2

RADC-TR-89-376, Vol I (of two)
Final Technical Report
February 1990

SPECIFICATION/VERIFICATION OF TEMPORAL PROPERTIES FOR DISTRIBUTED SYSTEMS: ISSUES AND APPROACHES

Odyssey Research Associates, Inc.

Edward A. Schneider, D.G. Weber, Tanja de Groot

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Services (NTIS) At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-89-376, Vol I (of two) has been reviewed and is approved for publication.

APPROVED: Emilie J. Siarkiewicz

EMILIE J. SIARKIEWICZ
Project Engineer

APPROVED: Raymond P. Urtz Jr

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: Igor G. Plonisch

IGOR G. PLONISCH
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-89-376,.Vol I (of two)

6a. NAME OF PERFORMING ORGANIZATION Odyssey Research Associates, Inc.	6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COTD)
6c. ADDRESS *(City, State, and ZIP Code)* 301A Harris B. Dates Drive Ithaca NY 14850-1313		7b. ADDRESS *(City, State, and ZIP Code)* Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center	8b. OFFICE SYMBOL *(If applicable)* COTD	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-87-D-0092, Task 0005

8c. ADDRESS *(City, State, and ZIP Code)*
Griffiss AFB NY 13441-5700

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.
35167G	1069	QB	05

11. TITLE *(Include Security Classification)*
SPECIFICATION/VERIFICATION OF TEMPORAL PROPERTIES FOR DISTRIBUTED SYSTEMS: ISSUES AND APPROACHES

12. PERSONAL AUTHOR(S)
Edward A. Schneider, D. G. Weber, Tanja De Groot

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Final	FROM Feb 89 TO Jun 89	February 1990	110

16. SUPPLEMENTARY NOTATION
N/A

17. COSATI CODES

FIELD	GROUP	SUB-GROUP
12	05	

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*

Computer Security	Temporal Properties
Distributed System	Fault Tolerance
Specification/Verification	Adaptive Policies

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This report identifies problems, models and solutions in the area of specification and verification of temporal properties for secure distributed systems. The temporal properties studied are security, progress, determinism, and real-time requirements. Also included is work on the specification of fault tolerance and adaptive security policies. This effort is a "first look" at these issues.

In a companion volume, The Gypsy Verification Environment is evaluated with regard to its ability to handle the temporal properties discussed here.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Emilie J. Siarkiewicz	22b. TELEPHONE *(Include Area Code)* (315) 330-2158	22c. OFFICE SYMBOL RADC (COTD)

Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Contents

Chapter 0

Introduction

This study, prepared by Odyssey Research Associates, covers tasks one, three, and seven of a project by Sytek, Inc., ORA, RCA, and Computer Corporation of America that was originally intended to investigate, design, develop, and demonstrate tools to aid in the specification and verification of the security and functionality of distributed systems. It identifies problems and develops models and solutions in the area of specification of temporal properties of secure distributed systems. Included is work on the specification of fault tolerance and on adaptive security policies.

The temporal properties that have been studied for this report are

security. It should be possible to prove that a system being built using such tools is secure. Also, it must be possible for the security policies used to adapt to changing conditions over the lifecycle of a system.

progress. Properties to be proved about a system that are of the form "nothing bad can happen" are referred to as *safety* properties. However, a system that does nothing meets such properties. It is therefore also necessary to have *liveness* properties stating that eventually progress will be made and something good will happen.

determinism. Nondeterminism occurs in distributed systems due to the unpredictability of process speeds relative to each other. Processes are synchronized to control the nondeterminism.

real-time requirements. The distributed systems to be built with the tools will frequently have timing constraints that affect the scheduling of the processors and the allocation of other resources.

fault-tolerance requirements. Making a system fault-tolerant requires replication in either space or time. The tools must therefore allow for the specification of how the system parts should be replicated and agreement reached and of when and how restart should be applied. The tools must also provide for the implementation of these requirements.

Specifying and verifying probabilistic algorithms [40] has not been studied.

0.1 Distributed System Model

The model of a distributed system that will be used in this report is a collection of objects that interact only by passing messages. The only state global to two or more objects is the communication channels that connect them. Every piece of data must be contained in one of the objects and cannot be directly read from another. This model fits very well with a multicomputer, in which there is no memory shared between the nodes, and the objects can be distributed arbitrarily among the nodes. However, it can also be used when objects share a node and some global memory. (KR)

The activity of an object, known as a process, is a sequence of actions consisting of a mixture of communications and internal computing. From outside its object, the only behavior of a process that can be observed is the sequence of input and output communications that it participates in. Internal actions can only be inferred when a change in behavior occurs.

There are two different styles of communication in distributed systems: synchronous and asynchronous. In synchronous communication, the sender of a message is delayed until the message is received. Thus, a channel between two processes has at most one message in it and no additional buffering is required. Also, when the sender proceeds to its next action, it knows that the recipient has received the message. In asynchronous communication, the sender does not wait for the receiver. The channel between processes buffers the messages that have not been received. Characteristics of such a channel include the amount of buffering provided and the order in which waiting messages are delivered.

Synchronous communication will be assumed in this report. Synchronous communication is used by CSP[23], CCS[35], and Ada. Asynchronous communication can be easily simulated by explicitly creating buffer objects that synchronously accept messages from the sender or deliver messages to the receiver, but which allow the sender to proceed independently of the receiver. Decisions such as buffer size and the order in which messages are delivered are then part of the buffer, rather than part of the model. Also, the collection of undelivered messages is represented in the state of the buffer.

A *process* will be described as a set of traces, where each trace is a possible behavior of the process as observed over a finite period of time. Thus, a *trace* is a finite sequence of input and output actions. A trace may also be thought of as determining a *state* representing the future behaviors that the process can exhibit. Initially, no actions have occurred so the empty sequence is a legal trace. If $\langle a, b, c\rangle$ is a valid trace, then $\langle a\rangle$ and $\langle a, b\rangle$ would have been observed at earlier times and must therefore be legal (traces are *prefix closed*) [23]. Formally, a process is an *Event System* consisting of:

- $\mathbf{E}$, the set of events or actions, which is partitioned into $\mathbf{I}$ (the set of inputs) and $\mathbf{O}$ (the set of outputs).
- $\mathbf{T} \subseteq \mathbf{E}^*$, the set of finite traces such that
  1. $\langle\rangle \in \mathbf{T}$
  2. $s\hat{\ }t \in \mathbf{T} \Rightarrow s \in \mathbf{T}$

Nondeterministic behavior can be explained either by adding new internal events not contained in I or O [35] or by using failure semantics [7,9].

The behavior of a system composed of one or more processes is also represented as a trace. The system trace is an interleaving of its constituent process traces, in which the synchronous communication between two of the processes is treated as a single event. An event occurs in the system trace iff it occurs in at least one of the process traces. The order of two events in a process trace is maintained in the system trace.

0.2 Remainder of the Report

The next chapter will discuss the temporal properties in the areas of security, adaptive policies, liveness (progress), determinism, real-time requirements, and fault-tolerance requirements. The following chapter then presents several formalisms that can be used to specify these temporal properties. These include temporal logic, a state-transition model, and timed CSP. Throughout these two chapters, the concepts will be illustrated using a live queue example and a file server example. The final chapter presents techniques of formal verification of fault tolerance that are one means for gaining greater assurance of the correctness of software. Precise specifications corresponding to the intuitive notions of "fault tolerance" and of "graceful degradation" are formulated. An analogy is constructed between these fault-tolerance specifications and a particular class of specifications for computer security. On the basis of this analogy, it is argued that formal verification of fault-tolerance will face some of the same problems, and benefit from some of the same solutions, as verification of security. Verification tools designed for one domain may be applicable in the other. The notation used for sequences and the CSP language used for specifications is given in the appendix.

In a companion volume, the Gypsy Verification Environment is evaluated with regard to its ability to handle the temporal properties discussed here.

0.2.1 Live Queue

Consider the queuer process defined in figure 0.1, which acts like a first-in, first-out buffer for one or more producer processes and one or more consumer processes. The queuer has two operations, enq and deq, which respectively enqueue and dequeue an item. If the queue is empty, the next operation is necessarily enq. The queuer can hold only finitely many items; if the queue is full, the next operation is necessarily deq.

The operations in figure 0.1 are represented as CSP channels; enq is an input channel and deq is an output channel. The events are enq?x (assign to variable x the next value from enq) and deq!x (output the value of x on deq). The first subscript of Q is the number of items in the queue and is easily shown to be the number of items input (enqueued) minus the number of items output (dequeued). The second subscript is a sequence representing the items stored, in order of arrival. The maximum number of items that can be stored is given by max, which is required to be greater than zero. The notation $\langle y\rangle{}^\wedge q{}^\wedge\langle x\rangle$ represents the sequence with first element y, last element x, and the (possibly empty) sequence q in the middle. The empty sequence is represented as $\langle\rangle$. Process e $\rightarrow$ P first participates in event e and then behaves like process P. Process P $\square$ Q behaves either like process P or like process Q. Thus, $Q_{n,q^\wedge\langle x\rangle}$ either inputs y from channel enq and then behaves like $Q_{n+1,\langle y\rangle^\wedge q^\wedge\langle x\rangle}$ or outputs x to channel deq and then behaves like $Q_{n-1,q}$.

The producer outputs an item to the queuer after doing some number of internal actions. Similarly the consumer inputs an item after doing some number of its own internal actions. The external behavior of these processes is defined by:

$$\begin{array}{rcll} \text{producer} & = & \mu p.\ \text{enq!x} \rightarrow p & \qquad \text{for varying x} \\ \text{consumer} & = & \mu p.\ \text{deq?x} \rightarrow p & \end{array}$$

A system composed of one producer, one consumer, and a queuer executing concurrently is:

$$\begin{array}{rcll} \text{queuer} & = & Q_{0,\langle\rangle} & \\ Q_{0,\langle\rangle} & = & \text{enq?x} \rightarrow Q_{1,\langle x\rangle} & \\ Q_{n,q^\wedge\langle x\rangle} & = & \quad\text{enq?y} \rightarrow Q_{n+1,\langle y\rangle^\wedge q^\wedge\langle x\rangle} & \\ & & \square\ \text{deq!x} \rightarrow Q_{n-1,q} & \qquad \text{for } 0 < n < \max \\ Q_{\max,q^\wedge\langle x\rangle} & = & \text{deq!x} \rightarrow Q_{\max-1,q} & \end{array}$$

Figure 0.1: queuer process

$$\mathrm{system} = (\mathrm{producer} \parallel \mathrm{queuer} \parallel \mathrm{consumer}) \setminus \{\mathrm{enq.x}, \mathrm{deq.x}\}$$

The traces for queuer are sequences of enq and deq events in which the difference between the number of enq and the number of deq events is between 0 and max and the sequence of values dequeued is an initial subsequence of the values enqueued. The only trace of system is the empty sequence, since all communications are internal and have been hidden.

This simple example, called the LIVEQ, provides a backdrop for discussing properties related to progress. For instance, by modifying the behavior of the constituent processes, the system may become subject to deadlock. Chapter 1 uses this example and modifications to illustrate progress properties. Later sections in this document describe other ways of specifying this example and related properties.

0.2.2 File Server

Consider the file system defined in figure 0.2, in which files may be created, read, and deleted. In order to avoid dealing with the problem of handling name conflicts (trying to create a file with a name already in use), the create operation returns a unique file name. The subscripts to FS represent the state of the file system: n is the number of files that have been created and f(n) is either the value of the file named by n or else error. The notation f[n : s] represents the function that is the same as f, except that the value at n is s. Note that $FS_{n,f}$ is defined recursively, with p representing the recursive call.

The file system is an object that receives inputs from channel in (input event in.x is abbreviated ?x) and sends responses over a channel c passed as a parameter in the request. The input events recognized by the file system are ?(create, s, c), which causes both a value to be output and a state change, ?(read, n, c), which causes a value to be output but leaves the state invariant, and ?(delete, n), which only causes the state to change. The output events are the responses returned to the sender of a create or a read. An error message is returned in response to a read if the file has not yet been created or if it has been deleted.

$$\begin{aligned}
\mathrm{filesystem} &= \mathrm{FS}_{0,f0} \text{ where } \forall n \in \mathrm{N}.\ f0(n) = \mathrm{error} \\
\mathrm{FS}_{n,f} &= \mu p.\ ?(\mathrm{create}, s, c) \rightarrow ((c!n \rightarrow \mathrm{SKIP}) \parallel \mathrm{FS}_{n+1,f[n:s]}) \\
&\qquad \Box\ ?(\mathrm{read}, n', c) \rightarrow ((c!f(n') \rightarrow \mathrm{SKIP}) \parallel p) \\
&\qquad \Box\ ?(\mathrm{delete}, n') \rightarrow \mathrm{FS}_{n,f[n':\mathrm{error}]}
\end{aligned}$$

Figure 0.2: File System

A new process is created to handle each output by splitting the file system object. This allows concurrency in handling requests and can be implemented by either sharing a single processor or by using several processors to respond to requests. Initiation of the processes occurs in the order in which requests arrive, but the completion time depends ou the amount of time required to output the value (this might depend on the length of the file for a read) and the amount of processing time provided by the underlying scheduler.

A trace of the file system is a sequence of input and output events such that every output event is preceded by a create or a read with which it can be paired. Thus, there are at least as many create and read events as there are output events. The value contained in an output event is uniquely determined by the corresponding input event and the sequence of input events in the trace prior to the corresponding input event.

Chapter 1

Temporal Properties

1.1 Security

Security for a computer system takes many forms. Physical security prevents damage to or theft of computer equipment. Encryption can be used to prevent the theft of information through wiretaps. User identifiers and passwords prevent unauthorized users from entering the system. Protection helps to maintain data integrity by controlling the way in which objects in the system are manipulated. While all of these topics are important, formal verification of them is not yet well understood. Therefore, this section will only deal with information confidentiality – preventing users of the system from receiving information that they are not authorized to receive. The **Progress** section deals with the related issue of assured service.

In order to be able to talk about the security of a computer system, there must be a precise description of the system, a method for controlling the flow of information in the system (the *security policy*), and a definition of what it means for a system to be secure. A computer system in this report is modeled as an *object* that interacts with its environment by sending and receiving "messages". Examples of objects include network services and individual processes, for instance the process which controls the operation of a file system. A shared memory is an object that receives writes and read requests and sends responses to the read requests. An object may be constructed from a collection of independent subobjects which communicate with each other by letting output messages from one component become inputs to another. Examples include a file system connected to a buffer of pending commands and two local networks joined together by a bridge. In this way, a large system can be decomposed into smaller, more manageable parts. Objects will be specified using CSP.

One form of security policy is *discretionary access control*, in which either each object in the system has an access control list, specifying by whom and how it may be used, or each user has a list of capabilities, specifying which objects it may access and how it may access them. Discretionary access control is especially useful for systems in which access to a particular piece of information is authorized for only a few users on a "need to know" basis. With this method, the security policy is adapted to new conditions by changing the access control lists or the capability lists stored in it. (These lists are themselves objects that must be protected.)

Another form of security policy is *mandatory access control* or *multilevel security*, in which each user and each object is assigned a value from a set L of *security levels* that is partially ordered by a *dominates* relation $\succeq$. A user's access to an object depends on how the corresponding levels relate to each other under this relation. For example, a system containing accounting information and information on weapon systems might have four security levels: N (neither), A (accounting), W (weapon systems), and B (both). Users at levels A and B can access accounting information and users at levels W and B can access weapon systems information. Thus, a user at level N can access fewer things than any other level user (N is dominated by all levels), while a user at level B can access more things than any other level user (B dominates every level). However, there are some things that a user at level A can access that a user at level W cannot and there are other things that a user at level W can access that a user at level A cannot, so neither A nor W dominates the other. A formal definition of security in a multilevel security model in terms of the flow of information will be given in section 1.1.1. Mandatory access control is especially useful for systems in which a new user or information is frequently added and for which access authorization does not need to be established on a case-by-case basis. Only mandatory control will be considered in the remainder of this report.

The Event System model is extended to a *Rated Event System (RES)* [59]

$$(\mathbf{E}, \mathbf{T}, \mathbf{I}, \mathbf{O}, \mathbf{L}, \mathbf{lvl})$$

consisting of a set E of events with disjoint subsets I (inputs) and O (outputs), a set $\mathbf{T} \subseteq \mathbf{E}^*$ of traces, a partially ordered set L, and a mapping $\mathbf{lvl} : \mathbf{I} \cup \mathbf{O} \rightarrow \mathbf{L}$ that associates each input or output event with a security level.

The most commonly-used class of security policies was defined by Bell and LaPadula [3]. A Bell-LaPadula policy divides the system into active subjects and passive objects and assigns a security level to each. Information is prevented from flowing from a subject through an object to a lower level subject by restricting how the subjects may access the objects. The basic rules are that a subject may only read objects that they dominate and write to objects that dominate them. A formal definition is presented in [3]. These policies are sometimes overly restrictive, in that the security level for some data items may be forced to too high a level, and also they have no rules to prevent information from flowing through covert channels, such as the security control mechanism itself.

1.1.1 Restrictiveness

In an office "system" consisting of people and actual files, the basic security policy is: only people whose security level dominates that of a file are allowed access to that file. This policy works for the office, but not for a computer: for instance, unauthorized users can write programs which, when used to process high-level information, leak information to the programmer via "Trojan horses" inserted in the code. The information leaks through covert channels not subject to the security controls in the system. The cooperation of a person cleared to see the information is not necessary. Also, the speed of the computer makes the bandwidth of covert channels much greater than for human systems. Since all the steps between the user and the information in question must be considered in order to eliminate covert channels, the definition of security will necessarily be more complicated.

For a rated event system to be *secure*, high-level inputs must not be deducible from that portion of the trace that can be observed [58]. For each security level $\mathbf{l} \in \mathbf{L}$, view_l denotes the set $\{e \in I \cup O \mid l \succeq \mathrm{lvl}(e)\}$.

Nondeducibility.

$$\forall t \in T, l \in L, \gamma \in E^*.\ t\hat{\ }\gamma \in T \Rightarrow \\ \exists \gamma' \in E^*.\ t\hat{\ }\gamma' \in T \ \wedge\ \gamma' \uparrow \mathrm{view}_l = \gamma \uparrow \mathrm{view}_l \ \wedge\ \gamma' \uparrow (I - \mathrm{view}_l) = \langle\rangle$$

This states that any future observable behavior of the system ($\gamma \uparrow \mathrm{view}_l$) can be produced without any nonobservable inputs (by γ'). Note that decreasing the amount of information available to an observer by increasing the level of an output (or deleting an output entirely) preserves nondeducibility. Additional properties may also be required.

For a proof of security about a large system to be practical, the security property must be *composable* in that it can be inferred from properties about its components and their interactions. This is McCullough's "hookup-up" requirement [33]. The security of an entire system should be a consequence of the security of its parts. While the nondeducibility property is not composable, it can be strengthened to a *restrictiveness* property that is composable [34].

The definition of restrictiveness is based on a state-machine representation of objects, rather than on traces. A state can be considered to be the set of event sequences that extend the current trace to an element of **T**. For each extension γ in state **s**, s_γ represents the state that results after γ occurs starting in **s**. For a system in which the sequence of outputs is uniquely determined by the sequence of inputs, the state can be equated with the trace of events that has occurred. However, for a nondeterministic system, each state may also need to include a failure set and a divergence set [9]. This state-transition model will be discussed further in the next chapter.

Restrictiveness. An object is restrictive if it is *input total* (inputs are never prohibited) and for each $l \in L$ there is an equivalence relation $=_l$ on the set of states S such that

- $\forall i \in I \cap \mathrm{view}_l,\ s, s' \in S.\ s =_l s' \ \Rightarrow\ s_{\langle i\rangle} \ =_l\ s'_{\langle i\rangle}$
- $\forall i \in I - \mathrm{view}_l, s \in S.\ s \ =_l\ s_{\langle i\rangle}$
- $\forall e \in O,\ s, s' \in S.\ s_{\langle e\rangle} \in S \wedge s =_l s' \ \Rightarrow\ \exists \gamma \in O^*.\ s_{\langle e\rangle} =_l s'_\gamma \ \wedge\ \langle e\rangle \uparrow \mathrm{view}_l = \gamma \uparrow \mathrm{view}_l$

If such equivalence relations can be found for an object, the nondeducibility property holds. **Also, nondeducibility holds for an object composed of several subobjects for which restrictiveness holds.**

Consider the infinite buffer $\mathrm{ABuf}_{<>}$. The subscript represents the state of the buffer

$$\begin{aligned}
\mathrm{ABuf}_{<>} &= ?x \rightarrow \mathrm{ABuf}_{<x>} \\
\mathrm{ABuf}_{<h>\hat{}\,tail} &= \ ?x \rightarrow \mathrm{ABuf}_{<h>\hat{}\,tail\hat{}<x>} \\
&\quad \square\ !h \rightarrow \mathrm{ABuf}_{tail}
\end{aligned}$$

(initially the empty queue). The buffer can either receive a value from channel in and append it to the end of the queue or else send the head element on channel **out**. **ABuf** can be shown secure by defining $s =_l s'$ iff $s \uparrow \mathrm{view}_l \ =\ s' \uparrow \mathrm{view}_l$.

$i \in I$ $s =_l s' \ \Rightarrow\ (s\hat{}\langle i\rangle) \uparrow \mathrm{view}_l \ =\ (s \uparrow \mathrm{view}_l)\hat{}(\langle i\rangle \uparrow \mathrm{view}_l) \ =\ (s' \uparrow \mathrm{view}_l)\hat{}(\langle i\rangle \uparrow \mathrm{view}_l) \ =$ $(s'\hat{}\langle i\rangle) \uparrow \mathrm{view}_l$ (Since input events are always possible, $\forall s \in S, i \in I.\ s\hat{}\langle i\rangle \in S$.)

$i \in I - \mathrm{view}_l$ $(s\hat{}\langle i\rangle) \uparrow \mathrm{view}_l \ =\ s \uparrow \mathrm{view}_l$

$e \in O \cap \mathrm{view}_l$ $\langle e\rangle\hat{}s =_l s' \ \Rightarrow\ \exists \gamma, s'' \in O^*.\ s' = \gamma\hat{}\langle e\rangle\hat{}s'' \ \wedge \gamma \uparrow \mathrm{view}_l = \langle\rangle \ \wedge\ s =_l s''$ (An output can occur iff it is the first element in the buffer and $(\gamma\hat{}s)_\gamma = s$.)

$e \in O - \mathrm{view}_l$ $\langle e\rangle\hat{}s =_l s' \ \Rightarrow\ s =_l s' \ \wedge\ \langle e\rangle \uparrow \mathrm{view}_l = \langle\rangle \uparrow \mathrm{view}_l$

A finite queue that throws away new inputs if the number of waiting messages equals a limit is not secure because an unclassified message might be discarded after the buffer has become full with secret messages. Thus, the secret messages can affect the sequence of unclassified outputs. The γ that was used to prove the security of the infinite buffer will not exist in this case. This problem can be corrected by placing bounds on the number of messages of each security level.

1.1.2 Synchronous Communication

The input totality assumption does not hold for synchronous communication. When considering security, if inputs are not always immediately received, as they are in ABuf,

synchronous communication must be represented as two events — the sending of the message and the return of an acknowledgement for which the sender waits. The message may arrive at any time, but will only be acknowledged when the receiver is ready for it.

For example, consider a system of three objects, one with secret information and the other two without any. Each of the nonsecret objects initially try to either synchronously send a message to the secret object or receive a message from the other nonsecret object. If one is successful in sending a message to the secret object, it then sends a message to the other nonsecret object. In this manner, the secret object has sent a bit of information by choosing the nonsecret object it is willing to receive a message from. An acknowledgement makes this information flow explicit.

The traces for **ABuf** pair an implicit input event with each explicit output (either a request-for-output preceding the output or an acknowledgement following it). If the security level of this implicit input is the same as the security level of the associated output, the proof of security given above is still valid. The traces for the Live Queue should contain an acknowledgement output immediately following each input and a request-for-output input preceding each output. The nondeducibility property does not hold for the **queuer** process. (Consider $\mathbf{L}=\{\mathbf{s},\mathbf{u}\}$ where $\mathbf{s} \succ \mathbf{u}$, $\mathbf{l}=\mathbf{u}$, $\mathbf{max}=1$, $\mathbf{t}= \langle \text{enq}_s\text{?v}, \text{enq}_s\text{!ack}, \text{enq}_u\text{?v}' \rangle$, and $\gamma= \langle \text{deq}_s\text{?rfo}, \text{deq}_s\text{!v}, \text{enq}_u\text{!ack} \rangle$.)

