequately express the assumptions implicit in the Markov model must be determined by human inspection, as must the validity of the rates attached to the transitions in that model.

The axioms of the Transition Specification are mapped onto the Replication Specification (described below) by two further axioms:

```
rep_err6:  axiom
           subset(poll_for_of(i,a),config_set(beginning(of(i,a))))

rep_err7:  axiom
           broken(p,t) implies not member(p,safe(t))
```

Using these mappings, the proofs for **rep_err4** and **rep_err5** were easy, but the proofs for **rep_err1**, **rep_err2**, and **rep_err3** were lengthy and difficult. The proofs depend not just on the correct behavior of the global executive for one iteration but on its correct operation for every iteration since the start of the system.

Additional mappings would be required in practice to show that, for each state of the Markov model, the corresponding configuration tables of the replication level provide sufficient replication for each critical flight control task so that **task_safe** can be ensured. **Task safe** for every critical task implies **system_safe**. In the absence of a set of schedules, these mappings and proofs were not performed, but they should not be difficult.

11

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 for a particular iteration of a task is referred to as its execution window, having a beginning 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 specification is any notion that a task is replicated and computed on a set of processors. At a lower level, we shall explain that the **task_result** value that 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 number of task replications and the number of working processors.

Briefly, the specification is organized as follows. Each task a in the set of all executive and application tasks computes a function, denoted by **the_function**(a), of its input values. The function **apply**($f, *$) takes as parameters a functional value and an argument list and produces the result of applying the function to the argument list.[1] The set of tasks providing inputs to a is denoted by **inputs**(a) . For task b that is a member of **inputs**(a), the most recently completed iteration of b prior to the execution window of the iteration of a provides the input to an iteration of a. A derived function **to_of**(b, i, a) denotes the iteration of b providing input to the ith iteration of a. Because all tasks iterate once per frame, one can prove (and indeed we do) that **to_of**(b, i, 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, **task_result**(a, i) denotes the set of output values which are produced. In order to map task iterations to subframe time, the function **of**(i, a) is used to denote the time interval $[t_1, t_2]$ comprising the execution window of the ith iteration of a. The functions **beginning**(i, a) and **ending**(i, a) denote the beginning and ending of the execution window, respectively.

The overall structure of task configurations within the I/O Specification is illustrated in Fig. 7. For a task such that the predicate **task_safe**(a, i) is true, a will produce exactly one output value during its execution window. The output of a task which is not safe during its iteration is unspec-

[1]This use of an explicit apply to avoid the use of higher order logic was a consequence of the constraints of an earlier specification and verification system. EHDM provides higher order expressions, but the prior representation was not changed.

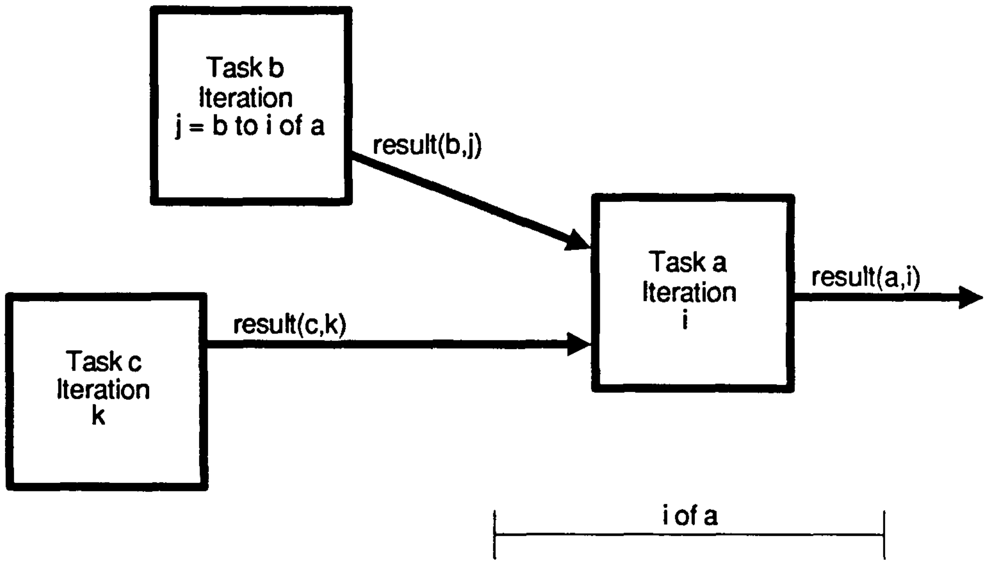

Figure 7: Three tasks in the IO Specification.

ified. Because the configuration of tasks is different for different phases of the flight, not all tasks necessarily compute each iteration. A predicate **on_during**(a, i) determines whether **task_result**(a, i) is expected to compute a function of its inputs or to return a special **bottom** 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 $ic(a)$, its associated function **the_function**(a) is defined to be the identity function. Recall from our discussion in Section IV that 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 requiring that safe tasks compute functions of their designated inputs. It is believed that these eight axioms are simple enough to be readily understood.

