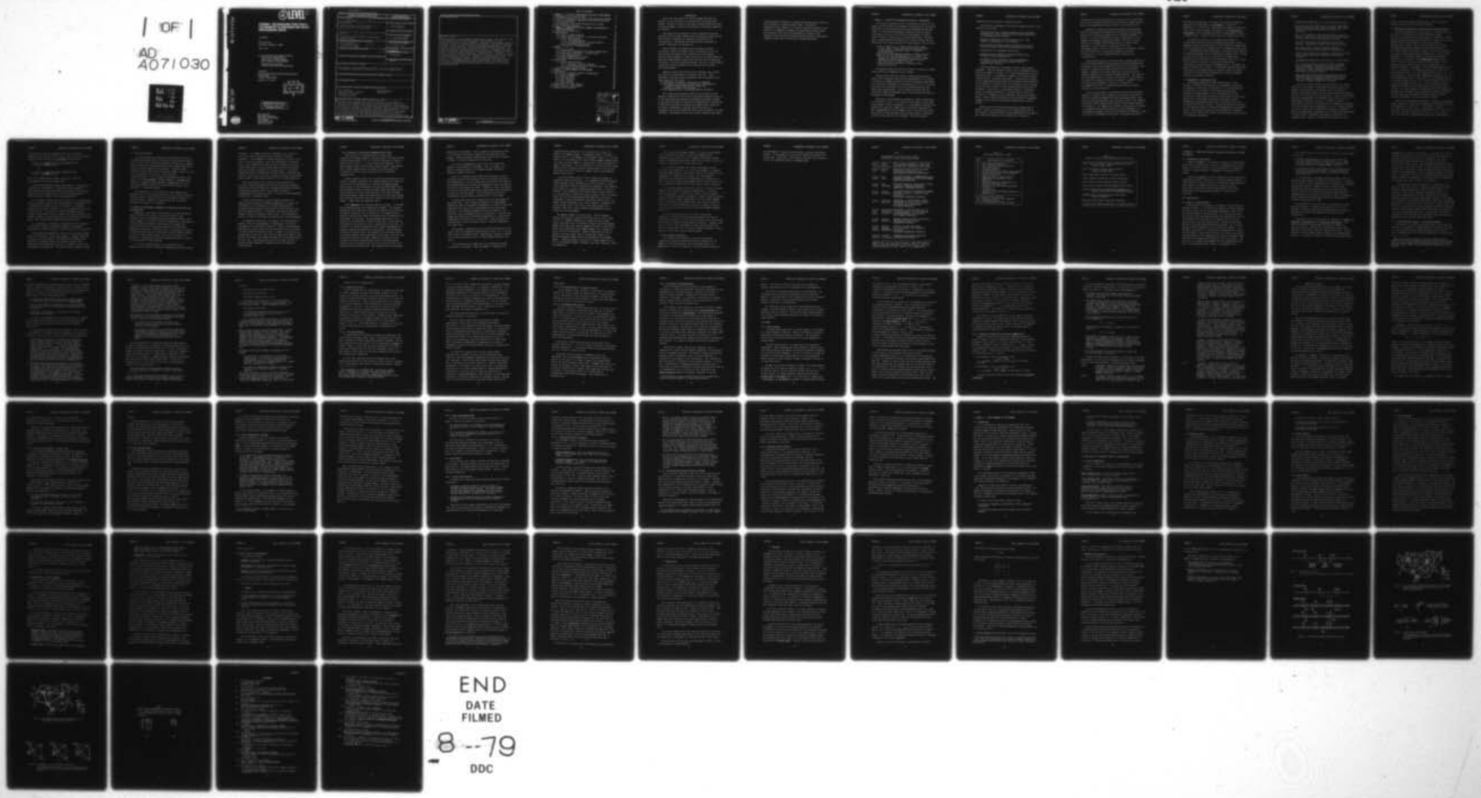
AD-A071 030 SRI INTERNATIONAL MENLO PARK CA COMPUTER SCIENCE LAB F/G 9/2
FORMAL TECHNIQUES FOR FAULT-TOLERANCE IN DISTRIBUTED DATA PROCE--ETC(U)
APR 79 J GOLDBERG, W KAUTZ, L LAMPORT DASG60-78-C-0046
NL

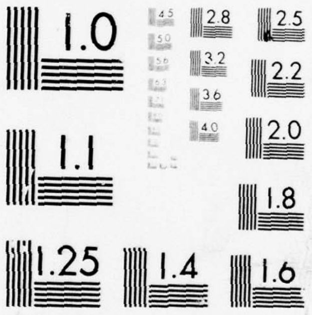
UNCLASSIFIED

| OF |

AD
A071030



END
DATE
FILMED
8--79
DDC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

SC

FORMAL TECHNIQUES FOR FAULT-TOLERANCE IN DISTRIBUTED DATA PROCESSING (DDP)

Final Report

SRI Project 7242
Contract No. DASG60-79-C-0046

April 3, 1979

By: Jack Goldberg, Project Leader and Director
William H. Kautz, Staff Scientist
Leslie Lamport, Computer Scientist
Peter G. Neumann, Program Manager

LSR
Computer Science Laboratory
Computer Science and Technology Division

Prepared for:

Ballistic Missile Defense Advanced Technology Center
P.O. Box 1500
Huntsville, Alabama 35807

Attention: Mr. J.E. Scalf

SRI International
333 Ravenswood Avenue
Menlo Park, California 94025
(415) 326-6200
Cable: SRI INTL MNP
TWX: 910-373-1246

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER Final Report	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) Formal Techniques for Fault-Tolerance in Distributed Data Processing (DDP).		5. TYPE OF REPORT & PERIOD COVERED (9) Final Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) Jack Goldberg, William Kautz, Leslie Lamport and Peter Neumann		8. CONTRACT OR GRANT NUMBER(s) (15) DASG60-78-C-0046
9. PERFORMING ORGANIZATION NAME AND ADDRESS SRI International 333 Ravenswood Avenue Menlo Park, CA 94025 (12) 72p.		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS SRI Project 7242
11. CONTROLLING OFFICE NAME AND ADDRESS (11) 3		12. REPORT DATE April 3, 1979 13. NO. OF PAGES 63
		15. SECURITY CLASS. (of this report) Unclassified
14. MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office)		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this report)

Distribution of this document is unlimited. It may be released to the Clearinghouse, Department of Commerce, for sale to the general public.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

fault tolerance
distributed data processing
hierarchical structure
processing networks
multiple data sets
diagnosability

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Distributed data processing can lead to systems with greatly improved reliability when compared with conventional systems, and can offer flexible strategies for redundancy, self-testing, and reconfiguration. DDP systems also have potential advantages in security, efficiency, and evolvability. The work in this contract is aimed at providing a suitable theoretical basis for the development of practical techniques for DDP system design and verification. Three fundamental issues have been investigated and are reported here: hierarchical structure in DDP systems; achievement of consistent redundant data

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified 410 646

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19. KEY WORDS (Continued)

20 ABSTRACT (Continued)

representations in faulty networks--specification, design, and validation; and self-diagnosability in DDP networks. This study indicates that the notion of hierarchical virtual machines is applicable to the client's application programs as well as to distributed operating systems (including multiprocessor systems). A study was made of general hierarchical organizations of recovery and initialization processes in networks with massive fault conditions. The study of consistency in multiple data sets points out a fundamental and serious problem that has not been discussed in the DDP literature. New results here give the precise conditions under which consistency may be achieved in a network containing multiple faults of the "erroneous" or "lying" processor types. The study of diagnosability include a review of existing literature on the structure of diagnosable networks, establish conditions under which multiple faults can be distinguished and located in DDP networks, and identify some key research problems in DDP-network diagnostics that need to be solved for the BMD application.

Table of Contents

INTRODUCTION

This is the final report for Contract DASG60-78-C-0046, for research on fault tolerance in distributed data processing (DDP) systems. The work is aimed at providing a suitable theoretical basis for the development of practical techniques for DDP system design and verification.

Distributed data processing can lead to systems with greatly improved reliability when compared with conventional systems, and can offer flexible strategies for redundancy, self-testing, and reconfiguration. DDP systems also have potential advantages in security, efficiency, and evolvability.

General methods for designing and verifying distributed processors to achieve particular levels of reliability do not yet exist because the possible fault conditions are very complex. Effective design methods can result from the use of formal methods for specifying system behavior under fault conditions and for analyzing the behavior of faulty interacting systems. The use of formal methods is made particularly desirable by the difficulties in testing distributed systems.

Research at SRI during 1978 and early 1979 under this contract has focused on formal methods for fault-tolerant DDP. Three fundamental issues have been investigated, and are discussed in the three chapters of this report.

(1) Hierarchical structure in DDP systems (P.G.Neumann)
(2) Achievement of consistent redundant data representations in faulty networks; specification, design, and validation (L.Lamport and M.C.Pease)
(3) Self-diagnosibility in processing networks (W.H.Kautz)

This study indicates that the notion of hierarchical virtual machines is applicable to the client's application programs as well as to distributed operating systems (including multiprocessor systems). A study was made of general hierarchical organizations of recovery and initialization processes in networks with massive fault conditions. The study of consistency in multiple data sets points out a fundamental and serious problem that has not been discussed in the DDP literature. New results here give the precise conditions under which consistency may be achieved in a network containing multiple faults of the "erroneous" or "lying" processor types. The study of diagnosibility has resulted in a review of existing literature on the structure of diagnosible networks, has established conditions under which multiple faults can be distinguished and located in DDP networks, and has identified some key reseach problems in DDP-network diagnostics that need to be solved for the BMD application.

1. Chapter 1: The Use of Hierarchical Structure in DDP Systems

This chapter discusses the use of hierarchical structure in the design and verification of DDP systems, extending some of the notions developed by the existing SRI Hierarchical Development Methodology (HDM). It includes some preliminary thoughts on specific uses of hierarchical structure to facilitate the development of reliable, secure, efficient, flexible, and verifiable DDP systems. The basic goal is termed "system controllability", by which is meant that the system shall have certain properties of good behavior, such as that it

* must be robust (i.e., it shall tolerate faults among a given class of faults, possibly including the ability of the system to reconfigure itself in a degraded performance mode),
* must be free from deadlocks (largely a software problem),
* must be free from disintegration (i.e., it shall not collapse into distributed components that can no longer communicate) wherever possible for faults outside of the class that it is able to tolerate,
* should do as well as possible in recovering from nontolerated faults.

The approach taken also notes the relation of recovery to system initialization and integrates the two concepts.

Hierarchically structured levels of abstract machines (also called virtual machines) have been used advantageously in the design of various new systems, particularly those with advanced requirements for reliability, recoverability, and security [22, 15, 13]. Such use facilitates the verification of critical system properties, and evolutionary system growth. Initial exploration shows that very significant potential advantages result from using such structures in the design of DDP systems.

The notion of virtualization provided by a level of abstraction is illustrated in Table 1.1 (at the end of the chapter), which shows several generic levels of abstraction. In this generic hierarchy, the operations at each level depend only on lower-level operations for their implementation, with the highest level at the top of the table. The notions of fault-tolerance that are either hidden or explicitly visible at that level are also indicated. Use of distributed implementations is indicated in upper-case letters.

Some potential advantages of hierarchical design structure in DDP systems are as follows.

- Controllability (e.g, freedom from deadlock and from system disintegration [loss of connectivity], continued possibility of reconfiguration and recovery)

- Efficient organization of fault-tolerance (e.g., fault diagnosis, reconfiguration, and recovery)

- Data security and system integrity (isolation of critical functions and data, with controlled sharing where desired)

- Extendability to accommodate new functions

- Portability of the system to different hardware and to different software configurations

- Verifiability of the design and of its implementation (including correctness of both computational and communications functions)

Various problems must be handled to achieve these potential advantages. These problems deal primarily with accommodating faulty behavior throughout the system, e.g., in data communications (errors, excessive delays), in processing, in memory, and in system software (synchronization, inadequate algorithms). In software the problems involve recognizing and recovering from inconsistent states among the virtual machines in different parts of the system, as well as among various different levels of virtual machines within a given part of the system. (The problem of inconsistent states in a distributed system is discussed extensively in the next chapter.) The basic solution is to integrate recovery completely into the basic system design. There is good understanding of this problem for integrated systems, but some issues in distributed systems require further research.

Another key problem involves designing and implementing the system to assure its verifiability, e.g., system controllability. This problem is being addressed in various system efforts through the use of HDM, which employs a strict separation of design and implementation, formal specifications of each part of a system and of the interactions among the parts, and a hierarchical design structure. Many of its techniques contribute directly to verifiability [19].

Although the concept of hierarchical design thus has numerous potential advantages in DDP systems, the existing notions developed primarily for centralized systems need to be extended somewhat. For example, the notion of hierarchical levels of abstraction is very appealing in designing conventional systems. This notion is also valuable in the design of DDP systems, and can be made directly applicable -- with the use of a particular view of the intercommunications. The use of such abstraction can greatly increase the controllability of a distributed system and the formal verifiability of critical system properties contributing to controllability.

The philosophical basis of the hierarchical approach used here is that distributed systems of hierarchically structured components can be designed cleanly and implemented efficiently, and that most of the benefits of using such structure can carry over to distributed systems. Therefore, this report takes the view that each of the component systems has a hierarchical design structure. The nature of the intercommunications is studied, and the extent to which they too can preserve hierarchical structure is examined. In general, there are significant advantages to some sort of hierarchical intercommunication (e.g., in assuring fault-tolerance, deadlock prevention, and verifiability).

