Reliability Issues in Computing System Design

B. RANDELL
P. A. LEE
P. C. TRELEAVEN
Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, NE1 7RU UK

This paper surveys the various problems involved in achieving very high reliability from complex computing systems, and discusses the relationship between system structuring techniques and techniques of fault tolerance. Topics covered include: 1) protective redundancy in hardware and software; 2) the use of atomic actions to structure the activity of a system to limit information flow; 3) error detection techniques; 4) strategies for locating and dealing with faults and for assessing the damage they have caused; and 5) forward and backward error recovery techniques, based on the concepts of recovery line, commitment, exception, and compensation. The ideas described relate to techniques used to date in systems intended for environments in which high reliability is demanded Three specific systems the JPL-STAR, the Bell Laboratories ESS No. 1A processor, and the PLURIBUS are described in some detail and compared.

Keywords and Phrases: error, failure, fault, fault tolerance, fault avoidance, hardware reliability, software reliability, system structure

CR Categories: 4.30 6.20

INTRODUCTION

This paper attempts to provide a general framework, based on concepts of system structuring, for the analysis and comparison of contrasting approaches to the goal of providing continuous and trustworthy service from a computing system. Discussion includes brief descriptions of some specific systems designed for highly demanding environments. It covers both software and hardware reliability problems, including those arising from design inadequacies, although one very important topic—software correctness—is largely ignored since it has been the subject of a number of survey and tutorial papers [ELSP72, LOND75, FOSD76, HANT76].

System reliability is sometimes interpreted rather broadly as a measure of how a system matches its users' expectations. See for example [NAUR77]. The trouble with this view is that the expectations themselves can be mistaken and can change almost arbitrarily, based perhaps on experience with the system. In this paper a somewhat narrower interpretation of system reliability is taken, more in line with typical formal, and often quantitative, assessments of hardware reliability. Thus sys-

CONTENTS

tem reliability is related to the success with which a system provides the service specified. By this means the concept of the *reliability* of a system is separated from that of the *reliance* placed on it.

It is of course to be hoped that the reliance placed on a system will be commensurate with its reliability. When this is not the case, one or other will have to be adjusted if the system is to be retained. For example, users of a time-sharing service that has a tendency to lose the contents of current work spaces probably would learn to take the precaution of frequently requesting the saving of copies of their work space, thereby satisfying themselves with the quality of the service that they are getting. Notions of reliance, therefore, can be as much bound up with psychological attitudes as with formal decisions regarding the requirement that a system is supposed to satisfy.

In fact the history of the development of computers has seen some fascinating interplay between reliance and reliability. The reliability of early computers caused relatively little reliance to be placed on the validity of their outputs, at least until appropriate checks had been performed. Even less reliance was placed on the continuity of their operation—lengthy and frequent periods of downtime were expected and tolerated. As reliability increased so did reliance, sometimes in fact outdistancing reliability so that additional efforts had to be made to reach previously unattained reliability levels. During this time computing systems were growing in size and functional capacity so that, although component reliability was being improved, the very complexity of systems was becoming a possible cause of unreliability, as well as a cause of misunderstandings between users and designers about system specifications.

The subject of system specifications and of how these can be arrived at; documented, validated, and updated, is a large and complex topic, well worthy of discussion in its own right. However, given the interpretation of system reliability that we have chosen, it is inappropriate to pursue the topic further in the present paper, which takes as its main starting point the informal but hopefully rigorous definitions of concepts relating to system reliability given in Section 1, Basic Concepts. These definitions presume the existence of some external specification of the requirements that the system is supposed to meet.

As discussed in Section 2, Reliability Issues, the increasing complexity of the functions that computing systems are asked to provide, and the increasing reliance that is wished to place on them—for example, in environments where unreliability can lead to huge financial penalties or even loss of life—have spurred the search for a greater understanding of reliability issues. Aspects of computing systems, such as their software, which were previously virtually ignored in many discussions of reliability problems, are now being addressed, causing earlier approaches and solutions to be reevaluated.

The basis for this reevaluation is provided by recent ideas on system structuring and its relationship to reliability and, in particular, to fault tolerance. The likely importance of system structuring was well expressed in the 1973 SRI survey of fault-tolerant computing [NEUM 73]. It stated that: "Our assessment of the structured design approach is that it has the potential for providing highly flexible and economical fault tolerance without greatly compromising system cost, system performance, and system efficiency. Some qualities of structure are found in the current art, but full realization of this potential requires further development.... A serious weakness in the current art is the absence of a design methodology that integrates hardware and software into a systems concept addressing reliability, availability, security, efficiency, and functional capability in a unified way. For example, significant benefits can be expected from techniques for structural design and implementation..."

This SRI survey provided a very useful account of the state of the art of hardware fault tolerance, an account which is still of value. Because of the existence of this survey and much other relevant literature, such as the proceedings of the IEEE Symposia on Fault-Tolerant Computing, the present paper does not attempt to describe the present vast array of techniques for hardware fault tolerance in great detail, but rather concentrates on the overall system aspects of reliability. System structuring therefore forms one of the major topics, and is the subject of Section 3. It is this section which provides a basis for describing, in Section 4, Fault Tolerance Techniques, different approaches to attaining reliable operation, such as masking redundancy, and forward and backward error recovery.

The present paper is a condensed version of a much lengthier survey originally prepared as lecture notes for the Advanced Course on Operating Systems, Munich 1977 [RAND78]. Several sections of this survey were based closely on various earlier papers [LOME77, MELL77, RAND75] also emanating from the research project on computing system reliability at the University of Newcastle upon Tyne. The survey includes eight appendices, each giving a detailed description of a particular highly reliable computing system, and an analysis of the reliability strategies it uses. In Section 5, Fault-Tolerant Computing Systems, just three of these systems, namely the JPL-STAR, the Bell Laboratories ESS No. 1A processor, and the PLURIBUS, are described and used to illustrate the general concepts discussed in earlier sections.

1. BASIC CONCEPTS

The terminology we use is intended to be suitable for both hardware and software systems, and to correspond broadly to conventional usage. However, the definitions of some terms differ from previous practice, which typically has paid little attention to design inadequacies as a potential source of unreliability.

Systems and Their Failures

We define a *system* as a set of components together with their interrelationships where the system has been designed to provide a specified service. The *components* of the system can themselves be systems, and we term their interrelationships the *algorithm* of the system. There is no requirement that a component provide service to a single system; it may be a component of several distinct systems. The algorithm of the system is, however, specific to each system individually.

Example: Figure 1 is a simple, schematic representation of a system consisting of a processor, a console, and an interconnecting cable. These three components are interrelated by being plugged together. Interconnecting lines represent these interrelationships, rather than any physical component.

The *reliability* of a system is taken to be a measure of the success with which the system conforms to some authoritative specification of its behavior. Without such a specification, nothing can be said about the reliability of the system. When the behavior of a system deviates from that which is specified for it, this is called a *failure*. A

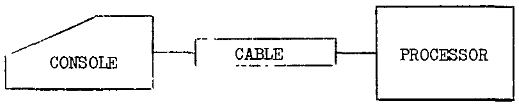

FIGURE 1. A three component system.

failure is thus an event, with the reliability of the system being inversely related to the frequency of such events. Various formal measures related to a system's *reliability* can be based on the actual (or predicted) incidence of failures, and their consequences (see, for example, [SHOO68]). These measures include Mean Time Between Failures (MTBF), Mean Time to Repair (MTTR), and *availability*, that is, the fraction of the time that a system meets its specification. Further measures can of course be defined which take into account classifications of the type and seriousness of the failure.

The *internal state* of a system is the aggregation of the external states of all its components. The *external state* of a system is an abstraction of its internal state. During a transition from one external state to another, the system may pass through a number of internal states for which the abstraction, and hence the external state, is not defined. The specification defines only the external states of the system, the operations that can be applied to the system, the results of these operations, and the transitions between external states caused by these operations, the internal states being inaccessible from outside the system.

The service provided by a system is regarded as being provided to one or more *environments*. Within a particular system, the environment of a given component consists of those other components with which it is directly interrelated.

Errors and Faults

In contrast to the simple, albeit very broad, definition of "failure" given above, the definitions we now present of "error" and "fault" are not as straightforward. This is because they aim to capture the element of subjective judgment which we believe is a necessary aspect of these concepts, particularly when they relate to problems which could have been caused by design inadequacies in the algorithm of a system.

We term an internal state of a system an *erroneous state* when that state is such that there exist circumstances (within the specification of the use of the system) in which further processing, by the normal algorithms of the system, will lead to a failure which we do not attribute to a subsequent fault. The subjective judgment that we wish to associate with the classification of a state as being erroneous derives from the use of the phrases "normal algorithms" and "which we do not attribute" in this definition. The first of these implies the possible
will typically be the error recovery algorithms.

The term "error" is used to designate that part of the state which is "incorrect." An error is thus an item of information, and the terms *error*, *error detection*, and *error recovery* are used as casual equivalents for *erroneous state*, *erroneous state detection*, and *erroneous state recovery*.

A *fault* is the mechanical or algorithmic cause of an error, while a *potential fault* is a mechanical or algorithmic construction within a system such that (under some circumstances within the specification of the use of the system) the construction will cause the system to assume an erroneous state. It is evident that the failure of a component of a system is (or rather, may be) a mechanical fault from the point of view of the system as a whole.

We hope it will now be clear that the generality of our definitions of failure and fault enables the notion of fault to encompass design inadequacies such as a mistaken choice of component, a misunderstood or inadequate specification (of either the component or the service required from the system), or an incorrect interrelationship among components (such as a wrong or missing interconnection in the case of hardware systems, or a program bug in software systems), as well as, say, hardware component failure due to aging.

It can be very difficult to attribute a given failure to a specific fault. A demonstration that further processing can lead to a failure of a system indicates the presence of an error, but it does not suffice to identify a specific item of information as the error. Consider a system affected by an algorithmic fault. The sequence of internal states adopted by this system will diverge from that of the "correct" system at some point; the algorithmic fault being the cause of this transition into an erroneous state. But there can be no unique correct algorithm. It may be that any one of several changes to the algorithm of the system could have precluded the failure. A subjective judgment as to which of these algorithms is intended determines the fault, the items of information in error, and the moment at which the state becomes erroneous. Some judgments may of course be more useful than others.

The significance of the distinction between faults and errors may be seen by considering the repair of a database system. Repair of a fault may consist of the replacement of a failing program (or hardware) component by a correctly functioning one. Repair of an error requires that the information in the database be changed from its currently erroneous state to one which will permit the correct operation of the system. In most systems, recovery from errors is required, but repair of the faults which cause these errors, although very desirable, is not necessarily essential for continued operation.

2. RELIABILITY ISSUES

Requirements

The reliability requirements of different environments can differ enormously. One extreme is the case of air- and space-borne computers where only momentary cessation of service can be tolerated, no maintenance or manual repair activity is feasible, and incorrect results are completely unacceptable. In most other cases, however, maintenance and manual repair are usually possible. In some cases the reliability goals can be met only by allowing for such repairs while the system is in service. Thus the repair activity is concerned with faults rather than with system failures.

In contrast, in many environments obtaining very high reliability from a computing system is not worth the expense because many other failure-prone devices, e.g., communications lines and mechanical peripherals, are being used, or because the cost of failure is comparatively low. Often certain types of failure are regarded as comparatively unimportant. For example, in computerized telephone systems, relatively infrequent, isolated small breakdowns can be tolerated, as long as the overall system remains operational.

Another type of reliability requirement is that typical of on-line database systems or, indeed, any systems intended to retain extensive amounts of valuable data over lengthy periods of time. In many such systems safeguarding the data held is more important than providing continuity of access to that data.

These examples highlight the fact that reliability is a commodity whose provision involves costs, either direct, or arising from performance degradation. In theory the design of any nontrivial computing system should involve careful calculations of tradeoffs between reliability, performance, and cost. In practice the data and relationships which would be needed for such calculations in complex systems, are quite often unknown, particularly with regard to unreliability caused by residual design faults.

Types of Fault

An enumerative list of fault categories is hardly likely to be exhaustive. Such a list could include hardware component failure, communication faults, timing problems, mistakes by users and operators, design inadequacies, and the like. Faults due to hardware component failures, are often classified by duration, extent, and value. *Duration* refers to whether the fault is permanent or transient (after agreeing on some time scale); *extent* applies to whether the effect of the fault is localized or distributed; and *value* indicates whether the fault creates fixed or varying erroneous logical values. Extension of this classification to software faults is possible, but not particularly helpful. What is significant about software faults is, of course, that they must be algorithmic faults stemming from unmastered complexity in the system design. Because hardware systems have, in the past, been much simpler than those constructed using software, algorithmic faults in hardware are less common, although not unknown.

A further category of faults is that caused by erroneous interactions with the system, for instance by a user providing invalid input data. While the occurrence of such faults can be reduced by appropriate checking, some interactions may be valid with respect to the system specification although discovered later to be incorrect. Such faults can be subtle, serious, and difficult to deal with.