The need for an acknowledgement can be avoided by making the communication asynchronous through the use of a secure buffer to hold the messages arriving at an object, as defined in figure 1.1. The function **ASynch** modifies a process P so that messages sent to it on channel **in** are buffered using **ABuf**. Process **P** communicates with the buffer over channel **int**, which is hidden from other processes. Thus, messages sent over **int** are not included in the traces of **ASynch(P)**.

1.1.3 File System

To express security properties in programs, the CSP notation must be extended to label input and output events with security levels. A message **m** communicated on channel **c**

$$\begin{aligned} \text{ASynch(P)} &= (\text{intout}(\text{ABuf}_{<>}) \parallel \text{intin(P)}) \setminus \{\text{int.x}\} \\ &\quad \text{where intout(!x)} = \text{int!x} \qquad \text{intin(?x)} = \text{int?x} \\ &\qquad\quad \text{intout(x)} = \text{intin(x)} = \text{x otherwise.} \end{aligned}$$

Figure 1.1: Buffer to allow asynchronous input

with security level l will be written as **c.l.m**. Internal events, such as communication over local channels, do not need to be labeled. The notation c!l.m will be extended to represent the process c!l.m → SKIP. Also, the notation l≤l' will be used for the dominates relation "l'⪰l".

In the file system, file names have been extended from being natural numbers to being pairs consisting of a security level and a natural number (figure 1.2). In this way, the creation of a classified file cannot be detected through the file names. The file-system state contains a function **N** from the set of security levels to the natural numbers to handle this extension. The output following a **create** or a **read** acknowledges those inputs. An acknowledgement is returned after a **delete**.

The security policy for the file system will be expressed entirely in terms of security levels. Nothing will prevent one user from deleting a file created by another user, provided the level of the file is high enough. The restrictions imposed will be that the level of a file must dominate that of the creator and the deleter ("write up") and be dominated by any readers ("read down"). The **read** and the **delete** commands compare the security level of the command against the security level of the file before performing the command. If the file has a greater security level than a **read**, an error is returned. Since the decision to return the error is based entirely on information contained in the message and not on the state, the sender learns nothing about the file system.

The main process repeatedly reads a message from channel **in** and potentially spawns a new process to output a value. The input events are the messages, which contain their security level. The output events are the process creations, at the security level contained in that process' output message, and the acknowledgements of **delete** messages.

To prove that the main process is secure, the state must be extended with the set of pending output-process creations, **P**. For each **p∈P**, let **lev**(p) be the security level of its output. Define $(N,f,P)=_l(N',f',P')$ iff

- $\forall l' \leq l.\ N(l') = N'(l')$

$$\begin{aligned}
\mathrm{filesystem} &= \mathrm{FS}_{\mathrm{N0,f0}} \text{ where } \forall l \in L.\ \mathrm{N0}(l) = 0 \text{ and } \forall m \in L \times \mathbb{N}.\ \mathrm{f0}(m) = \mathrm{error} \\
\mathrm{FS}_{\mathrm{N,f}} &= \mu p.\ \ ?l.(\mathrm{create}, s, c) \rightarrow (c!l.m \parallel \mathrm{FS}_{\mathrm{N}[l:n+1],\mathrm{f}[m:s]}) \\
&\qquad\qquad\qquad\qquad\qquad \text{where } n = \mathrm{N}(l),\ m = (l, n) \\
&\qquad\quad \Box\ ?l.(\mathrm{read}, m, c) \rightarrow ((c!l.f(m) \lhd m_1 \leq l \rhd c!l.\mathrm{error}) \parallel p) \\
&\qquad\quad \Box\ ?l.(\mathrm{delete}, m) \rightarrow (\mathrm{FS}_{\mathrm{N,f}[m:\mathrm{error}]} \lhd l \leq m_1 \rhd p)
\end{aligned}$$

Figure 1.2: Secure File System

- $\forall l' \leq l, n \in \mathbb{N}.\ f(l', n) = f'(l', n)$
- $\{p \in P \mid lev(p) \leq l\} \;=\; \{p \in P' \mid lev(p) \leq l\}$

The input events **create**, **read**, and **delete** and the output event of creating an output-process (thus removing it from the pending set) are easily seen to be restrictive for these equivalence relations.

For the output processes, there are no inputs and the only state is the value to be output or empty. They are trivially restrictive. Since each process in the file system has been shown to be restrictive, by the compositionality property the file system is secure.

1.2 Adaptive Security Policies

The security policy for an information system that will have a long lifetime, during which it will evolve, and that must operate under a variety of conditions, such as normal operation, after the failure of a component, and during battle, cannot be static. As the system evolves, new parts will be added while others are removed or replaced. Also, data maintained by the system may change in importance or become obsolete, thus requiring less protection. As conditions change, security may need to be sacrificed in favor of speed or greater functionality. Therefore, *adaptive* security policies are necessary.

In order to describe how security policies can be adapted, a model of a security policy must be given. This description of the basic model is followed by an analysis of how a variety of security changes can be formulated in it. Finally, a proof technique is given that can be used to detect leaks in security caused by a policy change. Throughout this section, the low-water-mark problem will be used.

1.2.1 Security Policies

A security policy for an object specifies restrictions on the security levels of messages sent to the object and assigns security levels to its output messages. It can be expressed as an *acceptance predicate* for the input messages and a security level *assignment rule* for the output messages. For example, in a Bell-LaPadula policy a **write** message is only accepted if its level is dominated by the level of the object and all output messages are assigned the object's level. The world outside the information system is also viewed as a set of objects, each with an acceptance predicate to filter out inappropriate messages from the system, and an assignment rule to classify messages sent to the system.

The acceptance predicate for an object specifies whether the security level of an input message is acceptable. It depends on the type and the value of the message. The message is rejected by the system if the predicate is false. The acceptance predicate for the file system example when given a *read request* input message is whether the level of the message dominates that of the file to be read, as determined by the file's name. The input-totality requirement for restrictiveness is satisfied by regarding an acceptance predicate as a separate filter object; the main object is then regarded as accepting all inputs received from this filter. Thus, any rejected inputs can be ignored when doing a proof of restrictiveness.

An alternate implementation of the file system is to store the security level of the file with the file, rather than in its name. Each file is a subobject of the file system, with its own acceptance predicate. The acceptance predicate for the file system would accept all *read requests* and forward them to particular files, where they would then be rejected if their level did not dominate that of the file. The file system could continue to reject *create* messages that were not dominated by the proposed level of the file.

The assignment rule for output messages is a function of the state of the object. In the file system, receiving a *read request* input with security level l sets the object's state so that it responds at level l.

Perceived Changes to Security Policies

The external view of an object is the sequence of outputs produced in response to a sequence of inputs (its trace). The security policy affects this view by rejecting some of the inputs, preventing them from influencing the outputs produced, and by labeling the outputs with security levels, controling which other objects will accept them. The security policy therefore helps determine view_l.

An actual change in security policy for an object is the replacement of the acceptance predicate and/or the assignment rule. For some levels l, view_l will no longer be consistent with the old policy and the change is perceived, while for other levels l, view_l will continue to be consistent with the old policy. In the file system example, a policy change might be to prohibit any files with levels not dominated by some system maximum level m. This can be accomplished by changing the acceptance predicate to reject any *creates* and *reads* for files with a level not dominated by m. This policy change will not be perceived by any level dominated by m. Likewise, a change in the system maximum will not be perceived by any level that is dominated by both the old and the new maximum.

Since a policy change can be perceived by some of the observers of the object, the event that triggers it must be included in the trace of the object. If the level of this event is not dominated by the levels of all views that perceive the change, the change itself will leak information about the occurrence of that event. Thus, a command changing the maximum level of the files in the filesystem should have a level dominated by both the old and the new level.

Low-Water-Mark Problem

Consider the low-water-mark problem [11]. An object that stores a value accepts **read**, **write**, and **reset** inputs. A **read** causes the object's value to be output at the security level of the **read**, a **write** changes the value of the object, and a **reset** causes the value to become undefined.

The acceptance predicate for an object permits only those **reads** that dominate the current value, as determined by the last **write** or **reset**. Thus, a **write** or a **reset** changes the acceptance predicate for future **reads**. Such a change can be perceived at any security level that dominates either the level of the old value of the object or the level of the new value. The object will be secure only if any of the levels at which a change can be perceived dominates the security level of the change event (the **write** or the **reset**). Therefore, these events can themselves only be accepted if their security level is dominated by the level of the current value of the object. It is also necessary to require that the levels are linearly ordered, since a **write** can affect the acceptance of future **writes** and **resets**.

1.2.2 Situations for Security Policy Modifications

There are several situations when a security policy might be modified. There can be multiuser terminals connected to the information system. Each is considered an object with a security policy. When no one is logged on at a terminal, the acceptance predicate will only accept the very lowest security class messages. The assignment rule can assign either the lowest class to all messages sent from the terminal to the system, under the assumption that a user will not send anything of importance until she has identified herself, or the highest class, because the user identifiers and passwords are extremely sensitive. After login is complete, the security policy for the terminal can be set to reflect the authorization of the user. During a session, the user might be allowed to change the assignment rule to reflect different sensitivities of the data being sent to the system. After logout, the security policy must revert back to the default.

The set of security levels **L** can be changed by adding or deleting levels. For instance, a new use of the system may require adding a compartment. Adding a new level is accomplished by changing some of the assignment rules so that some outputs are assigned the new level. Acceptance predicates can also be modified to accept or reject inputs at the new level, although most acceptance predicates are expressed as a range of levels and will automatically handle the new level, as determined by the *dominates* relation. Deleting a level is also a change to the assignment rule such that no output is assigned the deleted value. No change is required to the acceptance predicates in this case.

Reconfiguration consists of adding or removing objects. An object that is not part of a system may be considered to have the *absent* security policy that rejects all inputs and labels all outputs so that no other object can see them (actually, the assignment rule can be anything if the object produces no outputs in response to no inputs). Thus, removing an object is equivalent to changing its security policy to absent and adding an object is equivalent to changing its policy from absent. Also, the failure of a node can cause the objects running on it to be removed and appear to have the absent security policy. Detection of the absent policy can therefore be used to trigger recovery actions, possibly including reconfiguration.

The sensitivity of information can change, requiring a change in the assignment of security levels to the messages carrying it. Such reclassification might occur during a battle, when availability of information is more important than security, or according to some time schedule. One method of handling reclassification is to pass messages through a "trusted" object (that cannot be proved secure, but is considered secure anyway) that modifies security levels if appropriate. This object could possibly be a human. Another approach is to use "extended security levels" [60], instead of the simple levels used in this report, that assign inputs a security level based on the current trace. Thus, the level of an input can change as the trace of the object changes, perhaps permitting the level assigned to outputs that are based on that input to decrease. Note, however, that reclassification does not involve changing the security policy and will not be considered further here.

1.2.3 Changing Policies

The security provided by each potential policy can be evaluated individually and a determination made on the acceptability of any potential information flows. The concern here is over the security of changing from one policy to another. There are two problems: the change itself can be observed, thus causing a flow of information, and information input by the object and saved in its state under one policy will be released by the new policy at too low a level.

The first factor that must be considered is whether a single set of equivalence relations can be used to prove the security of both the old and the new policies. If there is not, and for some level l there are states that are equivalent under the relation for the old policy but not for the new policy, the amount of information available at level l has expanded. This is then an indication that information has been downgraded by the change in policies. Similarly, for a new level l the test for downgrading of information to level l is whether there are states that are not equivalent at level l under the relation for the new policy, but are equivalent under the relation for the old policy at all levels dominated by l.

In order to determine the level at which a change of security policies resulting from event e can be detected, assume that the equivalence relations for the old and the new policies are the same. The equivalence relation $=_l$ is extended to state, acceptance predicate, and assignment rule tuples as follows: $(s,AP,AR) =_l (s',AP',AR')$ iff $s =_l s'$, $\forall i$, $l \succeq lvl(i)$.

$AP\ i = AP'\ i$, and $\forall l' \preceq l, s, o.\ AR\ (s,o) = l'$ iff $AR'\ (s,o) = l'$. Informally, acceptance predicates are equivalent at security level l if they behave the same on all inputs with levels dominated by l. Assignment rules are equivalent at security level l if they assign outputs the same level, up to level l. The proof of security is modified such that, for any event that changes the security policy, the tuples are used instead of just the state. Inputs that are not accepted change neither the state nor the security policy and therefore can be ignored. Also, if $l \succeq lvl(i)$ and $(s,AP,AR) =_l (s',AP',AR')$, either both AP and AP' accept i or neither does. Thus, for equivalent states, a visible input that is accepted by one and potentially causes a change in security policies will be accepted in the others.

In the low-water-mark problem, the acceptance predicate and the assignment rule are modified by a **write** or a **reset** input. Concentrating on just the acceptance predicate and the assignment rule, let i be a **write** or a **reset** that is accepted. For a level $l \succeq lvl(i)$, the new acceptance predicate and assignment rule are defined without regard to the previous tuple and therefore all tuples that result from i have equivalent acceptance predicates and assignment rules. For a level $l \preceq lvl(i)$ and input $i' \preceq lvl(l)$, if i' is a **read** it would have been rejected by any acceptance predicate that accepted lvl(i) and if i' is a **write** or a **reset** it would have been accepted. After receiving i, i' will be accepted or rejected under exactly the same conditions. Therefore, the acceptance predicates before and after receiving i will be equivalent at level l. Also, no outputs would have been assigned a level of l or less either before or after i. Thus, the assignment rules before and after receiving i will be equivalent at level l.

1.3 Progress

The properties in this category have to do with showing that a process makes noticable progress and is not stuck at one point. A process can fail to make progress because its environment stops attempting to communicate with it, because it stops attempting to communicate with its environment, or because resources that it needs are never allocated to it. In general, showing that a system of processes continues to make progress is equivalent to showing that it does not halt and is therefore undecidable. However, techniques exist that can be applied to specific programs to show that progress properties hold.

1.3.1 Liveness

Liveness is the collection of properties that state that a system or a process will make progress. A liveness property for the Live Queue is that each item enqueued will eventually be dequeued. A liveness property for the File System is that each request to read a file will eventually be replied to. Another is that each file that is created will eventually be read.

One reason why a liveness property might not be satisfied is that the service provided by a process is not needed. For example, if the value of a file is not needed, it will not be read. Also, a process that handles exceptional conditions will never run if no such conditions occur. This failure to achieve liveness is not a system error, but rather is part of the normal operation of the system. Liveness should be proved for such processes under the condition that their service will be requested.

Starvation and Fairness

A liveness property can fail to hold because a process is continually bypassed by the scheduler. In the File System, if the readers of a file are never scheduled the file will never be read. Also, if the File System continually ignores a read request in favor of other input messages, there will never be a reply to the read request. Such a process that is always bypassed is said to *starve.*

A scheduler that does not allow starvation is called *fair* [15,20]. One formalization of fairness states that a process which is infinitely often able to proceed will eventually do so. For the file system, since there is no limit on the number of messages that can be processed, fairness requires that every attempt to send a message to the file system will eventually be accepted. Another aspect of fairness is that the file system will not continue to accept inputs while postponing outputs arbitrarily long.

Fairness is a property of the way processes and messages are handled, rather than of a system of processes. It is proved for the process-management code level of a system, rather than for the collection of processes that is managed. At the process level, fairness must be assumed in order for liveness properties to be proven.

Deadlock

When a group of processes is unable to proceed because each is waiting for one or more of the others to take some action, the processes are *deadlocked.* The queuer process can become part of a deadlock if its producer also can consume. If the producer tries to consume when the queue is empty, the queuer is waiting to receive a value from the producer while the producer is waiting to receive a value from the queuer. The file server can become part of a deadlock if each of its users tries to receive a message from it before sending it a request. In the model used here, waiting only occurs when a process is trying to communicate.

While the occurrence of a deadlock is the failure of a liveness property, its absence is a safety property that is proved for all executions of the program. One possible safety property is that cycles of communication requests over channels internal to a set of processes cannot occur [8]. A proof of this property is possible if the system is structured so that the communication paths are linear.

Another form of deadlock is in the allocation of system resources by a resource allocation process. Such a process either must execute a deadlock-prevention algorithm when making allocations or execute a detection algorithm when requests arrive that cannot be filled. If a deadlock is detected, resources must be removed from some of the processes in order to break the deadlock. There are well-known algorithms that can be used for prevention or for detection. These could possibly be proven correct using a verifier.

Divergence

Divergence occurs when a process stops attempting to communicate with its environment because it continually chooses to do internal actions. If the producer and the consumer processes only communicate with the queuer, the combination of the three processes diverges since it does not communicate with its environment. An observer may find it difficult to distinguish a diverging (live) process from a deadlocked one, since in neither case can the process communicate with its environment [9].

1.3.2 Specifying Progress Properties

Specifications in the Event System model used in this paper are properties that must hold for the set of traces. Safety properties, including the absence of deadlock, are invariants that hold for all traces. Liveness properties, such as the eventual willingness of a process to communicate with its environment (absence of divergence), must *eventually* hold for each sequence of events. For a particular trace t, eventually means that either a property P holds for that trace or it holds for every extension of that trace of sufficient length. Formally,

$$\exists k \in \mathrm{N}.\, \forall t' \in T.\, |t'| \geq k \ \wedge \ t \leq t' \ \Rightarrow \ \exists t'' \in T.\, t \leq t'' \leq t' \ \wedge \ P(t'')$$

Temporal logic, as described in the next chapter, includes *always* and *eventually* operators that have proven useful in specifying safety and liveness properties [28,37,38].

Temporal logic is also appropriate for expressing fairness assumptions [15]. These assumptions are used to prove liveness properties. They cannot be proved about specifications, but rather are requirements that must be satisfied by any scheduling algorithms used in the implementation.

Process and resource schedulers that have access to secure information can be a source of leaks of that information. They therefore should either be proven to be secure (using the techniques discussed in the section on security) or be identified as a communication channel. (Scheduling decisions based on priorities that are assigned using secure information may be unavoidable.) In addition to security, fairness assumptions should also be proven of any schedulers.

1.4 Nondeterminism

A system of one or more processes is nondeterministic if the trace of its outputs cannot be predicted with certainty from the trace of its inputs. A deterministic system may be treated as a function from finite input sequences to output sequences, while a nondeterministic system is a function to sets of output sequences. For a sequential process, nondeterminism is introduced by allowing the process to make a random choice (represented in CSP with the □ operator). For a system with several concurrent processes, nondeterminism is also introduced by varying the relative speeds of the processes. Different relative speeds may result in different orderings of the messages internal to the system (and therefore of the inputs to the various processes) and of the output messages.

In the Live Queue, the producer, the consumer, and the queuer are deterministic (although the interleaving of the outputs with the inputs in the queuer is nondeterministic). However, a system consisting of two producers and a queuer is nondeterministic, since the output is an arbitrary interleaving of the outputs of the producers. The output for a particular run of the system is determined by their relative speeds and how they are scheduled.

In the File Server, filesystem is a system of processes (created using the ‖ operator). Each output process is deterministic (it just outputs a value and halts), but filesystem is nondeterministic, since each output is generated by a different process and their sequencing is determined by how these processes are scheduled and by how long they take to execute.

1.4.1 Synchronization

Sometimes, nondeterminism caused by the arbitrary progress of several processes is undesirable. In a system with a producer process and a bounded queue that discards inputs when it becomes full, a fast producer (relative to the rate at which values are removed from the queue) can result in items becoming lost. If the producer can be delayed whenever the queue is full, no items will be lost and the system will be deterministic.

Various forms of synchronization mechanisms have been suggested to control the relative rates of processes. In the model used here, processes are synchronized through message passing. In the Live Queue with a single producer, the producer and the queuer are synchronized such that their combination is deterministic. Without synchronization, the set of traces for a system of processes is the arbitrary interleavings of the traces for the individual processes. With synchronization, some of the interleavings are not possible and therefore the set of system traces is smaller.

A special form of synchronization requirement, called *serializability*, is found in database management systems. A transaction is a sequence of atomic actions whose effect should be deterministic. A database management system that handles several transactions concurrently should ensure that the transactions do not interfere with each other, or are serializable. Using traces, serializability can be expressed as follows:

> For every trace t in which the events of several transactions may be interleaved, there exists a trace t' in which the events of the transactions are not interleaved and t' is a permutation of t.

Techniques for specifying and verifying serializability are further discussed in the report on database consistency.

1.4.2 Race Conditions

An alternative to using synchronization to avoid undesirable nondeterministic choices is to rely on the scheduler and the timing of the events in the system. This is known as a race condition (it is assumed that the "good" sequences will always beat the "bad" sequences). Normally, however, race conditions should not be used in reasoning about an execution. Modifying the program or changing the compiler or the hardware on which the system is run can change the timing. Also, faults during execution can affect the timing.

1.4.3 Verification

In verifying a nondeterministic program, the properties being proven must be shown to hold for each possible execution. In a system without any synchronization, the number of possible executions is exponential in the number of processes. Therefore, verifying each possible execution is impractical.

A solution to this problem is to prove that a property is an invariant. The property is shown to hold initially and then is also shown to hold regardless of which choice is made provided that it was true before the choice was made. In the model used in this report, processes do not interfere with each other in that the truth of a property of some process depends only on the sequence of events of that process. Therefore, a property proven to be an invariant for a sequential process must hold for that process in a system containing other processes. The conjunction of the individual invariants must be an invariant of the system.

When considering security, a synchronization mechanism provides a two-way flow of information. For synchronous message passing as used in this study, an implicit acknowledgement is returned to the sender of a message. (The acknowledgement can frequently be combined with a reply to the message.) The acknowledgements can be ignored if each component interacts only with a communication network that delivers messages asynchronously and is proven to be secure. (There is no information flow from the receiver of a message to the sender.)

Another possible source of security problems is information passed through the scheduling algorithm. For example, the scheduler could signal information by increasing the service given to one process in relation to that given to another. The scheduler must either be a trusted piece of the system or it must only receive information, such as process priorities and the creation or deletion of a process, at the lowest security level of the system.

1.5 Fault Tolerance Requirements

1.5.1 Introduction

Distributed computing and fault tolerance are closely related. Making a system fault-tolerant requires that parts of it be replicated and that computations can be restarted. Tools that aid in the specification and verification of constructs that achieve fault tolerance in secure distributed systems must therefore allow for the specification of how the system parts should be replicated and restart/recovery applied and they must provide for the implementation of these requirements. Some general discussions of fault-tolerant systems can be found in [1,53,54,55].

This chapter will indicate a number of the problems that have to be dealt with by a mechanism that implements fault tolerance by means of software replication. The model of a distributed system based on an object-oriented system approach was described in the introduction. It views the system as a collection of objects that interact only by passing messages. We will use the term *component* to indicate a collection of replicated objects that perform the same function, together implementing the fault-tolerant function. Replicated objects may either be identical, i.e. copies of a single object, or different, i.e. implementing the same function by using different algorithms. A *faulty component* is a component of which one or more of the composing objects are faulty. However, as long as the faults of the objects are masked, the component's behavior satisfies its specification, i.e. the component is fault-tolerant.

This chapter contains a survey of the mechanisms used to support fault tolerance and a summary of the issues which must be considered when designing a fault-tolerance mechanism.

Failure models

Failures can be partitioned into two classes, depending on whether repair is required following a failure. *Hard failures* influence the future operation of the component, and cause the need for repair. These are opposed to *transient failures* that do not influence subsequent operations [50].

An *object failure* occurs when the behavior of the object no longer satisfies its specification. A *component failure* occurs when it can no longer mask or recover from object failures. It will then show behavior that no longer satisfies its specification. Behavior of objects in response to a failure can be classified according to the nature of the disruption it causes. Four models can be distinguished that describe this behavior:

Crash failures An object halts in response to a failure and there is no way for other objects or components to detect this.