An intrinsic problem in hierarchically constituted distributed systems is to assure that the component hierarchies of abstraction within the system are compatible in some constructive sense -- and that the protocols associated with the various levels in the different hierarchies provide adequate controllability (particularly freedom from deadlock). The compatibility problem can be solved fairly simply in a homogeneous distributed system, but is also manageable in a nonhomogeneous system as long as the hierarchies are comparable. Thus the notion of "conformable" hierarchies is introduced here and its implications are discussed. In essence, two hierarchies among the components of the DDP system are conformable if any desired communication is restricted to take place only between two compatible levels, one in each hierarchy. The system is then hierarchically controllable if the intercommunications can be guaranteed to result in no harmful loops -- either in control flow or in communication flow.

A hierarchical solution to the deadlocking problem in a nondistributed system is given by Dijkstra's hierarchical locking strategy [2] used in the THE system, in which deadlocks between levels can be avoided generically through the use of hierarchical priorities, i.e., a downward-only locking dependence. An analogous result applies to distributed systems, and is characterized informally here. One way to achieve controllability is to constrain the communications themselves to be hierarchically order preserving. Note that it would also be possible to achieve a controllable system without hierarchically constraining the intercommunications -- although much greater care must then be taken: in the general nonhierarchical case, verification of each system would be required, rather than generic verification of characteristic properties for classes of systems adhering to particular constraints.

1.1 Fundamentals of Hierarchical DDP

Good design methodology for DDP should maintain a clear distinction between design decisions and implementation decisions. For example, at each level of abstraction (i.e., for each abstract machine) there is a potential choice among different design alternatives regarding what is to be distributed and how, and decisions as to what should constitute the abstraction at that level. Within each of these design alternatives there may be several implementation alternatives under which the abstraction can be be realized efficiently and reliably. Several key notions are used here to illustrate the variety of design and implementation alternatives. However, hierarchical structure is used throughout as a framework within which to describe the options, without having to deal with each of the options individually.

SPECIFICATION ISSUES (What is to be distributed, and how?):

- At what levels should distribution of function take place? What should be centralized, what distributed? (Note hybrid designs.) Decisions involve security, reliability, efficiency.

- What is to be hidden? Should distribution be virtual or explicit? Options exist at each level, some being dependent on lower levels, some being independent thereupon.

- Reliability and security both imply that trust may be necessary. What must be trusted? What should be proved?

- What must be done in design to enhance provability, anticipating proofs of both design and implementation.

- How can initialization and recovery be incorporated into the design, taking constructive advantage of hierarchical structure?

IMPLEMENTATION ISSUES (How is distribution to be achieved?):

- For virtual distribution, at any particular interface, should irredundant or redundant distribution be used [in space and/or time]? (The latter can employ direct replication, various differing historical versions, and techniques like recovery blocks.)

- What algorithms will guarantee controllability? How can consistent choices of protocols between system parts be made? Can deadlock avoidance be proved? Can insecure information leakage (propagation) be avoided?

Virtual distribution implies that distribution (e.g., remote execution or data access, or parallelism in data and control) is hidden by the interface to the relevant level of abstraction. Explicit distribution implies that the distribution must be known to the user of the relevant level of abstraction. That is, the user must explicitly designate which file system a particular file is on, or which system a program is to run on. In general, explicit distribution at one level of abstraction can be masked (virtualized) at the next higher level of abstraction, e.g., in a virtual directory that tries to find files in different file systems. Futhermore, hybrid strategies may be interesting (e.g., using virtual distribution on initial use for convenience, and explicit distribution on subsequent use). However, this must be done carefully to avoid pitfalls such as the aliasing problems that produce incompatible effects from using different would-be synonyms.

1.2 Conformable Hierarchies and Controllable Distributed Systems

This section presents an initial attempt to introduce rigor into the notion of verifiably controllable distributed systems. Intuitively, two hierarchies are controllable if there are no harmful communications among the component hierarchies. Note that this can be achieved through nonhierarchical as well as hierarchical communications. However, the use of hierarchically constrained intercommunications among compatible hierarchies appears to be particularly effective and easy to analyze. This can be achieved if the relative ordering among levels of abstraction in two different components of a distributed system can be maintained whenever communication is permitted. Here communication implies explicit control flow (as in remote procedure calls and returns, or as in coroutine calls), implicit control flow (by signalling), or a combination thereof. For example, a potential deadlock may arise if level 7 in hierarchy A can communicate directly with level 5 in hierarchy B when level 6 in A can communicate with level 6 in B. This is an example of a nonhierarchical communication situation, e.g., a circular dependency chain, in which level 5 in A can depend on level 7 in B which can depend on level 6 in B which can depend on level 6 in A, which can depend on level 5 in A. Note that even if the only communication between the two hierarchies is that between the two levels 6, it is still possible for deadlocks to arise. However, such deadlocks can be analyzed directly, whereas the deadlock resulting from the circuitous loop involving different levels in each of different hierarchies is far more difficult to detect.

Definition of Conformable Hierarchies. Consider two hierarchical design structures Di and Dj, each consisting of a hierarchy of levels of abstraction, Di = [Ai0, Ai1, ..., AiI] and Dj = [Aj0, Aj1, ..., AjJ], where each Aik is a level of abstraction in Di (0 < k <= I), and each Ajk is a level of abstraction in Dj (0 < k <= J). (Level 0 [i.e., Ai0, Aj0] is in each case the lowest level of abstraction in its hierarchy.) Consider the mth level Aim in Di, which can communicate with the nth level Ajn in Dj. Then, the two hierarchical design structures Di and Dj are _conformable_ if and only if

> no level in Di _higher_ than Aim can communicate with a level in Dj _lower_ than Ajn,
>
> and
>
> no level in Di _lower_ than Aim can communicate with a level in Dj _higher_ than Ajn.

That is, the communication between Aim and Ajn is the only intercommunication involving either level.

Desired Theorem (Simple Dijkstra-like result). If every pair of hierarchies is conformable in a distributed system composed of intercommunicating hierarchical design structures, and if each such design structure is deadlock free, then the resulting system is deadlock-free with respect to the intercommunications.

As in Dijkstra's THE System, deadlocks may still occur within one level; in addition, they may occur here because of a dependency loop in communications between a level of abstraction in one design structure and a level in another (i.e., in the communication from Aim to Ajn and back). Note also that freedom from deadlock can be obtained with constraints that are weaker than those indicated by the theorem. However, the above approach gives a simple and easily proved generic way of simplifying the deadlock-avoidance problem.

It is desirable to investigate the generality of this approach, to see whether designs of distributed systems of conformable hierarchies are overly restrictive. This will be done in future work.

Some people have expressed the concern that distributed systems might not be amenable to hierarchical design. It appears that their concerns are not well founded. The same beliefs had been voiced previously in designing conventional systems, although recent experience now indicates that not only is it feasible, but that the advantages can be significant. The same conclusion appears to relate to distributed systems.

The concerns about the feasiblity of hierarchical structure in nondistributed systems seem to have arisen from the apparent need to have one level (e.g., processes) dependent on another (e.g., memory), and then observing a reverse dependence (that processes need virtual memory and virtual memory needs processes). A general design technique exists to get around this problem, namely to split one level into two separated levels, with a resulting proper hierarchicalization. In the example of processes and memory, this results from having _variable_ processes dependent on _virtual_ memory dependent on _fixed_ processes dependent on _fixed_ memory; see [15]). This general design technique for nondistributed systems is also applicable to distributed systems.

As noted above, it is also possible to have communications between a level in one hierarchy and either of two levels in another hierarchy without losing controllability, although the analysis is more complicated. However, for many applications the approach discussed here appears to be adequate.

1.3 Control of Performance, Degradation, Hierarchical Recovery, and Reconfiguration

Depending on the recovery time available, real-time systems may employ a variety of strategies to cope with losses in state information and in processing resources resulting from faults. One simple response is to enforce a deliberate degradation of performance, in which tasks that generate erroneous data are isolated from fault-free tasks, and low-criticality tasks are allocated fewer computational resources. Given more time, it is desirable to restore information about the state of external and internal system variables to the most recent correct values. Given still more time, it is desirable to reconfigure the available computing resources to provide the maximum possible computing power.

All of these techniques apply in real-time DDP, but are complicated by the need to coordinate corresponding activities in many processors. For example, some state information (e.g., about an externally observed object) may be distributed over several processors. If one component of state information is found to be in error, it may be necessary to label data accordingly or to discard data in other components that may have been corrupted by the error. A further complication in fault-tolerant DDP is the use of redundant, nominally identical computations at multiple nodes. It may take considerable effort to ensure that all such copies are provided with identical input data and that their outputs are properly synchronized despite delays in some results caused by local roll-back operations.

Given the possibility of different recovery times within concurrent recovery processes, and given the varying [and possibly substantial] time delays required to propagate data among processors, a considerable amount of skew in the timing of updates in corresponding in state data is likely. In fact, it may be impossible to get back to a consistent state.

The use of hierarchical structure appears attractive for organizing the processes of performance degradation, recovery, and reconfiguration. For example, all of these functions may be structured hierarchically and distributed at various levels of the executive system hierarchy. In operation, many of the lower-level processes would be invisible at higher levels. This organization tends to increase the efficiency and verifiability of such mechanisms.

The design of hierarchical, distributed, real-time systems for maximally efficient accomplishment of degradation, recovery and reconfiguration is essentially unexplored. However, many applicable techniques are closely related to those arising in the initialization of a hierarchically structured system. In general, initialization is accomplished one level at a time, from the lowest level upward. Each properly initialized level becomes the basis for the initialization of the next level. The same approach seems to be highly relevant in achieving minimal degradation, maximal recovery, and effective reconfiguration.

1.4 An Illustration: Distributed Implementations of PSOS

These concepts are illustrated here by considering a hierarchically designed computer operating system, the Provably Secure Operating System (PSOS) [15], [3]. The basic hierarchical design of PSOS is given in Table 1.2. From the table, PSOS appears to be a simple general-purpose nondistributed computer system. Nevertheless, PSOS is highly appropriate as an example, precisely because its hierarchical structure lends itself naturally to distributed implementations.

The most important levels of the hierarchical PSOS design are the capability manager (responsible for creating capabilities, which form the basis for all addressing in the system), the segment manager (which permits virtual addressing), the abstract-type manager (which permits dynamic creation of types and typed objects and helps enforce the encapsulation of their implementations), the directory manager (which maps symbolic names into capabilities), the user object manager (which maintains directory entries for objects), and the user process manager. These levels are summarized below.

A PSOS capability is a protected name for an object. It provides the only means by which that object may be accessed. It contains a system-wide unique identifier (UID) that is unique for the lifetime of the system, along with a set of access bits. For capabilities of objects of any particular type, the access bits are interpreted by the type manager for that type. The type manager defines all of the operations to be permitted on objects of that type, and also defines the meaning of the access bits for all capabilities of that type. All accesses to an object are mediated by the relevant type manager and require an appropriate capability for that object. The type manager interprets the access bits on that capability and determines whether the intended operation may be performed on that object. The fact that a purported capability is indeed a capability is enforced by the hardware, since every capability has a tag bit that unmistakeably identifies it as a capability and that can be set only by one of two capability-creating instructions. Thus capabilities are nonforgeable (assuming correct hardware). That a capability represents an object of the appropriate type must be assured by the appropriate type manager. Capabilities may be passed around from one user to another -- subject to certain communication restrictions.

In a typical implementation, most of the lower levels are expected to be implemented in hardware. Thus, for example, the capability manager and much of the segment manager would be in hardware or possibly microcode.

Table 1.3 illustrates how the PSOS system concept can lead to distributed versions of the system, first in the design, and then in the implementation. The table shows various levels at which potential intercommunications and protocols may meaningfully exist, and some of the forms the distribution can take. In general, virtual distribution and explicit distribution may both be accomplished by the introduction of new levels into the design. Virtual distribution may also be accomplished invisibly within an existing level of the design.

As noted above, each level in PSOS acts as the type manager for some object type. This conceptual distribution of type managers suggests one mode of distributed implementation for PSOS, in which the implementations of the various type managers can be distributed separately. The complete encapsulation of typed objects for any particular type also suggests a mode of distribution, in which a type manager may itself have a distributed implementation, coordinated by communication at that level or a lower level. These two modes are discussed next, at each of several levels of interest, beginning at the lowest level.

The creation of capabilities is centralized conceptually in the lowest level of the PSOS basic design shown in Table 1.2. It is thus used by each of the higher levels. Nevertheless, capability creation can be distributed with the aid of a simple system convention (see below).

The interpretation of capabilities is intentionally already distributed in the design. Each type manager is responsible for interpreting the capabilities for all objects of its type. For example, the segment manager -- implemented largely in hardware -- knows about all of the segments in the system of Table 1.2. It could, however, be distributed so that each of several segment managers knows only about its local segments -- using either virtual distribution or explicit distribution. On the other hand, a collection of distributed PSOSes could be conceived in which the distribution is done at the user object level and in which the segment level of each component system is left intact.

In any distributed version of PSOS, universal uniqueness of capabilities must be maintained. In a homogeneous distributed system of PSOS systems, global uniqueness is trivial to achieve, if each unique identifier contains a SITE_UID and a LOCAL_UID. (The LOCAL_UID can be implicit in locally interpretable capabilities, or could be explicit.) Since SITE_UIDs can be fixed forever at site creation, it is relatively easy to ensure that they are unique. Thus it is easy to ensure that the distributed-system-wide capabilities are unique. Consequently a central mechanism for generating capabilities is not needed, and each component can do its own creation. Recognition of foreign capabilities at this level is thus easy, based on the SITE_UID. Note that in such a scheme the SITE_UIDs must also be trusted and thus nonforgeable.

The segment manager is outlined above. The most commonly executed instructions in PSOS are those that read or write at some location in a segment designated by a capability and an offset. Virtual distribution can be achieved by having a distributed segment manager that knows about nonlocal segments and that can redirect the access to the appropriate local segment manager (e.g., basing its action solely on the SITE_UID). Explicit redirection can also be handled at the user object level by redirecting a symbolic but globally meaningful segment name to the appropriate local segment manager. (Permitting explicit requests for foreign segments at the level of the segment manager is probably a bad idea, although it could be implemented.)