Fault Avoidance and Fault Tolerance

The traditional approach to achieving reliable computing systems has been based largely on *fault avoidance* (termed *fault intolerance* by Avizienis). Quoting Avizienis [AVIZ76]:

> The procedures which have led to the attainment of reliable systems using this approach are: acquisition of the most reliable components within the given cost and performance constraints; use of thoroughly-refined techniques for the interconnection of components and assembly of subsystems; packaging of the hardware to screen out expected forms of interference; and carrying out of comprehensive testing to eliminate hardware and software design faults. Once the design has been completed, a quantitative prediction of system reliability is made using known or predicted failure rates for the components and interconnections. In a "purely" fault intolerant (i.e., nonredundant) design, the probability of fault-free hardware operation is equated to the probability of correct program execution. Such a design is characterized by the decision to invest all the reliability resources into high-reliability components and refinement of assembly, packaging, and testing techniques. Occasional system failures are accepted as a necessary evil, and manual maintenance is provided for their correction.

There are several situations in which the fault avoidance approach clearly does not suffice. These include situations where the frequency and duration of repair time are unacceptable, or where the system may be inaccessible to manual maintenance and repair activities. An alternative approach to fault avoidance is that of fault-tolerance. This approach, which at present is largely confined to hardware systems, involves the use of protective redundancy. A system can be designed to be *fault-tolerant* by incorporating additional components and abnormal algorithms which attempt to ensure that occurrences of erroneous states do not result in later system failures. The degree of fault tolerance (or "coverage") will depend on the success with which erroneous states corresponding to faults are identified and detected, and the success with which such states are repaired or replaced.

There are many different degrees of fault tolerance which can be attempted. For example, a system designer might wish to reduce the incidence of failures during periods of scheduled operation by designing the system so that it will remain operational even in the presence of, say, a single fault. Alternatively, he might wish to increase the average length of periods of uninterrupted operation by designing the system so that it can tolerate not only the presence of a fault, but also the activity involved in repairing the fault.

Fault-tolerant systems differ with respect to their behavior in the presence of a fault. In some cases the aim is to continue to provide the full performance and functional capabilities of the system. In other cases only degraded performance or reduced functional capabilities are provided until the fault is removed. Such systems are sometimes described as having a "fail-soft" capability.

Example: It is now typical for the computer terminals used in banks to incorporate significant processing and storage facilities. Such terminals enable data input and possibly some limited forms of data validation to continue even when the main computer system is not operational.

Schemes for fault tolerance also differ with regard to the types of fault which are to be tolerated. In particular, many systems which are designed to tolerate faults due to hardware component aging, electrical interference, and the like, make no specific attempt to cope with algorithmic faults in either the hardware or the software design. This in fact illustrates that fault tolerance and fault avoidance are better regarded as complementary rather than competitive approaches to system reliability—indeed the two different approaches are often used within the same system in an attempt to deal with different types of fault.

Design Fault Tolerance

It is only recently that efforts have been undertaken to extend the fault-tolerant approach to cover design faults. The *a priori* elimination of design faults, assumed in the fault avoidance approach, is the normal (and praiseworthy) aim, so that many writers have equated the notion of reliability with that of correctness, particularly in the case of software. Virtually all research relating to the practice of software development can, therefore, be claimed to be directly relevant to software reliability; examples include the design of high-level languages, formal verification techniques, program design methodologies and tools, debugging aids, etc. However, important as all of these topics are, they can not guarantee that a complex software design is ever entirely fault free or that modifications to the design might not introduce new faults. When this is admitted the only alternative to simply accepting the resulting (probably unquantifiable) reliability is to seek to improve matters by the use of design fault tolerance.

Most existing approaches to the design of fault-tolerant systems make three assumptions: first, as mentioned earlier, that the algorithms of the system have been correctly designed; second, that all of the possible failure modes of the components are known; and third, that all of the possible interactions between the system and its environment have been foreseen. However, in the face of increasing complexity of systems, the validity of these assumptions must be questioned. At some stage it must surely become impractical to rely on enumerating all of the possible types of fault which might affect a system, let alone design algorithms to detect or accommodate each possible type of fault individually.

Thus the problem in using design fault tolerance is essentially that of how to tolerate faults which are not or cannot be anticipated as opposed to those previously enumerated and categorized. We would therefore argue that design fault tolerance requires a considerable rethinking of the techniques which have in the past proved suitable for tolerating various kinds of hardware component faults.

3. SYSTEM STRUCTURE

Considerations of the reliability problems of complex computing systems, and of means for coping with them, are closely interwoven with various notions that can be collectively termed "system structuring."

Static Structure

The definition of system given in Section 1 indicates that each system has what might be termed a static structure, which indicates what components it is regarded as comprising and how these components are interrelated.

One can of course visualize a given system in terms of many different structures, each implying a different identification of the components of the system.

Example: A programmer visualizes a CDC 6600 as having a single sequential main processor and a set of independent peripheral processing units, but the maintenance engineer sees it as consisting of a set of parallel function units, and a single time-shared peripheral processor.

Some static structures will have a more visible reality in the actual system than others—in the case of hardware systems, for example, by corresponding to the interrelated physical components from which the system is constructed. The important characteristic of such "actual" (as opposed to "conceptual") structuring is that the interrelationships between its components are constrained, while the system is operational, ideally to just those that the designer intended to occur. The stronger the constraint, the more the structuring is actual, and the more reasonable it is to base provisions for fault tolerance on that structuring. (The "strength" of a constraint is in fact a measure of the variety of different faults that it prevents from affecting the planned interrelationship.)

Example: When the various registers and functional units of a central processor are implemented using separate hardware components, the designer can have reasonable confidence that (presuming the absence of electrical or mechanical interference) only those registers and functional units that are meant to communicate with each other do so (along the interconnecting wires that the designer has provided).

The software of a computing system serves to structure that system by expressing how some of the storage locations are to be set up with information which represents programs. These then control some of the interrelationships among hardware components, for example, ensuring that the potential communication path between two I/O devices via working store is actually usable.

However, the software can itself be viewed as a system and its structure discussed in terms of the programming language that was used to construct it. Thus in a block-structured language each block can be regarded as a component, which is itself composed of, and expresses the interrelationships among, smaller components such as declarations and statements (including blocks).

The operational software will have "actual" structure matching that of its source language version only to the extent that it consists of components with constrained methods of interacting.

Example: The scope rules in a block-structured language are often enforced by a compiler, which then emits unstructured code. However the compiler could emit code in which the variables of each different block are kept, say, in different segments, and some form of protection mechanism used to impede access to those variables which are not supposed to be currently accessible.

Dynamic Structure

Just as the system itself can be regarded as having a static structure, so can its activity be regarded as having a *dynamic structure*. In fact each static structuring of a system implies a dynamic structuring of its activity. The static structure is important for understanding the kinds of faults that might exist in the system and the provisions that have been made for trying to cope with them; the dynamic structure is of equal importance for understanding the effects of these faults and how (or whether) the system tolerates them in order to continue functioning.

The activity of a given system can be visualized in terms of many different structures, depending on which aspects of this activity one wishes to highlight or to ignore. One basic and now well-established concept used for describing some important aspects of the dynamic structure of a system's activity is the "process." Depending on the viewpoint chosen, quite different processes with their interrelationships might be identified as constituting the structure of the system's activity. Again, a dynamic structure will be "actual" rather than merely "conceptual" to the extent to which the interrelationships are constrained to be as the designer intended.

Example: Reasonably "actual" dynamic structure exists in the situation where processes correspond to the application of programs to sets of data, if the programs and data are suitably protected and processes are impeded from interacting other than, say, via an explicit message passing system.

The sequencing, or control flow, aspects of process structuring, namely the creation, existence, and deletion of processes, can be shown graphically in some form such as Figure 2. However, matters of information

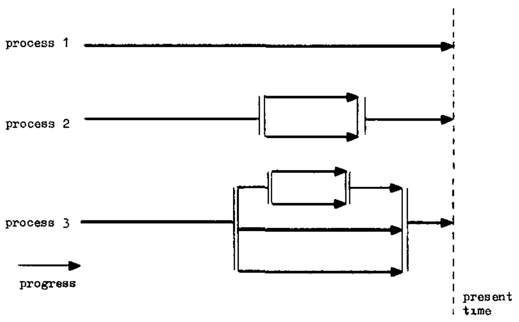

FIGURE 2. Control flow aspect of dynamic structure.

flow (intended or unintended) between processes are at least as important as control flow when it comes to considerations of reliability, particularly the problem of determining the possible damage that a fault has caused. Therefore, we need a concept such as *atomic action* [LOME77] as a part of our means of expressing the dynamic structure of the activity of a system.

Atomic Actions

The activity of the system is made up of primitive or atomic (i.e., apparently instantaneous) operations that are carried out by the components of the system. *Atomic actions* provide a means of generalizing such atomic operations. They are, in fact, a means by which a system designer can specify what process interactions are, if possible, to be prevented in order to maintain system integrity, without having to indicate how this is done. They do this by enabling the designer to indicate the sections of the activity, i.e., the sequences of atomic operations, of a process, or a group of processes that are themselves to be executed "atomically." Such "atomic" execution has the property that there is no information flow in either direction between the process (or group of processes) and the rest of the system.

Example: Consider a message-passing system that maintains a pool of buffers for holding messages and uses the variable "*i*" as a buffer frame pointer. The action of inserting an item into the buffer might involve the sequence of operations "*i*:=*i* + 1" and "buffer(*i*):= item". It is essential that this sequence of operations is executed as a single atomic action if the buffering scheme is to work properly.

An atomic action could involve several processes, since 1) a process executing a simple atomic action could temporarily create one or more further processes to assist it; and 2) two or more processes could cooperate directly in a shared atomic action, so that their activity is atomic with respect to the remainder of the processes in the system.

These various possibilities are shown in Figure 3, which is based on Figure 2, but where the ovals indicate atomic actions, and incomplete ovals represent atomic actions that are still in progress. The lines indicate processes, and should themselves be regarded as consisting of miniscule ovals, placed end to end, corresponding to the primitive operations out of which the processes are constructed.

The figure illustrates that "atomicity" has to be regarded as relative rather than absolute, and that atomic actions can themselves involve atomic actions, as well as the basic atomic operations. It also, by implication, illustrates that atomic actions, by their very nature, cannot overlap. Methods of implementing atomic actions lie outside the scope of this paper. However, it is worth mentioning that such methods include the use of separate processors, the disabling of

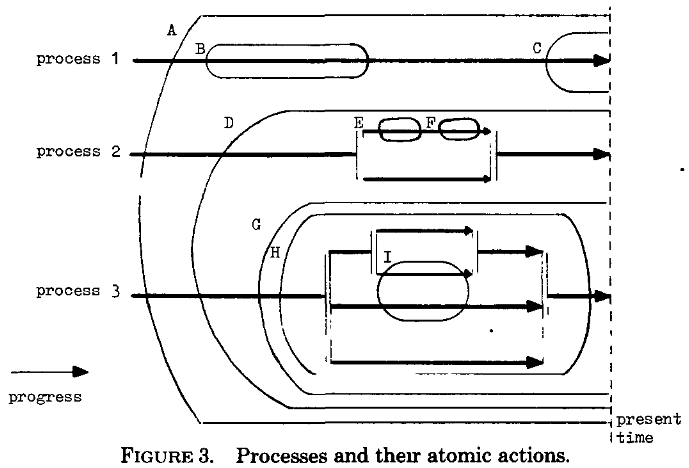

FIGURE 3. Processes and their atomic actions.

interrupts on an individual (multiprogrammed) processor, and synchronization schemes such as monitors and resource locking strategies.

We have so far described atomic actions merely as a means for a designer to indicate what system integrity constraints should be met. However, it should be clear that they are of direct relevance to techniques for achieving fault tolerance. This is due to the fact that they provide a means of error detection, and more importantly, a means of delimiting the possible consequences of a fault.

An atomic action is in fact a generalization of the concept of a "transaction" introduced by database designers [ESWA76, GRAY75, LOME77]. The transaction scheme allows a database system user to arrange that a sequence of interactions with the database will be treated as atomic, in order that desired consistency constraints on the database can be realized. A transaction can thus be viewed as an atomic action involving the user and the database system (or, more exactly, the process which is the user's activity, and one of the terminal support processes of the system). For reasons of performance, it is usually necessary to execute many such transactions concurrently, so atomicity can be provided by file- or record-locking strategies (for example, see [ESWA76, GRAY75]).

The concept of an atomic action is itself generalized by being made independent of the notion of a sequential process, and is defined using occurrence graphs, in [MERL77]. However in this paper we will content ourselves with using the present informally described and process-based concept.

Levels of Abstraction

In choosing to regard a system (or its activity) as composed of certain components and to concentrate on their interrelationships while ignoring their inner details, one is deliberately considering just a particular abstraction of the total system. Thus the sorts of structuring that we have discussed so far can be described as structuring within a single level of abstraction, or *horizontal structuring.* If we consider further details of a system (or part of a system), our view focuses on a lower level of abstraction which shows how components and their interrelationships are implemented and act, in terms of some more detailed components and interrelationships. These will of course in turn just be abstractions of yet more detailed components and interrelationships, and so on.

In choosing to identify a set of levels of abstraction (each of which might relate to the whole system, or just some part of the system) and to define their interrelationships, one is once again imposing a structure on a system, but this is a rather different form of structure which we will refer to as *vertical structuring.* Thus vertical structurings describe how components are constructed, whereas horizontal structurings describe how components interact.

The importance of levels of abstraction is that they allow one to cope with the combinatorial complexity that would otherwise be involved in a system constructed from a very large number of very basic components. The price that is paid is the requirement for well-documented specifications of the external characteristics of each level. Such specifications can be thought of as the *abstraction interfaces* interposed vertically between levels, much as the interrelationships defined between interacting components within a level function as what could be termed *communication interfaces.* In each case the interface will, if well chosen, allow the designer to ignore (at least to some extent) the workings of those parts of the system which lie on the far side of the interface.