These axioms are related to the scheduling of task iterations and 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. These axioms are

```
io_a1_1:  axiom
          beginning(of(i,a))+1 < ending(of(i,a))

io_a1_2:  axiom
          ending(of(i,a)) ≤ beginning(of(i+1,a))

io_d1:  axiom
        member(b,inputs(a))
        implies beginning(of(i,a)) ≥ ending(of(to_of(b,i,a),b))
                and beginning(of(i,a)) < ending(of(to_of(b,i,a)+1,b))
```

The main axiom defining the Input/Output behavior of a task (together with its associated set constructor axiom) is the following:

```
io_a2:  axiom
        on_during(a,i)
        and task_safe(a,i)
        and (forall b: member(b,inputs(a))
                       implies card(task_result(b,to_of(b,i,a)))=1 )
        implies singleton(task_result(a,i),apply(the_function(a),set_va2(i,a)))

io_a2a:  axiom
         member(source(vt),inputs(a))
         and member(value(vt),task_result(source(vt),to_of(source(vt,i,a)))
         iff member(vt,set_va2(i,a))
```

The main axiom **io_a2** states that for any iteration i of a task a, such that a is both on and safe, if each task b providing input to the ith iteration of a returns exactly one output value during its corresponding iteration, then a will return exactly one output during its iteration (i.e., that **task_result**(a,i) will be a singleton set). The value produced will be the result of applying its designated function **the_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 **task_result**$(t$,to_of$(t,i,a))$. 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, two additional axioms govern their characteristics.

```
io_a3:  axiom
        ic(a) and ic_task_safe(a,i) implies card(task_result(a,i)) = 1

io_a4:  axiom
        ic(a)
        implies card(inputs(a)) = 1
                and member(b,inputs(a)
                    implies card(poll_for_of(to_of(b,i,a),b)) = 1
                and ( singleton(vinputs,vt) and member(source(vt),inputs(a))
                      implies apply(the_function(a),vinputs) = value(vt)
```

The second of these defines basic properties of an interactive consistency task, that it has only a single input, that the source of its input is an unreplicated task, and that its associated function is the identity function. Axiom **io_a3** requires that an interactive consistency task which is safe during its iteration will always produce a single value as output (i.e., the same value in every processor). By **io_a2**, if its input task is safe and thus provides a single output, the interactive consistency task will perform its associated function (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 criteria for the interactive consistency algorithm. We did not carry out a mechanical proof of the algorithms used for interactive consistency to satisfy this axiom—a hand proof can be found in [7].

One axiom is required to ensure that tasks that are not currently scheduled to execute nevertheless have a defined null result value.

```
io_a5:  axiom
        on_during(a,i) and task_safe(a,i)
        and member(b,inputs(a)) and not on_during(b,to_of(b,i,a))
        implies singleton(task_result(b,to_of(b,i,a)),bottom)
                and task_safe(b,to_of(b,i,a))
```

One last axiom expresses a constraint on the clock to ensure that it does not suffer from excessive skew or jitter. This is not the axiom that requires clock synchronization between processors for, at this level of abstraction, individual processors are not visible. It merely requires that the consensus clock, resulting from synchronization, should not deviate too far from real time.

```
io_a6:  axiom
        t2 > t1
        and (forall i: ending(of(i,clock)) ≤ t2
                       implies task_safe(clock,i) )
        implies (t2 - t1)*(1 - lambda)-epsilon < real_time(t2) - real_time(t1)
                and real_time(t2) - real_time(t1) < (t2 - t1)*(1+lambda)+epsilon
```

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.

12

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 here is the information flow communicating the error reports from individual processors 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_of**(i, a) denotes the set of processors assigned to compute the ith iteration of task a. The I/O Specification predicate **on_during**(a, i) is derived within the Replication Specification as

rp_d7: axiom
  on_during(a,i) iff card(poll_for_of(i,a)) $>$ 0

With the concept of processor computation occurring in the Replication Specification, the **task_safe** predicate, which appears as primitive within the I/O Specification, can be derived within the Replication Specification in terms of working processors. The Replication Specification includes a predicate **safe**(t_1) which denotes the set of "safe" processors at any given time, while **safe_for**(t_1, t_2) denotes the set of processors safe during the interval $[t_1, t_2]$. At the Activity Specification 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, the set safe 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 **dw_for_to_of**(b, i, a) is defined to be the time interval [**beginning**(**of**(**to_of**(b, i, a)),b),**ending**(**of**(i, a))]. Based on this function, we define **dw_for_of**(i, a) to be the interval extending from the beginning of the execution window of the earliest input task to a and extending to the ending of the execution of the ith iteration of task a (i.e., **of**(i, 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 Specification, as follows:

```
rp_d9a:  axiom
         task_safe(a,i)
         iff
         not on_during(a,i)
         or card(poll_for_of(i,a))
            < 2*card(union(poll_for_of(i,a),safe_for(dw_of(i,a)))
```