Given a distributed segment manager, the abstract-type manager could be extended to take into account multisegment object implementations in which the various segments forming an abstract object are themselves distributed. Note that this effect can also be achieved by extensions to the user object level.

The distinction between the abstract type manager and the particular type managers (e.g., the directory manager) is important. Each type manager may use virtual and/or explicit distribution of its implementation, depending on its needs. Each type manager is responsible for encapsulating the implementation of objects of its type, although the abstract-type manager facilitates the isolation of the implementation capabilities from the abstract object capability.

The directory manager lends itself nicely to a virtual directory system in which there is distributed implementation of various directory structures. It would also be easy to provide redundant (e.g., distributed) directory information. However, explicit separate directory roots are better accomplished at the user object manager level. At the user object manager level, both virtual and explicit distribution of object creation and deletion is possible, with the options of distributed directories and distributed segment managers noted above.

At the user process level, both virtual interprocess communication (across distributed system boundaries) and explicit signalling are possible. At the user input-output level, both virtual input-output and explicit foreign names are possible. Similarly, both options exist for user environments (virtual name distibution and explicit foreign names) and for the user request interpreter (virtual command distribution or explicit remote logins).

1.5 Future Research Directions

This chapter has introduced the notion of hierarchical communications among hierarchically structured components in the design of a distributed system, and has discussed some of its implications. This notion appears to have considerable potential in the development of reliable, fault-tolerant, and secure distributed systems. It also appears to contribute to system controllability in various ways. Until now, it has received relatively little attention, but will be considered further in subsequent work.

Table 1.1

Some Examples of Virtualization of Fault-Tolerance in a Hierarchical System Design.

Visible functions	Hidden functions	Fault-tolerance mechanisms. [Protocols]. DISTRIBUTION IS INDICATED IN UPPER CASE.
Application	Network, nodes	Application rollback features and DATA DISTRIBUTION (redundant or not). [APPLICATION-TO-APPLICATION PROTOCOLS.]
Virtual network	Other nodes	DISTRIBUTED CONTROL OF COMMUNICATION,coding; REDUNDANT DISTRIBUTION OF DATA AND PROGRAMS [NODE-TO-NODE PROTOCOLS.]
Virtual system	Other subsystems	Compartmentalization, data security, system integrity; COOPERATING SUBSYSTEMS [SUBSYSTEM TO SUBSYSTEM PROTOCOLS.]
Virtual process	Process scheduling	REPLICATED PROCESSES; INDEPENDENT ALTERNATE PROCESSES; automatic rollback; process directories. [HIGH-LEVEL INTERPROCESS PROTOCOLS.]
Virtual i-o	Asynchrony Buffering	*DISTRIBUTED I-O ARCHITECTURES, REMOTE* CONTROLLERS; safe asynchrony; extensive handshaking and cross-checking. [I-O DEVICE PROTOCOLS.]
Virtual file system	Inaccessible directories, archiving	REPLICATION OF CRITICAL DATA, e.g., on on different media; FILE ARCHIVING AND ROLLBACK/RETRIEVAL; cross-checking. [FILE TRANSFER PROTOCOLS.]
Virtual memory	Storage addresses	Redundant physical address calculations; REPLICATION OF CRITICAL DATA. [INTERDEVICE PROTOCOLS.]
Virtual uniprocessing	Multiprogramming: dispatching	Reliable interrupt mechanisms; process isolation (e.g., domains of protection). [LOW-LEVEL INTERPROCESS PROTOCOLS.]
Multiprocessing	Processor coordination	Redundant interprocessor signalling [INTERPROCESSOR PROTOCOLS.]

Highest level is at the top of the table. Each level tends to hide some internal functionality from lower levels. Each level depends exclusively on lower levels for its implementation.

Table 1.2
Hierarchical Structure of PSOS

Level	PSOS Abstraction or Function
16	user request interpreter *
15	user environments and name spaces *
14	user input-output *
13	procedure records *
12	user processes * and visible input-output*
11	creation and deletion of user objects *
10	directories (*)[C11]
9	types and abstract objects (*)[C11]
8	segmentation and windows (*)[C11]
7	paging [8]
6	system processes and input-output [12]
5	primitive input/output [6]
4	arithmetic and other basic operations *
3	clocks [6]
2	interrupts [6]
1	registers (*) and addressable memory [7]
0	capabilities *

* = visible at user interface
(*) = partially visible at user interface
[i] = hidden above level i
[C11] = creation/deletion hidden by level 11

Table 1.3
Levels of Potential Intercommunications and Protocols

user request interpreter (virtual command distribution; explicit remote logins)
user environments (virtual name distibution; explicit foreign names)
user input-output (virtual i-o; foreign names)
user processes (virtual ipc; explicit signalling)
user objects (virtual; explicit remote creation)
directories (virtual directory system; explicit roots)
abstract types (virtual distributed implementation; explicit distribution of multisegment objects)
segmentation (virtual and explicit distribution based on SITE_UID)
primitive input/output (controller signalling)
capabilities (creation distributed; no dynamic protocol)

2. Chapter 2: Formal Specification, Design, and Validation of Robust DDP Systems

2.1 Robustness and Reliability

A system is robust if it continues to function correctly despite the failure or unavailability of some of its components; for a given design, it functions correctly as long as enough of its components continue to function properly. Robustness is considered in this chapter.

A system is reliable if it has a sufficiently high probability of functioning correctly. Reliability is attained through robustness. A robust system is reliable if there is a sufficiently high probability that enough of its components continue to function properly. Reliability as measured in terms of the probability of correct functioning is not considered here except, in the context of robustness.

2.2 Specification

2.2.1 What is a Specification?

A specification of a system is a statement of how the system is supposed to behave. It is perhaps best viewed as a contract between the system's buyer/user and its designer/implementer. When the two parties have agreed on the specification, (1) the buyer has stated that he will accept any system that meets that specification, and (2) the implementer has stated that he is legally bound to deliver a system that will meet that specification. Clearly there are many degrees of precision which specifications may have. (All too often, the buyer and the implementer are so closely tied together -- and frequently completely out of touch with the people who will actually use the system -- that the specification is not useful as a contract. Such specifications often prematurely bind various irrelevant design decisions, while completely avoiding other critical decisions. The result can be a system being designed [and even implemented] before there is any understanding as to what it is supposed to do.)

For a specification to serve as such a contract, it must satisfy two criteria.

1. It must be understandable to the buyer, so he can be sure that the system will behave the way he wants it to. Since the buyer is human (or a group of humans), this means that the specification must be understandable to humans.

2. It must be precise, so the implementer will know exactly how the system he provides must behave. (Deciding whether the contract has been fulfilled should be a technical decision, not a legal one.)

In the Hierarchical Development Methodology (HDM), there is a distinction made between requirements and specifications. In general, a requirement is a global property that all operations at a particular interface must satisfy. Examples in secure systems include system-wide security properties such as the military multilevel security policy used in KSOS and the properties concerning the nonbypassability of the capability mechanism in PSOS. An example in a reliable system is the Markov model defining the probability of recovery following faults in SIFT.

A specification, on the other hand, is a definition of what a particular operation or module of the system is to do. It is desirable to be able to show that each specification satisfies its requirements, and that each programs is consistent with its specification.

A specification can be broken into two components: a _logical, or functional, specification_ and a _performance specification_. The logical specification defines what the system (or module, or operation) must do. The performance specification defines what resources it may demand to meet its logical specification. Typical resources are time, computer memory, processors, and communication channels. For example, consider an airplane autopilot system that is implemented by a network of processors.[1] It receives input from the airplane's sensors and the pilot's controls, and sends output to the airplane's actuators and the pilot's instruments. The logical specification defines precisely what the output should be as a function of the input history. The performance specification defines (1) how long it may take to respond to inputs (how much time it may require) and (2) how robust it must be (how many properly functioning processors and communication links it requires to produce the correct ouput). (In general, the time and processor/communication resources required need not be independent of one another; slower response may be acceptable when fewer processors are available.)

It appears that these two components of a specification can always be identified. However, they may interact with one another. Different logical functions may require different amounts of resources. In particular, "graceful degradation" of service is specified by requiring that different logical functions require different amounts of processor/communication resources. For example, in an autopilot we may specify that attitude control must be performed correctly (e.g., the airplane must not go into a tailspin) despite the failure of two processors, but that navigation need only be performed correctly (the pilot's instruments must report where the airplane is) if at most one processor has failed. (As seen here, this type of interaction between the logical and performance specifications poses problems that have not been completely solved.)

2.2.2 How To Specify a System -- an Informal Discussion

The behavior of a distributed system consists of the concurrent activity of its components. Human beings tend to think of one thing at a time and have difficulty understanding concurrent activity. (This is demonstrated by the difficulty people have in writing correct

---

[1]Rather than choosing examples of ballistic missile defense applications, we here use examples which have been carefully studied. We trust that the reader will recognize the relevance of our work to ballistic missile defense despite the absence of explicitly related examples.

concurrent programs; such programs usually behave in ways not expected by their authors.) This presents a serious problem in trying to write a specification of a distributed system that will be understandable by humans. Our solution to this problem consists of writing the specification in the following three parts:

1. A sequential algorithm which describes the logical behavior of the system. We call this algorithm the _system machine_.

2. The relation between the input/output of the system machine, and the input/output of the individual processors which form the system.

3. Performance requirements on the execution of the system machine by the processors.

In terms of our decomposition of the specification into logical and performance specifications, the first two parts comprise the logical specification, and the third part is the performance specifications.

We illustrate this approach with two examples. First, we return to the example of an airplane autopilot implemented by a network of processors. The two parts of its specification are described below.

1. The autopilot may be described logically by an algorithm that at regular intervals -- say every 50 milliseconds -- accepts as input a vector (airspeed, altitude, angle of climb, throttle position, ...) of values and produces a vector (aileron position, rudder position, ground speed display value, ...) of outputs. The system machine consists of a precise specification of the algorithm that produces the output as a function of both (i) the input and (ii) the values of certain state variables. (Those state variables contain the values maintained by integrators in analog autopilots.) We discuss below the problem of giving a precise, formal specification for such a system machine.

2. The system machine defines the system as a single entity. In reality, the autopilot system consists of a collection of processors. The second part of the specification consists of a description of how the input and output of the individual processors are related to the input and output of the system machine. For example, a single value of the airspeed is part of the system machine's input. This value is derived from several sensors that measure the actual airspeed, each sensor being read by one or more processors. The airspeed input to the system machine must be defined as

a function of the airspeed sensor values read by the processors. This function must be designed so that it will produce a reasonable system machine input value despite the malfunction of some sensors, or of some processors reading the sensors. Part of the system machine's output is a rudder position. The actual control signals that cause the rudder to attain this position are sent by several processors. (Robustness dictates that the failure of one processor should not cause the rudder to assume an incorrect position.) We must define the control signals sent by the processors to the rudder as a function of the rudder position output of the system machine. Thus, part two of the specification consists of the collection of all of these functions that relate the processors input and output to the system machine input and output.

3. The third part of the specification consists of performance requirements for how the processors must execute the system machine. Below are some sample requirements.

    1. If a majority of the processors are functioning properly, then they should (collectively) correctly execute the system machine.

    2. A processor is considered to be correctly executing the system machine only if it reads all of its input values within 100 microseconds of the beginning of the system machine's 50 millisecond iteration period, and generates its output values at most 25 milliseconds later.

Our second example is a robust distributed data base system -- one that will maintain the correctness of the data despite the failure of some of the processors on which the data is stored. The data are assumed to consist of disjoint, named items. A user types requests to read or modify data items. For simplicity, we assume that a user sends all of its requests to a single processor, and that each request may either read or write a single item of data. (A data item is created when it is written for the first time.)[1] The specification consists of the following parts.

1. Every millisecond, the system machine executes one step. Its input consists of a sequence of requests. Each request

[1] It is quite easy to generalize this example to one in which the system is responsible for preserving the mutual consistency of different data items, but doing so would complicate the example.

contains:

- The identity of the requesting user
- The name of the data item
- The type of request (read or write)
- The value to be written (if it is a write request)

The output consists of a sequence of acknowledgements, one for each input request. An acknowledgement consists of:

- The identity of the user,
- An indication of whether there was an error, and, if so, the nature of the error (e.g., read of a nonexistent data item),
- For a correctly executed read request, the value read.

A single system machine step is performed by executing the sequence of input requests in order, updating the values of data items, and generating acknowledgements in the obvious way.

2. Each processor receives requests from many users. We may consider these requests to form a sequence. The system machine input is defined to consist of a sequence of requests formed by merging the sequences of unprocessed requests of all the individual processors. (An unprocessed request is one that was not included in the input to a previous step of the system machine.) The processors' outputs are defined so that each acknowledgement generated as output by the system machine is routed to its indicated user.

3. We make the following performance requirements for the system.

   - If a majority of the processors are functioning properly, then: (1) they will correctly execute the system machine, and (2) a properly functioning processor will generate an acknowledgment for a request within 0.5 seconds of its receipt.
   - The number of messages sent among the processors will equal, at most, 17 times the number of requests received.

   Of course, the values 0.5 and 17 are just illustrative -- whether this particular specification can be met will depend on the exact nature of the network of processors. The second requirement means that if no requests are are received, no messages are generated. This is a very strong constraint on the implementation.[1]

2.2.3 Formal Specification

We have discussed how the specification of a system can be broken into three components and given some informal examples. We now consider the problems involved in writing formal specifications. By "formal", we mean defined with sufficient precision to be suitable for mechanical manipulation. Given a formal specification of a system, it is in principle possible to verify mechanically that some particular implementation is correct. This would provide the proof that the implementer has met the contractual obligation defined by the specification. Formal specifications are useful even if such a mechanical verification is not feasible. Perhaps the most significant intellectual contribution of computer science is the idea that we really understand how to do something only if we can program a computer to do it. A formal specification is therefore one for which we really understand how to decide whether an implementation is correct.