Example: The system shown in Figure 4 has a vertical structure comprising four levels, all but the topmost of which are implemented by an interpreter. Each interpreter is programmed using the set of apparently atomic facilities (objects, operations, etc.) that is provided at one abstraction interface, and has the task of providing the more abstract set of (again apparently atomic) facilities that the next higher abstraction interface defines. Because each of these abstraction interfaces is fully specified and documented, the designer of the implementation of any one level will normally need little or no knowledge of the design, or perhaps even the existence, of any other levels.

As with horizontal structuring, so can many different vertical structurings be used

```
4   APL statements

    ---------------------(APL Machine)

3   Instructions

    ---------------------(S/370 Machine)

2   Micro-instructions

    ---------------------(IBM micro-instruction Machine)

1   Hardware logic,
    storage etc.
```

FIGURE 4. A fully interpretive multilevel system

to visualize a given system. Equally, some vertical structurings will have a more visible reality in the actual system than others. Once again, the important characteristic of such "actual" structuring is that, while the system is operational, the rules of the abstraction interfaces are, to some degree, constrained or "enforced." The greater the extent of this enforcement, the more the vertical structuring is actual. The role of the enforcement will be to try to prevent faults (or more likely, just certain of the more likely types of fault) from invalidating the abstraction that a level is designed to provide.

Example: In Dijkstra's THE system [DIJK68] the levels are almost entirely conceptual. They were used as a means of factoring the design effort, and of facilitating the validation of the system design. However no attempt was made to incorporate mechanisms in the system which would perform run-time checks on the rules relating to the usage of the facilities provided at various levels. Thus, for example, if a memory parity error was detected there was no way of relating this to a particular level of abstraction, much less of directly incorporating appropriate provisions in each level for coping with such faults.

However, the more the vertical structuring is actual, the more reasonable it is to base provisions for fault tolerance on it.

Example: Consider a multilevel interpreter similar to that of Figure 4, but where the microprogram and the program are each held in a (separate) part of the same store. Naturally, we assume that the microprogram has been designed to constrain the program from overwriting the part of the store holding the microprogram. Then the microprogrammed and programmed interpreters might well each have their own distinct means of recovering from a reported store parity error.

Faults and Structuring

We have already discussed some examples of how the provision of "actual" structure in a system makes feasible the provision of certain types of fault tolerance. Subsequent sections develop these points further. However the relationship between faults and structuring is really very basic, as well as subtle. In fact our whole categorization of faults into mechanical faults, algorithmic faults and faults due to invalid interactions with (i.e., misuse of) a system is based on system structure. Only after having chosen a particular perspective on a system, and having identified a vertical and horizontal structuring, can one say to which of these three categories a given fault belongs.

Worse than this, the process of identifying a structuring seems to involve a conscious (or more likely unconscious) assumption about the sorts of fault that could occur, and should be considered. Putting this thought another way, just as it seems impossible to consider any object as being completely unstructured, it seems that we can never, at any one time at least, avoid limiting the fault possibilities that can be conceived. Structurings that are useful from the point of view of considering the reliability of a system (i.e., structurings which are "actual" structurings) are those which enable designers to think, and to think simply, about those faults which are likely to occur. This is not just a question of considering the relative likelihood of problems with particular pieces of hardware, or from particular types of interface, but in complex systems is also a question of the likelihood of mistakes being made by the designers themselves. Good structuring should reduce the number of such mistakes but gives no guarantee of their absence.

4. FAULT TOLERANCE TECHNIQUES

Techniques for attempting to achieve fault tolerance can be classified in various different ways. Here we choose to regard them as comprising, in general, strategies for 1) error detection, 2) fault treatment, 3) damage assessment, and 4) error recovery.

The particular strategies used may vary in different parts of a single system, and at different times during its operation. Indeed it is not always possible to make a positive identification of the components responsible for each of the constituent strategies used in a given fault tolerance technique. The order in which these strategies are carried out can vary, and there can be much interaction between them, but the starting point is always the detection of an error. The additional components and algorithms that provide these various strategies in a fault-tolerant system constitute what can be termed "protective redundancy."

Protective Redundancy

One common way of classifying protective redundancy is to differentiate *masking redundancy* from *dynamic redundancy*. This distinction is in fact related not as much to the type of strategy used, as to the way in which it is fitted into the structure of the system.

Masking redundancy is redundancy used to mask or hide the effects of faults in a component. Thus, as far as the environment of the component is concerned, the component works perfectly, despite internal faults, at least while the masking redundancy is effective. This contrasts with the situation where redundancy is used inside a component to provide explicit or implicit indications among the outputs of a component as to whether these are erroneous. This internal redundancy would then have to be complemented by external redundancy, in the form of provisions for recovery, in the system that uses the component. Redundancy schemes can also be related to the structure of a system in such a way that they possess attributes of both classes. Such *hybrid redundancy* schemes will mask the effects of some types of fault, and provide error indications for some other types.

Masking redundancy is often equated to a form of redundancy which is termed *static redundancy* because the redundant components that it involves remain in use, and in the same fixed relationship, whether or not any errors are detected. Examples of masking or static redundancy include fault masking via coding, and fixed replication with voting. The canonical example of the latter is Triple Modular Redundancy (see Types of Check, below).

Error Detection

The purpose of error detection is to enable system failures to be prevented by recognizing when they may be about to occur. Ideally, the checks made by error detection mechanisms should both be based only on the specification of the service that the system is supposed to provide, and be independent of the system itself. Without such independence there is the possibility of a single fault affecting both the system and the check, and so preventing error detection.

Example: The PRIME system [BASK72] attempts to detect and limit the propagation of errors by requiring that an independent check be made on every critical function performed by every processor [FABR73]. The independent check is performed either by another processor making a consistency check or by hardware making a validity check. Moreover, the system software has been designed with the aim of having sufficient redundancy to ensure that every critical decision can be checked independently. This approach is in fact an example of the use of decision verfication [DENN76]

If such checks could be designed to cover all of the aspects of the system specifications, and complemented by appropriate means of error recovery, then no single algorithmic or component fault would lead to system failure. In practice, of course, one has to make do with much less rigorous checking than this. For a start, the system specification may be expressed in terms of information which is external to the system, and in a way which is not amenable to a computational verification. Any checks would therefore have to ignore some aspects of the system specification, because of the necessity of expressing them in algorithmic form which involved only information that was available within the system. Furthermore, the independence of the check and the system being checked cannot be absolute; assessments of the degree of independence will be based on implicit or explicit assumptions about what faults might occur and what faults will not (for example it might be assumed that the software is correct). Even then considerations of cost and effects on performance might further decrease the quality of the check.

Example: Because of the requirement that checks have to be based on information available in the system, a stock control program cannot guard against being fed valid but incorrect data. It therefore cannot ensure that the actual stock supply situation matches that represented by the information it is holding.

For all these reasons, therefore, the sorts of checks which can be made in a system will be ones which attempt to enforce some lower standard of behavior than absolute correctness, a standard usually referred to as "acceptability." All that can be aimed for is the choice of tests for acceptability which provide a very high degree of confidence that any error will be detected.

Types of Check

Ideally the checks on the function of a system will be made on its "results" immediately before they leave the system; this is the principle of minimizing "Time of Check to Time of Use" that is described by McPhee [MCPH74].

Example: Acceptance tests [HORN74] are a programming language construct which is in accord with this principle. Such tests are performed on the values of selected variables that exist when the end of a program block is reached. The acceptance test takes the form of a Boolean expression based, in general, both on the current values of these variables and on the values that existed when the block was entered. Means of checking that changes were made only to variables listed in the acceptance test, i.e., checking that the block did not have any unanticipated effects, are described in [RAND75].

It is often possible, through knowledge of the algorithms of the system, to recognize that certain values of internal data items are erroneous; to do so will save useless processing and enable more speedy recovery. However, such checks are an inadequate substitute for checks which are made at the last possible moment and based on the external specifications of the system. Rather, such internal checks, which necessarily depend on the internal algorithms of the system, may lack independence from those algorithms.

In many cases, the task of establishing the acceptability of the results of a system involves some form of replication, followed by the checking of the consistency of the results obtained. The replications might, for example, involve two or more systems of independent design, two or more copies of the same design or repeated use of the same system, depending on the sorts of faults that are expected, and on the cost/performance constraints on the system. Such techniques can also be the means by which fault masking is provided, so that separate error recovery is not needed.

Example: Triple Modular Redundancy (TMR) involves the use of three subcomponents (or "modules") of identical design, and majority voting circuits which check the module outputs for exact equality. It is thus designed to mask the failure of any single module, by accepting any output that at least two of the modules agree on. (The TMR structure shown in Figure 5 is assumed to be part of a larger TMR structure, and so has triplicated inputs and outputs.)

Such a TMR organization was adopted in the SIFT (Software Implemented Fault Tolerance) computer [WENS72] where each task is executed by three (or more) processors and the results voted on by three subsequent processors.

TMR schemes are based on the assumption that failures in the different modules are independent of each other. Thus a (hardware or software) design fault which caused all three modules to produce identical but incorrect outputs would not be masked. Equally, it is necessary that the modules do not interact with each other, i.e., that the activity of each is an atomic action. This requirement is difficult to guarantee for modules within a single integrated circuit package, where TMR use is therefore inappropriate.

A somewhat different kind of check, a reversal check, is sometimes possible. This involves processing the results of the system activity in order to determine the corresponding inputs and check them against the actual inputs.

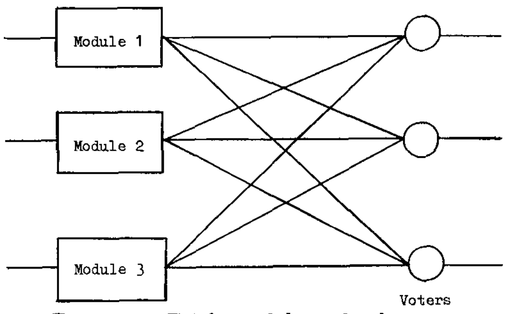

FIGURE 5. Triple modular redundancy.

Example: A reversal check on a system which produces a set of factors of an integer is very convenient, but this is not the case if the system, say, produces a single bit result, indicating whether or not a given integer is a prime number.

Many very effective error detection techniques involve the use of coding as a means of reducing the cost of the check. Techniques such as parity checks, Hamming codes, and cyclic redundancy checks are well established, particularly for detecting certain types of operational error. Checking the acceptability of large and complex masses of data is often infeasible without the use of coding techniques.

Example: Rather than undertake the task of checking that one large set of data is a permutation of another set, it might suffice to confirm that their checksums are correct, and identical.

Any form of error detection based on coding is, however, at best a limited form of check, whose acceptability must be based on assumptions about the probable characteristics of the errors to be detected. Thus, for instance, parity checks are regarded as suitable for core stores, but as quite inadequate for telecommunications, because of the different types of fault incurred and hence the different characteristics of the errors.

Error detection mechanisms can themselves suffer from faults. Provided that these faults do not cause damage to the system being checked, just two cases need be considered: the mechanism can detect nonexistent errors, or fail to detect actual errors. The detection of an error which does not exist cannot of itself cause the system to fail, though it will use up some of the recovery capability of the system. (If this capability is insufficient a failure will of course ensue.) One can expect that such faults in error detection mechanisms, since they draw attention to themselves, will be quickly identified and remedied. Unfortunately the reverse is true for faults which cause errors to remain undetected.

Interface Checking

Ideal or "complete" checks on the functioning of a system, as described above, would take no account of the design of the system, or of its intended internal structuring. Internal checks are typically based on a combination of suspicion and faith in the structuring of a system. Faith in "actual" structuring is based on the presumed effectiveness of the constraints that are applied to interfaces; faith in "conceptual" structuring on the quality of the thought that went into the design.

Error detection mechanisms within components that serve to check interactions across interfaces (either abstraction or communication) are one means of providing constraints. Checks for division by zero, protection violation, lack of response to a message within a time limit, and power supply irregularities are typical examples of such checks. All these checks of course can be viewed from inside a component as checks on the attempted use made of the component (i.e., system) by its environment. Clearly therefore they can at most guarantee validity of use, as opposed to correctness with respect to environment-level criteria.

The notion that a particular structuring obscures the possibility of certain types of fault has as its counterpart the fact that the choice of an interface makes some sorts of checking of the parts of the system on the far side of the interface impossible or at least impracticable.

Example: It is quite practicable to program a computer to perform a given task in such a way as to incorporate some checking on the correct functioning of the computer (for example of its arithmetic unit or backing store). However it would hardly be feasible to incorporate checks on the instruction fetching and decoding, or the store addressing into such a program.

This point leads us to the notion of diagnostic checking—that of periodic attempts to determine whether a component (e.g., the instruction decoder) is presently functioning satisfactorily, interspersed with periods when it is assumed that it is. To be effective the diagnosis must be capable of revealing the existence of likely faults, and the demands made on the component by the diagnostic scheme should approximate to or preferably exceed the demands made during normal use. Such schemes are applicable only in the case of faults that arise through uncontrolled changes to the system (such as component degradation caused by aging or perhaps inadequately planned modifications), though the diagnosis technique may be invoked in response to a failure, in an attempt to locate its underlying cause. The adequacy of diagnostic checking schemes will also depend on the amount of time and resources involved during diagnosis periods, and hence the frequency with which they can be undertaken, as compared to the arrival frequency of faults. The trouble with such schemes is that errors might go undetected for a long period, while damage spreads throughout the system, and beyond.