This definition states that a task a is safe if the task is not **on_during**(i) or if a majority of the processors assigned to compute the task are working for the data window of the task. It is necessary that the processors are in the set of safe processors 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 **on**(a, i, p), which denotes the set of outputs produced by processor p for the ith iteration of task a. In a manner left unspecified by this level, processor p communicates its results to all other system processors. The function **on_in**(a, i, p, q) denotes the value that processor q has reportedly received from processor p for the ith iteration of a. The relationship between **on** and **on_in** is defined by the following axiom.

```
rp_a2:  axiom
        member(p,union(poll_for_of(i,a),safe_for(dw_of(i,a))))
        implies (member(v,on(a,i,p))
                 iff (exists q: member(q,safe_for(of(i,a)))
                                and v = on_in(a,i,p,q)))
```

This defines that for a processor p in the poll set which is safe for the data window, the result set **on**(p) is equal to the set of values that processors safe for the execution window have reportedly received from p. 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 **in**(a, i, q) is used to define the result of processor q voting on the output of the ith iteration of a based on the results communicated to it.

The overall structure of the Replication Specification is illustrated in Fig. 8. The task structure shown is a refinement of the task configuration illustrated in Fig. 7.

As we shall show shortly, the I/O primitive **task_result**(a, i) for a safe

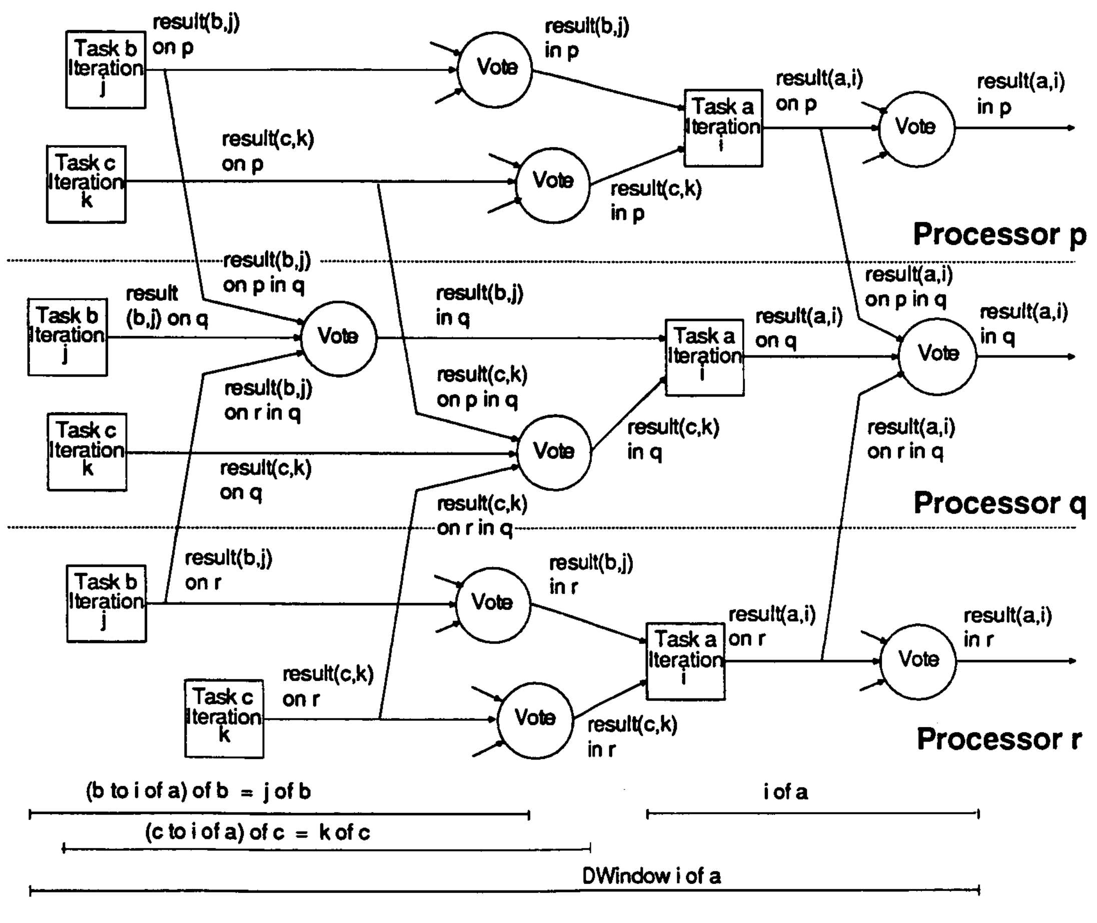

Figure 8: Three Tasks in the Replication Specification.