2.2.3.1 The System Machine

The system machine is basically a sequential program that takes input values and produces output values. Considerable work has been done on methods for formally specifying sequential programs. Some of these methods provide very useful tools for specifying the _design_ of a system. However, we feel that they are not adequate for the type of "contractual" specification we have been discussing, because the specifications they produce for nontrivial programs are too difficult for a human being to understand.

The fact that these specification techniques lead to such complicated specifications has led many to believe that real programs are too complicated to have precise, formal specifications. However,

---

[1]This requirement is oversimplified. In practice, a small number of messages will be needed to maintain synchronization even if no requests are received. The point is that the 1 - millisecond repetition rate of the system machine should not mean that messages must be generated every millisecond.

we do not hold that view. Mankind has developed a very good language for precise human communication: the language of mathematics. It is a generally accepted assumption of the physical sciences that if we understand something, we can precisely describe it with mathematics. We believe this to be true in the world of programming too. If we understand what a program is supposed to do, then we can express that understanding in the language of mathematics in a way that is both precise and understandable to other human beings. If we cannot write such a mathematical specification, then it means that we do not understand what the program is supposed to do.

This belief has been borne out in the two simple but nontrivial examples described in [7] and [8].

We do not mean to imply that any program has a simple specification, merely that it has a humanly understandable specification. If the program is very complicated, then understanding its specification may require a considerable effort. To simplify the task of understanding the specification, hierarchically structured specifications may be used. One may view the "Reliability Model" and the "Allocation Model" of the SIFT system described in [22] to comprise a hierarchically structured specification of that system. Such a hierarchical decomposition corresponds to the ordinary mathematical technique of defining simple notations to represent more complex concepts.

Our experience indicates that to obtain understandable specifications, one must be able to use the extreme power and flexibility that the language of mathematics has developed during the past 2000 years. Current formal specification methodologies seem to be too restrictive and lack this flexibility. However, traditional mathematics is not formal in our sense of the term. The challenge that faces us is to develop formal methods -- methods susceptible to mechanization -- that will have the power and flexibility of traditional, informal mathematics. Boyer and Moore [1] have developed a mechanical verification system that is extremely flexible. We hope that it can provide the basis for the needed formal specification methodology.

2.2.3.2 The System Machine / Processor Relation

In the systems we have considered, it has been fairly simple to specify the relation between the input/output of the system machine and the input/output of the individual processors. We feel that methods which are sufficiently powerful to specify the system machine will easily be able to specify this relation.

2.2.3.3 Performance Specification

We first consider robustness specifications. As we have seen in our examples, robustness is expressed by stating that the system machine must be executed correctly if enough of the components are functioning properly. Correct execution of the system machine means that (1) the system machine input is based upon the correct input of all the properly functioning processors, and (2) all properly functioning processors correctly generate their output based on the same (correct) system machine output. This has been defined precisely in [9], using the language of traditional mathematics. To obtain more formal specifications, we are faced with the same problem for the system machine: devising formal methods that have the power of traditional mathematics.

This method of specifying robustness only allows an "all or nothing" specification -- it does not handle the problem of specifying "graceful degradation". A more general specification method dealing with this problem is described below.

Specifying other performance requirements, such as the requirement that an acknowledgement must be generated within 0.5 seconds of the receipt of a request, seems to be straightforward. Given a sufficiently powerful specification language that allows us to express the necessary concepts, there seems to be no difficulty in writing the specifications. However, such formal specifications are only useful if formal methods exist for verifying that an implementation meets these requirements. This problem will be discussed later.

2.2.4 A Hierarchy of System Machines

We have discussed how one can specify a single degree of robustness for a system by specifying robustness requirements for the implementation of the system machine. However, we often want to specify different degrees of robustness for different system functions. For example, in an autopilot, attitude control requires greater robustness than navigation, since losing control of the airplane is more serious than getting lost.

Our solution to this problem is a two-level hierarchy of system machines, consisting of a single high level executive machine and a collection of lower level task machines.[1] Each task machine specifies a collection of system functions requiring the same degree of robustness. For an autopilot, one task machine might specify the attitude control functions, and anotner task machine might specify the navigation functions. The executive machine specifies how resources are to be allocated to the execution of the different task machines. The executive machine's input consists of diagnostic information that it can use to determine which components (processors and communication links) are faulty. Its output consists of intructions to the processors that determine what system functions (which task machines) each processor is to execute, and when those functions should be executed.

The executive machine thus acts as a scheduler, and is responsible for controlling the graceful degradation of the system in the face of component failures. The executive machine must have the greatest degree of robustness, since its correct execution is necessary for the execution of any task machine. The robustness of each task machine is determined by the executive machine. To verify that a task machine's robustness specification is met we must (1) verify that the executive machine's robustness specification is met by its implementation, and (2) verify that the robustness of the executive machine implies the necessary robustness of the task

[1]This hierarchy of system machines should not be confused with a hierarchical specification of a single system machine.

machine. The latter verification involves only the executive machine's specification, and is independent of its implementation.

This type of hierarchy has been used in the SIFT system. The SIFT global and local executives essentially form an executive machine. In the description of that system [22], only this executive machine is specified. The applications tasks run by the SIFT executive constitute the task machines.

Our description of the hierarchy of system machines has been informal. The SIFT system is the only case in which such a hierarchy has been worked out in detail. We do not yet have a general method for writing a formal system specification in terms of a hierarchy of system machines.

2.3 Design

2.3.1 Basic Problem

The system machine specifies the logical behavior of the system by a single sequential algorithm. However, the behavior is actually produced by a distributed system of processors. The processors' behavior must be properly synchronized to correctly implement the system machine.

Processor synchronization is complicated by the robustness requirements. Many mechanisms have been proposed for synchronizing distributed systems of processors, but few have rigorously considered the problem of failed components. In order to be able to discuss the problem in a sufficiently precise fashion, we now introduce some formal notation for describing the concepts introduced informally in the preceding section.

We begin by precisely defining the concept of the system machine. We assume that we are given a sequence of times T_0, T_1, ... , a set I of possible inputs, a set O of possible outputs, and a set S of possible states. The system machine is specified by functions $\underline{sm}.\underline{next}.\underline{state}$ and $\underline{sm}.\underline{output}$ from I X S into S and O, respectively. If $\underline{sm}.\underline{state}(\underline{T_i})$ and $\underline{sm}.\underline{input}(\underline{T_i})$ represent the system machine's state and input at time T_i, then sm.output(sm.input(T_i), sm.state(T_i)) represents the system machine's output at time T_i, and sm.next.state(sm.input(T_i), sm.state(T_i)) represents its state at time T_{i+1}. To formally specify the system machine, one must formally specify the sequence T_0, T_1, ... , the sets I, O, S, and the functions sm.output, sm.next.state.

Let us assume that the system is to be implemented by a fixed set of processors labeled 1 through N. We now precisely define the concept of the correspondence between the system machine's input/output and the processors' input/output. For each processor k, let I_k and O_k denote the set of all possible inputs and outputs for that procesor. The system machine - processor correspondence is specified by a function <u>sm.input.merge</u> from I_1 X ... X I_N to I, and N functions <u>p.output.demerge</u>$_k$ from O to O_k. If p.input$_k$(T_i) represents the input to processor k at time T_i, then sm.input.merge(p.input$_1$(T_i), ... , p.input$_N$(T_i)) represents the input to the system machine at time T_i. If sm.output(T_i) represents the system machine's output at time T_i, then p.output.demerge$_k$(sm.output(T_i)) represents the output of processor k at time T_i. To formally specify the correspondence between the system machine's and the processors' inputs/outputs, one must formally specify the sets I_k, O_k and the functions p.output.demerge$_k$ for each k; and the single function sm.input.merge.

Synchronizing the processors means first of all guaranteeing that for each time T_i, each processor k produces the output p.output.demerge(sm.output(T_i)) for the same value of sm.output(T_i), and that the processors all produce these values at approximately the same time. However, we must do more than that; we must guarantee that the value sm.output(T_i) the processors use is the "correct" output of the system machine. We now consider just what constitutes correct output. We have defined the system machine's output and next state in terms of the functions sm.output and sm.next.state. Hence, correct output involves the application of these functions to the correct values of the current state and the current system machine input. The correct current state at time T_i is defined inductively to be the state correctly computed at time T_{i-1}. We would like to define the correct current system machine input to equal the value obtained by applying the function sm.input.merge to the correct inputs of all the processors. However, we cannot expect any processor to be able to determine the correct inputs to a faulty processor. Instead, the best we can hope to achieve is to have all the non-faulty processors execute the system machine step at time T_i using as input the same value sm.input.merge(v_1, ... , v_N), where the v_k satisfy the following property: if processor k is nonfaulty, then v_k equals the correct input to processor k at time T_i.

This condition can be achieved only in the absence of serious communication failure. If communication failure should destroy all communication paths between two processors, there is no way that they can learn about each other's inputs. Hence, they cannot be expected to determine the same user machine input. We therefore begin by considering perfect communication.

2.3.2 Interactive Consistency with Perfect Communication

Assume a collection of N processors, communicating with one another by means of a perfect communication network. Each processor k generates a value i/p(k) -- its correct input at time T_i . Every processor j computes a vector of values ($v_j(1)$, ..., $v_j(N)$) -- where $v_j(k)$ represents the value that processor j thinks is the correct value for i/p(k). These vectors of values must satisfy the following two conditions for all i, j, and k.

1. If processors j and k are nonfaulty, then
   $$v_j(k) = i/p(k)$$
   i.e., processor j obtains the correct value for processor k's input.

2. If processors i and j are nonfaulty, then
   $$v_i(k) = v_j(k)$$
   i.e., processors i and j compute the same vector of values.

If these conditions are met, then we have achieved <u>interactive consistency</u>.

We have developed an algorithm to achieve interactive consistency in the case of perfect communication. To illustrate the algorithm, we consider the case N = 4. We make the following assumptions:

1. The communication network linking the processors is complete. This means that any processor can send a message directly to any other.

2. The communication network is such that there is no way a non-faulty processor can be fooled about who originated a message. There might be distinct transmission lines between each pair of processors, for example. A message that processor j receives from processor k, for instance, may be pure garbage if k is faulty, but j can still observe which line it came in on and so know with absolute certainty that it came from k.

3. Every message originated by any processor has either the form F defined as

   F: / k / data /

   where the data is originated in processor k, or the form F' defined by

   F': / k / Msg /

   where Msg is a message of the form F or F'. That is, every message either originates the transmittal of data or is a relay of a message that has been received. In the latter case, the route the message has followed is indicated in the message, although the early part of the route may be stated falsely if the message has passed through a faulty processor.

4. The network contains just four processors, of which just one, say Processor 1, is faulty.

Consider the case in which processor 1 is the source of the data. Being faulty, it can send out any combination of the same or different values to the other processors. Consider the various cases:

Case 1 Processor 1 sends out the same value to all the other processors. There is no problem. None of the others is faulty, the value processor 1 sends out cannot be changed by any of the others. There is, no faulty behavior exhibited.

Case 2 Processor 1 sends one value, say A, to one processor, say processor 2, and a different value, say B, to the other two. This is interesting. Processor 3 and processor 4 both tell processor 2 they received the value B from processor 1. Since there is only one faulty processor, processor 2 knows that processor 3 and processor 4 cannot both be faulty. Therefore at least one of them must have truly received the value B. This is different from the value processor 2 received, so processor 2 can be certain that processor 1 is faulty. However, processor 2 cannot use this information.

Consider what processor 3 sees. It receives value B from processor 1 directly. Processor 4 says it received B also. However, processor 2 tells processor 3 it received the value A. There is no way processor 3 can tell if it is processor 1 that is faulty, or processor 2. If processor 1 were actually good, it would have sent processor 2 the value A, but, if processor 2 is faulty, it could be sending false information about what value it received.

To resolve this difficulty, consider the following algorithm. Let each of processor 2, processor 3, and processor 4 send to the others the value it received from processor 1. Each processor then has three values, the one it received from processor 1 directly, and the two values that each of the other processors says it received. If two of the three values a given processor has agree, let that value be used. Otherwise, draw the conclusion that processor 1 is faulty and ignore its data.

Processor 2 receives the same value B from both processor 3 and Processor 4. Therefore it uses this value, even though it knows processor 1 is faulty. Processor 3 receives the value B directly from processor 1, and the same value from processor 4. It does not know if processor 1 or processor 2 is faulty, but it uses the value B anyway. Similarly, processor 4 receives the value B directly from processor 1, and the same value from processor 3. Therefore processor 4 uses the value B. All three non-faulty processors use the value B. Interactive consistency has been obtained.

Case 3

Processor 1 sends distinct values to each of processor 2, processor 3, and processor 4, say A, B, and C, respectively. Then processor 2, for instance, received A directly from processor 1, B from processor 3, and C from processor 4. Since no two agree, processor 2 asserts that processor 1 is putting out garbage and should be ignored. Processor 3 and processor 4 reach the same conclusion. All agree that processor 1 is faulty, and interactive consistency has again been obtained.

The algorithm described under case 2 is an example of one that assures interactive consistency under the stated conditions. Algorithms have been developed that apply under broader conditions. In particular, if assumptions 1 through 3 are retained, but there are N processors of which not more than M can be faulty, an algorithm that achieves interactive consistency if N is at least as large as (3M +1) is given by [16]. We have also shown that this condition relating N and M is a hard one under assumptions 1 through 3. For any smaller value of N, the faulty processors can act in a way that will fool the non-faulty ones, so that interactive consistency can no longer be assured.