Fail-stop failures An object halts in response to a failure and other objects or components can detect this. No faulty behavior occurs because the faulty object stops [46].

Omission failures Every now and then some information gets lost. This failure does not influence the subsequent operation of an object. .

Byzantine failures An object exhibits arbitrary and malicious behavior [30].

Crash, fail-stop and Byzantine failures all fall into the class of hard failures, of which Byzantine failures are the most disruptive. Allowing Byzantine failures is the weakest possible assumption that could be made about the effects of a failure. Since a design based on assumptions about the behavior of faulty components runs the risk of failing if these assumptions are not satisfied, it is prudent that life-critical systems tolerate Byzantine failures. Omission failures are transient failures and are often exhibited by communication lines. A message that is in transit might be corrupted, but subsequent transmissions will succeed.

Requirements for fault tolerance are usually specified in terms of MTBF (mean-time-between-failures), probability of failure over a given interval, and other statistical measures [16]. However, there are advantages to describing the fault tolerance of a system in terms of the maximum number of failures that can be tolerated over some interval of interest. We shall call a component consisting of a set of distinct objects *t fault-tolerant* if it satisfies its specification provided that no more than t of those objects become faulty during the interval of interest. Asserting that a component is t fault-tolerant makes explicit the assumptions required for correct operation; MTBF and other statistical measures do not. Moreover, t fault tolerance is unrelated to the reliability of objects that make up the component and therefore is a measure of the fault tolerance supported by the component architecture, in contrast to fault tolerance achieved simply by using reliable objects. Fault tolerance of an actual component will depend on the reliability of the objects used in constructing that component. In practice, t should be chosen based on statistical measures of object reliability in a way that minimizes the probability of more than t failures during the interval of interest.

Software Replication

Failures—be they hard or transient—can be detected only by replicating actions in a failure-independent way. One method to do this is by performing the action using objects that are physically and electrically isolated. We call this *replication in space*. The validity of the approach follows from an empirically justified belief in the independence of failures at physically and electrically isolated devices. A second approach to replication is for a single object to repeatedly perform the action. We call this *replication in time*. Replication in time is valid only for transient failures.

Techniques used to detect failures are well-known and widely used, so we will not discuss them here any further. Some examples are duplication, timeout mechanisms, and consistency checks [55].

In response to a detected failure, a fault-tolerant system can behave in two different ways. One way is to mask failures, i.e. to hide the effects of failures by substituting correct information. This involves *masking redundancy*, which can be provided by having multiple replications of an object in combination with some voting mechanism.

The other way is to allow dynamic reconfiguration or recovery of the system. This is referred to as *dynamic redundancy*, and uses redundant objects, stand-by sparing, and recovery (rollback) mechanisms.

Many fault-tolerance techniques are a combination of the techniques mentioned above. Thus, classifying a technique using a single term often is not possible. We will use combinations of the terms mentioned above to indicate the type of a technique, but the main differentiation will be in techniques that use *masking redundancy* or *dynamic redundancy*.

This study investigates the constructs that are currently used to support fault tolerance by means of replication. It discusses the specification and verification of these constructs and their possible implementations. The development of a construct for replication also requires a profound understanding of the problems that result from having multiple copies of software objects in a distributed system. The construct must adapt the syntax and semantics of I/O commands to incorporate replication. Implementation of the construct must solve the problems of communication, synchronization, and identification of the replicated objects.

Software Restart/Recovery

Both masking and dynamic redundancy techniques will fail after a certain number of objects has become faulty. Eventually, human intervention will be necessary to handle the repairs required. However, before this point is reached, restart/recovery techniques may try to recover faulty objects.

Software failures can be overcome by restart/recovery mechanisms if a system provides dynamic redundancy. This implies that state information has to be saved periodically as *checkpoints*, and, when a failure occurs, computation must be restarted from (rolled back to) its last checkpoint. If a failure is due to a hardware failure of the system on which an object resided, replacement of the faulty system part may be necessary. Information must be provided on how to reconfigure the system and on how to reintegrate objects that resided on the faulty system part. The time and information needed to perform these actions are critical factors influencing the fault tolerance property of a system.

If a system provides masking redundancy some caution is required. If the results of performing a set of replicated actions disagree, a failure has occurred. Without making further assumptions, this is the strongest statement that can be made. In particular, if the results agree, we cannot assert that no failure has occurred and the results are correct. This is because if there are enough failures, all of these might be corrupted, yet still agree. However, this only occurs if there are more failures at one time than the system is able to handle correctly according to its specification, in which case it can be expected to exhibit arbitrary behavior.

Some appropriate action has to be taken to handle faulty objects that participate in masking failures. One way to cope with faulty objects is to adapt the voting scheme in a way that reflects the degradation of the system (*graceful degradation*). However, in order to preserve the system's masking ability some form of dynamic redundancy has to be added.

Besides being a support to the redundancy techniques described above, restart/recovery techniques can be used as fault-tolerance techniques all by themselves. In fact, the first efforts to implement fault tolerance involved restart/recovery techniques. These techniques can provide either *backward error recovery* after a failure has occurred, or it can provide *forward error recovery* after detecting an exceptional state. Examples of both mechanisms will be discussed in the next section.

1.5.2 Fault-tolerance techniques

Fault tolerance techniques are in most cases a combination of three mechanisms: replication, which can be subdivided into masking redundancy and dynamic redundancy, and restart/recovery techniques. This section presents a survey of fault tolerance techniques and the different structures that have been proposed to support them.

Replication

Fault detection is the first step in accomplishing software fault tolerance. As mentioned above, fault-detection techniques are fairly well understood and are not discussed here. Our main interest concerns ways to handle replication of software objects to provide actual tolerance of faults that have been detected. The two basic approaches to software fault tolerance by means of replication are discussed below.

Masking Redundancy Masking redundancy provides fault tolerance by either isolating or correcting faults that are the result of faulty components before these results reach other (non-faulty) components. Trying to prevent the effects of faults from spreading out over the system by isolating them is also referred to by the term *fault confinement.* Masking redundancy is a static form of redundancy in that no intervention occurs from elements outside the component. This implies that when masking redundancy is exhausted, any further faults will cause component failures. Several masking-redundancy techniques are described below.

N-Modular Redundancy is a technique that involves multiple replications of an object, and a voting mechanism. Fault tolerance is achieved by executing all replications, either sequentially or in parallel, and then sending the results to the voter. The voter decides on the outcome of the fault-tolerant component.

Three replications of an object give enough redundant information to mask one faulty object in the component. This is accomplished by a majority (two-out-of-three) vote on the outcomes. If a system is made t fault-tolerant by replication, $t+1$-fold replication permits failure detection but not failure masking when there are as many as t failures. When there is disagreement among $t+1$ independently obtained results, one cannot assume that the majority is correct. Masking t failures requires $2t+1$-fold replication, since then as many as t values can be faulty without causing the majority value to be faulty.

In this scheme, the voter is a single point of failure. If the voter is faulty, the component is not fault-tolerant in that it still can produce faulty outcomes. To overcome this problem, replication of the voter is proposed. The fault-tolerant component will always have one point of failure in the voter where the results of all voters come together to produce a single component output.

Synchronization of the multiple replications in N-modular redundancy is necessary to prevent faulty outputs. This can be accomplished by associating a logical or physical clock with each replication [27], and running an agreement protocol to synchronize clocks. Several different agreement protocols that solve the consensus problem in distributed systems exist and are discussed in [14,48].

A variation of N-modular redundancy is provided by a technique called **N-version programming** or **design diversity** [2]. Instead of having identical replications, a number of independently-developed algorithms are used that perform the specified function. Making the algorithm different for each execution producing the results to be voted on may result in some protection against hard failures caused by software design faults. However, experience has shown that, in spite of independent development, design faults tend to cluster in the more difficult parts of the different algorithms resulting in *coincident* errors [13]. Thus, the improvement of system reliability using this scheme is not as high as was expected.

Moreover, the assumption of failure independency was shown not to hold for a certain general application [26].

Different voting schemes can be found if the values voted upon can lay in a small interval of analogue values, i.e. slightly different values are accepted to be correct. Methods are to take the mean instantaneous value (the average) or an average weighted by a priori probability of input source reliability. Another possibility is to take the mean of the two most similar values. Yet another scheme, called pseudo voting, chooses the median of three signals [55].

State machines A general method for achieving fault tolerance and implementing distributed control in distributed systems is presented in [51]. It is based on N-modular redundancy of a module that implements the concept of a *state machine*. A state machine consists of *state variables*, which implement its state, and *commands*, which transform its state. Each command is implemented by a deterministic program; execution of the command modifies the state variables and/or produces some output. A *client* of the state machine makes a *request* to specify execution of a command. The request specifies the state machine, the command, and any information needed by the command. Outputs of a state machine are completely determined by the sequence of requests, independent of any other action in the system. Thus, they satisfy the deterministic security requirement.

The File System example defines a state machine with the operations "create" that both changes the state and returns a value, "read" that returns a value, and "delete" that changes the state. The Live Queue example defines a state machine with "enq" and "deq" operations.

A state machine is a general programming construct that can be implemented in various ways. It can be implemented as a collection of procedures that share data, as in a module; as an object that awaits messages containing the requests and performs the actions they specify; and as a collection of interrupt handlers, in which case a request is made by causing an interrupt.

A t fault-tolerant state machine (FTSM) can be implemented by replicating it and running a copy on each of the processors in a distributed system. The scheme for implementing a t fault-tolerant state machine is based on fault-tolerant implementations of two abstractions.

Agreement Every non-faulty copy of the state machine receives every request.

Order Requests are processed in the same order by every non-faulty copy of the state machine in a way that is consistent with potential causality.

Agreement can be implemented by any protocol that, whenever a client makes a request, disseminates this request to all copies of the state machine. Depending on what type of failure the t FTSM is to tolerate, $t+1$ (fail-stop failures), $2t+1$ (Byzantine failures with digital signatures), or $3t+1$ (Byzantine failures) replications are sufficient to mask t failures. Agreement protocols for the different failure models can be found in [14,49]

Order can be implemented by having clients assign unique identifiers to requests and having state machines process requests according to a total ordering relation on these identifiers. This ordering relation must be consistent with potential causality. In order to produce identifiers that satisfy the Order requirement either logical clocks or approximately synchronized real-time clocks can be used [27,48].

Ensuring correct output outside the system requires the two points of failures of this scheme, the voter and the output device, to be replicated. Each voter drives one output device copy. Thus, the critical voting on the output is pushed out of the system. If the output of a state machine is to a client inside the system, this client can vote on the outputs of state machine copies itself. The voter—a part of the client—is faulty exactly when the client is, so the fact that an incorrect output is read by the client due to a faulty voter is irrelevant.

Ensuring correct time-varying input of a client can be done by making a fault-tolerant sensor. The input source is replicated; the client that reads from the time-variant input source is restructured as a state machine *SM* and a collection of clients. Each client reads from a different copy of the input source; *SM* combines the values obtained from all the clients. If a client cannot be restructured as a state machine, *defensive programming* of a state machine, by adding tests and restrictions on its inputs, may limit the effects of faulty requests.

The configuration of a system consists of three sets of objects: clients, state machine copies, and output devices. If identification of faulty objects is possible, a higher degree of fault tolerance can be achieved by reconfiguration. A recovery mechanism uses a collection of clients, called *configurators*, one for each element of the system. Upon detection of a failure of its element, a configurator makes a request to alter the configuration. Adding and removing faulty elements is fairly straight forward. Integrating an element requires some information which can be obtained by storing it in state variables or by requesting it from identical elements. Synchronization of integrated elements is based on recent values of request identifiers and the local synchronized clock.

Multiple Modules A second construct to support fault tolerance using N-modular redundancy is described in [31]. It is based on a concurrent programming model consisting of communicating sequential processes (CSP). In particular, it addresses the problem that certain kinds of nondeterminism can produce inconsistency in majority voting even in the absence of faults. CSP allows the expression of two types of nondeterminism with respect to communication: *local nondeterminism*, which occurs when a process decides independent of other processes for which communication to wait. It is introduced by using Boolean guards; and *global nondeterminism*, which is resolved by inspecting the other processes with respect to their willingness to communicate. Only mutual willingness may result in communication. This kind of nondeterminism is introduced by using I/O guards.

A *Multiple Module* ${}^{n}P$ is a set of copies $P_1, \ldots, P_n$ of P with the following properties:

1. $P_1, \ldots, P_n$ all have the same specifications as P.
2. Associated with each P_i are copies of the input and output channels of P.
3. Each message sent from ${}^{n}P$ is obtained by majority voting on all the messages sent from its copies. Each message sent to ${}^{n}P$ is sent to all its copies.
4. All the copies $P_1, \ldots, P_n$ of a multiple module ${}^{n}P$ resolve nondeterminism (local or global) in an identical manner.
5. The copies $P_1, \ldots, P_n$ are physically independent.

A syntactical construct which indicates that a process P is n-multiple is given in the paper. Syntactically, a multiple module ${}^{n}P$ has the following form:

$$[\| P(1) :: CL \| P(2) :: CL \| \ldots \| P(n) :: CL \|]$$

where CL is the command list associated with each process P(i). The syntax expresses that the concurrent command will fail *only if* more than n/2 of P(i) fail (n is assumed to be statically fixed). The syntax of I/O commands can be expressed in term of concurrent commands.

The semantics of I/O-commands involving multiple modules are defined in the paper. The modular redundancy semantics considers a state to be a mapping from variables to values. A special state fail is used to denote a failing computation. Transition to fail occurs when no agreement is reached in voting upon the states of the copies of the multiple module ${}^{n}P$.

The semantics defined for multiple modules requires that every implementation of communication must satisfy the following conditions:

1. All copies of P resolve nondeterminism (local or global) in an identical manner.
2. All copies of P process the contents of the channels which are involved in any form of global nondeterminism in an identical order.

The violation of the second condition will generally result in the sending of different messages.

Two possible implementations are described. The first solution consists of using a particular interprocessor communication structure. It is based on indivisibility of processor communications which is obtained by using a parallel bus and requiring that the bus be released only after all copies of the receiver module have received the message. Condition 2 is satisfied by marking messages. The solution uses marks that indicate the moment of reception of a message. The marks imply a total ordering of the received messages from the various copies of a multiple module ^nP. The centralized communication structure assures the same ordering for all copies. This implementation of send is proved to satisfy condition 2. Nondeterminism is solved by the receiver(s) by processing the message with the smallest mark value.

In the second solution the processors run a decentralized agreement protocol on the value of the mark which must be assigned to each received message. A signed message algorithm of interactive consistency is used which is periodically started by each of the receiving processors. It determines the set of messages already received but not yet inserted into the actual channels of the multiple module ^nP. This solution is proved to satisfy condition 2. Nondeterminism is solved by choosing the guard referring to the channel containing the message that precedes all other messages according to the marking order. In contrast to the previous solution that imposes a fixed nondeterministic strategy, this solution allows for the definition of a suitable nondeterministic strategy.

Both solutions allow N-version programming to be incorporated. Neither solution involves synchronization of the replicated processes and/or planned scheduling.

Replicated Procedure Calls A third construct for implementing masking redundancy combines remote procedure calls with replication of program modules [12]. The set of replicas of the module is called a *troupe*. In a program constructed from troupes, an intermodule procedure call results in a replicated procedure call. A distributed program will continue to function as long as at least one member of each troupe survives. Communication between troupes is based on á paired-message protocol which provides message sequence numbers that uniquely identify each pair of messages among all the ones exchanged by a pair of troupe members.

A *client troupe* represents a replicated caller, a *server troupe* implements a replicated procedure. A client troupe member does a one-to-many call to the server troupe, and a server troupe member handles a many-to-one call from the client troupe. Each member of the client troupe receives the results of all server troupe members. To distinguish unrelated calls from calls that are part of the same many-to-one call, the concept of a *distributed call stack* is introduced. An entry of this stack consists of a set of contexts, one from each member of the troupe implementing the called procedure. The stack is used to trace back the identifier of the troupe (the root troupe) that initiated the chain of calls. Two messages that are part of the same replicated call have the same root identifier. A set of related messages is reduced into a single result by a *collator*, a function that checks the equivalence of messages and raises an exception if necessary. This structure can implement any kind of voting mechanism.

To implement concurrent replicated calls some ordering of messages must be enforced that ensures the same ordering at all server members. This may be achieved by any concurrency control mechanism which can be properly coordinated with replicated procedure call, such as nested atomic actions in combination with a two-phase locking protocol.

Crashes are assumed to be detected by the paired-message protocol (timeout). Replacement of crashed members might be done by a copying variant of process migration (DEMOS) or using checkpointing to stable storage.

Nested Transactions An extension of the concept of atomic transaction is proposed in [36]: nested transactions that allow concurrency within as well as among transactions, and enable error recovery after a failure occurred. A traditional transaction now can expand to a tree of transactions. Transactions from the same level in a tree are synchronized among themselves using read-write locking. Resources are locked by transactions, and a parent transaction inherits the locks as its children complete. If a child fails, only little recovery has to be done because modifications are made permanent only when a top-level transaction completes. The parent can implement recovery of the child's action by retrying or alternate actions. A method for recovery of nested transactions is **shadowing.** Whenever a transaction begins to hold a lock for an object, a *backup* or *shadow* copy of the object is made. The transaction operates on one copy, and another copy is available for re-installation. Thus, a number of different versions of an object exists, one for each transaction holding a write lock for the object.

In a distributed environmer', management of nested transactions can be accomplished by some extra bookkeeping, oy extending two-phase commit to coordinate the saving of recovery information, ar by detecting and aborting orphan transactions. Policies for locking objects that have N distributed copies are described in [4].

Dynamic Redundancy Fault-masking techniques improve system reliability by allowing a system to operate correctly in the presence of failures, but it is limited by its static configuration: a system employing a fault-masking technique cannot heal itself, but only hide its failures. In contrast, dynamic redundancy techniques involve *reconfiguration* of the system in response to failures. Reconfiguration requires information about the global system state. A way to access global state information is presented in [10]. Further, reconfiguration requires the ability to correctly locate faulty objects. This is often achieved by some form of comparison, and also by periodic testing of objects. A method to implement the latter which uses a watchdog processor is proposed in [63]. Fault-masking can be part of a dynamic redundancy scheme and allows postponing of reconfiguration and repair actions until faults threaten to become unmaskable because of the degradation of the system.

One of the drawbacks of N-modular redundancy with voting (NMR) is that fault-masking ability deteriorates as more objects fail. The faulty replications eventually outvote the good ones. However, an NMR system could continue to function if the known bad objects could be discounted in the vote. Two methods of reconfiguration based on NMR, called **Reconfigurable N-Modular Redundancy** techniques, realize this potential.

The first, **hybrid redundancy,** replaces failed replications with previously unused spares. At any time N replications are used, with their output voted upon to produce the component output. When a disagreement is detected, the objects in the minority are considered to be failed and are replaced by an equivalent number of spare objects. As long as there are never more than $\lfloor N/2 \rfloor$ failed objects before reconfiguration can take place, the component can tolerate the failure of $P = (t + S)$ of its objects, where S is the number of spare objects. This technique has to solve the problems of detecting the faulty objects, integrating spare objects into the system when necessary, and synchronizing these new objects.

The second method is to modify the voting process dynamically as the system deteriorates. This technique is called **adaptive voting**. Each voter input n_i is weighted by a certain factor a_i. In the pure form of adaptive voting, the decision is based on the sum $\Sigma a_i n_i$, using a threshold detector. The a_i are modified over time by the accumulated history of disagreements and fault detection. In practice the a_i are usually zero or one. Two adaptive voting techniques are the following. When one object fails, it and another object are removed from the system, leaving an $(N - 2)$ modular redundancy system, which preserves the property that no tie is possible in voting. The other possibility is found in systems that provide nonvoting mode operation. Upon failure the system switches to a lower-redundancy scheme. It can repeat this until it has reached the state that only one object is left. Of course, the system reliability degrades accordingly unless faulty objects are repaired and reintegrated into the system.

Software Implemented Fault Tolerance (SIFT) The SIFT system is an ultra-reliable computer for critical aircraft applications that achieves fault tolerance by the replication of tasks among processing units [66] and a hybrid redundancy mechanism. The system executes a set of *tasks*, each of which consists of a number of *iterations*. Reliability is achieved by having each iteration of a task executed independently by a number of sites. Then, a two-out-of-three vote on the outcomes of the replicated iteration determines the input of the next replicated iteration. Optimization is achieved by voting on state data only at the beginning of each iteration. This implies that communication only needs to be loosely synchronized. An advantage of this scheme is that it is less likely for simultaneous transient errors of iterations at different sites to result in correlated failures. The number of replications of a task varies with the task and with time, depending on how critical the task is. Allocation of tasks is done dynamically by a task called the global executive. This task is replicated at each site and diagnoses failures. Based on this knowledge it decides if an object has become faulty and reconfigures the system. Further, a task called the local executive takes care of local error handling, scheduling, and voting. Other fault-tolerance mechanisms as N-version programming and recovery blocks may be incorporated into the system.

Formal specifications of the SIFT executive software have been written using the SPECIAL language developed at SRI [44]. A hierarchy of models is used to express the different aspects of correctness. The SIFT system itself may be viewed as the lowest level model. A higher-level model then must be proved to accurately describe the next lower-level one.

A model consists of a set of states and a transition relation that allows nondeterministic transitions. System behavior is expressed by a sequence of states. Concurrency can be represented by transitions that change disjoint components of a state, so that the order in which they occur is irrelevant.

In the *reliability model* a state (h,d,f) represents the number of handled, detected, and occurred failures respectively. Associated with each value of h is a *safety factor* SF(h) which reflects the additional number of failures the system can successfully cope with. A *safe state* is one in which $f - h \leq SF(h)$. Two properties are proved to hold which ensure that SIFT meets its reliability requirements.

1. If the system remains in a safe state, then it will behave correctly.
2. The probability of the system reaching an unsafe state is sufficiently small.

In the *allocation model* a transition represents the execution of one complete iteration of all tasks. The correctness property of SIFT—property 1 above—, for this model expressed in terms of I/O correctness, is proved to hold and the correctness proof for this model is derived. Further, the correspondence between the two models is shown.

Fault-Tolerant Multiprocessor (FTMP) Another system that implements fault tolerance is the fault-tolerant multiprocessor (FTMP) described in [24]. This system also uses hybrid redundancy, but voting always involves three objects. This is referred to as *Triple Modular Redundancy* (TMR). FTMP has adopted a fully synchronous approach to communication, which allows system management to be effected by majority rule. Objects in the system can be retired from participation in a triad and given the spare status, and/or reassigned in any triad configuration under executive control. Reconfiguration is carried out periodically to search for latent faults. Failure detection and correction is done by the voters which also identify the faulty objects. Recovery involves assignment and startup of a spare object to the triad that discovered a failure. As a second level of fault tolerance a rollback/restart mechanism is provided.

FTMP has several failure models, each of which is amenable to a different mathematical tool. Survival probability is modeled as a a set of Markov processes, whereas the probability of failure due to exhaustion of spares is modeled using combinatorial methods. An extensive analysis of the system has been done using the different models. This showed that for random hard failures the system meets its requirements. However, not much is stated on how these models can be validated.

Restart/Recovery As was mentioned above, recovery techniques can be used in combination with replication, but also as stand-alone techniques. Recovery techniques have two major approaches: *backward error recovery* and *forward error recovery*. Backward error recovery mechanisms correct the system state by restoring the system to a state which occurred prior to the manifestation of the failure. A system structure that supports backward error recovery is the *recovery block scheme.* Forward error recovery techniques aim to identify the failure and, based on this knowledge, correct the system state containing the failure. *Exceptions, signal-* and *raise-operations*, and *exception handlers* are common mechanisms to provide forward error recovery. Also combinations of the two approaches are proposed.

Many techniques for supporting fault tolerance have used the property of *atomicity.* Recoverable atomic actions conform to the "all or nothing" view, which requires that either all the modifications made by the action reach their final state, or no modifications are made at all. Recoverable atomic actions specify that both indivisibility and recoverability are fundamental requirements for atomicity.

We will describe a number of restart/recovery techniques in the rest of this section. First, forward recovery schemes are presented of which there are not many. Second, backward recovery schemes are discussed.