Fault Treatment

A detected error is only a symptom of the fault that caused it, and does not necessarily identify that fault. Even where the relationship between the fault and the detected error appears obvious, it will be found that many other possible faults could have caused the same error to be detected. It is often contended that the errors caused by different kinds of faults have fundamentally different natures, that for instance hardware component faults and software design faults are essentially different. We would argue that the differences are superficial. It may be that a pattern-sensitive type of fault, which generates an error only for specific patterns of inputs, is more common in software, whilst an abrupt change in behavior due to a change in internal state is more frequent in hardware. But both types of fault can be present in both hardware and software components (for example a program can damage its database and be unable to give any further service, and a hardware circuit can be affected by crosstalk).

The task of locating and removing a fault can therefore be a very complex one, whose automation is feasible only in situations which are (or are presumed to be) very simple. An alternative strategy is to ignore the fault and to try to continue to provide a service despite its continued presence, after having dealt in some way with the damage it might have caused. This also involves assumptions, this time about the maximum possible extent of such damage. (In practice it would be unwise not to log the occurrence of an error, and perhaps some information which might aid the later identification of the fault by maintenance personnel. Thus fault repair is not so much avoided, as delayed and left to manual techniques—such techniques can of course be aided considerably by the provision of appropriate tools.)

Continued usage of a component in which there is evidence of the presence of a fault makes sense only when the fault is believed to be one that is effectively a transient fault. For example, it is appropriate for faults which arise from sensitivity to a very limited set of specific input patterns, or from occasional time-varying behavior of subcomponents.

Example: Continued usage could be made of a magnetic tape unit, even though it is occasionally necessary to make repeated attempts at reading a tape before one is successful. If the operating system keeps a record of unsuccessful attempts at reading, this can provide evidence for use during the next period of scheduled maintenance, unless the tape unit deteriorates so rapidly that more urgent remedial action is called for.

If, or more realistically, when, it is decided that some action must be taken to avoid the fault during further operation of the system, or indeed to repair the fault, it must first be located. More exactly, the (assumed) position of the fault must be located to within a component of a size which is acceptable as a unit of replacement (for example a printed circuit board or a sequence of instructions). The search strategy will perforce be influenced by some preconceived ideas about the structure of a system. These will, at least initially, mislead the searcher (whether human or electronic) if a point has been reached where the fault has caused violation of some intended interrelationship between components. This fault will have occurred, without being detected at the time, because of the absence or inadequacy of constraints on the interrelationship. Where such possibilities have not previously been anticipated, and reflected in at least some existing (preferably documented) viewpoints on the system and its structure, the task of locating the fault can be very difficult indeed, even for a human, since it will in essence involve the generation of a new viewpoint.

Example: This is what has to be done all the time while debugging a program. It can become a much more difficult task when it concerns residual bugs in a system which has successfully operated for some time, and whose programmers have gained increased, but misplaced, confidence in the adequacy of their understanding of the system and its structure.

Given that a component has been designated as faulty, and as one whose further use should be avoided, various strategies are possible. Borgerson [BORG73] divides these into replacement and reconfiguration strategies. *Replacement strategies* are those in which a previously idle component (a so-called "stand-by spare") is directly substituted for the designated component. *Reconfiguration strategies* involve arranging for some or all of the responsibilities of the component to be taken over by other components which are already in use by the system. Reconfiguration therefore necessarily involves some degree of performance and/or functional degradation.

Borgerson further classifies such strategies as being either *manual*, *dynamic*, or *spontaneous*. In the first category the system takes no part in the strategy, which in the case of hardware may involve manual recabling and the like. In the second category, external stimuli cause the system to make use of provisions it contains for reorganizing its future activity. Spontaneous replacement and reconfiguration are strategies that are carried out entirely automatically by the system itself, and are sometimes referred to as "self repair" strategies.

Example: When a failure occurs in a hardware unit (such as a processor) in the PRIME system, the system is spontaneously reconfigured to run without that unit. Thus the system was designed as a "fail-softly" system which provides a graceful degradation in service when a failure occurs [BORG 72, BORG 73]. In contrast, the JPL-STAR computer discussed in Section 5 uses spontaneous replacement to deal with certain types of hardware fault.

To date there has been comparatively little work on the design of systems which are specifically intended to tolerate software faults, and on means of spontaneous replacement for this purpose. This is in fact what is provided by the recovery block scheme, (described below, under Backward Error Recovery), which can be regarded as a technique for using standby spare software components. Here the standby spares are of course of different design to that of the main component. A principal difference between this scheme and that usual with hardware standby sparing is that the spare component replaces the main component only momentarily. This is just in order to cope with the particular set of circumstances (such as the pattern of input data) that has caused a residual design fault to reveal its presence. Thereafter, attempts are made once again to rely on the main component. With hardware standby sparing, the spare component, being of the same design, should be as good as the main component, and would normally be used as a replacement at least until the main component has been repaired, or more probably until it itself starts to fail.

Damage Assessment

Damage assessment can be based entirely on a priori reasoning, or can involve the system itself in activity intended to determine the extent of the damage. Each approach can involve reliance on the system structure to determine what the system might have done, and hence possibly have done wrongly. The approach can be explained, and might have been designed, by making explicit use of atomic actions.

Figure 6 shows a set of processes and their as yet uncompleted atomic actions. (It is derived from Figure 3 but does not show the complete history of process creation and deletion, and of the lifetime of atomic actions. Rather, all completed actions and processes are elided.) Figure 6 therefore indicates, for example, that process 1 has been completely isolated since point C1 in its progress, but that process 2 has perhaps had some interactions with process 3 since they passed points D2 and D3 respectively, but not since process 3 passed point G3.

If process 3 is now detected to be in error, then one possible strategy is to assume initially that what it has done since point G3 is suspect. However if process 2 is the one that is detected to be in error the same strategy would involve an initial assumption that everything it has done since D2

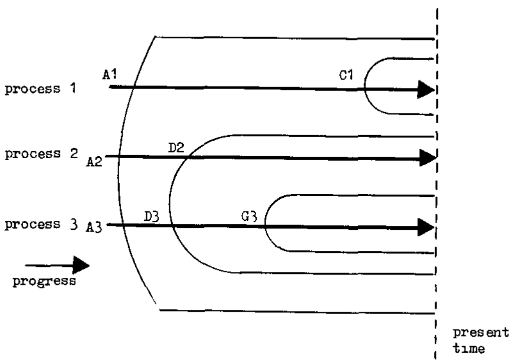

FIGURE 6. Extant atomic actions.

and everything process 3 has done since D3 is suspect. These assumptions may of course be too optimistic, in which case greater amounts of recent progress, perhaps on the part of a greater number of processes, will have to be treated with suspicion.

Atomic actions thus provide a simple choice of an a priori delimitation, or rather sequence of delimitations, of amounts of possible damage corresponding to each different error detection point. Such a delimitation can be used to select the area which is inspected for possible damage—a process which is feasible only to the extent that records have been kept of actual information flow and/or meaningful consistency checks can be performed on the system state. Alternatively, and much more simply, one can regard everything within the delimited area as being suspect, so that all must be abandoned. (This is discussed further in Backward Error Recovery, below.) However both these uses assume that the atomic actions are actual, rather than merely hoped for. Thus reliance on atomic actions for damage assessment is reliance on the constraints against unplanned information flow, i.e., reliance on "error confinement mechanisms" [DENN76].

In practice, damage assessment is often involved closely with efforts at error recovery and at dealing with faults, and is usually a rather uncertain and incomplete affair. Thus effort spent in trying to prevent the spread of damage, by careful definition and monitoring of interfaces between components, (e.g., using a capability-based protection scheme) is well worthwhile.

Error Recovery

Schemes for dealing with the damage that has been assessed as existing when an error is detected are usually classified into backward or forward recovery techniques. Such techniques aim to place the system in a state from which processing can proceed and failure can be averted. *Backward error recovery* involves first of all backing up one or more of the processes of a system to a previous state which it is hoped is error-free, before attempting to continue further operation of the system or sub-system. (The method of further operation will depend on the fault treatment strategy which is being used.) This technique is thus in sharp contrast to *forward error recovery*, which is based on attempting to make further use of the state which has just been found to be in error. (This distinction is in fact related to the structuring of the system—the system is regarded as having two levels, the lower level consisting of the components that provide the means of backward error recovery. It is only the system state visible at the upper level which is backed up, using information visible only at the lower level. If this classification of the information in the system were not made, one would have to regard all the activity which followed error detection as forward error recovery, by definition.)

Backward Error Recovery

Backward error recovery depends on the provision of *recovery points*, i.e., a means by which a state of a process can be recorded and later reinstated. Various techniques can be used for obtaining such recovery points. Checkpointing-type mechanisms involve foreknowledge of the resources that the processes could modify while they are in the atomic action (e.g. all of working storage). Audit trail techniques involve recording all the modifications that are actually made. Recovery cache-type mechanisms are a compromise which involve recording the original states of just those resources which are modified (see [ANDE76] and [VERH77a,b]).

The recovery block scheme [HORN74, RAND75], a technique of structuring sequential programs so as to provide means of tolerating faults whose exact location and consequences have not been anticipated, is based on the use of recovery cache-type mechanisms for backward error recovery. Recovery blocks in fact provide a means for expressing nested atomic actions in a sequential program, and for specifying a final programmed check (the acceptance test discussed previously in this section) on the results of an atomic action. The recovery block structure also enables zero or more alternative algorithms (called "alternates") to act as standby-spares for the atomic action. Figure 7, taken from [ANDE 76] illustrates the form of a simple recovery block example.

```
ensure
data still valid                                "acceptance test"
by
apply fast update                               "normal algorithm"
elseby
apply slow but sure update                      "standby-spare 1"
elseby
warning ("update not applied")                  "standby-spare 2"
elseerror
```

FIGURE 7. A simple recovery block.

The functioning of a recovery block is as follows: the first algorithm of the recovery block is executed, followed by the evaluation of the acceptance test. If the acceptance test fails, then the next algorithm in the block is attempted. (This also occurs if an error is detected during the execution of an alternate.) However, before this next algorithm is invoked, the state of the program is automatically reset to its state before the previous alternate was started (i.e., backward error recovery). Thus everything that the program has done since entering the current recovery block is discarded. If the final alternative algorithm (the last standby-spare) does not succeed in reaching and passing the acceptance test, this is treated as an error in the enclosing recovery block (if there is one) which therefore leads to the backward error recovery of the enclosing block.

Because each of the alternates can be designed on the assumption that they start from the same state, their designs can (and preferably should) be independent of each other. The designer of one alternate need have no knowledge of the designs of the other alternates, much less any responsibility for coping with any damage that they may have caused. Equally, the designer of a program that contains a recovery block does not necessarily have to concern himself with which of the various alternative algorithms of the recovery block was eventually used. It is therefore argued that the extra size of programs that incorporate recovery blocks as a means of design fault tolerance does not imply any increase in complexity. Clearly with this scheme the form of fault treatment is very simple. Since the intention is to cope with faults whose exact position and form are unanticipated, no attempt is made at fault diagnosis and location, although errors are logged for off-line inspection. Instead, after error recovery the next alternative algorithm is used as a temporary standby-spare to cope with just the particular set of input data that led to the error. On subsequent uses of the recovery block, with (presumably) different input data, the normal (first) algorithm will again be used. Thus the aim is to continue to provide service until there is an opportunity for manual fault diagnosis and repair.

The use of atomic actions as a basis for backward error recovery can be generalized so as to deal with systems in which there is concurrent activity, since atomic actions provide a means for specifying pre-planned limitations on information flow in such systems. All that is required is that recovery points are saved for a process each time it enters an atomic action and are retained for all processes involved in any given atomic action until all have reached the end of that atomic action. (The straightforward way of ensuring this latter requirement is to arrange that no process is allowed to leave a shared atomic action until all have indicated their readiness to do so. This is the scheduling restriction involved in "conversations" [RAND75].)

Given such a discipline of saving and retaining recovery points, the set of extant atomic actions directly defines the amount of system activity that will be abandoned should an error be detected in any process.

Example: In Figure 6, if an error is detected in process 1, it could be backed up to recovery point C1. However if an error was detected in process 2, it would have to be backed up to recovery point D2 and process 3 to recovery point D3.

More complex backward error recovery strategies are possible, either instead of, or as supplements to, this atomic action-based strategy. These all involve incorporating strategies into the actual system to determine what information flow has, or might have, occurred. Such strategies therefore can be regarded as ones whose tasks involve both damage assessment and the choice of which recovery points are to be used.

Thus, in contrast to the case of an atomic action-based scheme, the design of such strategies involves the possibly very difficult task of taking into account the effects of any other activity in the system that continues while the strategy is evaluated, and also the possibility of faults occurring during this time (see [MERL77]).

A major problem with strategies based simply on records of what information flow has, or might have, occurred is that they do not necessarily result in the location of a set of usable recovery points. The problem is illustrated in Figure 8. This shows three processes, each of which has three recovery points in addition to that taken at the point of its entry to the shared atomic action. The dotted lines mark occurrences of information flow between processes.