We have been assuming tacitly that a processor k always sends some message to each other processor containing a value for i/p(k). (If k is faulty, then it may send differing values.) However, we must be aware of the possibility that a failed processor will not send any messages. This possibility is handled by the use of timeouts. We assume that all processors have clocks, and the non-faulty processors' clocks are all synchronized to within some tolerance. (The problem of synchronizing the clocks is discussed below.) A processor sends its input value for time T_i when its clock reads T_i. The absence of a message with processor k as its destination can then be observed by processor k as the failure of the message to arrive by a certain time on its clock. Exactly when the message should arrive will depend on how many processors have to relay it.

The absence of a message is treated as if a message with the special value NIL has arrived. Conversely, the special value NIL can be sent by not sending any message. (In particular, processor 2 need not explicity inform processor 3 if it failed to receive a value from processor 1 -- it implicitly conveys that message by not relaying any value to processor 3.) This means that if i/p(k) equals NIL, processor k sends no messages, and there are no messages to be relayed. Hence processor k can inform all other processors that its input value equals NIL without any messages being sent! This is extremely important, because it makes our method feasible for situations in which input occurs relatively infrequently but quick response is still desired. In our distributed database example, user requests presumably occur sufficiently infrequently that for most of the one millisecond iterations of the system machine, there are no requests present. We simply let the value NIL denote the processor input consisting of the null sequence of requests. If no processors have any input requests, then the system machine input is the null sequence, so the system machine does not change state and produces the null sequence of outputs. In this case, there will be no messages produced (since all processors are sending the value NIL), and no processing needs to be done. It should be clear how the system can be implemented so that although the system machine is logically executing a "null iteration" every millisecond, there is no physical processing in the absence of user requests.

Assumption 3 may look fairly trivial but it is not. It is possible to allow a processor, when relaying a message, to add what we call an authenticator. Suppose processor k receives a message M from another processor. It relays this message in the form F" defined by

F": / k / M / S(M, k) /

S(M, k) is an encrypted version of the message M whose encrypting depends on M and k in a way such that it cannot be forged by any other processor. Another faulty processor faulty might accidentally create a valid authenticator for k , but we assume that it cannot do so intentionally. More precisely, we assume that the probability that it will accidentally create one is vanishingly small. The designing of an authentication mechanism depends on the type of faulty processors that can occur. If the faults are caused by random hardware failure, then an appropriate redundancy mechanism can be used. If the faults are caused by malicious intelligence, then cryptographic techniques must be employed. Such techniques have been developed [18], but are beyond the scope of this report.

We assume that the recipient of a message in form F" can examine the authenticator and either verify that it is the proper authenticator for M and for k , or else discover that k has not accurately relayed the message.

Note that if a message gets relayed several times, each time the relaying processor adds its own authenticator to the message. The message keeps growing longer, and the work of verifying it gets more elaborate. Nevertheless, authenticators do enable us to achieve interactive consistency without requiring the redundancy of having an N at least as great as (3M + 1). An algorithm for doing this is given in [16].

2.3.3 Interactive Consistency -- General Case

We now consider the general problem of achieving interactive consistency in an arbitrary network of processors and communication links when the communication links may be faulty. We define a communication link joining two processors to be functioning properly if it correctly transmits any message sent over it within some fixed length of time. A subnetwork of the network of processors is said to be functioning properly if all of its processors and communication links are functioning properly. The general statement of the interactive consistency problem for an arbitrary network of N processors is given below.

Assume that each processor k generates a data item i/p(k) at time T. Each processor j must compute a vector of values $(v_j(1), \ldots, v_j(N))$ by time T + D (for some fixed D) such that if a "large enough" subnetwork **M** of the network of processors is functioning properly, then for all processors i, j, k:

1. If j and k are in **M**, then $v_j(k) = i/p(k)$ -- if k is in **M**, then every processor in **M** obtains the correct input for processor k.

2. If i and j are in **M**, then $v_i(k) = v_j(k)$ -- all processors in **M** compute the same vector of values.

Of course, to complete the statement of this problem, we must specify precisely what constitutes a "large enough" subnetwork. We would like to make this "large enough" requirement as small as possible.

We have begun work on this general interactive consistency problem. We believe that, using authenticators, solutions are possible that satisfy the above requirement for arbitrary choices of the subnetwork M -- just as with perfect communication, authenticators permit solutions that will work despite any number of failures. We do not know what types of general solutions are possible without authenticators. However, it appears that for a solution to tolerate the failure of m processors, each processor must be connected directly to at least 2m+1 other processors.

2.3.4 Clock Synchronization

Our algorithms to achieve interactive consistency require that non-faulty processors have clocks that differ by at most some fixed quantity. Since real clocks do not run at exactly the correct rate, even if they are started in exact synchrony, they tend to drift apart. Thus, there must be some procedure for periodically resynchronizing them.

We assume that each processor can read its own clock and the clocks of all processors with which it is directly linked. We must design an algorithm in which each processor periodically makes these readings, and in which the values obtained are used to resynchronize the clocks. First suppose that each processor can read every other processor's clock and determine the difference between the two. We then get a matrix D of values, where $D_{i,j}$ is the difference between the clocks of processors i and j as measured by processor j. If either i or j is faulty, then $D_{i,j}$ might have any value. Assume that if i and j are both nonfaulty, then the value of $D_{i,j}$ differs from the true difference between the clocks by at most d. Then a synchronization method has been devised that results in the clocks of all nonfaulty processors differing by at most 4d after the syncrhonization has been performed. The method depends on the use of an interactive consistency algorithm to broadcast the values of $D_{i,j}$ to all the processors.

We are currently trying to generalize this method to an arbitrary network in which a processor can only read the clocks of its neighbors, so the full matrix of values D is not available. We are also trying to obtain more efficient methods, possibly employing authenticators directly instead of just using them to achieve interactive consistency when broadcasting clock readings.

2.3.5 Other Implementation Problems

Given syncrhonized clocks and a solution to the general interactive consistency problem, it is easy to devise an algorthm in which, if some "large enough" subnetwork M functions properly _at all times_, then every processor _in M_ will correctly execute the system machine. However, this would produce a system that might be unsatisfactory for two reasons:

1. It is requires that the same subnetwork M function properly at all times. For example, consider a network of three processors, in which any subnetwork containing two processors is "large enough". Such a solution would require that some two processors never fail. However, we might want a system in which processors could be repaired and brought back into the system. This would allow the system to continue functioning correctly even though every processor failed at some time -- so long as it never happened that two processors were faulty at the same time.

2. It allows a processor not in **M** to do anything. If a processor is functioning properly, but is isolated from **M** because of communication failure, we might want it to stop producing output.[1] (Of course, there may be situations in which it is better to do something than to stop, in which case we would want the processor to keep producing the best output it can.)

To solve both problems, it is necessary to enable a properly functioning processor to become aware that it is no longer part of any "large enough" properly functioning subnetwork **M** that is correctly executing the system machine. This could have happened for two reasons: (1) it lost communication with the rest of the system, or (2) it failed. The first case is easy to check for: the processor

[1]If a processor is faulty, we cannot expect to be able to stop it from doing anything.

just continually checks to make sure that it is using the same input as a "large enough" number of the other processors. It can turn itself off if it does not hear from enough other processors within a prescribed period of time.

Failure can be divided into two subcases: (1) hard failures that lead to detectable errors -- i.e., errors that can be detected by comparing the processor's output with the output of other processors, and (2) transient failures whose only result is to leave the processor with an incorrect version of the system machine state -- e.g., that causes it to have an incorrect value for some data item in the distributed database example. The first type of failure is easily detected. The second kind of failure leads to the problem of the "lurking fault" -- a fault that may not manifest itself until some future time. This can be extremely dangerous. For example, in the autopilot system, this error might only be discovered when the airplane enters some critical flight phase. The problem of detecting such lurking faults is addressed in the next chapter.

When a processor has discovered that it is no longer correctly executing the system machine, then the problem of restarting it is, in principle, straightforward. The processor must obtain enough information to allow it to update its version of the system machine state, and then it can simply join in with the other processors in executing the system machine. It must obtain redundant information from enough other processors (authenticated if it does not have direct communication with enough processors) to make sure that it obtains the correct system machine state. In practice, it will be necessary to minimize the amount of information transfer necessary to restart a procesor. Doing this will be a difficult programming problem, which will be very sensitive to the particular details of the system. [There does not seem to be much that one can say about this problem in general.]

2.3.6 Formal Design Methodology

Thus far, we have been describing an informal design for a system. There are two parts to this design:

1. The "logical" design -- the design of the system machine and the mapping functions sm.input.merge and $\text{p.output.demerge}_k$; and

2. The "distributed implementation" design -- the design of the robust method for implementing the system machine on the network of processors.

The first part is essentially the design of an ordinary, non-distributed sequential program, and is addressed by the SRI Hierarchical Development Methodology HDM ([15, 19]). Extensions to the formal development methodology to permit it to handle the second part of the design is now being explored in related work. Clearly, further experience and further research are needed.

2.4 Verification

We now consider the problem of formally verifying the correctness of the design. This means formally proving that the design meets the specifications. The specifications can be separated into (1) logical specifications and (2) performance specifications. These will be considered separately.

2.4.1 Logical Specifications

The design of a robust distributed system has been split into two subproblems:

1. Designing a sequential program for each processor k to implement the system machine and the input/output mappings sm.input.merge and $\text{p.output.demerge}_k$. (The parts of the program that implement the system machine and the mapping sm.input.merge can be the same for all processors.)

2. Designing a distributed algorithm to robustly implement the system machine by properly synchronizing the processors' programs.

Verification of the logical specifications involves first showing that they are satisfied by the sequential programs that implement the system machine and its input/output mappings on the individual processors, then showing that the distributed algorithm correctly synchronizes these programs. The methodology for verifying that the design of a sequential program satisfies such a logical specification is discussed in [19]. We now consider verification of the synchronization algorithm. We begin by describing the general work on the formal verification of concurrent programs, after which we discuss the particular problems presented by distributed systems.

2.4.1.1 Verifying Concurrent Programs

The fundamental method of verifying concurrent programs is described in [10]. It is an extension of the basic Floyd/Hoare method for sequential programs [5]. The method allows one to prove two types of logical properties:

1. Safety properties that state that "something bad does not happen" -- for example, that a bad output is not produced by a good processor.

2. Liveness properties that state that "something good must eventually happen" -- for example, that a good processor eventually produces some output.

A safety property is proved by proving that some assertion is invariant -- which means that if the assertion is true initially, then it will always remain true. Liveness properties are proved by a generalization of the type of "counting down" arguments used to prove the termination of loops in a sequential program. The proof of a liveness property usually requires first proving the invariance of some assertions.

The method is completely general and can be applied to any concurrent programming environment. However, unlike the method for sequential programs, a simple hierarchical decomposition of a proof is not in general possible. The basic reason for this is that with concurrent programs, two separate modules could be executed concurrently. This leads to a type of interaction between modules that is not present in the sequetial case. In particular, we cannot prove the correctness of a level of design without knowing how its modules are implemented by the lower levels. There are three possible ways to deal with this problem.

1. Restrict the possible forms of interaction between modules so that dangerous concurrent execution cannot occur. For example, one can prohibit the concurrent execution of any two modules that access the same data. This approach is feasible for systems that are implemented on a single computer, since such prohibitions are easy to enforce. It is used (for example) in the original design of PSOS [15]. However, this approach is not by itself applicable to distributed systems, because concurrent execution is a physical reality in a distributed system, and can only be prevented by very costly synchronization schemes.

2. Try to develop methods for specifying modules that contain all the information about their interaction with other modules that is necessary to prove the correctness of the design without having to know precisely how the modules are specified. Two different approaches at this are described in [11] and [12]. However, neither of these approaches have reached the stage of being applicable to large systems.

3. Recognize that we can only prove the correctness of the bottom level of the design, but use the hierarchical design process to simplify the construction of this proof. This is the approach described in [10], and is the only one now feasible for distributed systems. However, further research is needed to determine how well it works, and what sort of tools are needed to make such proofs practical for large systems.

2.4.1.2 Veryifying Distributed Concurrent Programs

The most obvious complication that distribution introduces into the verification of concurrent programs is that the communication mechanism must be regarded as a collection of processes -- separate from the processes representing the processors' programs. Typically, each communication line is represented by a process; the specification of that process consists of a precise specification of that communication line. Thus, there are many processes -- and the complexity of the proof tends to vary as the square of the number of different processes.[1]

Distributed synchronization algorithms seem to be inherently more complex than nondistributed ones. When a central control mechanism (such as a monitor in an ordinary multiprocess program) exists, the

[1]Two processes are not considered to be different if they execute the same algorithm with different constants (e.g., processor number).

invariant assertions used in the proof involve objects (such as variables) local to that control mechanism. However, for a distributed algorithm the assertions involve objects distributed throughout the entire system. The proof of correctness is therefore more "global" for a distributed algorithm, and it is more difficult to isolate the synchronization mechanism from the rest of the system.

Our initial efforts [4] indicate that it is possible to formally verify distributed algorithms. However, much more experience is needed to determine the limits of the current method, and what sort of further development is necessary.

2.4.2 Performance Specifications

Robustness specifications of the system become part of the logical specification of the synchronization algorithm. The requirement that the system performs correctly if "enough" components are functioning properly becomes the requirement that the synchronization algorithm satisfy certain correctness properties as long as "enough" of its processes execute their algorithms correctly. An informal proof of such a requirement is contained in [9]. No formal proofs of such robustness properties have been undertaken. We believe that the formal methods described above are adequate to verify these properties, and we intend to attempt such formal proofs in the future.