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 that produces a set of output values as in the I/O Specification, in the Replication Specification 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 the specification) on a word-by-word basis. We define voting via the following axioms.

```
rp_d4:  axiom
        member(q,safe_for(of(i,a)))
        and 1 ≤ y and y ≤ result_size(a)
        implies seq_elem(in(a,i,q),y) = majority(set_d4a(a,i,q,y)

rp_d4a:  axiom
         member(dp,set_d4a(a,i,q,y))
         iff (exists p: seq_elem(on_in(a,i,p,q),y) = value(dp)
                        and p = source(dp)
                        and member(p,poll_for_of(i,a))
```

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

The main execution axiom of the Replication Specification is now given.

```
rp_a3:  axiom
        member(p,union(poll_for_of(a,k),safe_for(dw_of(i,a))))
        implies singleton(on(a,i,p),apply(the_function(a),set_va3(a,i,p)))

rp_a3a: axiom
        member(vt,set_va3(a,i,p))
        iff member(source(vt),inputs(a))
           and value(vt) = in(source(vt),to_of(source(vt),i,a),p)
```

Axiom **rp_d6**, quite similar to its counterpart in the I/O Specification, defines that a working processor p, which is in the poll set for the ith 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 **task_result**(a, i). This is given by the following axiom.

```
rp_d6:  axiom
        member(v,task_result(a,i))
        iff (exists p: member(p,safe_for(of(i,a)))
                       and v = in(a,i,p))
```

This defines the set **task_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 (io_a2) is in outline:

> Assuming the antecedent of the I/O Execute Axiom (io_a2), that the task is safe and that there is only one value of the result of each input task, each safe processor voting is shown to obtain the same voted value to use as input to the computation for that task.
>
> By Axiom **rp_a3**, this implies that each safe processor applies the appropriate mathematical function to the same set of input values, and thus every safe processor produces the same correct output value.
>
> But our assumption that the task is safe asserts that a majority of the processors computing the task are safe. It follows that the majority of computed values must be the correct value.

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

13

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 specification of the activities and data structures which carry out the transformations specified at the Replication level. Whereas the Replication level defines the executed and voted value for each execution window of a task, the Activity level defines a schedule of execute and vote activities to compute these values within the execution window.

Within the Activity Specification 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, $\mathtt{start}(t,p)$ and $\mathtt{finish}(t,p)$ map subframe time on processor p to real times at which the subframe starts and finishes, as shown in Fig. 9. "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

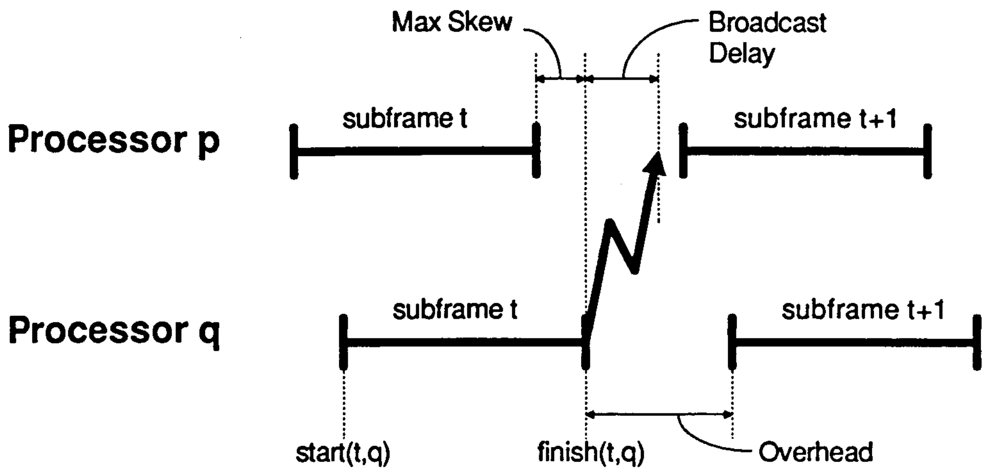

Figure 9: The Timing Relationships between Subframes on Asynchronous Processes.

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 [11]. The required synchronization is expressed by the following.

within_skew: axiom
clock_safe(p,t) and clock_safe(q,t)
implies finish(t,p) + broadcast_delay $\leq$ start(t+1,q)
and finish(t,q) + broadcast_delay $\leq$ start(t+1,p)

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 this data is attempted before that time, in order to map this asynchronous system up to the higher level, synchronous Replication and I/O Specifications.

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 **datafilein_for_on**(p, a, q, 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 **inputin_of**(p, a, 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, obtained by that processor itself voting the results of the replicated global executive task. At the start of a subframetime t, processor q obtains **config**(t, q) from the configuration subfield of **inputin_of**(q,global_exec,**start**(t, q)). 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. Fig. 10 illustrates the interaction between the data structures and scheduled activities.