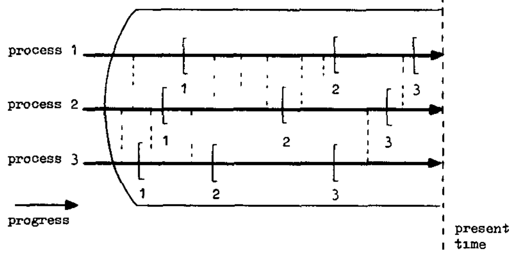

FIGURE 8. The domino effect.

If it should be necessary to back up process 1, this can clearly be done to its third recovery point, without affecting the other processes. If it is process 2 that requires backing up, it will be seen that if this is to its third recovery point, then process 1 will also have to be backed up, and to its second rather than its third recovery point, since its third recovery point postdates interactions between the two processes. However if process 3 has to be backed up, it and the other two processes will have to be backed up right to the start of the atomic action, due to what can be visualized as a sort of "domino effect" [RAND75].

The search for a usable set of recovery points for the process which is in error, and for any other processes which are also affected, is in fact a search for a set of consistent recovery points. Such a set we will term a *recovery line*. Each process will at any moment have associated with it a (possibly empty) sequence of recovery lines. A recovery line in fact identifies a set of recovery points, each belonging to a different process, such that

a) one of these processes is the process in question;
b) no information flow took place between any pair of processes in the set during the interval spanning the saving of their identified recovery points;
c) no information flow took place between any process outside the set and any process in the set subsequent to its identified recovery point.

Thus defined, a recovery line identifies a set of process states which might have all existed at the same moment, and since which all the processes have been isolated from the rest of the system, so that abandoning the activity which postdates these states is a straightforward task. (The concept of recovery line is very similar to that of "sphere of control," introduced by Davies and Bjork [DAVI72, BJOR72] as a method of representing the system information used for scheduling processes, recording information flow, and preserving recovery points.)

Figure 9 is based on Figure 8, with all the recovery lines drawn in. The line represent-

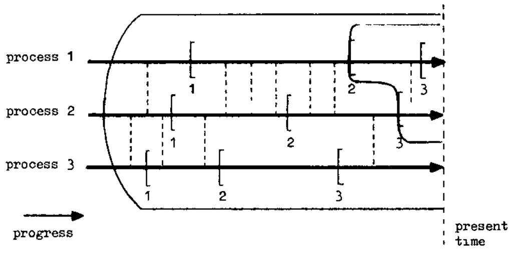

FIGURE 9. Recovery lines.

ing the shared atomic action is also a recovery line, since recovery points saved at the start of an atomic action automatically constitute a pre-planned recovery line. Without such pre-planned recovery lines, it may well be the case that, due to the domino effect, one or more processes have no recovery lines at all. Such processes must be regarded as being "out of control."

Clearly, therefore, if the scheduling constraints (and hence performance impact) of shared atomic actions can be accepted, they provide a much preferable basis for backward error recovery than what might be termed "information flow analysis" techniques. These scheduling constraints have so far been described in such a way as to imply that shared atomic actions can be used only where there is an explicit requirement for a set of processes to have some private interactions, in furtherance of some common goal. In fact one can imagine the sequencing constraints being imposed on a set of separately designed processes, merely because of a fear of accidental interactions. (An extreme example of this is the step of stopping all the activities of a system so as to take a system-wide checkpoint.) Unfortunately, in some situations one must resort instead to the more complicated and risky information flow-based techniques, because of an absence or scarcity of pre-planned recovery lines.

Schemes based on information flow analysis——The essence of such schemes is the search for a recovery line. Recovery lines come into existence through the saving of recovery points. When a recovery line ceases to exist we say that a *commitment* occurs. A commitment is therefore an impediment to future recovery. (This definition of commitment differs greatly from the various and varied descriptions of commitment given by Davies [DAVI72] and by Bjork and Davies [BJOR72] but is claimed to capture the essential idea involved.) There are several important varieties of commitment, including what we term "explicit commitment," "interaction commitment," and "accidental commitment."

The deliberate discarding of a recovery point (for example at the end of an atomic action), and hence of any recovery lines of which it was a member, constitutes an *explicit commitment*. Such explicit commitment is therefore the means by which the extent of the resources devoted to error recovery provisions is limited. In some schemes these limitations can be quite severe, so that only a few recovery points ever exist.

Example: File processing systems often have just two recovery points, implemented by the retention of so-called father and grandfather files.

If information is allowed to flow between processes whose latest recovery lines are not identical, and there is no means of "compensating" for this information flow, this constitutes an *interaction commitment*. It means that all recovery lines, for each process, which postdate the latest recovery line which is common to all the processes cease to exist. Such interaction commitments are to be expected for unplanned recovery lines—recovery lines that correspond to uncompleted atomic actions will cease to exist if, for example, the resource locking mechanism fails.

Example: We can re-interpret Figure 6 as a representation of a set of recovery lines for the three processes. Thus process 1 currently has recovery lines (C1) and (A1, A2, A3); process 2 has recovery lines (D2, D3) and (A1, A2, A3) and process 3 has (G3), (D2, D3) and (A1, A2, A3). If there is now an uncompensatable information flow between process 2 and process 3, process 3 will lose its latest recovery line, i.e., (G3).

Compensation [BJOR72] is in fact a form of forward error recovery, involving the provision of supplementary corrective information, a technique which is discussed later in this section. There is however one special case of information flow which does not need compensation, and which does not cause any commitment. This is the case when the recipients are not relying on the accuracy of the information they receive, to the extent that any one of a defined set of possible values will be acceptable, whether or not the sender later regrets having sent it. Bjork and Davies [BJOR72] term such information *reference information*, although *insignificant information* would perhaps be a better term. (A possible example of such information is that obtained by a process which is monitoring the progress of another process.) Barring means of detecting that information is being treated as insignificant, some explicit indication of this intention is needed. Such indications can be used to help determine when back-up is needed.

Clearly the task of establishing which recovery lines still exist, rather than which have disappeared because of interaction commitments can be a very difficult one. The rational design of strategies for this task must be based on the assumption that, in the absence of known constraints, information flow will have occurred unless it can be established otherwise. (See for example [EDEL74].)

The third form of commitment poses even worse problems. This is *accidental commitment*, which occurs when something damages the information or the mechanisms that provide one or more (perhaps many more) recovery lines, thus making them unusable. The effects of this sort of commitment can be particularly insidious, since its occurrence might be, from the designer's viewpoint, totally unrelated to the activities of many of the processes it may affect. Moreover the commitment may not be noticed at the time that it occurs, so that its effects are only felt later if and when error recovery is needed.

Our approach to understanding, and perhaps even coping rationally with, such problems involves the concept of multilevel error recovery, discussed later in this section.

Schemes based on information validation——The designs of the backward error recovery strategies described above do not involve any critical assumptions as to which of the processes involved is, or is not, responsible for a particular error. However several important approaches to backward error recovery are based on what are often quite reasonable assumptions about error causes.

For example, some data base systems use audit trails of transactions in order to shield users from the consequences of the system having had to be backed up. Rather than request the user to resubmit the sequence of requests which postdated the recovery point, the copies of these messages that are held in the audit trail file are reprocessed instead. This technique presupposes that the user's messages were neither themselves the cause of the error nor mistakenly based on erroneous messages from the system itself. (The scheme described by Russell [RUSS76] generalizes this approach somewhat, so as to deal with networks of interacting processes, rather than a single system interacting with a set of users.)

In effect, such techniques assume that the inputs to a process can be presumed valid or can be checked completely. A contrasting scheme would allow a process some means of indicating when its outputs should be regarded as having been certified as correct (with respect to its inputs, and assuming that the process was executed correctly). Then when an error was subsequently detected, such certifications would be regarded as indicating which recovery lines were not worth using because more global error recovery was needed. Whether such global error recovery is actually attempted is another matter.

Forward Error Recovery

The relative simplicity of backward error recovery is due to two facts: first, that questions of damage assessment and repair are treated quite separately from those of how to continue to provide the specified service; and second, that the actual damage assessment takes virtually no account of the nature of the fault involved. In forward error recovery these questions are inextricably intermingled, and the technique is to a much greater extent dependent on having identified the fault, or at least all its consequences. Thus generalized mechanisms for backward error recovery are quite feasible.

In contrast, forward error recovery schemes must, it seems, be designed as integral parts of the system they serve. Despite this entanglement with all of the other aspects of a system, forward error recovery techniques can be quite simple and effective. However this simplicity will be based on the assumed simplicity of the faults and the ensuing damage that they are expected to have to cope with. Where such assumptions are justified, forward error recovery can be both simple and much more efficient than backward error recovery. (After all, backward error recovery involves undoing everything a system has done since passing its last recovery point, not just the things it did wrongly.) However in many cases the simplicity owes much to the fact that the forward error recovery provisions are not even required to achieve complete error recovery.

Example: In many operating systems, recovery after a crash involves attempting to establish what system resources, and what parts of the data relating to current jobs, seem unharmed. These jobs will then be restarted. On the other hand, jobs which were being executed when the crash occurred, or perhaps all jobs that were in main store, may not be restartable or even recognizable as jobs. Thus although some (perhaps most) users will be unaware of there having been any trouble, others will find that their work has been lost or spoiled, and will themselves have to sort out what should be done about the situation.

The classification of faults that is implied by choosing a view of the system and its structuring provides us with a means of analyzing forward error recovery strategies. Thus they can be provided in a system in order to cope with faults either in the system components, or in systems it is interacting with, or in the design of the algorithm of the system. (It should however be noted that a forward error recovery strategy intended for one of these purposes might, more or less accidentally, on occasion cope with other classes of fault.) Forward error recovery strategies intended for coping with component faults are discussed next, under the heading Exception Handling, and our second category follows under the heading Compensation. It is our opinion that the third case, the deliberate use of forward error recovery to cope with residual design faults in the algorithm of which it is regarded as part, is inappropriate. Should one wish to provide means of tolerating such faults, backward error recovery seems much more appropriate. Forward error recovery would involve automated diagnosis of design faults, a task whose complexity is such as to be productive of design faults, rather than conducive to the design of reliable systems. Indeed, when the type and location of a design fault can be predicted, it should be removed, rather than tolerated. This topic is discussed more fully in [MELL77].

Exception handling——Rational design of an algorithm incorporating strategies for recovering from errors due to faulty components requires prediction of the possible faults, and of how they manifest themselves outside the component. In effect therefore it involves including various possible types of undesirable behavior among the activities that the component is specified as providing. As long as a component does perform one of its specified activities, it will not, by our definitions, have failed. However from the viewpoint of the system there will be a fault when one of the undesired activities takes place, and some (not necessarily easily identified) part of the algorithm of the system will be concerned with coping with it.

Example: A hardware system can incorporate forward error recovery strategies, based on the use of error correcting techniques and codes, to cope with faulty information storage and transmission components. A given coding scheme will be usable for correcting just a particular, limited, class of errors (e.g., those involving not more than *n* successive bits).

It is often desirable to have some means of distinguishing, from each other, and from the main part of an algorithm, those of its parts which have the task of providing forward error recovery for the different kinds of component fault which have been envisaged. Programming language designers have catered to this need by means of language facilities for "exception handling." The PL/I language provided an early form of such facilities with its "ON conditions";

more recent proposals include those by Goodenough [GOOD75], Wasserman [WASS76] and Levin [LEVI77].

Example: Wasserman's proposals are linked closely with procedure declarations and calls. A procedure can contain statements which will cause particular exceptions to "hold" (e.g., arithmetic overflow, end of tape, array bound check, etc.). Then each procedure call explicitly, or by default, indicates what is to be done when each possible condition occurs. This is illustrated in Figure 10. The procedure CHECKNUM provides an exception handler MESSAGE which will be invoked by the procedure READ if the OVERFLOW, UNDERFLOW or CONVERSION exceptions occur, and the exception handler CRASH for the IOERR exception.

Since we regard exception handling as a means of programming forward error recovery, we do not regard it as appropriate for coping with residual bugs in programs. However forward and backward error recovery techniques should be thought of as potentially complementary, rather than competitive, techniques. Thus exception handling can be combined with the recovery block scheme for backward error recovery. An example of such a combination is

```
VAR NUM: INTEGER;
EXCEPTION GOOF = OVERFLOW OR UNDERFLOW OR
                 CONVERSION;

PROCEDURE CHECKNUM (-> J:INTEGER);
   VAR COUNT: INTEGER; INIT COUNT<- Ø;
   PROCEDURE MESSAGE;
   GLOBAL COUNT: ASSIGNED;
   BEGIN
      COUNT <- COUNT + 1;
      WRITE ("PLEASE TRY AGAIN");
      IF COUNT ≥ 3 THEN
      BEGIN
         WRITE ("THREE STRIKES- -YOU'RE OUT");
         SIGNAL FAIL;
      END          ELSE RETRY
      FI
   END MESSAGE;
   PROCEDURE CRASH;
    local declarations
   BEGIN
       any required cleanup
      SIGNAL FAIL
   END CRASH;
BEGIN
 body of CHECKNUM
      READ (J); GOOF: MESSAGE, IOERR: CRASH
END CHECKNUM;

BEGIN
    main program
   •
   •
   CHECKNUM ( -> NUM);
    program terminates if FAIL is signaled
   •
   •
END.
```

FIGURE 10 An example of exception handling, taken from [WASS 76].

given in [MELL77]. This example shows a program which deals with readily forseeable simple faults, such as invalid input data, by means of exception handling and less likely faults, including those arising from design inadequacies in the exception handlers, by recovery block techniques.