Other performance specifications of the system become performance specifications for the synchronization algorithm. To verify that these specifications are met, certain performance specifications must be placed on the processors' algorithms for implementing the system machine and its input/output mappings. For example, consider the problem of verifying the requirement for the distributed database system that an acknowledgement be generated within 0.5 seconds of the receipt of a request. The delay in generating the acknowledgement for a request received at time T_i consists of two parts: (1) the time needed for a processor to learn what the system machine input at time T_i should be, and (2) the time taken by the processor to process the list of input requests to produce its output. The first part of the delay is caused by the synchronization algorithm itself; the second part is caused by the processor's program that implements the system machine and its input/output mappings. The second part is usually assumed to be much smaller than the first, since it does not involve any interprocessor communication delays.

An informal verification of a synchronization algorithm's performance is given in [9]. Formal methods for verifying performance have not been developed. It appears to be possible to formally prove worst-case execution time bounds for sequential programs -- i.e., to formally derive upper bounds on the execution times of programs in terms of the execution times of individual program statements. A method for doing this for concurrent programs was also indicated in [10]. However, it appears that this method will give upper bounds that are so pessimistic they are useless. Further research is needed to develop a method that provides more realistic upper bounds on execution times.

We know of no practical methods for formally verifying any other type of temporal specification -- e.g., specifications of average execution times. Since such properties are important, it would be very useful to know how to verify them. Hence, research should be undertaken in this area.

Other types of performance specifications seem to require ad hoc approaches to their verification. For example, it should be easy to determine how many messages an algorithm can generate, so specifications of maximum communication traffic should be easy to verify. However, we know of no general methodology for verifying such performance specifications.

3. Chapter 3: Fault Diagnosis in DDP Systems

3.1 Introduction

It is already well recognized and appreciated that fault tolerance is an essential requirement for the DDP system currently under development for BMD application. What may not be so well appreciated is the extent to which the requirement of fault tolerance may modify, and even dictate, design considerations normally governed by other more familiar factors in a conventional non-fault-tolerant design. Many of the issues and implications of fault tolerance in large digital systems have only recently been understood. Others, especially those having to do with computer networks, are still awaiting clarification, test, and practical application. It is very important, therefore, that fault-tolerance issues be given serious attention at an early stage in DDP-system planning and architectural design. If the last decade of digital system design has taught us anything, it is that certain system-wide features, such as security, reliability (including diagnosability and survivability) and extendibility, must be provided for early and not handled later as an add-on or retrofit.

The long-range objective of the diagnosis investigation, whose initial stage is reported in this chapter, is to provide a strategy, algorithms, and design techniques for the diagnosis of DDP networks of the class represented by the BMD system.

Diagnosis includes the detection, location, and handling of faults, which may arise from either random or malicious causes. The purpose of diagnosis is to enable the whole system to continue to carry out its mission as effectively as possible, even in the presence of multiple faulty units.

The goals of this first year's study have been:

- To achieve a fundamental understanding of fault diagnosis in DDP systems
- To identify the main issues and problems requiring further research

- To relate these issues and problems to other aspects of the design
- To explore approaches to the solution of any of these problems for which conventional approaches appear to be inadequate, to the extent that time and funds permit.

The next section of this chapter discusses the application of general diagnostic theory to DDP systems; fault location is the main area in which research results are needed. Next, a system model appropriate to fault diagnosis is presented. The fault location problem is then discussed in greater detail: the questions that need to be answered, relevant published and new results on local testing, and some new results on communication and decision requirements in computer networks. The chapter closes with a summary of the major problems and results and our recommendations for continuing research.

3.2 Application of Diagnostic Theory to DDP Networks

3.2.1 Fault Accommodation

The life history of a fault in any large system for which fault accommodation is provided is marked by four events, shown in Figure 3.1; namely:

Fault occurrence (FO): The fault occurs and remains latent until it gives rise to an error indication.

Fault detection (FD): The presence of the fault becomes known at some level within the system, but its exact location is not necessarily and not usually known.

Fault location (FL): As a result of repeated errors or deliberate diagnostic steps, the location of the fault is determined to a fineness determined by the size of the smallest replaceable or reconfigurable unit (SRU).

Fault handling (FH): Finally, action is taken to circumvent the harmful effects of the fault through masking, repair, or reconfiguration.

Note that a "fault" need not be a hardware failure but can also arise from a design or programming mistake, a signal transient, or another externally induced modification of data or programs.

In this sequence, then, the DDP network accommodates the occurrence of a fault (FO) in three successive operations: FD, FL, and FH. It is then ready to handle a second fault by the same sequence, albeit with somewhat reduced resources (a smaller network). This process continues with succeeding faults as long as enough of the network remains to carry out the mission-oriented functions and the FD, FL, and FH operations themselves.

3.2.2 Multiple Faults

In many systems faults occur randomly and infrequently, so that the chance that the sequence shown in Figure 1 might be interrupted by a second fault can be presumed to be negligible. In a system such as large DDP networks of BMD type, however, the probability of simultaneous or correlated faults can become so large -- or the times taken for FD, FL and FH operations so long -- that the possibility of a second fault arising while the first one is still being diagnosed can no longer be neglected. Some of the possible sequences of FO, FD, FL,and FH events are illustrated in Figure 2.

If the chance of near simultaneous faults is not negligibly small, the designer has to provide specifically for a diagnostic capability to deal with two or more faults occurring at about the same time. Multiple failures can result from the chance occurrence of a widespread transient, such as a lightening strike; from a hardware or program weakness that can be manifested simultaneously in replicated units (such as similar nodes of a network); or from a malicious act, such as sabotage or enemy attack, which could disable two or more units at the same time. The time required for FL and FH can be kept small by careful system design.

It will become apparent shortly (if it is not immediately obvious) that the system cost of multiple fault accommodation -- measured in terms of required hardware, communication capacity, running times of programs, and file sizes -- increases very rapidly with the number t of simultaneous faults that need to be diagnosed simultaneously. Consequently, in specifying overall system requirements for a network, careful attention is required to:

- Determine the required overall system reliability
- Estimate the achievable reliability of individual SRUs and the communication links
- Determine the likelihood of occurrence of up to t near-simultaneous faults.

3.2.3 Fault Detection

Currently available FD techniques are many and varied; they encompass both hardware and software at all levels of design. The cost of adding or incorporating FD into a design is usually relatively small, normally less than 10 percent, and in any case is much less than the redundancy required for reconfiguration or masking.

New FD techniques are not likely to be required in DDP systems. Design choices must be made, of course, but the costs and effectiveness of each alternative can readily be determined from information available in the course of design. No fundamental difficulties in making these decisions are anticipated. Therefore, even though FD will surely play an important role in the actual design of a DDP system, particularly as the essential element of FL, it is not a critical research issue.

3.2.4 Fault Location

FL techniques deal mainly with the problem of how to select a non redundant set of FD tests that are collectively sufficient to pinpoint one or more faults to the SRUs in which they fall. Some systematic methods for FL are now available, but they were devised for gate networks, memories, and small Integrated Circuit packages and are not directly applicable to networks whose SRUs are the size of minicomputers or larger. There is only a little in the technical literature and in current practice to draw from in deriving a strategy and general procedure for identifying the faulty elements in a distributed network. In fact, the FL problem for DDP networks has not even been adequately formulated, nor have the critical aspects of the problem been identified. These matters are discussed in Section 3.4 below.

3.2.5 Fault Handling

The only effective method for FH in a system containing as much local autonomy as a DDP network, yet requiring rapid recovery following failure, is by programmed reconfiguration. Repair is too slow, and hardware masking is too costly. Moreover, to protect against the propagating effects of typical hardware faults, the SRUs must be programmmatically and electrically isolated from one another to the maximum extent possible consistent with effective communication. Reconfiguration must be controlled from a high, software level of the system hierarchy. Programmatic isolation of SRUs demands that all inter-SRU communication be carried out in such a manner that erratic or badly timed data from one SRU cannot block correct operation of another. For example, all data transfers may be carried out according to the rule that no item of information received from another SRU will be used until it has been confirmed by comparison with corresponding items of identical information recovered from other SRUs. Electrical isolation is required to reduce to very low the probability that a faulty SRU may forcibly jam the operation of another, non-faulty SRU to which it is connected by a communication link. Software and hardware fault isolation require some design effort but present no serious conceptual problems.

Handling faults by reconfiguration, assuming the minimal degree of fault isolation suggested above, is actually the same as providing a scheduling/assignment (S/A) algorithm for a fault-free network. The only apparent difference is that some of the SRUs and/or links of the fault-tolerant network will already have been identified as faulty and will have been disabled or will be disregarded for S/A purposes. The DDP network architecture must anticipate these losses by providing enough redundancy and flexibility of function that satisfactory S/A can be carried out for critical task programs, in the face of all possible deletions of disabled SRUs and/or links up to the allowed maximum number t. The S/A algorithm is otherwise independent of fault diagnosis procedures; that is, it does not depend on how the faulty elements have been identified, nor does it dictate how the FD and FL procedures should be carried out.

An efficient and effective S/A algorithm is an important part of the distributed operating system of any DDP network, with or without fault tolerance. It will be used fully in the fault recovery process. However, its design is not greatly affected by the diagnostic strategy adopted and the diagnostic procedures used, and this strategy and these procedures do not themselves depend critically on the S/A algorithm.

FL therefore emerges as the only one of the three steps of fault diagnosis requiring serious research attention.

3.3 System Model for Fault Diagnosis

Before delving more deeply into the FL problem, it will be necessary to set up a model of the class of DDP networks under consideration. This model is best expressed mathematically in terms of a _graph_ G with suitably labeled nodes and edges.

The conventional and obvious representation will be employed. It identifies the _nodes_ of the graph with the SRUs of the DDP network -- not all of which are necessarily separated geographically. The _edges_ of the graph correspond to the direct communication links between SRUs. These are presumably bidirectional, though there is no implication that the communication capacities (bandwidths) in the two directions are the same. A typical small graph G is shown in Fig. 3.

In a full network model created for DDP design, each node would be characterized (labeled) to indicate the type of processing and memory capabilities that it provides. Edges would be labeled similarly to indicate the type and grade of communication. For diagnosis, however, it is not necessary to label the nodes fully but only to distinguish three types:

- _Executive nodes_. Full power for controlling the network resides in this set of nodes. This capability includes the diagnostic interpretation of sets of reports from tests conducted locally in the network for FL, most if not all of the operating system, and the S/A program used for reconfiguration control. The subset of executive nodes can be expected to be richly coupled by edges.

- _Regular nodes_. These have the capability to test themselves and other nodes, and can be tested by others, but have no authority to take action on the results of these tests beyond sending the results to the executive nodes.

- Slave nodes. These can be tested by other nodes, but have no power to test other nodes.

This graph model can also aid the development of an S/A algorithm. For this purpose each network task would be described by a set of alternative labeled subgraphs (each normally consisting of one or a few interconnected nodes) sufficient to execute that task. The corresponding running-time estimates would also be specified. Again, however, for diagnostic purposes it is adequate and merely to assume that every node has sufficient processing and memory capability to (1) store and apply testing programs to itself and (for non-slave nodes) to selected neighboring nodes, (2) initiate the transmission of the test results to all of its neighbors and (3) relay all to its neighbors. received messages regarding test results.

Another extension, to be considered later, provides for several fault status levels. It has been assumed so far that each node (SRU) is either faulty or nonfaulty. Actually, it will probably be worthwhile to distinguish between nodal faults having different probabilities and different consequences, thereby providing various levels of confidence and utility at individual nodes. It will not then be necessary to reconfigure a node out of use for a fault that affects only an infrequently run, low-priority program; the first occasion of a transient fault; or a difficult-to-locate fault that may have actually occurred in another node. Also, a properly designed node will usually retain its capability for relaying received messages to other nodes, even though it is crippled internally. Finally, it may be possible to retain memory readout capability from a node despite failure of its procesor, so that data and programs can be transferred to a non-faulty node rather than be completely lost.

Inclusion of any of these alternatives will require an increase in the number of levels of fault status to more than two. This extension will complicate the FD, FL, and FH programs but it can be expected to yield much higher network reliability for a given degree t of fault protection.

3.4 Fault Location in DDP Networks

FL requires three steps:

- Testing, in which tests are conducted among small sets of nodes in the network.

- Communication, in which the test results are relayed through the network to the executive nodes.

- Decoding, in which the executive nodes calculate which nodes of the network are actually faulty.

At the global (network) level, all three steps must somehow be carried out correctly, even in the presence of up to t nodal faults.

These three steps will now be discussed one at a time.

1. Testing

Two types of error reports from testing can be distinguished:

1. A report generated spontaneously by a node and communicated to its neighbors as a result of its own self-diagnosis action.

2. A report generated by one or more neighbors of a node as a result of testing operations conducted on that node by these neighbors.

Type 1 can probably be relied on to detect, during normal operation, almost all faults at the relatively low cost required for even high-level self-diagnosis. Type 2, though more expensive in time and hardware, will be necessary to ferret out the remaining faults whose probability is lower but whose effects on overall reliability can be much greater: latent and lurking (long-term latent) faults, and the various multiple faults affecting two or more nodes (SRUs) at about the same time. Only type 2 requires the development of a test strategy and an algorithm for fault diagnosis.

In both types the initial error report resides in the node in question or its neighbors, ready to be recommunicated through the network to the executive nodes.

In the simplest case, a local test consists of a node A of the network testing another node B, as indicated in the two-node subgraph shown in Figure 4. The directed edge or <u>arc</u> between A and B distinguishes which node is the tester and which is being tested. (The arrow designates the direction of control -- not information flow, which must be bidirectional.) For testing purposes, then, the testing <u>graph</u> takes the form of a <u>directed</u> graph G_T whose nodes correspond to those of the original graph G (hence, to the SRUs of the DDP network), but whose arcs indicate which nodes are capable of testing which others. Clearly, the set of arcs of G_T must be a subset of the set of edges of G. The graph shown in Figure 5 illustrates a possible testing graph; it corresponds to the example given in Figure 3.