Forward error recovery schemes The word 'forward' refers to the fact that recovery does not involve a previously reached system state, but that control is transferred to a special action that will try to correct the system state. Forward error recovery involves the following mechanisms. Failures are detected by entering unusual states that will signal or raise exceptions. The action that deals with an exception is called an *exception handler.* If several exceptions are raised concurrently, an exception resolution scheme selects a single exception to represent the combination of the exception conditions. For this purpose, exceptions are organized into a tree in which the upper bound is the universal exception.

Atomic Actions Atomic actions, as a tool for controlling interactions when shared data are accessed, have been studied extensively. In [62] an extension of this concept is proposed which allows concurrency within atomic actions, execution of a single atomic action at different sites, and forward error recovery within an atomic action. An atomic action containing concurrency is structured as a (possibly changing) collection of processes, each of which is executing a sequence of subordinate atomic actions. This structuring implies an ordering constraint between the subordinate processes that belong to a single process. The scheme presented associates one process with each atomic action. Such a process can create concurrent processes for subordinate atomic actions. To avoid the potential problems of this scheme (resource locking and process failure), the parent process is severely restricted during the execution of concurrent actions it has created. It takes the role of 'traffic director' and cannot access shared data nor can it execute nonelementary subordinate actions unless they are to execute concurrently. If these restrictions are enforced, then neither locking of resources nor exception handling present any special problems.

To implement forward error recovery, exception handlers to be called upon the failure of a certain function are specified in the function header. An exception is handled only after all subordinate concurrent actions have completed or also signaled an exception. Multiple exceptions are combined using an exception tree mechanism. Alternatively, subordinate atomic actions may be terminated by raising a special abort exception which ignores all other exceptions, rather than waiting for them to complete normally.

An arbitrary mechanism can be used to implement atomic actions. Locking is proposed for synchronization. On invocation of a subordinate concurrent action it is necessary to check precedence constraints, and hold the action pending if some are unsatisfied. When all local constraints are satisfied, a local process may be started for the action. If the action is to be executed at another site, a message containing the information needed to perform the action is sent to the destination site. This site is added to the *remote list* of the action. There, a tree is constructed for the action containing information on all its ancestors for termination purposes. After checking local precedence constraints, the concurrent action can be started. On termination, the parent action is notified, which may involve sending a message that includes the remote list of the termination action. If so, this list is merged into the remote list of the parent process. Information on locks is passed this way. When a top level action terminates all changes must be made permanent. This is done using a two-phase commit protocol.

Exception handlers are implemented as atomic actions, and thus are handled the same way. Backward error recovery can be incorporated easily by adding alternates to the atomic action. As no changes are made unless the atomic action commits, no special state saving has to be done.

Backward error recovery schemes All forms of backward error recovery require some redundant object-state information to be recorded as an action executes. The information is used to roll back an interrupted action after a failure has occurred to a point for which correct state information is known. Three forms of backward error recovery are considered: retry techniques, checkpoint techniques, and journaling techniques.

Retry techniques are the fastest form of error recovery, and conceptually the simplest. They are most commonly employed to tolerate transient failures. Immediately after an error is detected repairs are effected. This consists in pausing in the case of a transient failure, and in reconfiguration in case of a hard failure. Then, the action is retried. This necessitates knowing what the system state was before the action was first attempted. If the interrupted action had already irrevocably modified some data, the retry will be unsuccessful. A variation of this technique that also tolerates hard failures implies retrying with a different algorithm that performs the same function.

A method that tries to recover from transient failures by multiple retries is described in [63]. Ways to derive the number of retries for a given program are discussed, and the outline of an algorithm for insertion of rollback points is presented. The method uses a watchdog processor to initiate recovery actions through rollback.

In contrast to retry techniques, **checkpoint techniques** allow some failure latency, for the action is backed up to an earlier point in its execution. Some subset of the system state is saved at specific points (the checkpoints) which is necessary to the continued successful execution and completion of the action past the checkpoint, and which is not backed up by other means. Rollback is part of the actual recovery process and consists in resetting the system and action state to the state saved at the latest checkpoint. Hence the only loss is the computation time between the checkpoint and the rollback, plus any data received during that interval that cannot be recreated.

Important issues in checkpointing design are the following: the frequency of checkpoints which determines the length of rollbacks and the amount of overhead of saving system states; minimization of the amount of state information to be saved at each checkpoint; deciding which information is important for rollback actions; validation of saved state information; and the prevention of the *domino effect* [41]. The latter situation arises in systems with multiple concurrent processes communicating with each other. If one process is rolled back, any other process that receives data from it since the checkpoint must also be rolled back. If these processes in turn communicate with others, it requires them to roll back too, thus giving rise to the domino effect.

Journaling recovery is the simplest and least efficient method. A copy of the initial data is saved before the action starts. During execution of an action all transactions that affect the data are recorded. If a failure occurs, the action can be recreated by running copy of the saved data through the transactions. Journaling is better than completely restarting because it eliminates the loss of information involved in a restart.

Recovery Blocks A construct for backward error recovery is the *recovery block*. A recovery block consists of a conventional block (like in ALGOL or PL/I) which is provided with a means of error detection, called an *acceptance test*, a checkpoint mechanism, and zero ore more stand-by spares, called *alternate algorithms*. Alternates perform the same action in a different way or they are simpler and produce acceptable, though less desirable, results.

The *primary alternate* corresponds exactly to the block of the equivalent conventional program, and is entered to perform the desired action. The initial state is saved in the checkpoint prior to any modifications. The *acceptance test*, which is a logical expression without side effects, is evaluated on exit from any alternate to determine whether the alternate has performed acceptably. A further *alternate*, if one exists, is entered if the preceding alternate fails to complete, or fails the acceptance test. Before an alternate is so entered, the state of the process is restored to the state saved in the checkpoint. If an acceptance test is passed, any further alternates are ignored, and the statement following the recovery block is the next to be executed. If the last alternate fails to pass the acceptance test, then the entire recovery block is regarded as having failed, so that the block in which it is embedded fails to complete end recovery is then attempted at that level.

Checkpointing is done using a mechanism that saves nonlocal variables in a *recursive cache* just before they are first modified. Intermediate checkpoints can also be used inside a block to recover from transient failures.

In an environment with concurrent communicating processes the problem of the domino effect must be solved. A technique that deals with this problem structures process interactions into **conversations**. It involves a recovery block structure which is common to a set of processes. Within a conversation, processes may communicate freely between themselves, but may not communicate with any other processes. At the end of the conversation all the processes must satisfy their respective acceptance tests and none may proceed until all have done so. If any process fails, all processes must be backed up to the start of the conversation to attempt their alternates. As with recovery blocks, conversations can be nested and have additional error detection and recovery possibilities.

The fault tolerance technique of N-version programming can be used in specifying the alternates.

A technique that combines aspects of recovery blocks and N-version programming is the **Consensus Recovery Block** technique [52]. It uses independent versions of an algorithm, an acceptance test, and a voting procedure. The versions are ranked according to some "goodness" criteria. Upon invocation of the consensus recovery block, all versions are executed and submit their outputs to the voter. If no agreement is reached, a modified consensus block is entered. No input state recovery is needed since all versions execute concurrently.

Optimistic Recovery A journaling scheme [57] is based on dependency-tracking which enables computation, communication, checkpointing, and committing to proceed fully asynchronously. Dependency-tracking of a process on each other process's messages allows to detect if other processes have performed any computations which causally depend on messages which a failed process has lost. These computations can be undone by rolling the process back to the latest state which does not depend upon lost messages. Thus, arbitrary states can be recovered by restoring a checkpoint using the tracked dependency information and then rolling forward by replacing the appropriate number of input messages. In this scheme, a process cannot roll back too far and thus avoids the domino effect.

Fault-Tolerant Atomic Actions A construct for fault tolerance which uses both forward and backward error recovery is proposed in [25]. It supports fault tolerance in a system of communicating sequential processes (CSP) and is based on atomic actions. The construct is called a Fault-Tolerant Atomic Action (FT-Action). It is a distributed control structure that a group of processes may join or leave in synchrony. Communication is restricted to occur between processes in the FT-Action only. The FT-Action has the following properties:

1. Atomicity.
2. A recovery line for backward error recovery. It is established by the synchronized entry of all participating processes, and by recording a checkpoint for each process.
3. A test line for processes which contains a test for each process that determines if any failures have occurred. The exit statements together form a test line.
4. Recovery measures to be used inside the FT-Action. If one failure is detected, all processes must take appropriate recovery measures.
5. Nesting of FT-Actions that allow for hierarchical recovery techniques.

Backward error recovery constitutes a conversation [41] between the participating processes. Every process executes its primary and its acceptance test. If one process fails, an exception is raised, and every process invokes backward error recovery by executing the next alternate. All processes in the FT-Action are required to have the same number of alternates. Failure of one of the acceptance tests in the last alternate results in a signal that the FT-Action has failed. Forward error recovery is invoked if an exception is raised during execution of the FT-Action. The exception is notified to all participating processes, and each then will execute its handler for that exception. Recovery completes either when the handler is executed successfully or when a new exception is signaled to a containing FT-Action. In the latter case the FT-Action fails which may result in error recovery at a higher level. Exception collisions are solved by a resolution scheme. When applied to a raised and a signaled exception, the scheme ensures to raise a failure exception.

The *combined recovery mechanism* proposed associates a forward error recovery scheme with the primary algorithm which is invoked for the specified exceptions. The backward error recovery scheme would be invoked for other exceptions and any exceptions that might occur in the handler. However, many other ways to combine the two mechanisms exist.

Implementation of FT-Actions employing CSP primitives for communication and synchronization uses a voting technique based on a two-phase commit protocol [4]. Voting is done to check consistency of the list of each process which specifies the other processes that are allowed to participate in the FT-Action. The voting is implemented by message passing up and down the chain of the processes that try to enter the FT-Action. A similar scheme is used at the exit of a FT-Action to decide whether an exception has been detected. A timeout mechanism use used to detect *deserter processes*, which will result in the abortion of the containing FT-Action.

1.5.3 Design Issues

The model introduced in the first chapter presents a view of a distributed system as a collection of objects that can communicate only by message passing. This model nicely matches fault tolerance requirements since:

1. Defining and limiting the way objects interact allows an easier control of several problems concerning interactions.

2. Modularity is made easier and modifications inside a module can be easily performed, if no changes to the external interface of the module are made.

3. Failures can be prevented from spreading out to other objects more easily.

4. It allows the use of consistency checks among objects, building a logical level of error detection.

These attributes are very important, since the possibility of writing concurrent programs in a modular fashion is very useful for correct software design and modification. Also the possibility of structuring resource management and fault treatment with no central coordination may enhance reliability by eliminating physical as well as logical single points of failure [31].

The design of a construct that supports fault tolerance requires the following issues to be considered.

General issues

- **The failure model**: The type and the amount of failures that the construct is to tolerate must be investigated. In order to determine these issues, a study of existing software reliability models may be included. It may also depend on how critical the application will be.

- **Failure detection**: One or more ways to detect failures must be provided. In most cases, some level of failure detection will be incorporated into the fault-tolerant structure. Also, a method to diagnose failures may be necessary for classifying and locating failures.

- **Fault confinement**: The construct that supports fault tolerance must also allow for fault confinement. This can be achieved by modular design which implements protected environments, combined with masking techniques.

- **Verification**: The correctness of the scheme must be proved. This involves specification of syntax and semantics of the construct. It also involves proving that these specifications are consistent with the specification of the system model. Furthermore, the implementation must be proved to meet these specifications, and it must be verified that the implementation meets the reliability requirements of the fault tolerance mechanism.

- **Costs:** An evaluation of the costs involved with supporting fault tolerance must be made. It involves administration of memory and processing-time overhead which is added as a result of supporting the scheme.

Replication

When using replication as a means to support fault tolerance the following issues must be addressed:

- **Number:** It must be decided if a copy of a replicated object must reside on every site in the distributed system, or on only a subset. This will depend mainly on how critical the function of the object is. Further, there is a choice of using all copies in implementing the fault-tolerant function, or of assigning a spare status to some number of them. If copies can be added dynamically a mechanism to appropriately create and delete copies must be provided.

- **Allocation:** If the number of copies is less than the number of sites in the system, an optimal allocation of these copies must be found, for either static or dynamic allocation. In case of dynamic redundancy, the method must also handle the optimal allocation of spare objects.

- **Identification:** A mechanism that keeps track of the copies of an object is necessary in order to address messages. Also, a way to uniquely identify objects is needed for this purpose and in order to be able to identify faulty objects.

- **Consistency:** Consistency of the data associated with the copies of an object must be ensured. Protocols to agree on the value of a certain object, and ways to lock an object for exclusive access may be needed to achieve consistency.

- **Synchronization:** In order to determine the single output of a replicated function by a voting mechanism, the participating copies must be at least loosely synchronized.

- **The treatment of nondeterminism:** Fault tolerance requires that all copies of a replicated object resolve nondeterminism in an identical manner. Having a method that satisfies this requirement is essential in supporting fault tolerance in a system with concurrent objects which communicate by message passing.

- **Communication:** The fact that one or more objects participating in a communication may be replicated to support fault tolerance must be a transparent feature of the system. In order to achieve this, the I/O commands of objects must be able to handle different forms of communication. Two basically different communication mechanisms can be distinguished:

1. If the destination object is replicated, a message from the source object must be replicated and send to each destination copy. This is called a *one-to-many* communication. A method to perform this automatically must be provided. This method must also ensure that the copies of the message can be identified as belonging to the same communication in order to be able collect the response on the replicated message later.
2. If the source object is replicated, messages from the replicated object must be collected into a single message before it reaches the destination object. This is referred to as a *many-to-one* communication. This method must be able to distinguish between a message that is part of a replicated message and a single message. Furthermore, it must have a way to distinguish between related and unrelated messages to allow collecting and combining of replicated messages.

Supporting fault tolerance by means of replication requires that all copies of an object handles messages in an *identical order*. Moreover, this order must preserve potential causality. Thus, a mechanism that imposes a total ordering upon message identifiers must be provided.

Recovery

Recovery in response to a failure requires to have available some information on a state that is unaffected by the failure. The following issues must be included to support recovery:

- **State information**: First, a way of saving information in nonvolatile or stable storage (storage which is not affected by any failures) is needed. Second, it must be decided which information is critical for the recovery and restart of a faulty object, and thus must be saved. Of course, the amount of information saved should be as small as possible. Third, the time and frequency of saving state information must be determined. These issues influence the length of rollbacks and the amount of overhead involved with saving states. Fourth, it must be ensured that the state information saved is valid. This implies that the information must be saved in a way that prevents the incorporation of any effects of failures in the information saved. Finally, the problem of an uncontrolled rollback of processes, the domino effect, must be solved.
- **Handling faulty replicas**: If an object is detected to be faulty, a way of handling this object must be specified. In a system that will provide masking redundancy only, the faulty object can be excluded from participation, which results in a degradation of the system. Eventually, the whole system will fail. Thus, in general, it is advisable to support some form of dynamic redundancy also. Dynamic redundancy allows reconfiguration of the system in response to failures. It requires the availability of global system information on the status, the location, and the interaction patterns of objects. The following steps are involved in handling faulty objects:

1. Locating the faulty object.
2. Excluding the faulty object from participation in the support of a fault-tolerant function.
3. Replacing the faulty object by a spare one.
4. Notifying all other copies of the object of these changes.
5. Integrating the spare object by providing it with the appropriate information. This implies that startup information must be available. Further, the new object has to be synchronized with the other objects that implement the fault-tolerant function, i.e. it has to be placed in the same state of execution.

It is necessary to understand all of the issues mentioned above in order to design tools that will aid in the implementation of a mechanism to support fault tolerance in a secure distributed system.

1.6 Real-Time Requirements

Many secure distributed systems have performance constraints. These constraints will usually have the form "event e' must occur within t time units after event e". For example, following a disk interrupt a new disk transfer should be initiated before the head passes the start of the next block to be read. Also, the report of an abnormal event by a sensor on a rocket must be handled before the event causes the mission to fail.

Performance constraints must be feasible. The time interval t must be large enough to allow the processing required for e' to occur. This processing may require a sequence of messages to be passed among the various objects that must cooperate to perform the processing. Meeting performance constraints becomes harder when there are several such constraints that compete for system resources and the use of objects.

Situations may sometimes occur in which not all of the constraints can be met. For example, it may be possible to respond to a disk interrupt or to a sensor report, but not to both, within the required time. Priorities should be specified so that resources will be allocated to meeting the more important constraints. Efficient use of the disk can probably be sacrificed when an abnormal event has been sensed. Messages or channels can also be given priorities, so that an object can handle messages in order of their importance rather than their arrival time. Note that the use of priorities may cause fairness assumptions to be violated. When constraints cannot be met, a failure action should be specified.

When a message that must be handled quickly arrives at an object, the object may already be busy. In most computer systems today, when a peripheral sends a message to a busy processor, the processor is interrupted. Such a solution could also be used with objects. However, the object can be in the middle of a long modification of its state when the interrupt occurs and its state may be inconsistent.

The state of an object can be guaranteed to be consistent when an interrupt arrives if the state is never modified after the object is created. To change the state, a copy of the object containing the changes is created. This approach can be used in the file system example where files can be added and deleted from a copy of the file system while a file is being read.

Much of the copying can be avoided if the state is implemented as a linked structure. Only the parts that are being changed need to be copied. Links are used be point to the unchanged parts. For the file system, only the segments of the directory that are modified will be copied. Deleting a file is done by removing the link to it from the directory and creating a file is done by adding a link.

If several objects must stay consistent with each other and there are several concurrent requests to update the set of objects, accesses to these objects must be synchronized. Version control using timestamps [42] can be used. This topic is discussed further in the companion report on database consistency.

1.6.1 Time

Time has been ignored so far in this report. It is unimportant when proving safety properties or when proving liveness properties of the form "eventually event e will occur". The order in which events occur (the possible traces) is the primary consideration. When discussing performance constraints, however, time is important. Timing information can be added to traces [43], as explained in the next chapter.

When describing a system in terms of its traces, each event is treated as a single point in time. A trace defines a total ordering on the events. In an actual system, events have a duration and two events can overlap in time. In most cases, this will not be a problem. The time of an event can be identified with its start. Overlapping events are either the send and receive of the same synchronous communication and are represented by a single event in the system trace, or they have no effect on each other and are *serializable*. Serializable events produce the same result when one is performed before the other as they do when they are performed concurrently.

The amount of overlap of events is important to describe the speedup due to concurrency or to determine the effect on performance of a change in system resources. A semantics for CSP which models true concurrency has been developed [61]. Each element of a trace in this semantics is a *step*, or multiset of (possibly null) events.

In a distributed system, measuring time intervals is more complicated than in a sequential system[27]. If there is a single global clock, each of the various nodes will experience a different message delay when reading the clock. If each node has its own clock, the rates of the clocks will vary slightly. The clocks can be kept close to the same time (provided that the variance in the message transit time is small). However, there is an inevitable error in measuring time across several nodes of a distributed system.

1.6.2 Specification and Verification

The time required for a sequence of events can only be determined at a very low level of implementation. It depends on which high-level language constructs are used to implement the events, the compiler's translation of those constructs, and the hardware on which the events are executed. The time for a message transfer depends on the communication system and how busy it is. Therefore, performance requirements cannot usually be proven for a specification.

What can be done is to specify performance requirements that must be satisfied by the hardware, the process scheduler, and the resource allocators. The requirements may include maximum, minimum, or mean timing requirements and reliability requirements [67]. The specification of each event is extended to include an error action in case its performance requirement is not met. The effect of such extended events is nondeterministic. It also may be possible to prove that the specification is feasible if all of the performance requirements are met.

Real-time requirements can have a negative impact on security in several ways. Being able to read the time and to measure the duration of a sequence of events may allow a process to infer that it was delayed by high security-level activity. Also, high security-level activity may cause a low security-level activity to miss its performance requirements. Thus, the clock that a process reads must be suspended while higher-level processing occurs.

Chapter 2

Formalisms

2.1 Temporal Logic

Temporal logic [32,39] provides a formal way to express and to reason about properties that vary over time. By modeling a program or system of concurrent processes as a sequence of states, temporal logic can be used to specify and verify properties of such programs and systems. This presentation will be informal and brief. The references give a more comprehensive treatment of the subject.

2.1.1 The Logic

Temporal logic, which is a kind of modal logic, is not any *bigger* than ordinary first-order logic. That is, whatever can be expressed and reasoned about using temporal logic can also be expressed and reasoned about using ordinary first-order logic. The advantage of temporal logic is that it offers a concise notation for expressing time-varying properties by eliminating any explicit reference to time, i.e., time does not appear as a parameter. The rules of inference of temporal logic are derivable from the rules of inference of ordinary first-order logic, but allow shorter proofs.

The idea is that there is a sequence of elements, in which each element represents a state and the order of the states within the sequence corresponds to the order in which the states occur over time. A basic premise of temporal logic is that time is discrete. A temporal logic formula makes an assertion about the *current* state, which may be chosen arbitrarily, and the rest of the sequence. If there are no temporal operators in the formula, the formula is indistinguishable from a formula in ordinary first-order logic and it makes an assertion about only the *current* state.

Temporal logic adds to first-order logic the temporal operators $\bigcirc$ (next), $\Box$ (always), $\Diamond$ (eventually), and $\mathcal{U}$ (until). $\bigcirc$ asserts that its formula is true in the state immediately following the current state. $\Box$ asserts that its formula is true in the current state and in all states following. $\Diamond$ asserts that its formula is true either in the current state or in some state following. $\mathcal{U}$ is a binary operator; "P$\mathcal{U}$Q" asserts that Q is true for some state *s*, which is either the current state or a state following the current state, and P is true for every state from the current state until (but not necessarily including) state s. Temporal operators may be nested, which has the effect of assigning a (possibly) new current state for the nested temporal formula. Here are some examples:

> "$P \Rightarrow \Box P$" is a temporal formula which says, "if P is true for an an arbitrary state s, then for every state following s, P is also true."
>
> "$\Diamond(P \ \wedge \ \Box Q)$" says, "There is a state s which satisfies $P \wedge Q$ and for which every state following s satisfies Q."
>
> "$P \Rightarrow \bigcirc\neg P$" says, "Whenever P is true, then in the next state, P is false."

Some time-related properties require looking backward in time rather than, or in addition to, looking forward in time. To use temporal logic to express and reason about such properties, it can be extended with the *previous-time operators*: $-\bigcirc$ (in the immediately preceding instant), $-\Box$ (always in the past), $-\Diamond$ (sometime in the past), and $-\mathcal{U}$ (since). Additional rules of inference, again derivable from the rules of inference of first-order logic, allow reasoning about these kinds of properties. Example:

> "$P \ \Rightarrow \ -\Diamond Q$" says, "If P is true for an arbitrary state s, then for some state preceding s, Q is true".

2.1.2 Program Models

Several models of program execution have been developed which are compatible with using temporal logic to express and prove program properties. The kinds of properties that have motivated using temporal logic are those that have been previously called *progress* properties in this report: deadlock, starvation, liveness, etc. Since these properties are most interesting in the context of concurrent programs, the models encompass programs consisting of cooperating concurrent (or interleaved) processes. A model and variations are presented here which are representative of the models that have been developed.

A concurrent program consists of one or more processes, running in parallel. There is a set of program variables, which may or may not be shared among all the processes, depending on variations in the model. Each process is an independent transition graph with labeled nodes (locations) and edges (transitions). Each edge in a process transition graph is labeled by a pair consisting of an *enabling condition* and a *transformation*. The enabling condition is a predicate on the program variables and the transformation determines how the values of the program variables in the next state are derived from the values in the "current" state. A single node may have several edges emanating from it, in which case the enabling conditions need be neither exhaustive nor exclusive. Thus, deadlock and nondeterminism are allowed for single processes.

The Manna-Pnueli model [32] is a *shared-variables* model: the program variables are accessible and shared by all the processes, and provide for communication and synchronization among the processes. The program state consists of a set of values for the program variables and a set of nodes (locations), one from each process, indicating the *current* node for each process. Nonsensical programs, composed of processes whose edges specify conflicting values for the program variables, are avoided by assuming interleaved execution for the processes, i.e., the current program state and the next one differ in location values for no more than one process. Consequently the edge labels are considered indivisible (atomic).