The effect of an execute activity is specified by the following axiom.

```
br_a41: axiom
        working_during(p,t) and working_during(q,t)
        and member(activ, sched(config(t,q),t,q))
        and action(activ) = execute and task(activ) = a
        and (forall vt: member(vt,vinputs)
              iff member(source(vt,inputs(a)))
                 and value(vt) = inputin_of(q,source(vt),start(t+1,q))
        implies datafilein_for_on(p,a,q,finish(t,q)+broadcast_delay)
                 = apply(the_function(a),vinputs)
```

The set **working**(t) denotes the set of correctly functioning processors

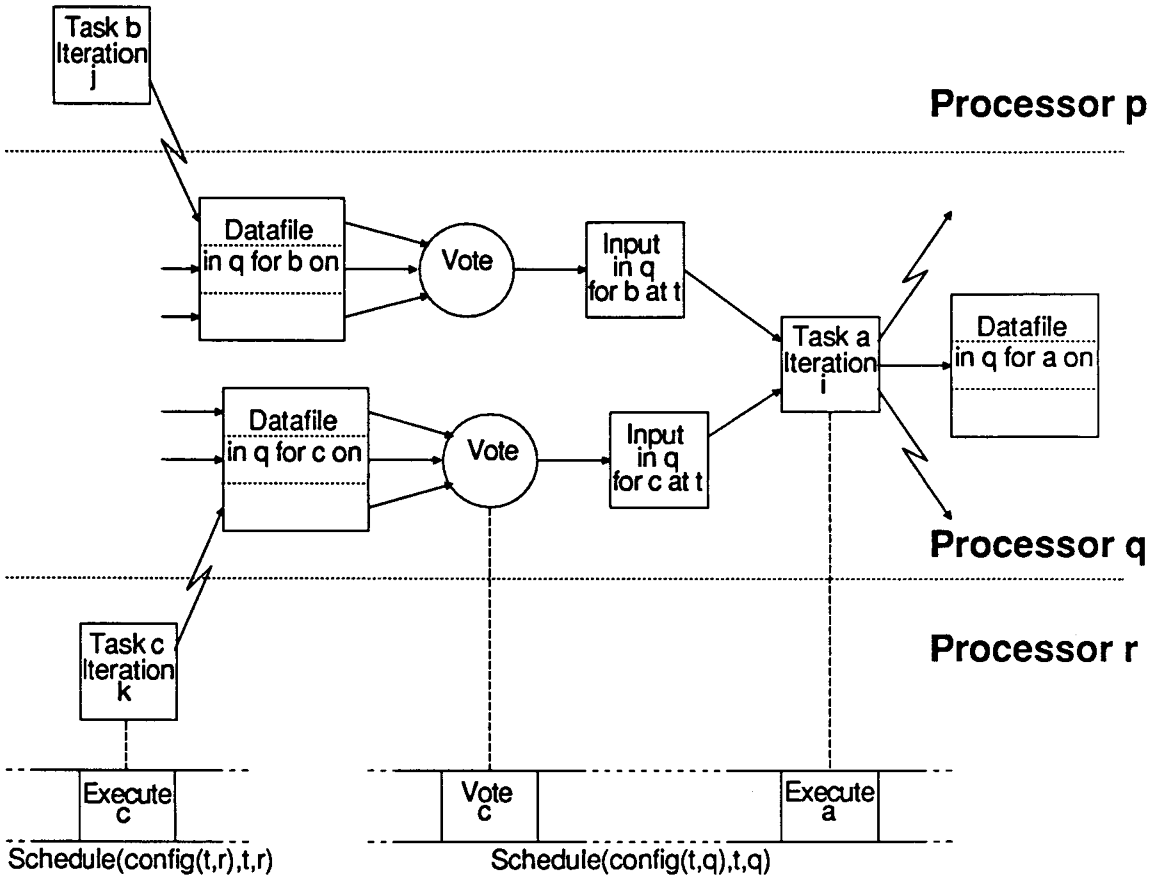

Figure 10: A Partial View of Three Tasks in the Activity Specification.

during subframe t, and the predicate working_during(p, t) is true if processor p is a member of that set. 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 necessary to demonstrate that, with respect to 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 its 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 axioms defining a vote activity scheduled for the subframe are the following:

```
br_a9c:  axiom
         working_during(p,t)
         and member(activ,sched(config(t,p),t,p))
         and action(activ) = vote and task(activ) = a and elem(activ) = y
         implies elem(inputin_of(p,a,start(t+1,p)),y)
                 = majority(set_a9c(p,a,t,y))

set_abstraction_a9c:  axiom
                      member(dp,set_a9c(p,a,t,y))
                      iff member(source(dp),pollby_for(p,a,t))
                      and value(dp)
                          = elem(datafilein_for_on(p,a,source(dp),start(t,p)), y)
```

Given a working processor p scheduled to perform a vote on the yth 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 **pollby_for**(p, a, t) denotes the set of processors determined by p at the time of the vote to have executed the last iteration of task a.

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 specification and constraining the composition of the schedule table.