The testing problem is not trivial. An error report e from the test A-->B reflects the true failure status f_B of node B (e=0 if B is nonfaulty; e=1 if B is faulty), but only if A is itself nonfaulty (f_A=0). If A is faulty (f_A=1), then the test result e may be 0 or 1 and is meaningless (e=X). This observation, summarized in Table 3., permits an e-label (0, 1, or X) to be attached to every arc in the graph G_T, provided only that all nodes are labeled with their f-values (0 or 1). In other words, if the fault status of every node is given, one may readily determine the values of all error reports.

The inverse problem -- that of determining a consistent node labeling corresponding to a given arc labeling -- is the <u>decoding</u> step mentioned above and discussed below. A valid solution must be independent of the Xs; that is, it must hold for an arbitrary assignment of values 0 and 1 to the Xs. It is obvious that if there are too many Xs, complete decoding is impossible. For any given graph G_T, only a maximum number of faulty nodes may be located. The graph must be designed so that a redundant and overlapping pattern of local tests will accurately locate up to t faults in the network.

Actually, local testing may take place in ways other than that indicated in Figure 4(a). The self-test shown in Figure 4(b) corresponds to an error report of type 1. Even though most faults can be expected to reveal themselves first during the running of a regular task program, a self-test is of limited use for FL since, as indicated in Table 4(b), the test result e = 0 provides no definitive fault information at all. Hence, the fault coverage of self-testing can be expected to be low. Indeed, no separate segment of hardware or program can ever be counted on to check fully its own fault status.[1]

Other modes of local testing are indicated in the rest of Figure 4. The alternative in Figure 4(c) is especially worth noting because, if nodes A and B are similar, identical programs may be run on them and the key variables can be compared frequently for agreement. Thus, fault coverage is high and there is prompt detection (normally with local location as well) when a fault first appears. In Figure 4(d) nodes A and C are required to test B (either or both of the dashed arcs may be missing), so a failure in either A or C can render the test result meaningless. Finally, in Figure 4(e) node A tests B and D simultaneously, but B and D each participate in the testing of the other. Clearly, many other test modes are possible. (A general network testing model encompassing these and other testing modes is described by Russell and Kime [20, 21].)

With this background, it can be appreciated that there are important fundamental questions of how the size and interconnection structure of a network are related to the type and extent of local testing required for a desired degree of fault locatability. Most crucial for design purposes, it will be necessary to determine the limits on the size and interconnection structure, and the types of local tests required, as functions of the prescribed number, t, of faults that are to be accommodated simultaneously. The total number of faults that can be accommodated before repair may also be relevant. These questions must be answered before the architectural design of the DDP network can be completed.

[1]The usefulness of self-tests may be improved statistically by "stretching" task programs into a diagnostic mode wherein they are run with artifical inputs, blocked outputs, and internal changes arranged to flex little-used portion of the hardware and software of the SRU. Nevertheless, the limitations of self-testing are fundamental.

Other problems in local testing, equally important but not so urgent, involve extensions to the fault model assumed here: how to handle faults on communication links (edges), transient and intermittent faults, and multiple fault status levels (nodal faults that are less than catastrophic).

A few results are available in the technical literature for the A-->B mode of the general testing problem [Figure 4(a)]. Preparata et al [17] have derived necessary conditions for the minimal size and the connection pattern of networks capable of locating t faults at the same time (one-step FL). They showed that the number n of nodes in G_T must be at least 2t + 1, and each node i must have in-degree $d_{ini} \geq t$. (Parallel arcs and self-loops are disallowed.) They also determined a family of node-symmetric networks based on star polygon graphs that achieve these numerical bounds exactly and are one-step t-fault-locatable. When multiple faults can be located in sequence instead of in one step, weaker limits apply: in terms of the number N of arcs, we have $N > n + 2t + 2$, instead of $N \geq nt$. This bound also is shown by construction to be achievable for t-fault location.

Hakimi and Amin [6] derived three sets of necessary and sufficient conditions for one-step t-fault locatability in A-->B testing. The first set contains the two inequalities stated above -- $N \geq 2t + 1$ and $d_{ini} \geq t$ for every node i-- plus a third condition: no two nodes test each other. That is, the first two inequalities must continue to hold for at least one set of arc removals that break all two-node loops of the type shown in Figure 4(c). The second set of necessary and sufficient conditions requires that $N \geq 2t + 1$ and $c(G_T) \geq t$, where c is the connectivity, defined to be the smallest number of nodes of the graph whose removal yields a graph that is not strongly connected. (The in-degree condition is satisfied automatically.) The third set of necessary and sufficient conditions contains the same two inequalities stated above, plus a third requirement that says, in effect, that every subset of n-2t+p nodes must have more than p successors, for all p = 0, 1, . . . t-1.

All three sets of conditions are very general but algorithmically awkward and would be difficult to implement efficiently. More workable conditions need to be derived, not only for the A-->B case but for other testing modes as well. It has not yet been demonstrated that this algorithmic complexity is fundamentally necessary.

2. Communication

The distribution of test results around the network, and to the executive nodes in particular, is one instance of the interactive consistency problem discussed in Chapter 2. The "value" of the message originating at each node A is now the set of outcomes of the tests performed by A on its neighbors in the graph G_T. Note that if B is faulty, the test result e_{AB} need not be the same as the results of other tests of B -- for example, e_{DB}, e_{EB}, -- unless we also assume that all tests of node B were identical and executed effectively simultaneously. This assumption is probably not justified for practical networks, in view of the possibility of marginal and transient faults if not for other reasons. Consequently, it cannot realistically be assumed that all fault-free testors of a given node will agree with one another.

The interactive consistency bound $N > 3t$ presented in Section 2 is directly applicable to the communication problem of diagnosis only when the graph G is complete and all nodes are executive nodes. Thus, this bound and algorithm need to be extended to cover the more arbitrary graphs under consideration here. Also, just as for testing, an extension is needed to allow for faults on communication links (edges) and certain multiple fault levels within nodes. Finally, it may be necessary to remove the assumption that the identity of the sender of a communicated message is always known and not subject to error.

The heaviest communication load requiring interactive consistency will arise from self tests rather than A-->B or other test modes, but the communication requirements are otherwise the same in both cases.

These extensions will require some investigative effort, but none of them appear to be insurmountable.

3. Decoding

The executive nodes must be provided in sufficient number n_E to compensate for faults in at most t of them -- namely, $n_E \geq 3t$. Assuming that each of them has received reports from all tests that have been performed, the problem remains of determining from this information which nodes are actually faulty. As a consequence of interactive consistency, one may be assured that all non-faulty executive nodes are working with the same set of test results. (They may have a common NIL value for some node, indicating the lack of an agreeing majority among reports received from that node by different routes). Consequently, as long as they employ the same decoding algorithm they will all reach the same conclusion about which nodes are faulty.

An example of the decoding problem for A-->B testing is illustrated in the three parts of Figure 6. The graph G_T in Figure 6(a) has been node-labeled with a hypothetical pattern of node faults (1s). In Figure 6(b) the resultant error reports are appended as labels to the arcs of G_T. In Figure 6(c) an arbitrary assignment of 0s and 1s to the Xs has been made, and the information regarding which nodes are at fault has been deleted. A successful decoding algorithm must regenerate these node labels from the arc labels.

Even when a network is known to be t-fault locatable, no general decoding algorithm is now available. This problem is similar to the classic decoding problem in coding theory; many codes have been devised whose error-correcting properties are known but for which decoding algorithms have never been found.

In general the solution to the decoding problem is not unique. For example, an arbitrary directed graph having an arc labeling consisting of all 0s could have arisen from a node labeling consisting of all 0s (no faults at all) or all 1s (all nodes faulty). Given an arc labeling, therefore, we seek the particular corresponding node labeling having the least number of 1s (the fewest faults)--that is, the labeling of <u>minimum weight</u>. Actually, we are interested mainly in those sets of arc labelings that could have risen from a maximum of t node faults. (Other arc labelings resulting from more than t faults may or may not be decodable.) The decoding problem is therefore one of determining for a given graph G_T the minimum-weight node labeling corresponding to any arc labeling that can arise (according to Table 3.4) from a node labeling having no more than t 1s.

A solution to this problem will now be presented for A-->B testing.

Following Meyer and Masson [14], the full set of test results may be conveniently represented in the form of an n x n matrix E whose general element e_{ij} is 0 or 1, according to whether the test of node j by node i did not or did indicate an error. Nonexistent arcs in G are designated with an "unspecified" value, e_{ij}=U. Each row E_i of this matrix designates the collective result of all tests performed by node i, and each column vector E_j^t designates the collective result of all tests performed on node j.

Note first that for a single fault at node i, all n-1 row vectors representing nonfaulty tester nodes must <u>agree</u> with one another in their non-U components , even in column i where any non-U value must be 1. When the number of faulty nodes does not exceed t, E will have at least n-t such agreeing rows, except possibly in columns corresponding to faulty nodes. Rows representing faulty test nodes will contain arbitrary values (Xs) in their non-U components; they may or may not agree with the other rows. Columns correspondng to faulty nodes will contain 1s except in rows also corresponding to faulty nodes, where the entries are arbitrary.

In these terms, then, the decoding problem is one of finding a maximal set of rows E_{i1}, E_{i1}, ... E_{ip} of the error matrix E that agree in all of their non-U components, except possibly in Columns i1, i2, ... ip. Restated, the problem is one of deleting the least number of rows and their corresponding columns such that all remaining rows agree in their non-U components.

The set D_E of rows and columns to be deleted may be readily identified by calculating the matrix product

$$F = EE^t$$

using ordinary scalar addition and a special multiplication rule given by the table

*	0	U	1
0	0	0	1
U	0	0	0
1	1	0	0

The entry f_{ij} of the (symmetric) matrix F will be 0 if and only if rows i and j of E agree perfectly in their non-U components; f_{ij} will be $\leq t$ if the disagreement is limited to t components and f_{ij} will be $>t$ otherwise. The rows of F, all of whose components are $\leq t$, then define the corresponding rows of E over which agreement holds. Let C be the binary consensus obtained by summing vector over all of these agreeing rows (formed, for example, by boolean addition with Us replaced by Os). The positions of the 1s in C identify the faulty nodes precisely.[1]

For testing in other than A-->B mode, a systematic procedure is not available. This is an outstanding problem whose solution is required before diagnosis programs for a DDP network can be written. If a practical decoding algorithm cannot be found for the class of networks known to satisfy the above conditions for t-fault locatability, then additional nodes or interconnections must be added to the network to permit decoding, thereby affecting the network architecture. This risk of network redesign must be acknowledged, but it is surely very low.

Two of the extensions mentioned in the last two subsections also

[1]This last step of returning to E to form C is necessary because one or more rows of E representing faulty nodes may happen to agree with the rows representing nonfaulty nodes, except in columns representing faulty nodes.

need to be made: the accommodation of faults on links as well as nodes, and provision for multiple fault status levels within nodes.

3.5 Summary and Conclusions

We have described the history of a fault from occurrence, through detection and location, to handling by *reconfiguration.*

The most important observation is that the complexity required for automatic fault diagnosis depends critically on the maximum number t of simultaneous or near-simultaneous faults that must be accommodated. The probability that multiple faults will need to be dealt with at the same time can be reduced by (1) running high-coverage FD programs frequently, (2) reducing the running time of FL and FH programs, and (3) following good hardware and software design practice to minimize the probability of highly correlated faults between similar units. In a system such as the BMD DDP network, protection must also be provided against maliciously induced faults that could appear on several nodes and/or links simultaneously. Further analysis of these fault conditions will be necessary, but it seems likely that tolerance against simultaneous malicious faults will dominate the requirements for network redundancy and for diagnostic capability. In any case, the minimum acceptable value of t must be determined from system considerations.

It has also been pointed out that, of the three aspects of diagnosis for which hardware and programs must be provided (FD,FL, and FH), virtually all of the unsolved problems in diagnosis concern FL. Further, all three steps of FL -- testing, communication, and decoding -- contain research problems still unsolved for the types of DDP networks expected in the envisioned application.

Work on these research problems should be guided by the goals of (1) determining how the costs and capabilities of FL programs are related to (a) the value of the parameter t and the classes of faults that can occur, and (b) the main design parameters (number and types of SRUs, and number and configuration of inter-SRU communication links); and (2) finding good algorithms for allocating local tests, for the communication protocol, and for decoding test results prior to reconfiguration.

The major conclusions to be drawn from this study of the diagnostic aspects of fault tolerance in DDP systems are:

1. The maximum value t of faults to be accommodated simultaneously should be determined as soon as possible from overall system requirements.

2. Research is needed for all three steps of FL: testing, communication, and decoding. Testing results are needed first.

3. Diagnosis requirements are crucial to the overall BMD system design, and should be incorporated early in the design effort, not later as an add-on.

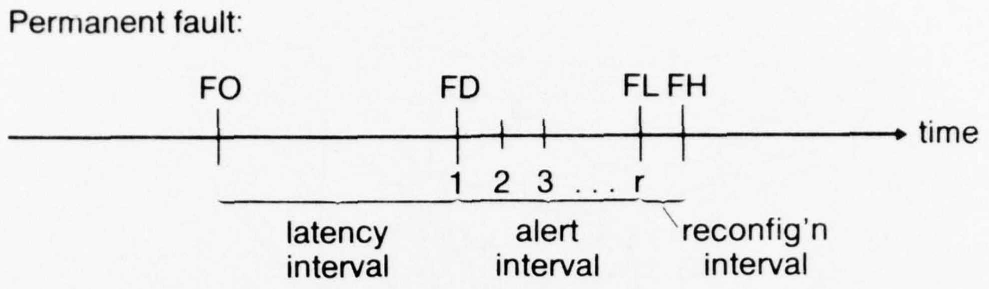


FIGURE 1 LIFE CYCLE OF A FAULT.