2.1.3 Temporal Logic Applied to Program Properties

An execution sequence is a sequence of program states, each non-initial state derivable from the previous one through an enabled transition effected for no more than one process. The behavior of a concurrent program is characterized by its set of (possible) execution sequences. Temporal logic, which is interpreted over sequences, can be used to state and prove properties of concurrent programs.

For example, consider the LIVEQ examples used earlier to introduce the various progress properties. In this section, we will model a system composed of a producer, queuer, and consumer, and specify the various progress properties using temporal logic. Below, a variable name preceded by a single quote refers to the value of the variable in the state immediately following the current state. If a program variable does not appear quoted in a transition, it is understood that the value of the variable is unchanged by the transition.

```
Constants:          max   (maximum # of items to be held in queue)
Program variables:  itemp  (item produced by the producer)
                    ?itemp (is available for queueing?)
                    itemc  (item consumed by the consumer)
                    ?itemc (is available for dequeueing?)
                    queue  (array of items)
                    size   (# of items currently held in queue)
Define enq(x) by 'size = size + 1 &
                 'queue['size] = x &
                 ~ '?itemp
Define deq(x) by 'size = size - 1 &
                 '?itemc &
                 'itemc = x

                    ~ ?itemp -> '?itemp & exists x. 'itemp = x
Producer --  Node P ---------------------------------------->|
                ^                                            |
                |____________________________________________|

                    ?itemp & size < max -> enq(itemp) & ~'?itemp
                   ------------------------------------------------->|
                  /                                                  |
Queuer --  Node Q                                                    |
              ^   \                                                  |
              |    ------------------------------------------------->|
              |     ~?itemc & size > 0 -> deq(queue[size]) & '?itemc |
              |                                                      |
              |______________________________________________________|

                     ?itemc -> ~ '?itemc
Consumer --  Node C ------------------------>|
                ^                            |
                |____________________________|


Program state: (itemp,?itemp,itemc,?itemc,queue,size,Ploc,Qloc,Cloc)
Initially:     size = 0 & max > 0 & ~?itemp & ~?itemc &
               Ploc = P & Qloc = Q & Cloc = C
```

Specification of Deadlock and Liveness

The system is stopped if each of its processes can proceed no further, i.e., if none of the enabling conditions on the edges emanating from the current nodes of the processes are met. After defining a predicate *?stopped*, system deadlock and system liveness for the LIVEQ example can be specified as follows:

$?\text{stopped} = ?\text{itemp} \wedge \neg(?\text{itemp} \wedge \text{size} < \text{max}) \wedge \neg(\neg ?\text{itemc} \wedge \text{size} > 0) \wedge \neg ?\text{itemc}$
System deadlock: $\Diamond ?\text{stopped}$
System liveness: $\Box \neg ?\text{stopped}$

A process can proceed only when one of its current enabling conditions is eventually met. Liveness, absence of deadlock, and absence of livelock (absence of starvation) are largely indistinguishable in a shared-variables model. Liveness and deadlock cannot be specified for a process out of the context of the rest of the system, since the values given by other processes to the shared variables determine whether any of the enabling conditions for a process are met.

Liveness for the producer: $\Box\Diamond\neg ?\text{itemp}$
Liveness for the queuer: $\Box\Diamond((?\text{itemp} \wedge \text{size} < \text{max}) \vee (\neg ?\text{itemc} \wedge \text{size} > 0))$
Liveness for the consumer: $\Box\Diamond ?\text{itemc}$

Responsiveness and Conservativeness

A responsiveness property for the system is, "Every item enqueued eventually gets dequeued." A conservativeness property is, "Every item dequeued was previously enqueued at some previous time." Both these properties are straightforward to specify using temporal logic.

Responsiveness: $\forall x.\ \text{enq}(x) \Rightarrow \Diamond\text{deq}(x)$
Conservativeness: $\forall x.\ \text{deq}(x) \Rightarrow -\Diamond\text{enq}(x)$

Fairness

In a model with interleaving, fairness is an assumption that must be made to ensure that each process gets a chance to execute.

Divergence

Divergence occurs when a process, whenever given the opportunity to execute, chooses only to make a transition that has no effect on the rest of the system. It is not clear whether divergence can be satisfactorily specified using temporal logic and this model.

2.2 State Transition Model

The following model of systems of concurrent processes is similar to the model presented in the section on Temporal Logic, but differs from it in important ways. In the new model:

- variables are not shared among processes,
- transitions (events), rather than shared variables, are the basis for synchronization and communication between processes, and
- the model is *compositional* in that a system is constructed by describing the processes that form its parts.

These differences, which make this model somewhat similar to the model of concurrent computation which underlies CSP, is largely taken from Lamport's Action-Axiom semantics [28]. We feel that this model is suitable for the underlying model for the DSL specification and verification language, which is intended for specifying distributed systems. It also corresponds well with the state machine technique for achieving fault tolerance. This model allows the progress properties discussed earlier to be expressed in a natural way and verified, with or without temporal logic.

Below, first the model is described. Then it is shown how first-order logic can be interpreted with respect to this model. Finally, it is shown how temporal logic can improve expressibility by allowing more concise statements of temporal properties than allowed by ordinary first-order logic.

2.2.1 The Model

A concurrent program consists of two or more processes running at the same time. Each process has its own set of variables, including a location counter, which are accessible only by that process. Processes interact by sending messages through synchronous channels, which are the only variables shared among the processes.

Single Process

A process is defined by a set of states and a set of transitions. Each state is an assignment of values to the process variables, including values for control predicates and input channels. Control predicates are defined for each transition t as follows:

at(t) Control is at the beginning of t.

after(t) t was the last transition to complete.

The value of an input channel is a sequence of values sent to it. Each transition is an enabling condition, which is a predicate on the states, and a transform, which is a mapping from states to states. The enabling condition for transition t implicitly includes **at(t)**. The transform can only be applied to those states for which the enabling condition is true. A transform can be a communication, which is either an input or an output over a specific channel. An input transition changes the state by removing the first value (the head) from the channel.

A state is nondeterministic when the enabling conditions of more than one transition are true at that state. Any of the corresponding transforms can be applied. A state is terminal when no enabling condition is true. If the process reaches such a state, it will remain there and is said to terminate. Notice that whether a process is in a terminal state (has terminated) is deterministic, but whether the process eventually terminates may be nondeterministic.

The execution history of a process consists of a sequence of the form:

$$s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \ldots$$

where s_0 is an initial state of the process and $\forall i > 0$ the enabling condition of t_i at state s_{i-1} is true and t_i transforms s_{i-1} into s_i. If the process terminates, the execution history will be finite, e.g.,

$$s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} s_n$$

It is convenient to deal only with infinite execution histories. A special **final** transition is added with an enabling condition that is true for the terminal states and with the identity transform. A process that terminates continues to perform this transition:

$$s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} s_n \xrightarrow{final} s_n \xrightarrow{final} \ldots$$

This model is sufficient to specify and verify properties relating to progress as well as determinism. For example, for a given execution history,

$$s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \ldots$$

the process terminates if $\exists i > 0.\ t_i = \text{final}$. Otherwise the process does not terminate, but remains live. A process *must necessarily* terminate if it terminates in every possible execution history. A process *can possibly* terminate if there is an execution history for which it terminates.

Concurrent Processes

A system of concurrent processes is modeled as a process, as described above, and a set of internal communication channels. The state is formed by combining the local states of the subprocesses, except that the internal input channels are deleted and input channels that occur in more than one process are combined. The transitions include all of the transitions of the subprocesses except for communications over the internal channels and for final transitions, all transitions formed by combining an input and an output from different subprocesses that each specify the same internal channel, and a final transition whose enabling condition is the conjunction of the enabling condition for the final transitions of each of the subprocesses.

An execution history for a system of processes can be *restricted* to execution histories for each of its subprocesses. The states of the restriction are formed from that subprocess' contribution to the system state, extended with the channels from which the process inputs. Because the sets of variables for the subprocesses are disjoint, only the transitions from a subprocess can affect these states and therefore all other transitions can be ignored. If a restricted history is finite and the last state is terminal, it is made infinite by adding final transitions and repeating the last state.

A system state is a *deadlock state* if no transitions are enabled at it. An execution history terminating in such a state is deadlocked. All other execution histories are infinite. Each restricted history of a deadlocked execution history must either have reached a final state or reached a state from which only internal communications are enabled and no other subprocess has a matching internal communication.

Additional progress properties, which relate a process to its environment, can also be expressed in this model. The fairness assumption is that for each nondeadlocked (infinite) execution history, the restricted execution history for each subprocess is infinite. A subprocess diverges if its restricted history contains only finitely many communication transitions. A process is responsive to its environment if it eventually takes some action in response to an event occurring in some other part of the system. For example, for a given execution history "if subprocess p queries subprocess p' in transition t_i, then subprocess p' responds in transition t_j, where $j > i$" expresses a responsiveness property.

2.2.2 Formal Specifications

To summarize what has been described above, a process consists of:

STATES is a set of states, each of which is a function from (local) variables (including control predicates and input channels) to values.

IC is an initial condition, which is a predicate on STATES specifying that the process is at a starting point and that no values have been sent on the output channels.

TRANSITIONS is a set of transitions, each consisting of an enabling condition EC_t and a transform mapping TR_t : STATES → STATES. A final transition can be applied when all of the other enabling conditions are false.

HISTORIES is the set of possible execution histories for the process, consisting of infinite sequences of states and transitions. STATE is a function that yields the state component of an element in an execution history. For $h \in$ HISTORIES and $i \geq 0$, $\mathrm{STATE}_h(i)$ yields the state after i steps of h. Also, $\forall h \in$ HISTORIES. $\mathrm{IC}(\mathrm{STATE}_h(0))$.

First-Order Logic

Security for a sequential process is proven for each security level l by creating an equivalence relation $=_l$ on states and a function

$$witness_l \ : \ TRANSITIONS_O \rightarrow TRANSITIONS_O^*$$

such that

1. $\forall t \in \mathrm{TRANSITIONS}_{I-l}.\ \mathrm{EC}_t(s) \ \Rightarrow \ s =_l \mathrm{TR}_t(s)$
2. $\forall t \in \mathrm{TRANSITIONS}_O.\ \mathrm{EC}_t(s) \wedge s =_l s' \ \Rightarrow$
   $\mathrm{EC}_{\mathrm{witness}_l(t)}(s') \ \wedge \ \mathrm{view}_l(\mathrm{witness}_l(t)) = \mathrm{view}_l(\langle t \rangle) \ \wedge \ \mathrm{TR}_t(s) =_l \mathrm{TR}_{\mathrm{witness}_l(t)}(s')$
3. $\forall t \in \mathrm{TRANSITIONS}_{I\cap l}.\ \mathrm{EC}_t(s) \wedge s =_l s' \ \Rightarrow \ \mathrm{EC}_t(s') \ \wedge \ \mathrm{TR}_t(s) =_l \mathrm{TR}_t(s')$

where $\mathrm{view}_l(h)$ is the sequence of those transitions in h dominated by l, $\mathrm{TRANSITIONS}_{I\cap l}$ are the input transitions dominated by l, $\mathrm{TRANSITIONS}_{I-l}$ are the input transitions not dominated by l, and $\mathrm{TRANSITIONS}_O$ are the output transitions. Security for a system of processes holds if it holds for each sequential process.

A safety property is stated as an invariant predicate P over STATES. It is shown in two steps:

1. $\mathrm{IC}(s) \ \Rightarrow \ \mathrm{P}(s)$

2. $\forall t \in \mathrm{TRANSITIONS}.\ \mathrm{P(s)} \wedge \mathrm{EC_t(s)} \Rightarrow \mathrm{P(TR_t(s))}$

The first step shows that the property holds for the initial state of a history. The second step shows that if the property holds at the start of a transition, it will also hold after the transition completes.

An example of a safety property is the absence of deadlock. The predicate that must be shown to be invariant is:

$$\exists t \in TRANSITIONS.\ EC_t(s)$$

To show this, the first step is:

$$IC(s) \Rightarrow \exists t \in TRANSITIONS.\ EC_t(s)$$

and the second step is:

$$\forall t \in TRANSITIONS.\ EC_t(s) \Rightarrow \exists t' \in TRANSITIONS.\ EC_{t'}(TR_t(s))$$

Since the existence of a true enabling condition on a state is not implied by the truth of enabling conditions on each of the restrictions of that state (the enabling condition for a communication over an internal channel is the conjunction of the enabling conditions for the communicating elements, only one of which may be true), the "absence of deadlock" predicate cannot be shown to be invariant by showing that it is invariant for each of its parts.

Liveness properties are statements that eventually something will happen. An example is that some value will eventually be communicated over internal channel c. A property L is live after the n^{th} step of history h if $\exists i \geq n.\ L(STATE_h(i))$. The statement that property L is satisfied infinitely often in history h is the same as stating that it is live after every step of h, or $\forall n \geq 0. \exists i \geq n.\ L(STATE_h(i))$. An assumption of fairness that is required for most liveness proofs can be stated as:

$$(\forall i \geq n.\ \exists j \geq i.\ EC_t(STATE_h(j))) \Rightarrow (\exists k > n.\ after(t)(STATE_h(k)))$$

This assumption states that any transition that is infinitely often enabled will be infinitely often executed.

2.2.3 Temporal Logic

Safety and liveness properties can be stated more concisely by using operators from temporal logic, rather than relying entirely on ordinary logic. The specifications are shorter and clearer and support for these operators can be built into a verifier.

Safety properties are expressed using the □ (always) operator. Thus, $\Box P$ states that P is a safety property and must hold for all states. It is proven by showing that it holds for any initial states and that it is an invariant for each transition. Absence of deadlock is expressed as $\Box\exists t \in \mathrm{TRANSITIONS}.\ \mathrm{EC}_t$.

Liveness properties are expressed using the $\leadsto$ (leads to) operator. The meaning of $P \leadsto Q$ is $P(\mathrm{STATE}_h(s_i)) \Rightarrow \exists j \geq i.\ Q(\mathrm{STATE}_h(s_j))$ and expresses the property that Q is live at any state for which P holds. The fact that property P holds infinitely often in every history is expressed as true $\leadsto$ P or alternatively as

$$IC \leadsto P \ \wedge\ \forall t.\, TR_t(P \wedge EC_t) \leadsto P$$

(Notice that a transition is applied here to a predicate rather than a state.) The fairness assumption is:

$$(\forall t' \neq t.\, TR_{t'}(EC_t \wedge EC_{t'}) \leadsto EC_t)) \ \Rightarrow\ EC_t \leadsto after(t)$$

The following rules can be used in a proof of $P \leadsto Q$:

- $P \leadsto P$
- $(\forall t \in \mathrm{TRANSITIONS}.\ P(s) \wedge \mathrm{EC}_t(s) \ \Rightarrow\ Q(\mathrm{TR}_t(s))) \ \Rightarrow\ P \leadsto Q$
- $P \leadsto R \ \wedge\ R \leadsto Q \ \Rightarrow\ P \leadsto Q$
- If e is an integer-valued expression such that $P(s) \wedge (s(e) \leq 0) \ \Rightarrow\ Q(s)$ and $e_n(s) = s(e) = n$ then $(P \wedge e_n \leadsto Q \vee (P \wedge e_m),\ m < n) \ \Rightarrow\ P \leadsto Q$
- If there is a finite set $T \subseteq \mathrm{TRANSITIONS}$ such that $P(s) \wedge \neg Q(s) \ \Rightarrow\ \exists t \in T\ (\mathrm{EC}_t(s) \wedge Q(\mathrm{TR}_t(s)))$ then $(P \wedge \neg Q \leadsto P \vee Q) \ \Rightarrow\ P \leadsto Q$

The last rule assumes that the history is infinite and fair.

2.3 Timed CSP Processes

The model described here was developed by Reed and Roscoe at Oxford University [43]. The notion of an event is generalized to a *timed event* (t, e), where t is a non-negative real number and e is an event, that represents an occurrence of e at time t. All times represent values of a (purely conceptual) global clock. Traces are replaced by *timed traces*, which are finite sequences of timed events $\langle(t_0, e_0), (t_1, e_1) \ldots (t_{n-1}, e_{n-1})\rangle$ such that $t_0 \leq t_1 \leq \ldots t_{n-1}$. The possibility that events e and e′ happen truly concurrently is admitted by allowing $t = t'$. For each communication e, $\hat{e}$ represents e occurring the instant that it becomes available.

One additional extension to traces is required. A timed trace has become *stable* when the process that it represents has made all of the invisible internal progress that it can and the only way for it to proceed is to engage in some communication. The time at which a trace becomes stable is its *stability time*. The notion of trace abstracts away from the invisible happenings occurring between its events, and that of stability puts some of that information back in the form of timing information.

As an example, suppose that the timed trace $\langle(1, \mathrm{b}), (3, \mathrm{c})\rangle$ becomes stable at time 4, and that the only visible events to occur have been $(1, \mathrm{b})$ and $(3, \mathrm{c})$. Between times 3 and 4 internal changes in the process may dictate changes in the externally visible behavior it is willing to undertake. It may extend an offer to communicate with the outside world at time 3.2 and then rescind the offer at time 3.5. Should no visible event occur by time 4, then internal change (at least as it affects the possibility of engaging in visible events) ceases. Notice also that the externally visible sequence $\langle(1, \mathrm{b}), (3, \mathrm{c})\rangle$ may result from any of several different internal paths of control. One such path may stabilize immediately, another stabilize at time 3.5, etc. The stability time associated with $\langle(1, \mathrm{b}), (3, \mathrm{c})\rangle$ is the earliest time at which all possible ways of producing the trace have become stable. There may be no such (finite) real number. Therefore, the values used to denote stability times are the elements of the set STAB $= [0, \infty) \cup \{\infty\}$ where $[0, \infty)$ are the nonnegative real numbers.

A timed process P is an internally consistent set of pairs of the form $(\mathrm{tr}, \mathrm{x})$ where tr is a timed trace and x, the stability time of tr, is an element of STAB. For example, the process DIVERGE – which uselessly loops forever – is identified with the singleton set $\{(\langle\rangle, \infty)\}$. That is, it engages in no visible activity and is never stable. The do-nothing process STOP is associated with the singleton set $\{(\langle\rangle, 0)\}$. It engages in no visible activity and is stable from the very start.

The internal consistency conditions are an extension of the consistency conditions for an untimed process. Here are some examples. To state them neatly, we'll introduce one further piece of terminology:

$$Traces(P) \ = \ \{tr \mid \exists x \in STAB.\ (tr, x) \in P\}$$

- If tr $\in$ Traces(P), then for any initial segment tr$'$ of tr, tr$'$ $\in$ Traces(P).
- No finite interval of time can contain infinitely many events.
- A given timed trace has only one stability time. (If $(\mathrm{tr}, \mathrm{x}) \in \mathrm{P}$ and $(\mathrm{tr}, \mathrm{y}) \in \mathrm{P}$, then $\mathrm{x} = \mathrm{y}$.)
- None of the timed events in a trace can occur later than its stability time. (If $(\mathrm{tr}\hat{\ }\langle \mathrm{t}, \mathrm{e}\rangle, \mathrm{x}) \in \mathrm{P}$, then $\mathrm{t} \leq \mathrm{x}$.)

- The first opportunity for any (visible) timed event to extend the timed trace tr cannot occur later than its stability time. (If $(tr, x) \in P$ and $tr\hat{}\langle(t, \hat{e})\rangle \in Traces(P)$, then $t \leq x$.)

Chapter 3

Specification and Verification of Fault-Tolerance

3.1 Introduction

3.1.1 Motivation

The more complex a system grows, the greater the number of its components which may fail. For very complex systems, it becomes necessary to consider ways in which the failure of the entire system can be made less dependent on failures of particular components. These considerations are usually called "fault-tolerance".

Distributed system design and fault-tolerance are closely related. One prime reason for building a distributed system is to reduce the dependence of that system on the reliability or failure of a single node. So distributed systems may spring from a desire for fault-tolerance. Conversely, once the processing taking place in a distributed system becomes dependent on complex interrelationships between nodes, the probability of failure for the entire system becomes greater than the probability of failure for single nodes. This is because the failure of any node may still disable the entire system. Therefore, complex distributed processing will not be reliable without fault-tolerance.

In this chapter, we will be concerned with specification and verification of fault-tolerance properties. We will be seeking precise definitions of the term "fault-tolerance", and asking what steps must be taken to prove that a system design in fact is fault-tolerant according to the definition. We will only be minimally concerned with strategies, designs, and algorithms used to implement fault-tolerant systems, and only then as examples to show why a particular definition of "fault-tolerance" is relevant.

Fault-tolerance is a general property of systems. Because it is a system property, its formal analysis should have certain aspects in common with analysis of other system properties, such as security, liveness, determinism, and so on. Some points of similarity are:

- The definition of "fault-tolerant system" must not depend on factors external to the system.
- Neither should the statement of the fault-tolerance property involve every detail of the system. It should be possible to make statements about "fault-tolerance" without referring to particular system designs. Therefore verification of fault-tolerance should not generally require verifying the correctness of every detail of the design.
- A fault-tolerant system may be implemented from the interaction of several constituent parts. Therefore, specifying and implementing fault-tolerance may involve specifying properties of various components.
- It is possible to make a mistake in the implementation of a fault-tolerant algorithm even though its theory is well understood. Therefore, formal specification of fault-tolerance and formal verification of its correct implementation may serve to reduce errors.

One previous effort toward verifying fault-tolerance can be found in the SIFT project (Software Implemented Fault Tolerance) [66]. SIFT was an ultra-reliable fault-tolerant computer designed for aircraft flight control. A precise model of this system was developed, and constraints (specifications) on the model which implied the correctness of the system were written down. However, the SIFT approach differs from ours, in that the property of "fault-tolerance" was never considered in isolation, but was always implicitly subsumed in the other system specifications. One could not prove, or even state, that SIFT was fault-tolerant without additionally stating and proving that it satisfied many other correctness properties as well. We intend to find a definition of fault-tolerance which can stand by itself.

It is usually desirable to build fault-tolerance into a system at many levels. For example, at a very detailed level of the design, we would like algorithms which store individual words of memory in a manner which is tolerant to faults in single bits of those words. At a higher level of abstraction, we would like algorithms which are tolerant to faults which cause the crash of entire nodes of a distributed system. The design at the higher level will take the fault-tolerance properties at the lower level for granted. Because fault-tolerance will occur at many levels of design, it is likely that advanced software-specification systems will allow designers to work at a high level of specification, largely ignoring fault-tolerance algorithms, but expecting that fault-tolerance will be built into a system automatically by a compiler or other transformation tools. An example of this kind of specification system is ISIS [6]. Does this approach to specification remove the need for explicitly verifying fault-tolerance properties? Perhaps so; however,

- we will consider a system to include potentially its entire design, at every level of detail. Therefore, we will be analyzing the fault-tolerance of algorithms built in automatically, even though they are submerged at a very detailed level of the design;
- it is not clear that building fault-tolerance into a system automatically will give great assurance of correctness. (Secure operating systems automatically build "security" into the interactions among their application programs, yet this "security" is notoriously unreliable).

3.1.2 Fault Scenarios and MTTF

A system's fault-tolerance is often expressed as a mean time to failure (MTTF). Obviously, a larger MTTF can be taken as an indication of a more fault-tolerant system. One goal of a verification methodology for fault-tolerant systems might be to prove that a system's MTTF exceeds some value.

However, the MTTF is not just a property of a computer system, but involves the environment of that system as well. For example, increasing the flux of cosmic rays through the computer system hardware will surely decrease its MTTF, as is found for space-borne systems. A measure of fault-tolerance which depends only on the system design, and not on its environment, would be preferred.

A **fault scenario** is a history of a system's interaction with its environment which includes not only its inputs and outputs, but also a description of faults, including which components failed, when they failed, and how each failure is expected to manifest itself in the future. Even simple systems will have a large number of fault scenarios (if the time at which a failure happens is a real number, then the number of fault scenarios will be infinite). We may suppose that a system's environment will determine whether a particular fault scenario happens or not. Circumstances which are (apparently) the same will sometimes produce one fault scenario, sometimes another. Therefore, we may suppose that a system's environment assigns probabilities to each fault scenario.