Compensation——Compensation, our second form of forward error recovery, fulfills a very different, and indeed much more important need, one which cannot be coped with by backward error recovery. It provides the means for trying to deal with the situation of an error being detected while a system is in communication with an environment which cannot be backed up. As explained earlier *compensation* is an act by which one system provides supplementary information intended to correct the effects of information that it had previously sent to another system. This requires that both (or more generally, all) the interacting systems are such that, when erroneous information is found to have been sent out or let out by a system, all of the other systems are capable of accepting the corrective information which is then sent out by, or on behalf of, the offending system.

Example: If a stock control data base has been updated because of an input message indicating the issuance of some stocked items, and it is later found that this message was incorrect, it might be possible to compensate for the wrong message by means of another message purporting to record the acquisition of replacement stock items. Such a simple compensation would probably not suffice if, for example, the stock control system had, as a result of the wrong information, recalculated optimum stock levels, or produced purchase orders for the replenishment of the stock involved.

As has already been described, determination of the extent of information flow, and hence of the requirements for compensation, can be aided by the concept of atomic actions, and by the careful planning of such actions. System algorithms that incorporate compensation strategies can be designed using such structuring techniques as exception handling. (See [LEVI77] for a detailed description of an exception handling scheme which is specifically designed to facilitate the programming of compensation strategies.) However the problem of designing compensation algorithms that will work well, in complex systems, is an immense one, with no obvious general solution. Clearly the simple, but unfortunately impracticable, solution is to avoid the need for any compensation by guaranteeing that no incorrect results are ever allowed to leave a system.

Example: Even recent data base systems which incorporate very sophisticated error recovery strategies can occasionally get the data that they hold into such a state that there is no alternative but to suspend service, and attempt manual correction of the data base, and manual assessment of the damage that has been done through the provision of wrong information to the organization using the system [GRAY77]. The basic requirements for such manual determination of what compensation is necessary are an understanding of the information flow in the environment of the system, and a complete record of all information flow to and from the system (see Bjork [BJOR74]). In practice what usually happens is that such manual compensation is neither guaranteed, nor expected, to be complete.

Multilevel Error Recovery

The advantage of multilevel designs, i.e., those characterized by the existence of specified (and hopefully well-documented) abstraction interfaces, is that each level can be designed independently of the internal designs of other levels. This advantage is especially desirable in situations which are complicated by the possibility of faults. Indeed it seems to be the only means we have of mastering problems such as the occurrence of further faults while a previous fault is still being dealt with.

Ideally the algorithm at any level of the system would be designed on the assumption that all of its components, that is the level below, worked perfectly (either faultlessly, or with complete masking of faults). Then error recovery would be needed only for purposes of coping with invalid use of the system and with design faults in the algorithm of the system.

Even when perfection of components is not assumed, it is possible to achieve considerable independence of the design of separate levels, and even of their provisions for error recovery. This in fact is conventional practice in the simple multilevel structures (namely those that relate directly to the physical construction of the system, and involve no sharing of components) that are typical of hardware designs.

Example: The three components, say processors, in a TMR structure might use such internal error recovery strategies as instruction rentry and error correcting codes. As long as the faults which cannot be masked by these strategies are independent, and do not destroy the TMR structure, it can mask their effects at the system level.

In more sophisticated forms of multilevel structure, and in systems which have to cope with more than just operational faults in correctly designed hardware components, the situation is more complicated. However in all cases rational design of a system is impossible if it incorporates faulty components which are known to leave themselves in internal states for which no external state (i.e., system level abstraction) is defined.

Therefore one approach, feasible only for faults whose exact consequences can be predicted, is to incorporate the description of the faulty behavior into the specification of the abstraction interface. The system algorithm then has to be augmented so as to deal with this specified undesirable behavior, for example by using explicit exception handling facilities, or static redundancy (such as in the above TMR example). Parnas and Wurges [PARN76] give a detailed discussion of one exception handling approach, couched in terms of techniques for programming hierarchies of abstract machines, and based on the use of traps as a means of reporting "undesired events" (i.e. exceptions). An account of how the approach is used in the HYDRA operating system, mainly for coping with hardware faults, is given by Wulf [WULF75]. (See also Denning [DENN76].)

The technique of augmenting and hence complicating the specification of an abstraction interface by incorporating details of all anticipated (unmasked) faults reduces the practical, if not the theoretical, independence of the designs of the levels it separates. The one alternative means of retaining a large measure of design independence is to require that unmasked faults in the components constituting the lower level cause this lower level to assume a defined state which corresponds to some previous external state. Thus from the system's viewpoint, some sequence of operations end up by having had no effect. In other words, errors detected at one level result in what can be seen from the next level as backward error recovery. A detailed explanation of a very basic version of such a scheme applicable to levels in a fully interpretive multilevel system (such as that shown in Figure 4) is described in [RAND75]. More sophisticated versions, which do not require levels to be fully interpretive, are described in [VERH77a,b] and [SHRI78]. A detailed general account of multilevel system design and the problems of recoverability is given in [ANDE77].

Example: The scheme described in [SHRI78] is used to show how backward error recovery can be provided by an operating system to independent user processes which are competing for the resources (e.g., storage space) being managed by that operating system. Faults in one process do not affect other processes, and the specification of the operating system interface remains simple.

All strategies for independent design of separate levels of a multilevel system depend absolutely on the vertical structure being "actual," as was discussed in the section on levels of abstraction. With respect to error recovery specifically, there would, for example, be little point in having the checkpoints, audit trails, or whatever, that were being kept for different levels, in the same failure-prone storage device. If this were done, and a failure did occur, the level structure would be an impediment to understanding what had happened, rather than an aid to designing some means of tolerating such occurrences.

Multilevel recovery schemes therefore have to be designed with very careful regard for the possibility of an accidental commitment (i.e., destruction of recovery lines) caused by the failure of some system component which is common to the recovery mechanisms used for separate levels. Many possible types of multilevel recovery schemes are possible, with widely varying strategies for achieving a measure of independence between the recovery mechanisms used for the different levels. Perhaps the most common scheme is one which provides two levels of recovery after a system crash, one referred to as enabling a "warm start," the other a more brutal "cold start" (one in which the system retains no information from any of its activity prior to the crash), for use when warm start cannot be achieved. Both levels of recovery are usually incomplete, since even the warm start scheme normally ignores the effects of some interaction commitments, so that some users are left to fend for themselves. This is the case even with the very sophisticated scheme described by Lampson and Sturgis [LAMP76]. However this scheme, which is based on some quite plausible assumptions about the ways in which storage, communication channels, and processors can fail, does guarantee the integrity of the system's distributed shared files, no matter when the crash(es) occur.

5. FAULT-TOLERANT COMPUTING SYSTEMS

There have been many fault-tolerant computing systems designed (and built) to satisfy high reliability requirements. The terminology adopted in the description of such systems has in general been sufficiently varied that comparison between the systems is often difficult or impossible. One of the aims of this paper has been to present some coherent but general reliability terminology which can be used to describe and compare such fault-tolerant systems from a common viewpoint.

As an aid to further explication of the ideas, and to appreciating their manifestations and significance in practice, a number of different systems have been studied by the Newcastle Reliability Project, and overall descriptions of some of these systems, using the terminology adopted in this paper, have been presented in [RAND78]. This present section discusses just three of those systems, namely: 1) the Jet Propulsion Laboratory STAR computer, designed for unmanned spacecraft applications; 2) the Bell Laboratories ESS No. 1A processor, designed for electronically controlled switching systems such as telephone exchanges; and 3) the Bolt, Beranek and Newman PLURIBUS, which was designed as a message switching node for the ARPA network. The descriptions are based solely on the particular documents that are referenced and so do not necessarily give an up-to-date or complete account of the systems. The particular selection of systems has been chosen merely to illustrate the diversity (but common underlying principles) of approaches used in systems that were developed to operational status, and to show how the different environments in which the systems were designed to function have been reflected in the manner in which fault tolerance has been provided. (The selection deliberately does not include a large data base system—such systems are the subject of the companion paper by Verhofstad.)

The JPL-STAR Computer

The Jet Propulsion Laboratory Self Testing and Repairing (JPL-STAR) computer was the result of studies, initiated in 1961, into the design of fault tolerant computer systems. The principal goal stated in [AVIZ71] for the design of the STAR was to achieve fault tolerance for a variety of hardware faults, namely transient, permanent, random and catastrophic. In order to achieve this degree of fault tolerance, a variety of techniques was used: coding, monitoring, standby spares, replication with voting, component redundancy, and program rollback and repetition. No specific provisions were made for possible software faults—rather it would appear that programs that are run on the STAR are assumed to be correct.

The STAR was designed as a general-purpose fault-tolerant computer, whose main characteristics were chosen to match the requirements of a spacecraft guidance, control and data acquisition system which would be used on long unmanned space missions. Thus the reliability requirements for the STAR were for 100,000 hour survival with a probability of 0.95, and with a maximum time requirement for recovery of 50msecs. An experimental laboratory version of the STAR was constructed and operational in 1969, although it did not implement all of the features of the STAR design. This description will therefore concentrate on the design presented in [AVIZ71] rather than on the particular implementation of the STAR. Some of the results from experimentation with the laboratory STAR are presented in [AVIZ72], and [ROHR73] discusses details of the system software.

System Description

The STAR computer may be regarded as a three level system. The bottom (hardware) level of the STAR supports two software levels consisting of a resident executive and the applications programs.

The hardware level of the STAR can be regarded as having a decentralized organization. A standard configuration of functional subsystems (i.e., components) implements the abstraction interface presented to the higher (software) levels, which essentially has the appearance of a single-cpu system with the required computing capability. Figure 11, taken from [AVIZ71], illustrates a static structuring of the bottom level of the STAR computer, consisting of the following functional subsystems:

1) Control processor (COP)—contains the index registers and contains and maintains the program location counter;
2) logic processor (LOP)—performs logical operations;
3) main arithmetic processor (MAP)—performs arithmetic operations;
4) read only memory (ROM);
5) read/write memory (RWM);
6) input/output processor (IOP);
7) interrupt processor (IRP)—handles interrupt requests;
8) test and repair processor (TARP)—monitors the operation of the computer and implements the recovery.

Communication between the various units is carried out on three buses: the memory-out (M-O) bus; the memory-in (M-I) bus; and the control bus.

The second level in the STAR computer, the resident executive, provides typical operating system features for use by the applications programs in the third level.

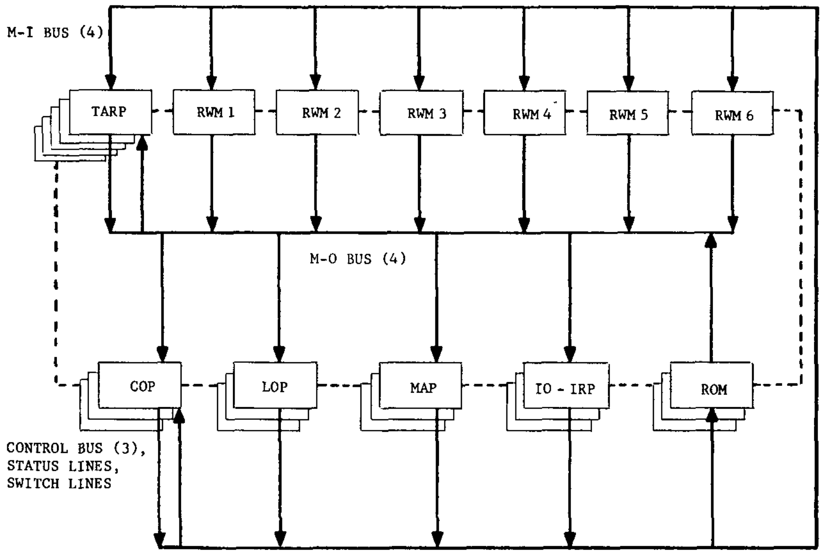

FIGURE 11. STAR computer organization.

These features include interrupt control, I/O processing and job scheduling.

The STAR computer operates in two modes: standard mode and recovery mode. In standard mode the stored programs are executed, and the "normal" algorithms of the TARP issue the principal clocking signals and continually monitor the operation of the system. Recovery mode is discussed subsequently.

Reliability Strategies

The STAR computer employs a variety of techniques, as already mentioned, to attain the desired hardware controlled self-repair and protection against many types of faults. The TARP implements, in hardware, the majority of these features. The correct operation of the rest of the system is based on the assumption that the TARP is always functioning correctly.

Error detection——The major error detection mechanisms in the STAR computer are implemented in the hardware level. In particular, the TARP is responsible for monitoring the operation of the computer, and detects errors by two methods: 1) testing every word sent over the two data buses; and 2) checking status messages from the functional units for predicted responses.

All machine words in the STAR are protected by various error detecting codes. The codes are preserved by arithmetic operations although not by logic operations. (Consequently operation of the logic processor is checked by replication, using two operational copies to indicate when disagreement occurs.) Thus the TARP can detect errors arising from faults in the storage, transmission and processing of words, i.e., the TARP checks the abstraction interface between the components at this level.

Each functional unit in the STAR computer generates status messages which are checked in the TARP against the responses that are predicted (independently) by logic internal to the TARP. Thus the TARP can identify, for example, both improperly activated units (unexpected message) and failed units (absence of an expected message).

The types of message that a unit can generate include "disagree with bus" message and "internal fault" message. The "disagree with bus" message is needed for duplex operation of units. The "internal fault" message is produced by monitoring circuits internal to each unit. These circuits utilize redundancy internal to the unit to (attempt to) detect errors in the internal state of that unit. For example, a reversal check is employed whereby "inverse microprogramming" deduces what the operation/algorithm should have been from the active gating signals. The deduced result can then be checked against the requested operation/algorithm. The status message generating circuits in each unit are themselves duplicated, enabling some errors in the status messages to be detected by the TARP.