FO = Fault Occurrence; FD = Fault Detection; FL = Fault Location; FH = Fault Handled.

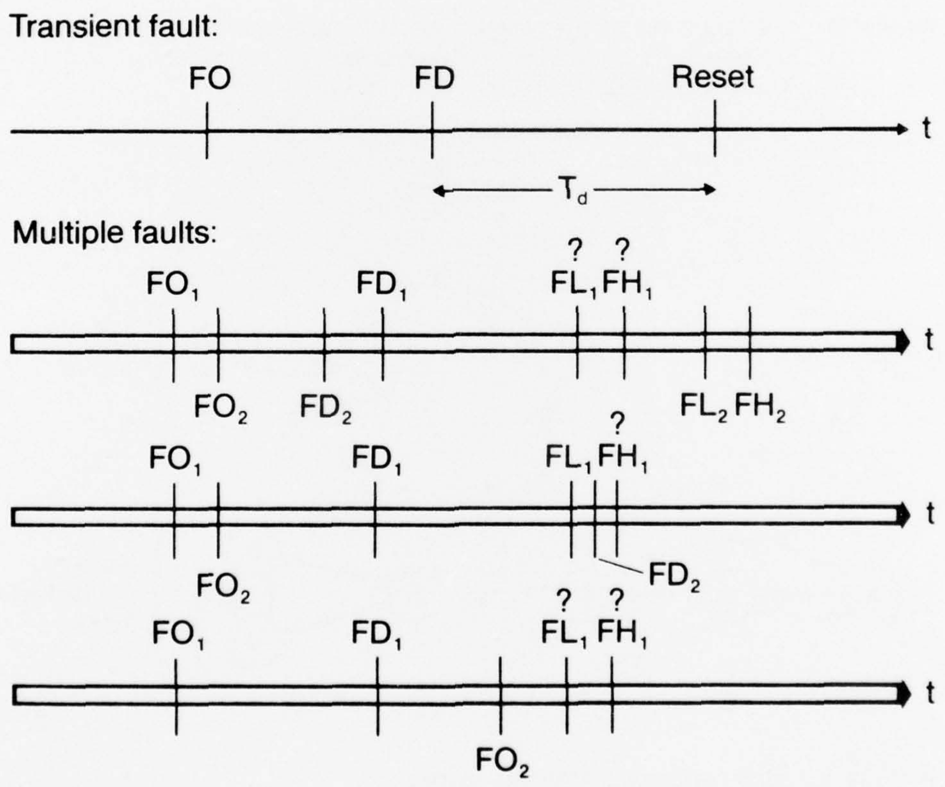


FIGURE 2 LIFE CYCLES OF SOME POSSIBLE MULTIPLE FAULTS

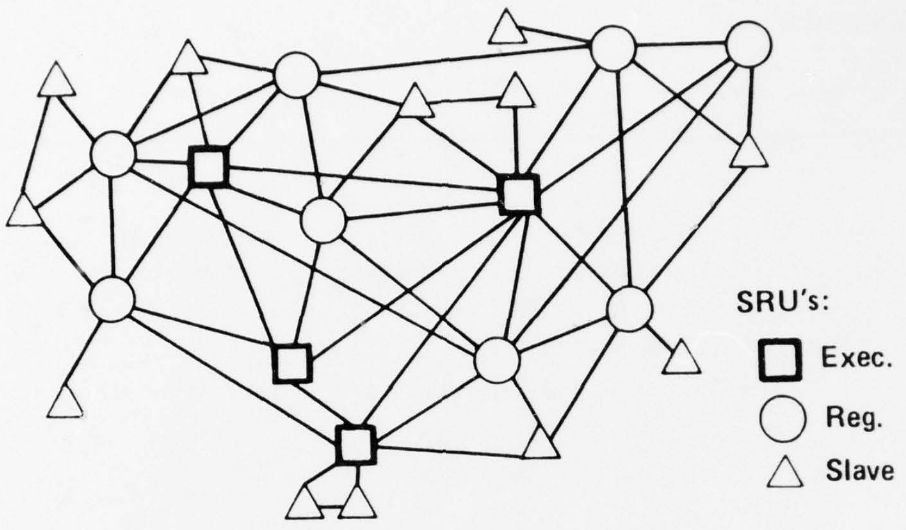


FIGURE 3 AN ILLUSTRATIVE EXAMPLE OF THE GRAPH OF A DDP NETWORK, SHOWING ITS SMALLEST RECONFIGURABLE UNITS (SRUs) AND THEIR INTERCONNECTIONS

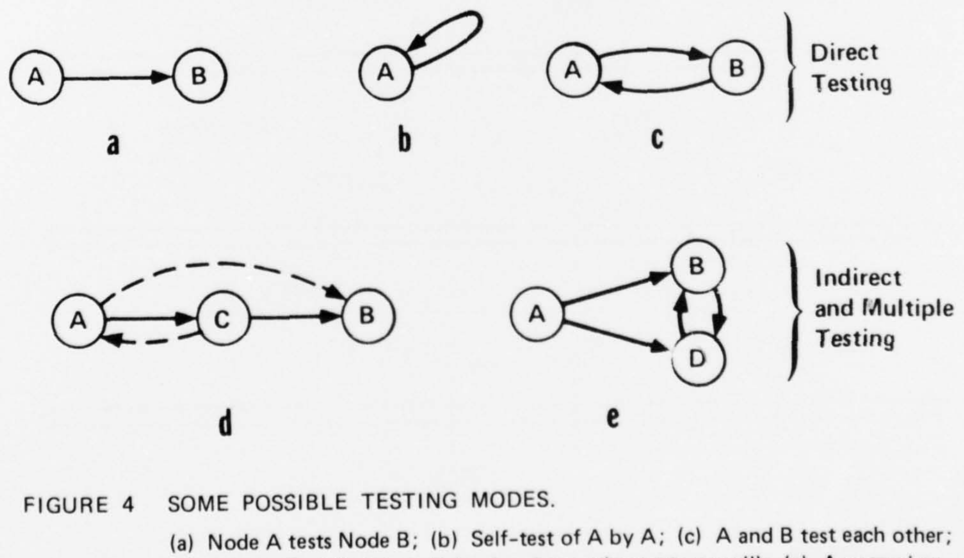


FIGURE 4 SOME POSSIBLE TESTING MODES.

(a) Node A tests Node B; (b) Self-test of A by A; (c) A and B test each other; (d) A tests C while C tests B (and perhaps other tests as well); (e) A supervises joint B–D test.

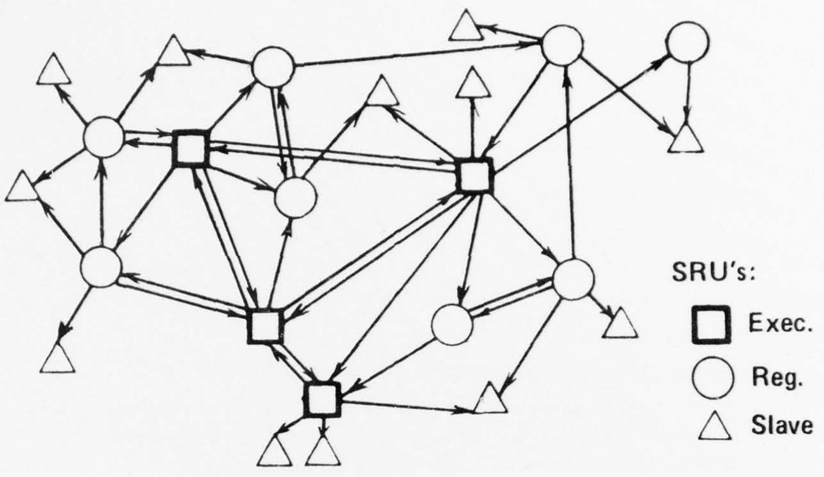


FIGURE 5 ONE POSSIBLE TESTING GRAPH CORRESPONDING TO THE DDP NETWORK GRAPH SHOWN IN FIGURE 3

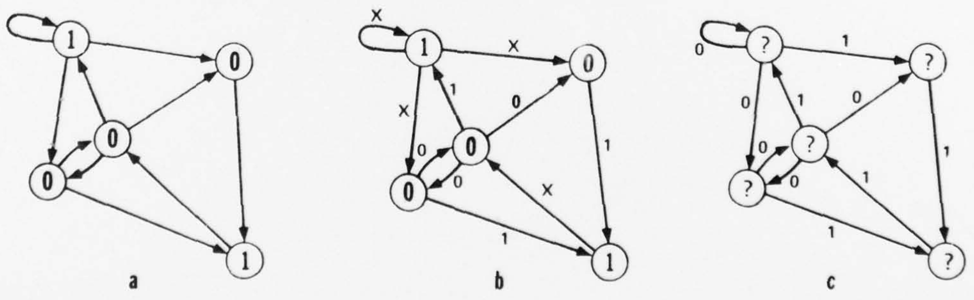


FIGURE 6 AN EXAMPLE OF A TESTING GRAPH ILLUSTRATING:

(a) Node labeling (SRU fault statuses); (b) Derivation of arc labeling (error reports) from node labeling; (c) The reverse problem of determining node labeling from arc labeling.

Table 1

ERROR TABLES CORRESPONDING TO FIGURE 4 (a) AND (b).

f_A, f_B = fault statuses of Nodes A and B (0 = nonfaulty, 1 = faulty); e = error report from test (0 = no error, 1 = error, x = test result ambiguous).

f_A	f_B	e
0	0	0
0	1	1
1	0	x
1	1	x

a

f_A	e
0	0
1	x

b

REFERENCES

[1] R.S. Boyer and J S. Moore.
A Computational Logic.
Academic Press, 1979.

[2] E. W. Dijkstra.
The Structure of the The Multiprogramming System.
Communications of the ACM 11(5):341-346, May 1968.

[3] R. J. Feiertag and P. G. Neumann.
The Foundations of a Provably Secure Operating System (PSOS).
NCC, 1977.

[4] R.J. Feiertag, et. al.
COL Final Report.
Technical Report, SRI International, Menlo Park CA, August 1978.

[5] R. Floyd.
Assigning Meanings to Programs, pages 19-32.
American Mathematical Society, 1967.

[6] S.L. Hakimi and A.T. Amin.
Characterization of Connection Assignment of Diagnosable Systems.
IEEE Transaction on Computers C-27:86-88, January 1974.

[7] L. Lamport. "The Specification and Proof of Correctness of Interactive Programs," S. Takasu (ed.), Proceedings of the International Conference on Mathematical Studies of Information Processing, University of Kyoto, 1978.

[8] L. Lamport.
On the Proof of Correctness of a Calendar Program.
Technical Report CSL-88, SRI International, Menlo Park, CA, January 1979.

[9] L. Lamport.
The Implementation of Reliable Distributed Multiprocess Systems.
Computer Networks 2:95-114, 1978.

[10] L. Lamport.
Proving the Correctness of Multiprocess Programs.
IEEE Trans. on Software Engineering SE-3(2):125-143, MAR 1977.

[11] L. Lamport.
A New Approach to Proving the Correctness of Multiprocess Programs.
ACM TOPLAS 1(1):, July 1979.

[12] L. Lamport.
The 'Hoare Logic' of Concurrent Programs.
Technical Report CSL-79, SRI International, Menlo Park, CA, November 1978.
Submitted to Acta.

[13] E.J. McCauley and P. Drongowski.
KSOS: Design of a Secure Operating System.
NCC 79, 1979.

[14] G.L. Meyer and G.M. Masson.
An Efficient Fault Diagnosis Algorithm for Symmetric Multiple Processor Architectures.
IEEE Transactions on Computers C-27(11):1059-1063, November 1978.

[15] P.G. Neumann, R.S. Boyer, R.J. Feiertag, K.N. Levitt, L. Robinson.
A Provably Secure Operating System.
Technical Report, SRI International, Menlo Park, California, February 1977.
Final Report, Project 4332.

[16] M. Pease, R. Shostak, L. Lamport.
Reaching Agreement in the Presence of Faults.
Technical Report CSL-87, SRI International, Menlo Park, CA, January 1979.
This work supported by NASA Langley Research and Ballistic Missile Defense Advanced Technology Center.

[17] F.P. Preparata, G. Metze, and R.T. Chien.
On the Connection Assignment Problem of Diagnosable Systems.
IEEE Transactions on Electronic Computers EC-16(6):848-854, December 1967.

[18] R.L. Rivest, A. Shamir, and L. Adleman.
A Method for Obtaining Digital Signatures and Public-key Cryptosystems.
Communications of the ACM 21:120-126, February 1978.

[19] L. Robinson, K.N. Levitt, P.G. Neumann, and A. R. Saxena. "A Formal Methodology for the Design of Operating System Software," R.T. Yeh (ed.), Current Trends in Programming Methodology, Prentice-Hall, 1977.

[20] J.D. Russell and C.R. Kime.
System Fault Diagnosis: Closure and Diagnosability with Repair.
IEEE Transactions on Computers C-24(11):1078-1089, November 1975.

[21] J.D. Russell and C.R. Kime.
System Fault Diagnosis: Masking, Exposure, and Diagnosability.
IEEE Transactions on Computers C-24(12):1155-1161, December 1975.

[22] J.H. Wensley, L. Lamport, J. Goldberg, M.W. Green, K.N. Levitt, P.M. Melliar-Smith, R.E. Shostak, and C.B. Weinstock.
SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control.
Proceedings IEEE 66(10):1240-1254, October 1978.