If we can now decide, for each fault scenario, whether the system design will fail, and at what time it fails, then we can (in principle) calculate the MTTF by averaging over all the fault scenarios:

$$\left(\begin{array}{c}\Sigma\\ \text{all fault}\\ \text{scenarios}\end{array}\right)\left(\begin{array}{c}\text{probability}\\ \text{of fault}\\ \text{scenario}\end{array}\right)\left(\begin{array}{c}\text{time until}\\ \text{failure in}\\ \text{fault scenario}\end{array}\right)$$

This calculation, which is usually quite complicated and involves many approximations and assumptions about the environment, will not be our concern in what follows.

Instead of calculating MTTF, we will consider verification of fault-tolerance to be proof that a system design will not fail for a given fault scenario or set of fault scenarios. For example, one might verify that failure does not happen in any fault scenario in which at most one fault occurs. This meaning of "fault-tolerance" depends only on the system design, and is independent of environmental factors. It now remains only to give a meaning to "failure".

3.2 Formal Specification of Fault-Tolerance

A precise definition of "fault-tolerance" can be used as a specification against which system designs may be verified. Several distinctions should be kept in mind when formulating these definitions.

Generally, the definition of "fault-tolerance" should be kept as abstract as possible. There are several advantages to concentrating on specification at a more abstract level:

- A more abstract property will involve fewer (hopefully, no) details of the design. In principle, it can be stated in advance of the design, thereby constraining system designers only in appropriate and minimal ways. It should be possible to change a design without needing to change its specification.
- A more abstract, higher-level property will correspond more closely to intuitive conceptions of "fault-tolerance".

Once abstract versions of the fault-tolerance property are stated, it may be possible to derive lower-level, less abstract, properties from it. If these lower-level properties imply the higher-level one, then it is sufficient to demonstrate that they hold for design X, and to infer from that fact the fault-tolerance of X.

An **extrinsic** property of a system is one which can be defined purely in terms of external interactions of the system, rather than in terms of internal states and state changes (i.e., **intrinsic** properties). Since an extrinsic property makes only minimal constraints on a design, it will generally be preferable for stating abstract system properties.

3.2.1 Specifying Faults

Before defining "fault-tolerance", it is first necessary to define "fault". Abstractly, a system or system component is faulty when it no longer performs according to its specification. This is an extrinsic definition, given in terms of behavior: up until some time, a component behaves according to its specification; after that time, it behaves according to another specification.

The literature of fault-tolerance identifies various "specifications" for components after they have failed. These include: Byzantine (a failed component exhibits arbitrary behavior), fail-stop (a failed component halts and its failure can be detected), and others.

In addition to characterizing faults extrinsically, it is also possible to define them at more detailed, less abstract levels. For example, if one is willing to identify the physical memory of a system as a certain collection of words, then one kind of fault will be that a single bit in one of the words becomes stuck "on". This approach allows a very precise definition of some faults in a simple manner. However, it requires that some knowledge of the design must exist before the specification can be given. It is therefore not an extrinsic specification.

We will use both extrinsic and intrinsic methods to characterize faults.

Some faults, although they may be characterized very easily at a detailed design level, cause very complicated changes in behavior at a higher level. The case of the single "stuck" bit, if that bit happens to be in a region of memory occupied by program code, can cause very intricate changes in behavior of that program. Generally, rather than characterize that behavior exactly at the higher level, we may conservatively choose to characterize it as "Byzantine", or "fail-stop", etc.

3.2.2 Non-Interference

The simplest way to define "fault-tolerance" is converse to our previous definition of "fault": a system is fault-tolerant if it meets its specification. This does not mean, however, that a fault-tolerant system experiences no faults; on the contrary, it must behave as though it were non-faulty, even in the presence of faulty behavior of its components, or of faults at lower levels of abstraction.

What will it mean that a system meets its specification even in the presence of faults? Denote by N the set of fault scenarios under which no faults occur. Let C be a set of fault scenarios under which we desire fault-tolerance; no loss of generality will result if we require that $N \subseteq C$. Suppose that D is a design of an algorithm which exhibits the behavior desired, and that FTD is a fault-tolerant version of D. There are now three methods by which we may show that FTD is fault-tolerant under the given set of fault scenarios, C.

1. We may show that the behavior of D, under scenarios N, is identical to the behavior of FTD under C.

2. We may characterize the behavior of D under N by some specification, S. Then we may show that FTD implements S under C.

3. We may show that the behavior of FTD under N is equivalent to the behavior of FTD under C.

The first of these methods is impractical. We would need to construct two separate implementations for one system.

The second method is the on‹ used in the SIFT project. The specification S describes the correct behavior both of D ; nd of FTD. We would then require simply that FTD behave correctly.

The third method captures the notion of fault-tolerance as a comparison of behaviors with and without occurrences of faults. However, it does so without referring either to design D or to its specification S. The relevant aspects of each can be derived from FTD alone. The behavior of FTD under scenarios with no faults should be equivalent to behavior of D under the same scenarios. Therefore, the third method has a clear advantage over the others: the property of fault tolerance becomes entirely a property of the behavior of FTD, and does not involve extra correctness constraints that may be required by S. It is this third method we will proceed to develop.

Our statement of fault-tolerance is now a relation on the behavior of the system under different sets of fault scenarios. With an aim toward defining fault-tolerance extrinsically, we make more precise the notion of "behavior". Let the possible ways a system may interact with its environment be called "events". Seq:ences of events will be called "histories", and a history that is possible for a system will be called a "trace" of that system. A way to characterize a design extrinsically is by giving the set of its traces. This is a simplification of the approach of CSP [7]. "Behavior" wil be defined by the sequence of events of a trace that are visible to a system's users.

Simple fault-tolerance is then: allowing certain fault scenarios does not change the visible aspects of the set of traces of a fault-tolerant system. We may say that the occurrence of fault events does not *interfere* with a system's behavior.

Formal Definition

Let a system be characterized by a tuple $\langle E, F, I, O, T \rangle$, where E is the set of possible events, $F \subseteq E$ the set of possible fault events, $I \subseteq E$ the set of possible input events, $O \subseteq E$ the set of possible output events, and T is the set of traces of events chosen from E. We will use the notation E^* to denote the set of all sequences of events in T; then $T \subseteq E^*$. Events not in I or O will be called internal events.

Sequences of events may be written explicitly when enclosed in angle brackets, e.g., $\langle e_1, e_2, e_3 \rangle$. The empty trace is written $\langle \rangle$.

For any sequence h and set of events S, we will use the notation, $h \upharpoonright S$, to denote the sequence derived from h but with all events not in S removed, and the ordering of the remaining events retained. For any trace h, what was called "behavior" above will now be denoted by the sequence of non-fault events, $h \upharpoonright \overline{F}$, where $\overline{F} = (E - F)$ is the complement of F with respect to E. Two sets of traces exhibit the same behavior if, for any trace in one, there is a trace in the other which has the same sequence of events in $\overline{\mathrm{F}}$.

We choose the most straightforward way to characterize fault scenarios: as a sequence of events. Each scenario is simply a history, whether or not it is a possible history for a given system.

We will now formalize the above definition of fault-tolerance for the system $A = \langle E, F, I, O, T \rangle$. Let C be the set of fault scenarios for which A is to be fault-tolerant; C will be a subset of E^*. The set of behaviors which may occur in the presence of fault scenarios in C is

$$\{\alpha \mid \exists\beta \in \mathrm{T}, \beta \in \mathrm{C} \text{ and } \beta \uparrow \overline{\mathrm{F}} = \alpha\}.$$

The set of behaviors which may occur in fault-free scenarios is

$$\{\alpha \mid \exists\beta \in \mathrm{T}, \beta \in \mathrm{C} \text{ and } \beta \uparrow \overline{\mathrm{F}} = \alpha \text{ and } \beta \uparrow \mathrm{F} = \langle\rangle\} =$$
$$\{\alpha \mid \alpha \in \mathrm{T} \text{ and } \alpha \in \mathrm{C} \text{ and } \alpha \uparrow \mathrm{F} = \langle\rangle\}.$$

Fault-tolerance as discussed above is the equivalence of these two sets.

$$\forall\alpha \in \mathrm{E}^*, (\exists\beta \in \mathrm{T}, \beta \in \mathrm{C} \text{ and } \beta \uparrow \overline{\mathrm{F}} = \alpha) \leftrightarrow$$
$$(\alpha \in \mathrm{T} \text{ and } \alpha \in \mathrm{C} \text{ and } \alpha \uparrow \mathrm{F} = \langle\rangle)$$

As before, we choose C to include any scenario (sequence) under which no faults occur. Hence,

$$\forall\beta \in \mathrm{E}^*, \beta \uparrow \mathrm{F} = \langle\rangle \rightarrow \beta \in \mathrm{C}.$$

Simplifying the above equivalence yields

$$\text{(FT1)} \qquad \forall\beta \in \mathrm{T}, \beta \in \mathrm{C} \rightarrow \beta \uparrow \overline{\mathrm{F}} \in \mathrm{T}$$

which we take as our basic definition of fault-tolerance. It says that from any given possible history that is also a fault scenario in C, we can construct a second possible history simply be removing all fault events from the first. If the given history has no fault events, then the second history is identical. This definition will be augmented in various ways in future sections.

Example: File System

As an example, consider a fault-tolerant file system. An external user or other client of the file system interacts with it through operations such as "read-file", "write-file", and so on. These operations, along with any of their associated parameters and return values, will be taken as the events from which system histories are constructed. Certain observable behaviors, or traces, are expected, e.g., if a user writes a file and then reads the same file, the contents returned should be the same as the contents written. So the history $h_1 = \langle$

write-file 'contents', read-file 'contents' ⟩ should be a trace, while the history h_2 = ⟨ write-file 'contents', read-file 'garbage' ⟩ where 'contents' and 'garbage' are different, should not be.

Fault events in the hardware supporting the file system may make various bizarre behaviors possible. For example, if the sequence h_3 = ⟨ write-file 'contents', fault, read-file 'garbage' ⟩ is a trace, then the system may appear as though it had actually executed the illegal history h_2. If, however, the file system is to be fault-tolerant with respect to fault scenarios including h_3, then our property (FT1) demands that h_3 won't exist if h_2 doesn't.

Why isn't property (FT1) equivalent to requiring that the file system work correctly? In fact, (FT1) is weaker. Suppose, for some reason, that when each file is written, the file system alters the contents with some arbitrary function Mod; when a file is read, the altered contents are returned. The traces of this new file system, if it is fault-tolerant, will include:

⟨ write-file 'contents', read-file Mod('contents') ⟩
⟨ write-file 'contents', fault, read-file Mod('contents') ⟩

and possibly many others. Like our previous file system that did not alter the content of files, this one meets (FT1) since the alterations are made regardless of the presence of a fault. But, if we have specified a file system that does not modify the content of files, this implementation is incorrect even though it is fault-tolerant.

Invariants

Fault-tolerance is implemented by use of redundancy, often redundancy of state information. Therefore, one might try to specify fault-tolerance by specifying the redundancy of state information in the design. A design would correctly maintain the redundancy as an invariant even in the presence of certain faults. For example, a system which uses an error-correcting code to store a word of memory should maintain the invariant: the stored bits are in a consistent, error-free state when the word is read. Note that the invariant does not literally *always* hold, but holds only at certain events during processing.

Using an invariant to specify redundancy will be an important, necessary condition for verifying higher-level fault-tolerance properties, such as (FT1), but does not substitute for them. In the file system example, suppose that each file's content is stored redundantly in more than two copies, that faults corrupt exactly one copy, and that the set of scenarios, C, includes all sequences with at most one fault. Then majority voting is a fault-tolerant algorithm satisfying (FT1). An invariant could be used instead to require that all copies be consistent when any operation on the file is completed. Unfortunately, an algorithm that replaced all copies with a corrupted version would also satisfy this invariant, but does not meet (FT1).

More powerful invariants can be found that imply FT1 []. These invariants, however, relate not only the values of state variables but other predicates on the state machine's history. They are often stronger conditions than FT1.

3.2.3 Analogy with Multi-Level Security

Property (FT1) is often called a "non-interference" property. Non-interference properties have been explored and used extensively in the context of multi-level computer security (MLS) [17,21,33,64]. It is reasonable to ask, then, what is the relation between fault-tolerance and multi-level security? We begin with a brief description of MLS.

In a secure computer system, it is desirable to prevent sensitive information from flowing to users who are not authorized access to it. In military systems, the sensitivity of information and the authorizations of users can be labeled by a partially ordered set of levels. Highly sensitive information is brought into the system by the inputs of users of the system with high levels of authorization. The multi-level security problem is the prevention of information transfer from those inputs to outputs which can be seen by users who are not so highly authorized.

The key to defining security in this way is the definition of "information flow". As described previously, a system can be characterized extrinsically, i.e., in terms of its interactions with the outside, and we may take this to mean at least the knowledge of the possible traces of inputs and outputs. An extrinsic definition of information flow is the ability of a particular user to use the observed behavior of the system, plus knowledge of its possible traces, to make deductions about its unseen behavior. Information will flow from unseen inputs to observed behavior if more can be deduced about those inputs than could be deduced if the system's behavior were not observed.

Non-interference can be taken (loosely) to mean that high-level inputs do not interfere with or influence processing, and hence outputs, on lower levels. Stated more precisely, the existence of high-level inputs cannot be deduced from observing a particular history of lower-level inputs and outputs.

An analogy between fault-tolerance and multi-level security properties can now be drawn. A non-interference property for multi-level security can be converted to a non-interference property for fault-tolerance by translating from "highly sensitive" inputs to "fault events", and from "less sensitive inputs and outputs" to "non-fault events", i.e., ordinary system behavior. A fault-event, e.g., a failure of a processor, hardware glitch, etc, is thus considered a type of input, although not from any user. The non-interference property for MLS can then be translated into the language of fault-tolerance: the existence of fault events cannot be deduced from particular histories of ordinary system behavior. We will see that this is almost exactly the property (FT1).

The analogy can be posed in another way if we consider the work of Biba [5] in extending MLS to handle some concerns over information integrity. His work added integrity levels to the security levels already used for marking the sensitivity of information and the authorizations of users. The Biba property, in effect, allows information to flow only from inputs to outputs of the same or lower integrity level. Given this property, high-integrity users would be prohibited from deducing (and hence from being corrupted by) information about inputs at lower integrity levels. The analogy between fault-tolerance and MLS can be recast in terms of integrity, in which case a fault-event is seen to be analogous to an input of low integrity level. This becomes an appropriate analogy if we consider that fault events are not usually a high-integrity source of information.

The analogy between fault-tolerance properties and multi-level security/integrity properties is not perfect, and breaks down in several ways.

- Unlike the inputs from users, which are the ultimate source of information in MLS systems, fault events are not external events. Therefore, the noninterference property for fault-tolerance cannot be truly extrinsic. The system design will be partially constrained by specifying the set of possible faults.

- Systems are never, in practice, tolerant to all fault scenarios. Some possible sequence of faults will cause the system to fail. This differs from the analogous multi-level security case, in which one desires to build systems that are secure under all possible histories of sensitive inputs. Users of an insecure system may conspire to transmit information by concocting unusual or unlikely sequences of inputs; fault events are assumed not to do this. Thus, the MLS analogue of the set of fault scenarios, C, is in fact the set of all traces, T. It is the set, C, which divides the most probable fault scenarios from those that are less likely.

Since fault-tolerance non-interference properties and MLS non-interference properties are formally similar, can the same kinds of implementation mechanisms be used for both? In other words, might we use access control to implement fault-tolerance? The differences given in the previous paragraph point out why this will not work. Faults are not external events, and therefore it is not possible for a system to decide, without further processing, whether they are fault events or not. A fault detection mechanism may be needed. In secure systems, however, inputs are associated with the authorization level of the user who causes them. Thus, even though there is a formal similarity between fault-tolerance and MLS properties, the designs used to implement them must be different.

Deducibility

We have discussed multi-level security in terms of information flow, and information flow in terms of deducibility. The non-interference statement of fault-tolerance given previously can also be seen as a restriction on deducibility, as we now show.

Suppose that an actual run of a fault-tolerant system $A = \langle E, F, I, O, T \rangle$ produces trace $\beta \in T$. This trace may contain fault events, i.e., events in the set $F \subseteq E$. But a user of system A interested only in the results of normal, fault-free processing will be able to view at most the events of the sequence $\beta \uparrow \overline{F}$. Since we suppose that A is fault-tolerant according to our non-interference definition, we have

$$\forall \beta \in T, \beta \in C \rightarrow \beta \uparrow \overline{F} \in T.$$

Now, if the trace β is a fault scenario for which this system is fault-tolerant, the user who knows the set of traces, T, (i.e., the design) will be able to conclude that the sequence $\beta \uparrow \overline{F}$ is also a trace. Since both are traces, the user cannot assuredly deduce that β happened rather than $\beta \uparrow \overline{F}$. The user cannot assuredly deduce that any of the fault-events in β happened.

A few notes are in order concerning this form of deducibility.

1. System A may be nondeterministic, and for a given sequence of inputs to A the possible output sequences may have different probabilities. If $\beta \uparrow \overline{F}$ is an extremely unlikely trace of A, then a user seeing this sequence of fault-free events may perhaps conclude that trace β is almost certainly the actual behavior of the system. However, this conclusion would not be certain. We have not analyzed this probabilistic kind of inference.

2. We have not said whether the events of E include timing information. If they do, then this fault-tolerance property demands that processing a trace containing faults (and which is in the set of fault scenarios, C) takes no longer than processing the behaviorally equivalent fault-free trace. If timing information is not included, then this fault-tolerance property becomes much easier to satisfy, but of course deducibility based on timing is no longer ruled out.

3. Property (FT1) rules out the ability to deduce the existence of faults in a fault scenario based on the visible behavior, but it does not rule out the ability to deduce that particular fault scenarios did *not* happen. For example, the system A may produce identical behavior under every fault scenario, except that if any fault has occurred, at some point A will loop forever without any external interaction. If a user of the system now sees A terminate, he can deduce that no fault has occurred during processing. So, our fault-tolerance property guarantees that information does not flow from fault events to visible behavior, but it does not guarantee that that behavior is *independent* of the actual fault scenario. If our formal definition of fault-tolerance is augmented by a converse property

$$\forall \beta \in E^*, \beta \in C \text{ and } \beta \uparrow \overline{F} \in T \rightarrow \beta \in T$$

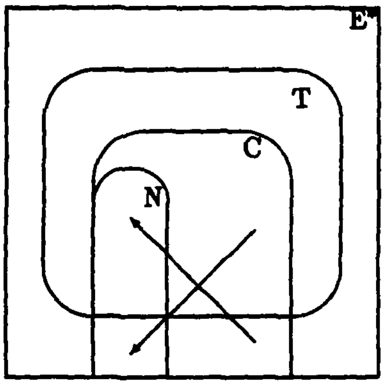

Figure 3.1: Relationship between the set of all histories, E*, all traces, T, of some given system, a set of fault scenarios, C, and the set of all fault-free histories N. Arrows are instances of the mapping $\uparrow \overline{\mathrm{F}}$ ruled out by properties (FT1) and (FT2).

then deducibility of the non-existence of faults is prevented. After combining with our original definition, *this results in the simple form:*

$$\text{(FT2)} \qquad \forall \beta \in \mathrm{E}^*, \beta \in \mathrm{C} \rightarrow [\beta \in \mathrm{T} \leftrightarrow \beta \uparrow \overline{\mathrm{F}} \in \mathrm{T}]$$

In a system satisfying (FT2), the scenarios in C are divided into equivalence classes in which every member shows the same behavior. Either every member of a class is a trace, or none are.

The rationale for property (FT2) can be better seen from figure 3.1. The regions E*, T, C, and N, represent sets of traces as defined previously. The arrows represent the mapping from traces into traces defined by the operation $\uparrow \overline{\mathrm{F}}$. The downward arrow is an instance of a trace whose fault-free behavior is impossible. This case is prevented by property (FT1), since by observing this fault-free behavior one could deduce that some faults had happened. The upward arrow is an instance of a fault-free trace for which a particular pattern of faults is impossible. For example, the pattern of faults may actually be one in which the system crashes. This case is prevented by property (FT2).

So far, each fault-event has been thought of as a kind of input event, effectively an element of the set I. It may be that there are other events, possibly in the output set, O, which are to depend on the existence of faults. For example, an audit report for the system may contain information about the previous history of faults. Then the production of the audit report would be an event in the set $O \cap F$, in other words, an output which contains information about the existence of faults. If the system is designed so that audit reports are produced regardless of whether faults have happened, then fault-tolerance property (FT1) will not hold. A sufficient generalization of that property is:

$$\begin{aligned}&\forall \beta \in T, \beta \in C \rightarrow \\ &\quad \exists \gamma \in T, \gamma \uparrow \overline{F} = \beta \uparrow \overline{F} \text{ and } \gamma \uparrow I \uparrow F = \langle\rangle.\end{aligned}$$

In this generalization, we are prevented from deducing the existence of a trace containing fault input events, but we may be able to deduce that fault output events have occurred. Note that this fault-tolerance property reduces to (FT1) in the case that $F \subseteq I$.

3.2.4 Graceful Degradation

Even though a system is verified as perfectly fault-tolerant under a given set of fault scenarios, C, we may still require more. We may also require that, for fault scenarios only slightly worse than those found in the set C, the system will not be reduced to chaos, but will rather behave only slightly less than perfectly. This aspect of fault-tolerance we will refer to as "graceful degradation".

As an example, consider a system which must perform two tasks, A and B. Suppose that fail-stop processors A_1 and A_2 are dedicated to simultaneous execution of task A, while fail-stop processors B_1 and B_2 are dedicated to task B. Ignoring the amounts of time needed for processors to compare final results, this system will be perfectly fault-tolerant for a fault scenario in which processors A_1 and B_1 fail: both tasks will complete, and they will complete in the same amount of time they would have taken if no faults had occurred. However, it will not be fault-tolerant for a fault scenario in which processor A_2 fails in addition to A_1 and B_1. In this case, processing of task A is interrupted, and will not be completed unless further steps are taken. "Graceful degradation" could mean, that for any fault scenario in which three or fewer processors fail, both tasks A and B will eventually complete. To implement this specification will require some reconfiguration of the system: interrupted processing of task A will have to be continued or restarted on a processor originally dedicated to task B, or vice-versa. The processing power of the system will then be degraded, since a single processor will take longer to complete both tasks than either one separately, but the response to faults is graceful since at least both tasks will be finished.

Like "fault-tolerance", "graceful degradation" is quickly grasped intuitively, and it is always a consideration in the design of fault-tolerant systems. Our goal in this section will be to show that certain kinds of "graceful degradation" may be reduced to formal specifications, just as certain kinds of "fault-tolerance" were. Once reduced to specifications, it is possible to decide unambiguously whether system performance "degrades gracefully".

The problem of specifying graceful degradation will have much in common with the previous discussion of specifying fault-tolerance. We expect that many systems will be fault-tolerant for some fault scenarios, gracefully degrade for others, and be chaotic for the rest. As a result, specifications for graceful degradation may be merely modifications or generalizations which weaken those we have already discussed for pure fault-tolerance. In fact, the specifications we will arrive at can be seen simply as a more comprehensive way to define fault-tolerance itself.

Classes of Fault Scenarios

In the example above, a system was perfectly fault-tolerant for any fault scenario in which at most one task A processor and at most one task B processor failed. Denote this set by C, as usual. It is gracefully degraded in a larger set of scenarios, C_1, in which at most 3 processors fail. A complete specification for this case would be the conjunction of two conditions: if a system's history is in C, then fault-tolerance; if it is in C_1, then a weaker condition holds.

In general, we will divide the fault scenarios into classes and treat each class separately. Let each of $C_1, \ldots, C_n$ be a set of fault scenarios. For any i, $C \subseteq C_i$. Specification FT will hold under C. We will find weaker forms of fault-tolerance, $GD_1, \ldots, GD_n$, that will hold in each class C_i separately. For any i, $FT \rightarrow GD_i$. Our complete specification is then a conjunction

$$\begin{aligned} \forall h \in T,\ h \in C &\rightarrow FT \text{ and} \\ h \in C_1 &\rightarrow GD_1 \text{ and} \\ &\ldots \\ h \in C_n &\rightarrow GD_n. \end{aligned}$$

Limited Interference

If fault-tolerance is to be expressed as a non-interference property, as discussed in section 3.2.2, then graceful degradation may be expressible as some limited interference of faults with external behavior. A specification of limited interference should be a generalization of a specification for non-interference. The form of the specification must then show the way in which interference of fault events with normal behavior is limited.