The read/write memory units in the STAR computer have two modes of operation, absolute and relocated. In absolute mode the unit will respond to its own wired-in name; in relocated mode the unit responds to an assigned name. The relocated mode can be used by the executive to provide duplicated or triplicated storage for programs and data.

Finally, errors in the TARP itself are detected (and masked) by triple modular redundancy. The intention is that three fully powered copies of the TARP will be operational at all times with their outputs decoded by a 2-out-of-(n+3) voter. An assumption made, therefore, by the rest of the system is that faults in the TARPs are always masked.

It is not clear from the documentation what error detection mechanisms (if any) are made available to the higher (software) levels in the STAR, apart from the availability of information concerning errors that occurred in the interface with the arithmetic unit (for example, overflow, division by zero).

Fault treatment——Once an error has been detected the TARP exits from standard mode and enters recovery mode (i.e., the "abnormal" algorithms are invoked). In this mode the TARP is responsible for locating and treating the fault. It can be seen from the previous section that the prediction logic of the TARP coupled with the status messages should, in general, enable the TARP to locate the faulty unit causing the error (assuming of course that this prediction logic itself is correct). However, for fault conditions which cannot be resolved by the TARP logic there is a wired-in "cold start" procedure (which is also invoked in the case of temporary power losses).

As the STAR was intended for unmanned space missions, permanent faults are treated by the automatic replacement of the faulty unit. There is no provision for their repair. The standard configuration of functional units is supplemented by one or more unpowered spares of each unit, and the TARP implements a spontaneous replacement strategy. A repeated fault indication in a unit leads to its replacement, implemented by the TARP power by switching. Spare TARP units are also provided; thus if one of the three operational units disagrees with the other two, then the faulty unit can be replaced by a spare.

Fault treatment is also performed by the resident executive in the STAR. Software assistance is required for memory replacement, both for assignment and cancellation of relocation names, and for reloading the replacement memory. Reference is also made in the documentation to a class of diagnostic instructions, which can, among other things, exercise the unit status messages and the TARP fault location logic, and also control the power switching to the spare units. The executive apparently implements diagnosis for faulty units, although the interface between this and the TARP is not made clear.

Damage assessment——It would appear that there is no dynamic damage assessment at any level in the STAR computer, as the error recovery described below is always invoked. However, much of the activity of the components in the bottom level could be regarded as consisting of simple atomic actions. First, each functional unit contains its own instruction decoders and sequence generators, as well as storage for the current operation code, operands and results. Apart from overall synchronization, each unit operates autonomously. It would appear, therefore, that once initiated a functional unit could operate atomically. Hence, if the internal monitoring circuits detected an error, the damage could be assumed to be localized to that particular unit. The second activity in which atomic actions can be identified is in the operation of the TMR-protected TARPs. Assuming that the activity of each TARP unit is atomic, then any TARP fault should be masked by the voter. Nevertheless, in both of these cases the more global error recovery described below is invoked.

Error recovery——The main form of error recovery in the STAR is implemented by backward error recovery of the software levels. When an error is detected and any replacements have taken place, the TARP issues a reset message which causes all operational units to assume an initial state (presumably the contents of the memories are not reset to an initial state). The program that was running is then forced to rollback, that is, to back up to a previous state.

The application programs are provided with a mechanism for establishing a recovery point. It would appear that it is the responsibility of the programs to establish recovery points as often as is needed for reliable operation [ROHR73], and to specify (correctly) the information that needs to be checkpointed. The program also assumes that this operation is performed reliably (and atomically). Moreover, as only one recovery point can be established at one time, an explicit commitment will occur whenever a new recovery point is established. There is no automatic compensation for information that has left the system prior to such a rollback—presumably this is the responsibility of the application programmers, as the recovery mechanisms provided would enable the program to perform compensation actions if it so required.

The executive level is responsible for implementing the recovery mechanism provided to the upper level. It uses the rollback point register in the TARP to achieve this. The rollback register can be updated by the executive level, and acts in effect as an interrupt vector, used to restart the executive when a rollback is invoked by the TARP. The rollback of the application program is then implemented by software in the executive. The executive also uses the rollback register to control non-repeatable events (for instance, input/output operations). The storage of checkpoints for the upper level programs is also the responsibility of the executive, and it provides duplexed storage for this purpose. [ROHR73] describes all of these strategies in more detail.

The only other form of error recovery in the STAR is provided by the "cold start" procedure in the TARP, augmented by a "cold start" capability in the resident executive.

Reliability Evaluation

According to [AVIZ71], early analytical studies using models of dynamically redundant systems had indicated that mean-life gains of at least an order of magnitude over that yielded by a non-redundant system could be expected from dynamically redundant systems, with standby spares replacing failed units. This gain compared favorably with the mean life gain of less than 2 in typical TMR systems. An analysis of a model of the STAR's reliability compared its reliability with that of a simplex computer of equivalent performance and with the Mars Mariner spacecraft computer (MM'69). Some of the results are indicated in Table I, taken from [AVIZ71], where the lower bound ($k = 1$) indicates an equal failure rate of powered and spare units, and the upper bound (k = infinity) indicates a zero failure rate of spare units.

It is stated in [AVIZ72] that initial experimental tests on the STAR have verified the effectiveness of the error detecting codes used on the STAR words and the coverage provided by the TARP for each functional unit. The tests, involving the introduction of noise bursts on the buses, demonstrated 99.5–100% proper recovery.

TABLE I. RELIABILITY VERSUS TIME FOR VARIOUS CONFIGURATIONS.

Mission Time (h)	MM'69 Computer	Simplex Computer	STAR Computer with S Spares			
			Upper Bound (K=∞)		Lower Bound (K=∞)	
			S=3	S=2	S=3	S=2
4368 (≈6 months)	0.928	0.82	0.9999998	0.99997	0.999995	0.99982
43 680 (≈5 years)	0.475	0.14	0.997	0.97	0.966	0.87
87 360 (≈10 years)	0.225	0.019	0.96	0.79	0.71	0.45

Limited tests on the adequacy of the TARP for error detection, fault diagnosis and recovery demonstrated 90-100% coverage for the various processors and memory modules.

Bell Laboratories ESS No.1A Processor

The Bell Laboratories Electronic Switching Systems (ESS) represent one of the first major attempts at incorporating extensive fault tolerance in a processor. In this case, the processor is the heart of the electronically controlled switching systems used in the main for telephone exchanges. The first system of this type was in service in 1965.

There are stringent reliability requirements to be met by the ESS systems. For example, those specified for the No.1A ESS [BELL77] require the system to be available for 24 hours a day, and that down time for the total system should not exceed 2 hours over its 40 year life. Moreover, the percentage of calls handled incorrectly should not exceed 0.02%. For the processor itself, the requirement is that its average outage time, that is the time during which established calls are retained but no new calls are established, is not greater than 2 minutes per year.

There are extensive references to the various ESS systems, particularly in the Bell System Technical Journals. This description will concentrate on the fault-tolerant techniques employed in the processor in the No.1A system, a description of which is given in [BELL77] and [CLEM74].

System Description

The No.1A processor is a self-contained high speed processor developed specifically for the control of Bell ESS systems. Its design is based on the experience gained from the earlier ESS systems. The instruction set of the No.1 processor is a subset of that of the No.1A, thus enabling the No.1 ESS call processing programs, described in [BELL64], to be used.

The system may simply be regarded as having two levels with hardware components at the bottom level supporting one software level. Figure 12, taken from [CLEM74], illustrates a static structuring of the hardware components of the No.1A processor. The heart of the system is the Central Control which is fully duplicated (CC0 and CC1). In general, both CCs operate in synchronization although one of the CC modules is designated as the standby, while the active unit controls the system.

The programs to be obeyed by the CCs are stored in the program stores (PS). The information relating to the processing of telephone calls and the translation (routing) information for the system is stored in the call store (CS) complex. This primary memory can be divided into "protected" and "unprotected" areas. The division is enforced by the active CC, which provides different instructions to write to these different areas, and contains the mapping registers which define the areas. The mapping registers are software controllable via other (special) write instructions. Basically, the protected area contains the parts of the memory that are not duplicated (e.g., the program stores), as well as those areas which vitally affect the operation of the system (e.g., the internal registers of the CC and the disk controllers).

The auxillary unit (AU) system at the bottom of Figure 12 consists of the bulk storage devices and their controllers, essentially disk file units (FS) and tape units (TUC). The disks provide a file store for the system, used mainly for holding the backup versions of the programs and data, and for holding infrequently used programs. The tape units hold the accounting information that is generated for complete calls, and also hold infrequently used parts of the system database. The components in the system are interconnected by the three (duplicated) buses indicated in Figure 12.

Basically there are two kinds of interrupt in the No.1A processor: "call processing" interrupts which invoke the "normal" (call processing) algorithms of the software, and "maintenance" interrupts which are discussed below. The call processing programs can be divided into two classes, deferrable and non-deferrable. The deferrable pro-

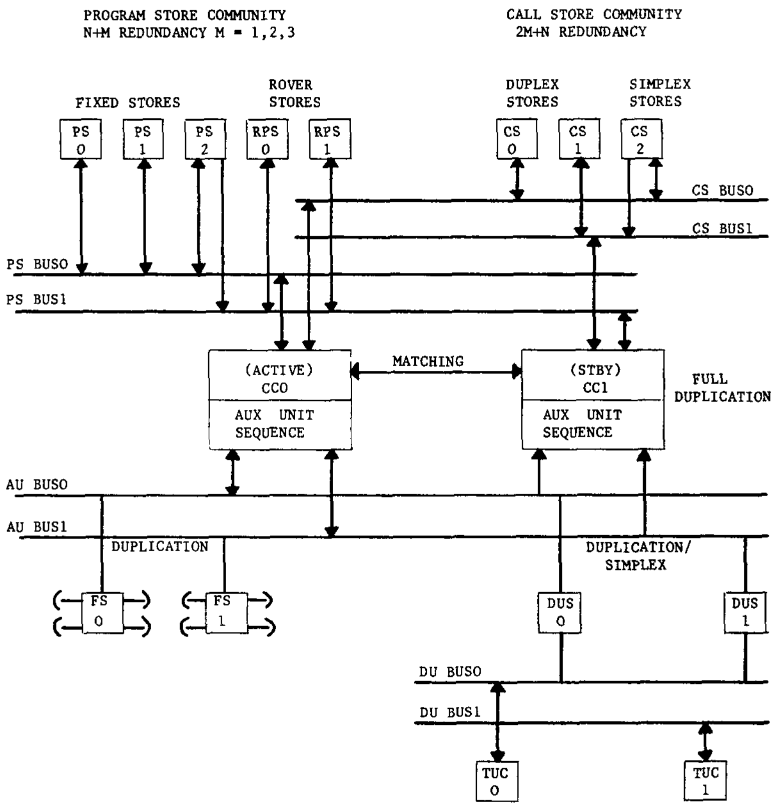

FIGURE 12. ESS No. 1A processor.

grams are those for which the data is already in the system, and the programs are not therefore critically synchronized to real time. The non-deferrable programs are those that must be executed on a strict schedule, otherwise data will be lost. These programs are generally the input/output programs, and are activated by a clock interrupt. The majority of programs in the system are deferrable, and run at what is referred to as the base-level, with all of the interrupts enabled.

Reliability Strategies

In order to attempt to reach the reliability requirement of an average of less than 2 minutes outage per year, comprehensive "maintenance" software is provided in the No.1A processor. The various strategies for error detection, fault treatment, damage assessment, and error recovery are closely related. In fact, many of the strategies employed in the No.1A processor have been based closely on the accumulation of experience from the various earlier installed ESS systems as to what sort of faults will occur, and with what exact consequences. Table II, taken from [BELL77], indicates the reliability objectives that have been set for each of the probable causes of outage time.

Although the 1A system makes use of the well-tested No.1 ESS call processing programs, it can be seen that "software deficiencies" are still expected to result in an outage of 0.3 minutes each year, although this time would appear to include outages caused by the integration of new software into the system.

The "hardware unreliability" category includes faults in the hardware which pre-

TABLE II. No. 1A SYSTEM OUTAGE ALLOCATION

	Outage time (minutes/year)
software deficiencies	0.3
hardware unreliability	0.4
procedural faults	0.6
abnormal algorithm deficiencies	0.7

vent a working system configuration from being established. Significant effort was placed into the design of the system hardware components in order to make them intrinsically reliable. However, the functions implemented in some units are so critical that the 1A system, in common with the earlier ESS systems, makes significant use of the full duplication of critical units. In general, full duplication is used for the parts of the system whose failure would affect a substantial number of customers, for example the central control, the bus systems, and the file stores. (It is stated in [BELL77] that reliability calculations have shown that redundancy greater than full duplication is not required.)

Replication of the core memory units is influenced by the ease with which the data contained in the unit can be regenerated. Thus the program stores are supplemented with standby spares, which can be loaded from the file store. The standby spares in the call-store complex are used to provide full duplication for the units containing transient data such as that relating to calls. Figure 12 illustrates the redundancy in the No.1A system.

The third cause of system outage time is that attributed to "procedural faults," which is expected to cause about 0.6 minutes outage per year. In the main, these faults will be caused by the human interface so that particular attention has been paid to the clarity and uniformity of the documentation, and to achieving a reduction in the number of manual operations required.