In terms of the functions of the Activity Specification, we can now define the mappings to the function symbols of the Replication Specification. The function **on_in** of the Replication level is derived with the following axiom.

```
br_re_mapping_6:   axiom
        v = on_in(a,i,p,q)
        iff (forall t,activ,y: beginning(of(i,a))≤t≤ending(of(i,a))
                               and 1≤y≤result_size(a)
                               and member(activ,sched(config(t,q),t,q)
                               and action(activ) = vote
                               and task(action) = a
                               and elem(action) = y
                               implies elem(v,y) = datafilein_for_on(q,a,p,start(t,q)), y))
```

Briefly, the mapping axiom defines each component y of the **on_in** result 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 during a subframe corresponding to the ith 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.

In an analogous manner, the mapping of the Replication Specification's in voted value is defined by the following axiom.

```
br_re_mapping_7:   axiom
        (exists t,activ: start(frame(t)) = i*frame_size
                         and 1≤y≤result_size(a)
                         and member(activ,sched(config(t,p),t,p))
                         and action(activ) = vote
                         and task(activ) = a
                         and elem(activ) = y
                         and d1 = elem(inputin_of(p,k,start(t+1,p)), y))
        implies elem(in(a,i,p),y) = d1
```

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

```
br_re_mapping_4:   axiom
        member(q,poll_for_of(i,a))
        iff (exists p,t,activ,y: start(frame(t)) = i*frame_size
                                 and 1≤y≤result_size(a)
                                 and member(p,safe_for(of(i,a)))
                                 and member(activ,sched(config(t,p),t,p))
                                 and action(activ) = vote
                                 and task(action) = a
                                 and elem(action) = y
                                 and member(q,pollby_for(p,a,t))
```

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

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

```
br_re_mapping_9x:   axiom
        member(p,safe(t))
        iff member(p,working(t))
           and clock_safe(p,t)
           and task_safe(global_exec,last(t,global_exec))
           and (forall pp: ic_task_safe(ic_error_reporter(pp),
                                        last(t,global_exec)))
           and (exists q: safe_for_ending(last(t,global_exec),
                                          ending(of(last(t,global_exec),
                                                    global_exec)),
                                          q,global_exec)
                 and member(p,configset(config(ending(of(last(t,global_exec),
                                                          global_exec)),
                                               q)))
                 and config(t,p) = config(ending(of(last(t,global_exec),
                                                    global_exec)),
                                          q)))
```

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 that

- The global executive task, and the interactive consistency tasks that supply error reports to it, have had sufficient replication to remain safe.
- Clock safety be recovered despite any transients affecting the clock in the past.

- There was a processor q that
  - was safe for the last iteration of the global executive
  - included p in its configuration
  - had the same opinion of the current configuration as p.

The proof of the relationship between the Replication Specification and the Broadcast Specification was quite challenging. The proof involved showing that

> The vote and execute activities, replicated on different processors and running during different subframes within the frame, use the same information for input.
>
> The various processors, operating independently and asynchronously with only local information, communicate with each other without mutual interference, the task schedules guaranteeing that results are always available in other processors when required, and are never accessed at any time when they might be modified by a broadcast.
>
> The distributed system has, as a valid abstraction, the synchronous, global characterization expressed in the Replication Specification.
>
> The axioms and schedule constraints imply consistency of configuration and schedule within a single processor and between processors during an execution window.
>
> If the initial configuration has a majority of processors that are safe and have identical opinions as to the configuration, then processors that remain safe remain in the configuration and all continue to have identical configurations.

14

PrePost and Imperative Levels

The Activity level represents a specification for each processor in the distributed, multiprocessor system; the PrePost level, very similar in abstraction to the Activity level, defines the behavior of a single, independent processor. The specification employs the data structure abstractions present in the actual Pascal operating system implementation and is intended to facilitate a connection 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 [12] was employed to prove precondition/postcondition procedure characterizations from the Pascal procedures, each treated as a sequential program. We explain only enough of the PrePost level and its specification for the reader to glean an overall understanding of this level of the specification and proof.

Within the PrePost Specification, the state of a processor is specified as a pair $< p, t >$, where p is a processor id and t is a subframe time. The accessor functions **proc** and **time** map states into processor component and time components, respectively. For a state pair $< p, t >$, the function **next**$(< p,t >) = < p, t+1 >$. Within the PrePost Specification, 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 identifier and task name into the array of integer values currently in the datafile. The input file is declared to be of type **array[task,integer] of integer**, mapping 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.

```
execute_activity:  axiom
working_during(proc(siftstate),subframe(siftstate))
and (exists j: 1 ≤ j ≤ max_activities
      and activity(index(index(index(index(sched_table,
                                                  index(real_to_virt(siftstate),
                                                        proc(siftstate))),
                                            config(siftstate)),
                                      subframe(siftstate)),
                                j))
          = execute
      and taskname(index(index(index(index(sched_table,
                                                   index(real_to_virt(siftstate),
                                                         proc(siftstate))),
                                             config(siftstate)),
                                       subframe(siftstate)),
                                 j))
          = a
      and (forall b,j1,y: 1 ≤ y ≤ index(result_size,t1)
                                implies index(index(inputs,a),j1) = b
                                       and not (b = null_task)
                                           implies index(index(inp,j1),y)
                                                  = index(index(input(next(siftstate)),
                                                                           b),
                                                             y)
implies index(index(datafile(next(make_state(subframe(siftstate),
                                                         proc(siftstate)))),
                   proc(siftstate)),
             a)
        = task_results(a,inp)
```