When we developed property (FT1), we required that system behavior with and without faults be *identical*. This prevents deducibility that *any* faults have occurred. Unlike the MLS case, though, it may not be a problem that one can deduce the existence of faults, so long as the system behavior in response to those faults is "good enough". Thus, we need not demand that behaviors be identical, but only that they be *acceptably equivalent*. If α and β are behaviors (sequences in which no fault events occur) then let $\alpha \equiv \beta$ mean that the two behaviors are acceptably similar. The relation '$\equiv$', called the **tolerance relation**, will be an equivalence relation on behaviors.

Adding fault scenarios to a system satifying (FT1) will not enlarge the set of behaviors. We now want a second property, (FT2), such that adding fault scenarios will not enlarge the set of possible equivalence classes of behaviors, where a "possible equivalence class" is one that contains at least one possible behavior. Repeating the analysis that led to (FT1), but demanding only equivalent instead of identical behavior, we find that

$$\text{(FT3)} \qquad \forall\beta \in \mathrm{T}, \beta \in \mathrm{C_i} \rightarrow \exists\gamma \in \mathrm{T}, \gamma \uparrow \mathrm{F} = \langle\rangle \text{ and } \gamma \equiv \beta \uparrow \overline{\mathrm{F}}$$

is the generalized fault-tolerance, or graceful degradation, property that results. This property says: for any fault scenario, we must be able to find an alternate possible history that is fault-free and is acceptably equivalent. Note that (FT3) reduces to (FT1) in the case that the tolerance relation is equality (and that $\mathrm{C_i}$ is C).

If β is the system's actual history, then behavior $\beta \uparrow \overline{\mathrm{F}}$ is observed. If the behavior is not itself a trace, an observer can deduce that faults have occurred. However, if the observer could not distinguish the behavior γ from $\beta \uparrow \overline{\mathrm{F}}$, then, just as for (FT1), the existence of faults could not be deduced.

It is the choice of tolerance relation that determines how faults interfere with behavior. A special case occurs when the tolerance relation is used to ignore certain events. For example, it may be that the observable, non-fault events possible for a system can be associated with different tasks the system is designed to accomplish. Let the tasks have different priorities, so that under fault scenarios in set $\mathrm{C_1}$, the system degrades gracefully by completing just the high priority tasks and abandoning the low priority ones. Let $\mathrm{L_1}$ be the set of events associated with the lower-priority tasks. Then $\mathrm{L_1} \subset (\mathrm{E} - \mathrm{F}) = \overline{\mathrm{F}}$, where E is the complete set of events and F is the set of fault events. "Graceful degradation" for this system can be stated as (FT3), in which the tolerance relation satisfies $\alpha \equiv \beta \leftrightarrow \alpha \uparrow \overline{\mathrm{L_1}} = \beta \uparrow \overline{\mathrm{L_1}}$ and $\overline{\mathrm{L_1}} = (\mathrm{E} - \mathrm{L_1})$. In other words, we require limited interference in which events in set $\mathrm{L_1}$ are ignored.

It is possible that a larger set, $\mathrm{C_2}$, of more severe fault scenarios would result in a larger set of tasks being dropped. (FT3) could then be used with a tolerance relation ignoring $\mathrm{L_2}$, a larger set of events. In general, $\mathrm{L_i}$ could be determined as a function of $\mathrm{C_i}$.

This form of limited interference can be viewed in terms of the analogy with multi-level security. Each event in set $\mathrm{L_i}$ may carry information about the existence of unseen fault events, and is analogous to events in secure systems that leak some information from high security levels to lower ones. In the language of secure systems, these are "downgrading" events, and the systems that produce them are not perfectly secure. A downgrading event may occur either in an explicit downgrading operation, or in a covert channel which leaks information about higher-level processing, and which cannot be avoided due to other design requirements. A secure system with downgrading events will not leak information from higher levels if the downgrading events are not used to make deductions about higher-level processing. However, the analogy ends here. In a secure system, the fact that downgrading events exist can often be manipulated to transmit arbitrary amounts of information to lower levels. This is not a concern in fault-tolerance, since the occurence of fault events is not maliciously manipulated.

Example: Timing

A special case of graceful degradation is a decrease in real-time responsiveness, or throughput. In this case, faults in a system history will interfere with visible behavior, but the interference may only be observed through changes in the *timing* of visible events. The example given at the beginning of this section is one such case: reconfiguration allows the system to complete the processing of all its tasks eventually, but final completion may take longer if some processors have failed. The actual sequence of visible events will remain unchanged under a fault scenario containing fault events, but due to the special processing needed to hide the effects of the faults, processing is slowed down, and some visible events occur later.

The simplest way to handle this kind of graceful degradation is to ignore it. Our previous definitions of fault-tolerance made no explicit mention of timing. If we suppose that timing information is not present in traces, then property (FT1) is sufficient. However, it fails to specify the property of graceful degradation.

There are at least two ways to add timing information to traces.

1. Make each event a pair, *(action, time)*. The first component records which operation was performed at the event; the second records the real-time at which it occurred. The time-components of events in a trace are required to be non-decreasing, i.e., if event B follows event A in a trace, then B's time component is at least as great as A's.

2. Do not directly associate times with each event, but instead add explicit clock ticks as events. The type of each tick event tells how much real-time has passed since the last tick. A lower bound on the time taken by a trace is the sum of the time intervals represented by clock ticks in the trace.

These methods for including timing are interconvertable. Given a sequence of events including real-time components, the appropriate number of clock tick events may be interleaved. Given a sequence of events including clock ticks, the ticks may be removed if the intervals they represent are summed and attached as real-time components to non-tick events; the uncertainty in the real-time components depends on the density of tick events.

We will work with the second method, since it involves less alteration of our previous trace description. For a system $A = \langle E, F, I, O, T \rangle$, let the clock tick events be drawn from a set $\tau \subset \mathrm{E}$. *A history*, α, *with timing* information can then be converted to one without: $\alpha \upharpoonright \overline{\tau}$. We suppose that the set τ is disjoint from each of F, I, and O.

Suppose that system A is fault-tolerant under fault scenarios C, but its throughput will be degraded under the scenarios in set $\mathrm{C}_1 - \mathrm{C}$. C_1 may be parameterized by δ, i.e., let C_1 be a function which maps a real number into a set of fault scenarios. The parameter δ can be thought of as the 'severity' of a particular collection of fault scenarios: the greater the severity, the greater the damage to throughput. One way to limit the damage is to require:

$$\begin{aligned} &\forall \beta \in \mathrm{T}, \beta \in \mathrm{C}_1(\delta) \rightarrow \\ &\qquad \exists \gamma \in \mathrm{T}, \gamma \upharpoonright \mathrm{F} = \langle\rangle \text{ and} \\ &\qquad\qquad \gamma \upharpoonright \overline{\tau} = \beta \upharpoonright \overline{\mathrm{F}} \upharpoonright \overline{\tau} \text{ and} \\ &\qquad\qquad \Delta(\beta \upharpoonright \overline{\mathrm{F}}, \gamma) \leq \delta \end{aligned}$$

The function Δ compares two traces for timing differences and returns a positive real number. This property requires that for any actual system trace, there is a trace with no fault inputs, which shows equivalent visible behavior if timing events are ignored, and whose timing difference from the actual trace is bounded in some way by the parameter δ. It is a special case of (FT3), in which the tolerance relation is $\alpha \equiv \beta \leftrightarrow \alpha \upharpoonright \overline{\tau} = \beta \upharpoonright \overline{\tau}$ and $\Delta(\alpha, \beta) \leq \delta$.

This property should reduce to (FT1) as a special case. So if $\delta = 0$, then let $\mathrm{C}_1(\delta) = \mathrm{C}$. Also, $\Delta(\mathrm{t}_1, \mathrm{t}_2) = 0$ if and only if $\mathrm{t}_1 = \mathrm{t}_2$.

The function Δ may take various forms. Assuming that every clock tick event records the same interval of time, then a simple form for Δ which compares the total real-time taken by alternate traces is:

$$\Delta(\mathrm{t}_1, \mathrm{t}_2) = |\ \mathrm{length}(\mathrm{t}_1 \upharpoonright \tau) - \mathrm{length}(\mathrm{t}_2 \upharpoonright \tau)\ |$$

With this Δ, our graceful degradation property stated above limits how much longer a faulty trace may take than would the slowest equivalent fault-free trace. Other, more complicated forms for Δ are possible. For example, the above Δ might be normalized by dividing by one of the lengths; it would then limit the fractional degradation in processing time. Instead, it might limit only the difference in processing times taken by the last task completed, or the difference in average times taken to complete all tasks so far. In general, Δ will express bounds on real-time which are system-specific.

Deducibility

Is there a noninterference property which strengthens the non-deducibility aspects of (FT3) in the same way that (FT2) strengthens (FT1)?

Let EqScen be an equivalence relation on fault scenarios such that two scenarios are equivalent if they exhibit the same "pattern" of faults. Naturally, there is more than one choice for EqScen, depending on the degree of detail in which a "pattern" of faults is described. Consider

$$
\begin{aligned}
&\text{(FT4)} \qquad \forall\beta \in \mathrm{T},\ \beta \in \mathrm{C} \rightarrow \\
&\qquad\qquad \forall\alpha \in \mathrm{E}^*,\ \alpha \in \mathrm{C} \text{ and } \alpha \uparrow \overline{\mathrm{F}} = \beta \uparrow \overline{\mathrm{F}} \rightarrow \\
&\qquad\qquad\qquad \exists\gamma \in \mathrm{T},\ \gamma \uparrow \overline{\mathrm{F}} \equiv \beta \uparrow \overline{\mathrm{F}} \text{ and } \mathrm{EqScen}(\gamma, \alpha).
\end{aligned}
$$

This property requires for any possible fault scenario, β, and any pattern of faults, represented by the sequence α, that there must be a trace, γ, which both has that pattern of faults and behaves equivalently to β. Therefore, observing behavior $\beta \uparrow \overline{\mathrm{F}}$ does not allow any pattern to be ruled out.

A simple choice for EqScen considers two scenarios as equivalent merely if they produce the same fault events in the same order, i.e., $\mathrm{EqScen}(\alpha, \beta) \leftrightarrow \alpha \uparrow \mathrm{F} = \beta \uparrow \mathrm{F}$. With this choice, (FT4) implies (FT3) (since α can be taken to be $\beta \uparrow \overline{\mathrm{F}}$). However, this choice retains so little information about the "pattern" of faults in a history that (FT3) implies (FT4) as well (since a γ can be constructed by placing all required fault events at the end of a trace).

A much more discriminating choice for EqScen considers two scenarios as equivalent if exactly the same faults appear in corresponding places in both. In other words, the nth event of one scenario is a fault if and only if the nth event of the other is the same fault event. With this choice, and replacing the tolerance relation by equality, property (FT4) reduces to (FT2).

Example: Agreement

As another example in which one may want to achieve fault-tolerance but does not expect to hide the existence of faults, consider algorithms for agreement. Consider a collection of processes, $A_1, \ldots, A_n$, each designed to choose a single value initially. The choices are events of the form v_{ij}, where i tells which process and j tells which value. The processes communicate with each other via a set of events, X; they terminate by outputing events of the form o_{ij_k}, each of which is a claim by process A_i that process B_j chose value k initially. Their goal is to arrive at agreement on the correct values chosen by each of the others. However, each process A_i is subject to a possible fault event f_i, which alters its behavior in some way. What is fault-tolerance in this situation?

If the faults are Byzantine, the problem of finding an algorithm by which all non-faulty processes can agree is Byzantine agreement [30]. Define a predicate, 'Agree', whose domain is the set of histories. 'Agree' holds when applied to h if:

> For any processes A_i and A_j, if neither f_i nor f_j appears in h (neither has failed), then exactly one o_{ik_l} and $o_{j_{km}}$ occurs for every value of k (both have terminated), o_{ik_l} occurs if and only if $o_{j_{kl}}$ does (agreement), and $o_{j_{ik}}$ occurs if and only if v_{ik} does (correct value).

Define

$$\begin{aligned}
&(\mathrm{FT4mod}) \qquad \forall\beta \in \mathrm{T},\ \beta \in \mathrm{C} \rightarrow \\
&\qquad\qquad \forall\alpha \in \mathrm{E}^*,\ \alpha \in \mathrm{C} \text{ and } \alpha \uparrow \overline{\mathrm{F}} = \beta \uparrow \overline{\mathrm{F}} \rightarrow \\
&\qquad\qquad\qquad \exists\gamma \in \mathrm{T},\ \mathrm{Agree}(\gamma) \leftrightarrow \mathrm{Agree}(\beta) \text{ and } \\
&\qquad\qquad\qquad\qquad \mathrm{EqScen}(\gamma, \alpha).
\end{aligned}$$

The property (FT4mod) guarantees at least: if the processes come to agreement when there are no fault events, then they can come to agreement when there are. If one looks only for agreement or disagreement and not at the actual outputs of the processes or the communications between them, then one cannot eliminate any pattern of faults in the set of fault scenarios, C. (FT4mod), however, is not equivalent to Byzantine agreement; it becomes so only if it is also known that the processes will come to agreement when there are no faults.

(FT4mod) cannot be placed in the form of (FT4) by a choice of tolerance relation because the definition of 'Agree' is not extrinsic: 'Agree' uses the presence of faults in a history to determine whether a given process has failed. It does not depend purely on behavior. This is the definition used in [30]. Other, extrinsic, definitions of agreement may be possible.

3.3 Verification of Fault-Tolerance

We have argued that non-interference specifications can be used to capture the intuitive notion of fault-tolerance. How can a system be verified, in practice, to implement this sort of specification correctly?

First, the constructs of "event" and "trace" that appear in the definition must be related to features of the implementation. One may then appeal directly to the definition in constructing a proof. However, for all but the simplest system, this approach becomes very complicated.

Many of existing verification tools [56,19] provide little help either. Typically, these are designed for proof of invariants, or more generally, of *embedded assertions*: conditions that hold at a particular point in an execution history. Unfortunately, an embedded assertion expresses a condition that applies to each history independently, whereas a non-interference specification applies to the entire set of possible histories at once. The help provided by these tools is not sufficient.

Because fault-tolerance specifications are formally similar to specifications for multilevel security, this same problem occurs in the verification of MLS. We know of no general solution. In that domain, however, specialized techniques can be applied to analyze special cases. For example, the technique of [65] is one in which the existence of some traces is shown by modifying others in appropriate ways. Demonstrating the existence of one trace, given another trace, is exactly what is needed in each of the fault tolerance properties considered in previous sections. This technique can in fact be applied either to designs in the MLS or fault-tolerance domains.

3.3.1 Modeling Fault Tolerance in MUSE

We now consider an example of fault-tolerance verification in the context of a particular set of tools.

The MUSE verification environment [22] has been developed at SYTEK as a successor to the Hierarchical Development Methodology (HDM) [56]. In this section we will identify some changes to the MUSE environment that would be needed to support efficient verification of fault-tolerance, defined in previous sections in terms of non-interference. Familiarity with HDM or MUSE is assumed. We make no claim that this list of potential changes is complete.

MUSE and HDM are both languages for describing state machines. For the most part, the conclusions we reach in this section will apply generally to other state-machine description languages as well.

Modeling Faults

Section 3.2.1 described two ways to model faults in a specification language:

1. as changes to the state of system components

2. as changes in the behavior of system componentε

The first of these is useful for modeling changes to data structures; the second of these is the more general, and can be used to model aberrant behavior of algorithms as well. Can these methods for modeling faults be used in MUSE?

It is easy to model in MUSE changes to the state of system components: this is what the precursor to MUSE, HDM, was originally designed to do. Faults which cause changes to data structures can be modeled simply by adding new OFUNs which modify the VFUNs representing those data structures. Because OFUNs can describe non-deterministic state changes, it will often be possible to describe a large class of faults using a single OFUN. For example, any single-bit fault in the data in a region of computer memory can be modeled as an OFUN which is an exclusive-or of possible changes to individual bits.

A set of fault scenarios, restricting the possible histories of faults under consideration, can be modeled in MUSE simply as a precondition on each OFUN describing state changes due to faults. MUSE does not give an explicit facility for describing an OFUN's preconditions; however, the preconditions can always be built into the OFUN's EFFECT.

Faults which are described as changes in the behavior of a component cannot be modeled simply in MUSE. Consider a module consisting of a collection of VFUNs, OFUNs, and OVFUNs. The collection of O- and OVFUNs is in effect an abstract description of an algorithm for interacting with other modules, via invocations of O- and OVFUNs, via values returned by OVFUNs, and via VFUNs shared with external modules. Changes to that algorithm which are caused by faults at less abstract levels, e.g., in the hardware executing the algorithm, are not easily described in MUSE. Instead, we may conservatively describe the effect of such a low-level fault by specifying that the module's behavior changes: by halting, for example, or by becoming Byzantine. So the module's behavior appears in two phases: a normal phase, followed optionally by a faulty phase. This phase change can be modeled in MUSE by building the possibility of both phases into each O- and OVFUN; this method is clumsy at best. O- and OVFUNs can grow quite complicated just describing normal processing, let alone the possibility of faulty behavior.

A description of a module's behavior in terms of traces offers a better alternative. Normal behavior can be described as one set of traces, N, while faulty behavior is described in terms of another set, F. The complete description of the module is then a set of traces each of the form $n\,\hat{}\,\langle e\rangle\,\hat{}\,f$, where $n \in N$, and $f \in F$, and e is a fault event. This decomposes the problem naturally into a separate description of each phase.

Specifying a set of traces which includes all those which are possible behaviors of a module and no others can become very involved in any but the simplest cases. Several points should be noted.

- Characterizing faulty behavior is usually not an exact science. Often, the set of traces is quite simple, e.g., a Byzantine-faulty process exhibits all traces.

- It will often be sufficient to give properties of the set of traces, rather than characterize it exactly.

- Ideally, the normal behavior can be described as a state machine, while the faulty behavior is described as a set of traces. This requires a hybrid specification language.

A MUSE environment which allows modules to be described either as non-deterministic state machines, as currently, or instead as sets of possible behaviors, would be ideal for modeling faults. Modules which are described using both methods would require verification that the two descriptions were consistent. Interactions between modules written using the two different methods would need a common language for their interface.

Modeling Replication

Replicating of data structures and algorithms forms a key component of the fault-tolerance toolkit. How does MUSE support specification of designs with replication?

Replication of data structures is easily modeled in MUSE. Any VFUN can be replicated simply by adding to it another parameter which varies over the various copies of the VFUN. For example, the parameter may be simply an integer, a host type, a process name type, etc.

Replication of algorithms and modules is also accomplished by adding parameters to OFUNs and OVFUNs.

Top-level Verification

The approach taken to verifying the top-level of a MUSE specification of a fault-tolerant system will depend on the sort of fault-tolerance property to be proved. We distinguish the various cases considered earlier in this chapter.

If it is simply required that the design expressed in MUSE satisfy its specification even in the presence of fault scenarios, then the MUSE methodology will certainly not be adequate for all cases, since there is an unlimited variety of specifications possible. This drawback is suffered by any verification system.

If it is required that at certain times during system processing the system state satify an invariant, then MUSE is well suited to the task. The OFUNs and OVFUNs must be structured so that the times at which the invariant is to hold are the points after the invocation of one O- or OVFUN and before the invocation of the next. Then it is sufficient to verify a MUSE ASSERTION.

If it is required that a module satisfy a non-interference property, then MUSE is not necessarily adequate for the task of verification. Suppose that a VFUN, h, of type history is used to record any and all information about module invocations and changes to the module's VFUNS. Then the value of every VFUN can be expressed as a function of h. Also suppose that the set of histories possible for this module is H. Then any property which can be stated as a MUSE ASSERTION for the module is of the form:

$$\forall h \in \text{history}, h \in H \rightarrow \text{ASSERTION}(h)$$

since the ASSERTION need be demonstrated only for histories which are possible. Note that the ASSERTION is not a function of H, the set of possible histories. H is characterized in MUSE via the set of O- and OVFUNs themselves, and although it may be possible to characterize H in other ways, (perhaps by defining a CONSTANT which could be a parameter to the ASSERTION), this second characterization would then need to be shown equivalent to the set of histories possible for the O- and OVFUNs.

However, every non-interference property mentioned in this chapter is effectively of the above form with an ASSERTION which is a function of H:

$$\begin{array}{l}\forall h \in \text{history}, h \in H \rightarrow \\ \quad \exists h' \in \text{history}, h' \in H \text{ and } P(h,h')\end{array}$$

where P is a relation between pairs of histories. Therefore, to state non-interference as a MUSE ASSERTION, H must be explicitly characterized within the ASSERTION. Except in the very simplest cases, this will be hopelessly complicated.

The problem above is simply that non-interference properties involve a relation between two different module histories, whereas the ASSERTION mechanism is designed to handle only one at a time. Two solutions have been proposed:

1. The non-interference property can be inferred from other, sufficient conditions which are more easily stated as an ASSERTION. The unwinding theorems of Goguen and Meseguer [18], and of Rushby [45], are examples of this sort of inference in the multilevel security domain. Whether these results can be generalized to other forms of non-interference, including the ones discussed in this chapter, is not clear.

2. Several histories of a module's behavior can be generated simultaneously under differing assumptions about inputs, and the behavior compared. As an example for fault-tolerance: a trace which includes faults, and one which doesn't, can be generated simultaneously from the module's specification, and the externally-visible parts of the two behaviors required to be identical. This solution will require a different method for generating verification conditions than now exists in MUSE. A method to generate such verification conditions for MLS non-interference properties in Gypsy is shown in the SDOS project [65]. This method is the basis for the example worked out in the next section. However, it can be applied only to a particular class of designs, and only to the simplest forms of non-interference specifications. Whether the method can be generalized is also not clear.

3.3.2 Example

In this section, we give an example of a simple fault-tolerant design, give formal specifications for that design which illustrate some of the fault-tolerance properties discussed in earlier sections, and present informal arguments showing why these specifications should be provable. In order to relate this example to the previous section, we present the design and its formal specifications in a MUSE- or HDM-like language. (The syntax used is a hybrid of MUSE and HDM. It probably can't be parsed by any existing verification tools.)

The design is a module which implements the storage of a single computer word, and uses triple redundancy to implement fault-tolerance. The normal functionality of this design should allow an external user to interact with the module by reading and writing the computer word in the normal fashion. However, the computer word is purely an abstraction, since the data contained in that word cannot necessarily be found in any particular part of the module's internal state. Instead, three words are used internally to redundantly store the abstract word. The design is fault-tolerant for any fault scenario which creates errors in at most one internal word at a time, and which has at most one fault occurring between any pair of read or write operations. At each 'write' operation, the contents of the three words are made consistent, and at each 'read' operation, if two or more internal words agree, then the agreed-upon value is output and the third word is made consistent with the others. Therefore, every 'read' or 'write' compensates for the possibility of a single fault. The performance of the design degrades ungracefully for fault scenarios worse than the ones described above.

The MUSE description of this design follows. In the design, a single type declaration is necessary for the (unspecified) DESIGNATOR type 'word'. The internal state of the module actually consists of three words, VFUNs 'w1', 'w2', and 'w3', which are intended to hold redundant data. There are two potentially state-changing operations: an OFUN 'write', which stores data into the abstract computer word, and an OVFUN 'read', which returns the data stored in the abstract word.

```
MODULE triple
TYPES
   word: DESIGNATOR;

FUNCTIONS
   VFUN w1() -> word w;
      HIDDEN;
   VFUN w2() -> word w;
      HIDDEN;
   VFUN w3() -> word w;
      HIDDEN;

   OFUN write(word value);
      EFFECTS
         'w1() = value;
         'w2() = value;
         'w3() = value;

   OVFUN read() -> word w;
      EFFECTS
```

```
         IF w1() = w2()
            THEN w = w1() AND 'w3() = w1()
         ELSE IF w1() = w3()
            THEN w = w1() AND 'w2() = w1()
         ELSE IF w2() = w3()
            THEN w = w2() AND 'w1() = w2()
         ELSE w = w1();
END_MODULE
```

To this design we must add a description of the possible faults. These will be included in a separate OFUN 'fault'. This new OFUN non-deterministically chooses one of the three internal words, and makes an arbitrary change to its value. We will permit OFUN 'fault' to occur, however, only if it fits into one of the fault scenarios for which this design is fault-tolerant. The restriction on when 'fault' may occur is expressed as an EXCEPTION 'not_in_fault_scenario'. Any ASSERTIONs which are proved will be contingent on the history of the system falling in the set of fault scenarios described this way.