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 $\mathbf{next}(siftstate)$, for each input task b indicated by the array **inputs**. The array **real_to_virt**, shown here as an explicit function of the state, maps a real processor identifier into a logical processor identifier, 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 $\mathbf{next}(siftstate)$ to contain the results of applying the appropriate 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 $\mathbf{next}(siftstate)$ 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 from an explicit-time, explicit-state specification to a Hoare sentence specification 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 of 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 noninterference proof at the Activity Specification level. Following this transformation, the execution axiom above becomes the following Hoare sentence

```
execute_activity:  axiom
       (exists j: 1 ≤ j ≤ max_activities
        and activity(index(index(index(index(sched_table,
                                                    real_to_virt(myproc)),
                                             config),
                                      subframe),
                              j))
             = execute
        and taskname(index(index(index(index(sched_table,
                                                     real_to_virt(myproc)),
                                              config),
                                       subframe),
                               j))
             = a
index(index(index(index(sched_table,
            and (forall b,j1,y: 1 ≤ y ≤ index(result_size,b)
                              implies index(index(inputs,a),j1) = b
                                      and not (b = null_task)
                                          implies index(index(inp,j1),y)
                                                  = index(index(input,b),y)

       {dispatcher}

       index(index(datafile,myproc),a) = task_results(a,inp)
```

15

Conclusions and Further Work

Our proof has demonstrated that

- Formal specification techniques can be used to state very abstract requirements such as the SIFT reliability requirement and the transition characteristics of the Markov reliability model.
- These formal requirements can be specified sufficiently succinctly that they can be validated by human inspection.
- It is possible to use hierarchical verification techniques to demonstrate with rigor that a detailed design satisfies its requirements.

The soundness of the axiomatic specifications must be demonstrated by the existence of an imperative model (the Pascal implementation) at the lowest level of the hierarchy, relative to interpretations for all unimplemented function and predicate symbols (such as **working**, the set of working processors). It is also necessary to assume the correct implementation of the Pascal machine, realized by the Pascal compiler and the Bendix BDX930 hardware.

15. CONCLUSIONS AND FURTHER WORK

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 [11]). The formal proof revealed that tasks not scheduled to execute did not regenerate their default result value during 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 construction of these proofs demonstrated that the current form of the Enhanced HDM is capable of design proofs of considerable complexity, but it also demonstrated that such proofs can, at present, only be constructed as a *tour de force.* The Revised Special specification language was quite satisfactory and was capable of expressing everything that was required, even at the very abstract levels of the IO and Transition Specifications.

This project was performed using the first working version of the Enhanced HDM System and encountered problems caused by the limitations of that first version. Later versions of Enhanced HDM, running on Symbolics and Sun workstations, show better time and space performance, and improved user interaction and theorem proving capability is being developed.

The verification system was rather slow for proofs as large as these. The form of man-machine interaction envisaged for Enhanced HDM requires rapid response from the computer to maintain the concentration of the human user, preferably within a minute. But the parse-typecheck-proof cycle for many of the proofs was closer to an hour than a minute, radically affecting the useability of the system. The Foonly F4 is a relatively slow computer and substantially faster workstations are now available. It is our expectation that personal workstations of an appropriate performance will become available within the next decade.

The system, running on a Foonly F4 computer, similar to a DEC System 20, was unable to accomodate a full specification in store at once. Consequently it was necessary to partition the specifications into modules that reflect the storage requirements of the system rather than the logical structure of the proof. Portions of the specification had to be replicated, and it was not possible to construct the dependency tree that demonstrates that all proofs have been performed. Subsequent versions of the Enhanced HDM system, running on Symbolics and Sun workstations, do not suffer from this limitation.

The existing proof construction and debugging aids of Enhanced HDM proved to be insufficient for complex proofs such as these. Enhanced HDM requires the user to provide all substitutions into the axioms and lemmas he or she cites to construct a proof. Only a few of these substitutions are subtle and require human insight to discover, insight beyond the capabilities of a computer. Many of the substitutions are obvious, but the number of substitutions required in a large proof tends to overwhelm the human user, and the proof aids provide only partial assistance in locating missing substitutions. Current development work on the Enhanced HDM system will generate almost all of the simple substitutions, greatly reducing the burden on the user. This development work will also greatly improve the effectiveness of the proof development aids. When these developments are available, large proofs, such as these, will be more feasible.

The construction of the proofs demonstrate that design proof is indeed possible, and perhaps even useful in very critical applications. However, we must conclude that continuing further development of the mechanical tools is necessary before large design proofs, such as that for SIFT, can be undertaken as a standard validation procedure. We also conclude that the current state of the resulting proof for SIFT, modularized for space conservation, is not a good model for future efforts unless completely remodularized.

Acknowledgements.

The design, specification, and verification of the SIFT project have involved many members of the Computer Science Laboratory, SRI International, past and present. J. Wensley led the original design effort for SIFT and conceived the basic architecture. Also involved in the design were J. Goldberg, K. Levitt, L. Lamport, R. Shostak, M. Pease, M. Green, W. Kautz, and C. Weinstock. L. Lamport collaborated on the early "paper formulations" of the SIFT specification hierarchy. R. Boyer, J. Moore, and M. Green contributed to the design proof effort; D. Hare and K. Levitt undertook a parallel code-level verification. J. Crow, J. Rushby, R. Shostak, and F. von Henke participated in the development of the Enhanced HDM specification and verification system. We particularly acknowledge the support and patience of our sponsors at NASA, N. Murray and R. Butler.

Bibliography

[1] J. Wensley *et al.*, "SIFT: Design and analysis of a fault-tolerant computer for aircraft control," *Proc. IEEE,* vol. 60, pp. 1240-1254, Oct. 1978.

[2] J. Goldberg, "SIFT: A provable fault-tolerant computer for aircraft flight control," *Proc. IFIP Congress 1980,* pp. 151-156.

[3] C. Weinstock, "SIFT: System design and implementation," *10th International Symposium on Fault-Tolerant Computing,* Oct. 1980.

[4] J. Goldberg, "Development and evaluation of a software-implemented fault-tolerant computer: SIFT hardware," Tech Report, SRI International, Nov. 1979.

[5] J. Goldberg, "Development and analysis of the software implemented fault-tolerance (SIFT) computer," Tech Report, SRI International, Sept./ 1982.

[6] K. Moses, "SIFT—An ultra reliable avionic computing system," *Proc. NATO AGARD Conference on Tactical Airborne Distributed Computing and Networks,* June 1981.

[7] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *JACM,* vol. 27, pp. 228-234, Apr. 1980.

[8] A. Hopkins, T. B. Smith and J. Lala, "FTMP—A highly reliable fault-tolerant multiprocessor for aircraft," *Proc. IEEE,* vol. 66, pp. 1221-1239, Oct. 1978.

[9] R. Shostak, R. Schwartz, and P. M. Melliar-Smith, "STP: A mechanized logic for specification and verification," *Proc. 6th Conference on Automated Deduction,* 1982, pp. 32-49.

[10] R. Shostak, "Deciding Combinations of Theories," *Proc. 6th Conference on Automated Deduction,* 1982, pp. 209-222.

[11] L. Lamport and P. M. Melliar-Smith, "Synchronizing clocks in the presence of faults," *JACM,* vol. 32, p. 1, Jan. 1985.

[12] S. Igarashi, R. London, and D. Luckham, "Automatic program verification: A logical basis and its implementation," *Acta Informatica,* vol. 4, pp. 45-182, 1975.

[13] J. Crow *et al.,* "SRI Specification and Verification System: User's Guide," Tech Report, SRI International, Apr. 1986.

[14] J. Crow *et al.,* "SRI Specification and Verification System: Preliminary Definition of the Revised Special Specification Language," Tech Report, SRI International, May 1986.

Standard Bibliographic Page

1. Report No. NASA CR-4097	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle Design Verification of SIFT		5. Report Date September 1987
		6. Performing Organization Code
7. Author(s) Louise Moser, Michael Melliar-Smith, Richard Schwartz		8. Performing Organization Report No.
9. Performing Organization Name and Address SRI International, Computer Science Laboratory 333 Ravenswood Avenue Menlo Park, CA 94025		10. Work Unit No. 505-66-21-01
		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 Contractor Report
		14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor: Ricky W. Butler
Final Report

16. Abstract

A SIFT reliable aircraft control computer system, designed to meet the ultrahigh reliability required for safety critical flight control applications by use of processor replication and voting, was constructed by the Bendix Corporation for SRI, and was delivered to NASA Langley for evaluation in the AIRLAB. To increase our confidence in the reliability projections for SIFT, produced by a Markov reliability model, SRI constructed a formal specification for SIFT, defining the meaning of reliability in the context of flight control. A further series of specifications defined, in increasing detail, the design of SIFT down to pre and post conditions on Pascal code procedures. Mechanically checked mathematical proofs were constructed to demonstrate that the more detailed design specifications for SIFT do indeed imply the formal reliability requirement. An additional specification defined some of the assumptions made about SIFT by the Markov model, and further proofs were constructed to show that these assumptions, as expressed by that specification, did indeed follow from the more detailed design specifications for SIFT. This report provides an outline of the methodology used for this hierarchical specification and proof, and describes the various specifications and proofs performed.

17. Key Words (Suggested by Authors(s)) Reliable computers Fault tolerance Reconfiguration Specification Verification	18. Distribution Statement Unclassified - Unlimited Subject Category 62

19. Security Classif.(of this report) Unclassified	20. Security Classif.(of this page) Unclassified	21. No. of Pages 84	22. Price A05