Expressing that no more than one fault will occur between and pair of reads and writes requires that we record some properties of the system history. The O- and OVFUNs, along with their parameters or values returned, are taken to be the events of this history, and types 'event' and 'history' are declared. The actual system history is maintained in VFUN 'hist', and each O- and OVFUN is modified to record its own occurrence in 'hist'. The property which defines the possible set of fault scenarios is then expressed in terms of 'hist' as the EXCEPTION to 'fault'.

Having made these modifications, an invariant ASSERTION can be stated. This ASSERTION shows that the three internal words are maintained consistent at each non-fault event. (Further invariant ASSERTIONs will need to be supplied to prove this specification; we have not shown these.)

```
MODULE triple
TYPES
   word: DESIGNATOR;
   eventtype: {read, write, fault};
   event: STRUCT_OF(eventtype operation; word parameter);
   history: SEQUENCE_OF event;

ASSERTIONS
   (hist() ~= NULL  AND
   LAST(hist()).operation ~= fault) =>
      ( w1() = w2()  AND  w2() = w3() );
```

```
FUNCTIONS
   VFUN w1() -> word w;
      HIDDEN;
   VFUN w2() -> word w;
      HIDDEN;
      INITIALLY w = w1();
   VFUN w3() -> word w;
      HIDDEN;
      INITIALLY w = w1();

   VFUN hist() -> history h;
      HIDDEN;
      INITIALLY h = NULL;

   OFUN write(word value);
      EFFECTS
         'w1() = value;
         'w2() = value;
         'w3() = value;

         'hist() = JOIN(hist(),
            SEQUENCE( STRUCT(operation:write, parameter:value) ));

   OVFUN read() -> word w;
      EFFECTS
         IF w1() = w2()
            THEN w = w1() AND 'w3() = w1()
         ELSE IF w1() = w3()
            THEN w = w1() AND 'w2() = w1()
         ELSE IF w2() = w3()
            THEN w = w2() AND 'w1() = w2()
         ELSE w = w1();

         'hist() = JOIN(hist(),
            SEQUENCE( STRUCT(operation:read, parameter:w) ));

   OFUN fault();
      EXCEPTIONS
         not_in_fault_scenario:
            hist() ~= NULL AND
            LAST(hist()).operation = fault;
```

```
    EFFECTS
      ('w1() = 'w1() AND 'w2() = w2() AND 'w3() = w3()) OR
      ('w1() = w1() AND 'w2() = 'w2() AND 'w3() = w3()) OR
      ('w1() = w1() AND 'w2() = w2() AND 'w3() = 'w3());

      'hist() = JOIN(hist(),
        SEQUENCE( STRUCT(operation: fault, parameter: ?) ));
END_MODULE
```

Knowing that the internal state is consistent is not sufficient to show fault-tolerance, however. For example, rather than use majority voting within OFUN 'read' to determine which value becomes accepted, we could have chosen the minority value instead. The ASSERTION would still hold, but the design would not be fault-tolerant.

Let us show that the basic non-interference property of section 3.2.2 holds for this design. To do this, we must show that for any history represented by 'hist', there is another, fault-free history which has the same history of reads and writes as does 'hist'. Since there is no predefined way to refer to alternate histories in MUSE, we will show that the required fault-free history exists by constructing it.

Every VFUN, including 'hist', is implicitly a function of the abstract state. We could explicitly show this dependence on the abstract state by adding 'state' as a parameter to each VFUN definition. However, since only two histories must be compared, we will simply duplicate every VFUN. This has been done in the following MUSE module: for every VFUN of the previous specification, there is a duplicate with the suffix '_alt'. 'hist_alt' is then the alternate, fault-free, history. In the OVFUN 'read', the return value has also been duplicated, since values returned need not be the same in the actual history and in the alternate history.

In the alternate history, we retain the effects of inputs and outputs represented by the O- and OVFUNs 'write' and 'read'. Therefore, the actions of these functions are duplicated on the '_alt' VFUNs, and on their return values. We are trying to construct an alternate fault-free history, so the action of the 'fault' OFUN is not duplicated on the '_alt' VFUNs.

To show fault-tolerance, expressed as non-interference, for this design, it is sufficient if the alternate fault-free trace produced by this duplicated specification shows behavior equivalent to the actual history. The defined function, 'fault-free', removes from a trace all fault events, and leaves all non-fault events intact, including their order and their parameters. Therefore, it is sufficient for non-interference if we prove an ASSERTION which states that the alternate history equals 'fault-free' applied to the actual history. The externally visible history of reads and writes for the two histories will then be the same.

As in the case of the consistency invariant, other supporting ASSERTIONs must be proved in order that the equivalence of histories can be proved. For example, the majority value of the word VFUNs in the fault history must be shown to equal the value of all word VFUNs in the fault-free history.

```
MODULE triple
TYPES
   word: DESIGNATOR;
   eventtype: {read, write, fault};
   event: STRUCT_OF(eventtype operation; word parameter);
   history: SEQUENCE_OF event;

DEFINITIONS
   history fault_free(history h) IS
      IF h = NULL THEN NULL
      ELSE IF LAST(h).operation = fault THEN fault_free(NONLAST(h))
      ELSE JOIN( fault_free(NONLAST(h)), LAST(h) );

ASSERTIONS
   hist() ~= NULL  AND
   LAST(hist()).operation ~= fault => ( w1() = w2()  AND  w2() = w3() );

   hist_alt() = fault_free( hist() );

FUNCTIONS
   VFUN w1() -> word w;
      HIDDEN;
   VFUN w1_alt() -> word w;
      HIDDEN;
   VFUN w2() -> word w;
      HIDDEN;
      INITIALLY w = w1();
   VFUN w2_alt() -> word w;
      HIDDEN;
      INITIALLY w = w1_alt();
   VFUN w3() -> word w;
      HIDDEN;
      INITIALLY w = w1();
   VFUN w3_alt() -> word w;
      HIDDEN;
      INITIALLY w = w1_alt();
   VFUN read_w_alt() -> word w;
VFUN hist() -> history h;
   HIDDEN;
   INITIALLY h = NULL;
VFUN hist_alt() -> history h;
   HIDDEN;
   INITIALLY h = NULL;

OFUN write(word value);
   EFFECTS
      'w1() = value;
      'w1_alt() = value;
      'w2() = value;
      'w2_alt() = value;
      'w3() = value;
      'w3_alt() = value;

      'hist() = JOIN(hist(),
         SEQUENCE( STRUCT(operation:write, parameter:value) ));
      'hist_alt() = JOIN(hist_alt(),
         SEQUENCE( STRUCT(operation:write, parameter:value) ));

OVFUN read() -> word w;
   EFFECTS
      IF w1() = w2()
         THEN w = w1() AND 'w3() = w1()
      ELSE IF w1() = w3()
         THEN w = w1() AND 'w2() = w1()
      ELSE IF w2() = w3()
         THEN w = w2() AND 'w1() = w2()
      ELSE w = w1();

      IF w1_alt() = w2_alt()
         THEN read_w_alt = w1_alt() AND 'w3_alt() = w1_alt()
      ELSE IF w1_alt() = w3_alt()
         THEN read_w_alt = w1_alt() AND 'w2_alt() = w1_alt()
      ELSE IF w2_alt() = w3_alt()
         THEN read_w_alt = w2_alt() AND 'w1_alt() = w2_alt()
      ELSE read_w_alt = w1_alt();

      'hist() = JOIN(hist(),
         SEQUENCE( STRUCT(operation:read, parameter:w) ));
      'hist_alt() = JOIN(hist_alt(),
        SEQUENCE( STRUCT(operation:read, parameter:read_w_alt()) ));

  OFUN fault();
    EXCEPTIONS
      not_in_fault_scenario:
        hist() ~= NULL AND
        LAST(hist()).operation = fault;
    EFFECTS
      ('w1() = 'w1() AND 'w2() = w2() AND 'w3() = w3()) OR
      ('w1() = w1() AND 'w2() = 'w2() AND 'w3() = w3()) OR
      ('w1() = w1() AND 'w2() = w2() AND 'w3() = 'w3());

      'hist() = JOIN(hist(),
        SEQUENCE( STRUCT(operation: fault, parameter: ?) ));
END_MODULE
```

This method of duplicating VFUNs to produce an alternate history is sufficient to show non-interference in cases such as this one. However, not every design satisfying a non-interference fault-tolerance property can be verified in this way.

It should also be obvious that a significant part of the final specification is devoted to artifacts needed in constructing the alternate history. The example is simple, but generating a provable specification in this way is not. However, the transformation from our first description of the design in MUSE, to our final description of alternative histories, is straightforward and can certainly be automated.

Chapter 4

Conclusion

This study has examined several aspects of the specification and verification of distributed systems, especially in those areas where these systems differ from traditional sequential systems. A distributed system has been viewed as a collection of objects, or processes, that execute concurrently and only interact through messages. The lack of a global memory is necessary for systems with components that are geographically separated, but it can also be used for tightly-coupled systems. Synchronous communication is assumed, but asynchronous communication can be modeled by introducing the communication medium as a component of the system in which the form of buffering is described.

Several different categories of properties that comprise requirements for distributed computer systems have been examined. These are Security (including policies that are adaptive), Progress, Nondeterminism, Real-time Performance, and Fault Tolerance. Proof techniques have been given for security, progress properties, and fault tolerance. Real-time properties have been discussed, but their proof is still a research topic. The effect of nondeterminism on the other properties has been described. Other than the requirement that the process scheduler be fair and that it not be a source for information flow, there is nothing to be proven here. A collection of techniques for providing fault tolerance was given. The state-machine approach is especially appropriate for a system of communicating objects.

Formalisms that permit specifying program properties in such a way that verification can be performed have also been examined. Both safety and liveness properties can be specified in the state-transition model augmented with temporal logic. Therefore, this model is appropriate for security and progress properties. It can also be used to express the fairness assumption for the scheduler. It is not capable of handling real-time requirements, however. For this purpose it must be extended, possibly using the methods from the timed-CSP model.

4.1 Security

A definition of security and a method for proving that a system is secure has been given. Part of the definition is the requirement that security be compositional. This means that security proofs only need to be carried out for the individual objects that make up a system, rather than for the system as a whole. It also means that secure components can be added or removed without affecting the proof for the rest of the system.

The synchronous communication used in this report required the security proofs to consider the transmission of a message as two events. In a distributed system, however, many of the message transmissions between components will be asynchronous, with buffering occurring either in the transmission medium or at the receiver. In these cases, the components interact synchronously with the communication system and the communication system also forms a component of the system and should be proven secure. In particular, it should be shown to deliver messages to the right objects, to label the messages with the correct security level, and not to alter the contents of messages.

A concern for security proofs is whether or not a system can be designed that *is* secure. There may be a few objects that are not secure. These are known as the *trusted* objects of the system. An insecure event must be ignored to prove such objects. The occurrence of such events is monitored and an argument made that the rate at which information can flow from these events is so slow that they do not pose a security risk.

Any schedulers that arbitrate among nondeterministic choices must be trusted. Normally, these schedulers should not receive any classified information on which to base their scheduling decisions. An exception is the use of scheduling priorities and time requirements in a real-time system. Such schedulers must be shown not to leak any of this information.

4.2 Secure Distributed System Developer's Workbench

The project of which this study was to have been a part was going to build a prototype workbench using existing tools and then to design a production-quality workbench for use in developing secure distributed systems. Such a prototype should include the state-transition model for verifying specifications about the systems being built. A system in this model is a collection of communicating state machines, each of which is defined as a set of states, a set of transitions, and an initial condition. If the state machine is not coupled to a secure buffer, its input transitions must always be enabled and an acknowledgement is returned when they are acted on. The states include control predicates. Several recommendations for a prototype workbench are:

- Research remains to be done on the verification of real-time requirements. Therefore, the prototype should not attempt to deal with these requirements.

- Temporal logic is desirable for specifying safety and liveness properties. The temporal operators provided should be supported by the verifier.

- Research remains to be done on the verification of fault tolerance and therefore the verifier should not specifically support such verification. However, the prototype can deliver a specified level of fault tolerance by automatically replicating objects as needed, using the state-machine approach to coordinate the copies.

- Synchronized clocks should be provided by the system for each object. These are needed to generate message identifiers and also in making choices required in a real-time environment.

Appendix A

Notation

A.1 Sequences

Sequences are represented as follows:

- $\langle\rangle$ is the empty sequence.
- $\langle a, b, c\rangle$ is a three-element sequence. $\langle x\rangle$ is a sequence with one element.
- $s\hat{}t$ is the concatenation of sequences s and t.
- $t \upharpoonright S$ is the sequence formed from t by removing any elements that are not in set S.
- $|s|$ is the length of sequence s.
- $s \leq t$ is a proposition stating that s is a prefix of t. That is, there is a sequence s' such that $t = s\hat{}s'$.

A.2 CSP

Processes will be described using the programming notation of CSP [23]. This notation has been extensively studied and a formal semantics has been developed for it [7,9] from which the sets **E** and **T** can be produced. Methods for proving properties of CSP programs have been suggested [29,47]. If an existing specification language was used instead, it would have to be modified and semantics would have to be developed for it.

The CSP notation that will be used in this study is as follows:

STOP is the deadlocked process

SKIP is the process that terminates successfully

$e \rightarrow P$ is the process that performs event e and then acts like process P

P;Q is the process that acts like process P until it becomes SKIP and then acts like process Q

$P \mathbin{\Box} Q$ is the process that acts either like P or like Q

P||Q indicates that processes P and Q should execute concurrently. Any channel that the two have in common is used only to communicate with each other. (In terms of the semantics given in [7], ignoring operators are assumed to be used.)

$P \mathbin{|||} Q$ is like P||Q, except that there is no communication between the processes.

P\S acts like process P, except that any events contained in set S are concealed.

f(P) acts like process P, except that all events are renamed as specified by f.

μp.P is a recursively defined process.

$P \lhd b \rhd Q$ acts like process P if b is true or like process Q otherwise.

Communication events will be expressed as c!v for the sending of value v on channel c, and c?x to define x to be the value received on channel c. Most processes will input from channel **in** and output to channel **out**. These events will be abbreviated as ?x and !v respectively.

Process names could have been used for communication instead of channels. However, using channels permits greater flexibility in that several processes may send to or receive from a channel. If a single process can receive from a channel, the channel acts like a variable name for that process.

Bibliography

[1] T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice.* Prentice-Hall International, Englewood Cliffs, NJ, 1981.

[2] Algirdas Avizienis. The N-version approach to fault tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, December 1985.

[3] D. Elliott Bell and Leonard J. La Padula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, The MITRE Corporation, May 1973.

[4] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

[5] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, The MITRE Corporation, April 1977.

[6] K. Birman, T. Joseph, T. Raeuchle, and A. El-Abbadi. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering*, SE-11, June 1985.

[7] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, July 1984.

[8] S. D. Brookes and A. W. Roscoe. Deadlock analysis in networks of communicating processes. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI Series F: Computer and System Sciences*, pages 305–323. Springer-Verlag, New York, 1985.

[9] S. D. Brookes and A. W. Roscoe. An improved failures model for communicating processes. In *Proceedings NSF-SERC Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 281–305. Springer-Verlag, 1985.

[10] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[11] Maureen Harris Cheheyl, Morrie Gasser, George A. Huff, and Jonathan K. Millen. Verifying security. *ACM Computing Surveys*, 13(3):279–339, September 1981.

[12] Eric C. Cooper. Replicated procedure call. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, pages 220–232, August 1984. Vancouver, Canada.

[13] Dave E. Eckhard, Jr. and Larry D. Lee. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE Transactions on Software Engineering*, SE-11(12):1511–1517, December 1985.

[14] Michael J. Fischer. The consensus problem in unreliable distributed systems. *Int. Conf. on Foundations of Computation Theory*, August 1983. Borgholm, Sweden.

[15] Nissim Francez. *Fairness*. Springer-Verlag, New York, 1986.

[16] Amrit L. Goel. Software reliability models: Assumptions, limitations, and applicability. *IEEE Transactions on Software Engineering*, SE-11(12):1411–1423, December 1985.

[17] Joseph A. Goguen and José Meseguer. Security policy and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, April 1982. Oakland, CA.

[18] Joseph A. Goguen and José Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pages 75–86, April 1984. Oakland, CA.

[19] Donald I. Good, Robert L. Akers, and Lawrence M. Smith. Report on Gypsy 2.05. Technical Report 48, Institute for Computing Science, The University of Texas at Austin, October 1986.

[20] Orna Grumberg, Nissim Francez, and Shmuel Katz. Fair termination of communicating processes. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, pages 254–265, August 1984. Vancouver, Canada.

[21] J. Thomas Haigh, Richard A. Kemmerer, John McHugh, and William D. Young. An experience using two covert channel analysis techniques on a real system design. In *IEEE Symposium on Security and Privacy*, pages 14–24, April 1986. Oakland, CA.

[22] J. Daniel Halpern, Sam Owre, Norman Proctor, and William F. Wilson. Muse – a computer assisted verification system. In *IEEE Symposium on Security and Privacy*, pages 25–32, April 1986. Oakland, CA.

[23] C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, Englewood Cliffs, NJ, 1985.

[24] A. L. Hopkins Jr., T. Basil Smith, III, and J. H. Lala. FTMP—A highly reliable fault-tolerant multiprocessor for aircraft. *Proc. of the IEEE*, 66(10):1221–1239, October 1978.

[25] Pankaj Jalote and Roy H. Campbell. Atomic actions for fault-tolerance using CSP. *IEEE Transactions on Software Engineering*, SE-12(1):59–68, January 1986.

[26] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, January 1986.

[27] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[28] Leslie Lamport. An axiomatic semantics of concurrent programming languages. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI Series F: Computer and System Sciences*, pages 77–122. Springer-Verlag, New York, 1985.

[29] Leslie Lamport and Fred B. Schneider. The "Hoare logic" of CSP, and all that. *ACM Transactions on Programming Languages and Systems*, 6(2):281–296, April 1984.

[30] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[31] Luigi Mancini. Modular redundancy in a message passing system. *IEEE Transactions on Software Engineering*, SE-12(1):79–86, January 1986.

[32] Zohar Manna and Amir Pnueli. Temporal verification of concurrent programs: The temporal framework for concurrent programs. In Robert S. Boyer and J Strother Moore, editors, *The Correctness Problem in Computer Science*, pages 215–273. Academic Press, New York, 1981.

[33] Daryl McCullough. Specifications for multi-level security and a hook-up property. In *IEEE Symposium on Security and Privacy*, pages 161–166, April 1987. Oakland, CA.

[34] Daryl McCullough. Noninterference and the composability of security properties. In *IEEE Symposium on Security and Privacy*, pages 177–186, April 1988. Oakland, CA.

[35] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1980.

[36] J. E. B. Moss. Nested transactions and reliable computing. In *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, pages 33–39, July 1982. Pittsburgh, PA.

[37] Van Nguyen, Alan Demers, David Gries, and Susan Owicki. A model and temporal proof system for networks of processes. *Distributed Computing*, 1(1):7–25, January 1986.

[38] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[39] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Symposium on the Foundations of Computer Science*, pages 46–57, November 1977. Providence, RI.

[40] Amir Pnueli and Lenore D. Zuck. Verification of multiprocess probabilistic protocols. *Distributed Computing*, 1(1):53–72, January 1986.

[41] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[42] David Patrick Reed. *Naming and Synchronization in a Decentralized Computer System.* PhD thesis, M.I.T., September 1978.

[43] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. In *13th Colloquium on Automata, Languages and Programming (ICALP 86)*, volume 226 of *Lecture Notes in Computer Science*, pages 314–323. Springer-Verlag, 1986.

[44] L. Robinson and O. Roubine. Special—a specification and assertion language. Technical report, SRI International, Menlo Park, CA, 1977.

[45] J. Rushby. Mathematical foundations of the MLS tool for revised Special. Draft internal note, Computer Science Laboratory, SRI International, April 1981.

[46] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[47] Richard D. Schlichting and Fred B. Schneider. Using message passing for distributed programming: Proof rules and disciplines. *ACM Transactions on Programming Languages and Systems*, 6(3):402–431, July 1984.

[48] Fred B. Schneider. Synchronization in distributed programs. *ACM Transactions on Programming Languages and Systems*, 4(2):179–195, April 1982.

[49] Fred B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145–154, May 1984.

[50] Fred B. Schneider. Abstractions for fault tolerance in distributed systems. *Information Processing IFIP86*, pages 727–733, September 1986. Elsevier Science Publishers B.V. (North-Holland).

[51] Fred B. Schneider. The state machine approach: A tutorial. *Proc. of a Workshop on Fault-Tolerant Distributed Computing*, December 1986.

[52] R. K. Scott et al. Experimental validation of six fault-tolerant software reliability models. *FTCS-14*, pages 102–107, 1984.

[53] Omri Serlin. Fault-tolerant systems in commercial applications. *IEEE Computer*, 17(8):19–30, August 1984.

[54] Daniel P. Siewiorek. Architecture of fault-tolerant computers. *IEEE Computer*, 17(8):9–18, August 1984.

[55] Daniel P. Siewiorek and Robert S. Swarz. *The Theory and Practice of Reliable System Design*. Digital Press, 1982.

[56] B. A. Silverberg et al. The HDM handbook, Vol. II. Technical Report A006, Proj. 4828, SRI, June 1979.

[57] R. E. Strom and S. Yemini. Optimistic recovery: An asynchronous approach to fault tolerance in distributed systems. *FTCS-14*, 1984.

[58] David Sutherland. A model of information. In *9th National Computer Security Conference*, pages 175–183, September 1986.

[59] Ian Sutherland. Relating "Bell-LaPadula style" security models to information models. In *Proceedings of Computer Security Foundations Workshop*, pages 112–126. The MITRE Corporation, M88-37, June 1988. Franconia, NH.

[60] Ian Sutherland, Stanley Perlo, and Rammohan Varadarajan. Deducibility security with dynamic level assignments. In *Proceedings of Computer Security Foundations Workshop II*, pages 3–8. IEEE, June 1989. Franconia, NH.

[61] Dirk Taubner and Walter Vogler. The step failure semantics. In F. J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *4th Annual Symposium on Theoretical Aspects of Computer Science (STACS 87)*, volume 247 of *Lecture Notes in Computer Science*, pages 348–359. Springer-Verlag, February 1987. Passau, FRG.

[62] David J. Taylor. Concurrency and forward recovery in atomic actions. *IEEE Transactions on Software Engineering*, SE-12(1):69–78, January 1986.

[63] J. Shambhu Upadhyaya and Kewal K. Saluja. A watchdog processor based on general rollback technique with multiple retries. *IEEE Transactions on Software Engineering*, SE-12(1):87–95, January 1986.

[64] Stephen T. Vinter et al. The Secure Distributed Operating System design project. Technical Report RADC-TR-88-127, Rome Air Development Center, June 1988.

[65] D. G. Weber and Bob Lubarsky. The SDOS project – Verifying hook-up security. In *Third Aerospace Computer Security Conference*, pages 7–15, December 1987. Orlando, FL.

[66] J. H. Wensley et al. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proc. of the IEEE*, 66(10):1240–1255, October 1978.

[67] Pamela Zave. An operational approach to requirements specification for embedded systems. *IEEE Transactions on Software Engineering*, SE-8(3):250–269, May 1982.

NOTE: Although this report references the following limited document, no limited information has been extracted.

RADC-TR-89-376, Vol II: USGO agencies and private individuals or enterprises eligible to obtian export controlled technical data IAW regulations implementing 10 U.S.C. 140c; Feb 90. Other requests RADC (COTD), Griffiss AFB NY 13441-5700.