The largest system outage time in the No.1A system is expected to be caused by "abnormal algorithm deficiencies," i.e. deficiencies in those algorithms in the system that are invoked when an error occurs. As with the earlier systems, the "abnormal algorithms" in the 1A system are mainly implemented at the software level, and the system is dependent on these for its recovery. However, it is recognized that there are a large number of variables involved and that system recovery is related to all other maintenance components (for example, recovery can be easily misled by an incomplete diagnosis). Indeed it is stated in [BELL77] that "there is no guarantee that all impending trouble will be identified and isolated before it can jeopardize system operation."

Error detection——Mechanisms for error detection are employed at both the hardware and software levels in the 1A system. The main hardware error detection mechanisms are essentially the same as those employed in earlier ESS systems, namely:

a) replication checks
b) timing checks
c) coding checks
d) internal checks i.e., self checking units

Replication checks are the primary mechanism for detecting errors caused by hardware faults in the CC. The CC is fully duplicated; in general two identical modules are fully operational (i.e., static redundancy) with each executing the same instruction atomically, with both units in synchronization. Each CC has special (duplicated) circuits which perform consistency (matching) checks between the two CCs. The overall action of the CC can be regarded as a shared atomic action which encloses the atomic actions of each processing module. Of course, this type of replication does not detect errors caused by design faults in either the software or hardware. Replication checks are also provided by the CC on the transmissions on the duplicated buses, and by the duplicated file stores.

Timing checks are used throughout all of the hardware components in the 1A processor. In particular, those on the CC are used to verify that its operations are proceeding in the correct manner. Timing checks are also provided by the CC to the software level (described below).

Coding redundancy is used to protect all of the words in the system. The codes used include M-out-of-N codes, parity, and cyclic codes, depending on the types of error expected.

Many of the hardware components in the 1A system have been designed to be self-checking. Information concerning the detection of internal faults in a unit is made available to the recovery programs (discussed below) so that extensive software checking may not be required to locate faulty units.

The hardware level also provides error detection mechanisms for use by the software level. These mechanisms are run-time interface and timing checks. The interface checks verify that the addresses used by the programs are within proper limits and that the program does not violate the memory protection described above. Units such as the tape and disk controllers also provide interface checks on the operations they are requested to perform. The timing checks are used to ensure that the program control in the system is not lost because of any software (or hardware) fault.

Essentially, there are two error detection mechanisms at the software level in the No.1A system, namely the audit programs and the diagnostic programs. The audit programs have been designed to detect (and correct) errors arising in the system database, and are run at regular intervals during the normal operation of the system, as well as after hardware failures have been detected. The audit programs provide an independent diagnostic check on the "actual" structure of the database, using the redundancy contained within the database. Replication checks can also be performed, comparing data with its replicated copy. Coding checks are also used by the audit programs, for example to check the non-transient data in the program and call stores.

As discussed below, the diagnostic programs play an important role in the treatment of faults in the No.1A system. However, the diagnostics are also used to detect errors in units. Periodically, each unit is removed from service and the diagnostic tests performed to (attempt to) detect latent faults in that unit.

Fault treatment——When an error is detected by the hardware mechanisms described above, the normal call processing operation of the system is interrupted and the fault treatment and recovery programs are invoked. Essentially, there are three priority categories for this interruption:

- immediate interrupt (maintenance interrupt)—if the fault is severe enough to affect the execution of the currently executing program
- interrupt deferred until the completion of the currently executing program—if the problem could affect several telephone calls
- interrupt deferred until detected by the routinely executed base-level jobs—if the problem affects only a single call

The main aim for the fault treatment and recovery programs is to re-establish the call-processing capabilities of the system as quickly as possible. This involves identifying and isolating the faulty hardware unit and (spontaneously) reconfiguring the system to utilize a spare unit. (There are special instructions implemented in the CC to control the configuration circuits in the subsystems.) For faults in the call store complex, where no standby spares are available, a unit operating in duplex mode will be pre-empted to replace a failed unit which was not protected by duplication.

An attempt is made to minimize the effects of the non-deferrable maintenance activities. In particular, a technique called "first look" is generally used. With this technique, the fault recognition program examines the information provided by the error detection hardware to determine the most likely cause of the problem. The system will then be configured to run without the implicated unit(s), and further diagnostics of those units will be scheduled for deferred processing at the base-level, i.e., the diagnostic tests on the faulty unit will be multiprogrammed with the normal operation of the system. For example, if a program store was removed from service, a deferred action would be to run diagnostic checks on that store.

Error records are collected by the fault treatment programs to indicate the units in which errors have been detected and the response of the treatment programs to those errors. If analysis of these records indicates that the system has not been restored to fault-free operation, then at the occurrence of the next error the "first look" strategy is abandoned and a complete check of the implicated units will be performed. If a unit passes all of the diagnostic checks then it is assumed that the fault was transient, and that the unit is now fault-free and can be returned to service. However, analysis of the error records will result in the isolation of units experiencing a high rate of transient faults.

It can be seen that fault location in the system (and the recovery described below) are program controlled and therefore require (and assume the existence of) a fault-free processor. In order to achieve this, "abnormal" algorithms in the hardware level implement a spontaneous reconfiguration mechanism to enable a (hopefully) fault-free processor to be configured. There are four steps that can be taken by this mechanism.

The first step involves the automatic configuration of a basic processor from the components of the system, consisting of a central control, program store and program store bus. The processor is not capable of full system operation, but only of running fault recognition and spontaneous reconfiguration programs in an attempt to resume normal operation of the system. If this step fails (e.g., detected by timing checks), then it is assumed that the program stores have been corrupted, and the second step involves repeating the first step and reloading the program stores from the disks. If step two fails, then step three will configure a basic processor to perform simplified tests, isolating various subsystems until (possibly) fault-free operation is restored, and normal operation of the system can be attempted. If this step fails then the last step is to stop the system and wait for manual recovery actions.

The repair of faulty units in the No.1A processor is performed manually. In order to meet the reliability requirements for the hardware (0.4 minutes outage per year) the average repair time for a unit has to be less than 2 hours. Thus a maintenance objective for the No.1A processor was that at least 90% of faults should be isolated to no more than three replaceable modules by the (automatic) diagnostic programs, and that such programs should detect at least 95% of the possible faults.

It is clear from [BELL77] that very extensive work has been carried out into the design and development of the diagnostic programs. The diagnosticians and hardware designers work as a team from the start of the design through to the completed system. The hardware and diagnostic designs proceed in parallel, and are used to verify each other. Furthermore, the diagnostic tests are used in many applications, from the initial development of the system (tested using simulation), to the testing of the various units while a system is being built and commissioned, and are finally used on-line in the operational systems. Thus a large amount of experience with the diagnostic programs is built up.

The design philosophy for the diagnostic program is essentially that all of the tests on a unit are run, and a post-processing scheme, the trouble location procedure (TLP), examines the results of all of the tests to attempt to determine the problem. Essentially, the TLP performs pattern matching between a condensed version of the results of the diagnostic tests and an office-resident database (held on magnetic tape), to produce an ordered list of suspected faulty modules. The database has been built up (off-line) by circuit analysis and by the simulation of permanent hardware faults using a physical or computer model.

Damage assessment——The main form of damage assessment in the No.1A system appears to be based on the a priori reasonings that either a fault will not result in any damage, or that any damage will manifest itself as damage to the system database. Damage to the database will be dynamically assessed by the audit programs, to invoke various stages of recovery as discussed below.

Clearly some form of dynamic damage assessment will be performed by the environment of the No.1A system. For example, if a call is lost or routed incorrectly, then the customer will assess the call as "damaged," and (hopefully) will retry the call. It is not clear from [BELL77] whether the possibility of the accounting information being damaged is considered, or whether this also relies on damage assessment by the environment.

Error recovery———The main form of error recovery in the 1A system can be viewed as forward error recovery at the software level. Forward error recovery techniques are used to provide recovery from faults in any of the storage modules in the system whose contents have been duplicated. For example, if a program store fails, its contents can be recovered from the copy on the disk.

Recovery from errors in the system's database is also handled by forward error recovery, implemented by compensation algorithms in the audit programs. In general, the algorithms may treat parts of the database as insignificant information, in that the erroneous information that is detected is removed (and discarded) from the database, without necessarily being corrected. This may result, for example, in a (it is hoped, small) number of telephone calls being lost, but enabling the normal operation of the system to be resumed. Clearly the environment of the system will provide the actions required to recover from the system, treating some information as insignificant; for example, customers whose calls were lost will provide a form of backward error recovery, abandoning the current state of that call and attempting to redial. The amount of information that is treated as insignificant will depend on the damage that is assessed. An "optimistic" approach is adopted for this damage assessment and compensation, i.e., those stages least disruptive to the normal operation of the system will be attempted first. If the damage is assessed as minor, then the error recovery will be multiprogrammed with the normal operation of the system. If the damage is more severe (i.e., too extensive to rely on the normal audit to correct the problems), then the call processing is suspended until the appropriate database reinitialization has been performed.

When an error occurs during the processing of the maintenance programs at the base-level of the system, the currently executing program is terminated (to protect the system against further disruption and to guard against invalid results from the program [BELL77]). However, these programs can specify an "abnormal termination algorithm" which is invoked (once) by the "abnormal" algorithms of the software level, if it is assessed that the program caused the error. Thus a program can attempt to provide its own forward error recovery actions. If these fail then that program is terminated, and the normal recovery programs are relied on to resume normal system operation.

Reliability Evaluation

It is stated in [BELL77] that, after extensive laboratory testing of the No.1A processor (100,000 processor hours) and greater than 8000 hours in service, the overall performance indicates that the design objectives of the system are being realized. Some early studies were also conducted to evaluate the reliability strategies employed in the system. In one study, 2071 single determinate hardware faults were inserted at random into a normally operating system. Automatic system recovery occurred in 99.8% of these cases, with manual assistance required for only five of the faults. Another study which analyzed, by simulation, the performance of the diagnostic programs in response to a random selection of 2400 hardware faults indicated that 95% of the faults were detected by the programs. A test of the trouble location procedure was also performed by inserting circuits which were known to be faulty into a system. In only five of the 133 simulated repair cases did the TLP fail to include the inserted circuit on its list of suspected modules, and in 94.7% of the lists the faulty component was located within the first five modules.

PLURIBUS

PLURIBUS [HEAR73] is a packet-switching communications multiprocessor designed as a new form of switching node (IMP) for the ARPA Network. Bolt, Beranek and Newman Inc. started design of the multiprocessor system in 1972 and a prototype 13 processor system was constructed by early 1974.

The machine, besides being capable of a high bandwidth in order to handle the 1.5 megaband data circuits which are planned for the network, is also intended to be highly reliable [ORNS75], operating 24 hours a day all year round. PLURIBUS is designed to recover automatically within seconds of detecting an error and to survive not only transient faults but also the failure of any single component. To achieve this fault tolerance, PLURIBUS uses such techniques as replication, isolation of components (e.g., processors and programs), monitoring, process restart, and reconfiguration. The information presented in this survey is primarily based on [ORNS75], while Figures 14 and 15 are taken from [HEAR73] and [ORNS75] respectively.

System Description

When discussing the system structure of PLURIBUS, particularly with regard to its reliability, it is useful to view the system as consisting of four levels and three interfaces, as indicated by Figure 13.

The bottom level is formed by the hardware (processors, memories, etc.) of PLURIBUS which supports three software levels. The first and second software levels are concerned with the control and reliability of the system, while the third level consists of the application program that performs the packet-switching role of the IMP.

Figure 14 illustrates the hardware level static structure of the PLURIBUS system. The system is designed around three types of bus (processor, memory, input/output)

Levels		*Interfaces*
4	*application program*	
		IMP
3	*second software level*	
		multi-computer
2	*first software level*	
		hardware architecture
1	*hardware*	

FIGURE 13. PLURIBUS level structure.

that are joined together by special bus couplers which allow units on one bus to access those on another. Each bus, together with its own power supply and cooling, is mounted in a separate unit. A processor bus contains two processors each with its own local 4k memory which stores code that is either frequently run or used for recovery purposes. A memory bus contains the segments of a large memory, common to all the processors, while an input/output (I/O) bus houses device controllers as well as central resources such as system clocks. A feature of PLURIBUS is its treatment of hardware units (processors, memory, buses) as set of equivalent resources. There is, for example, no specialization of processors for particular system functions and no assignment of priority among the processors, such as designating one as master.

The hardware level, in addition to providing conventional processing and storage facilities, contains three components: a "pseudo interrupt device" which facilitates processor scheduling, a "60 Hz interrupt" and a "bus timer." The functions of these latter two components relate to the problem of monitoring software activity and assuring its continued progress. The three levels of software seem to be conceptual, based on the assumed correctness of the software design, and to the hardware they appear as a single real-time program. This single program, which includes the application program (i.e. IMP job) as well as the control and reliability programs, is divided into small pieces called *strips,* each of which handles a particular task. Tasks can initiate other tasks but cannot communicate directly with them while running; communication is handled by leaving messages in common memory.

When a task needs to be performed the name of the appropriate strip is placed on a queue of tasks to be run. Each processor, when it is not running a strip, repeatedly checks this queue for work. When a name appears in the queue the next available processor will remove it and execute the corresponding strip. Since all processors frequently access this queue, contention for it is high. For this reason the queue is

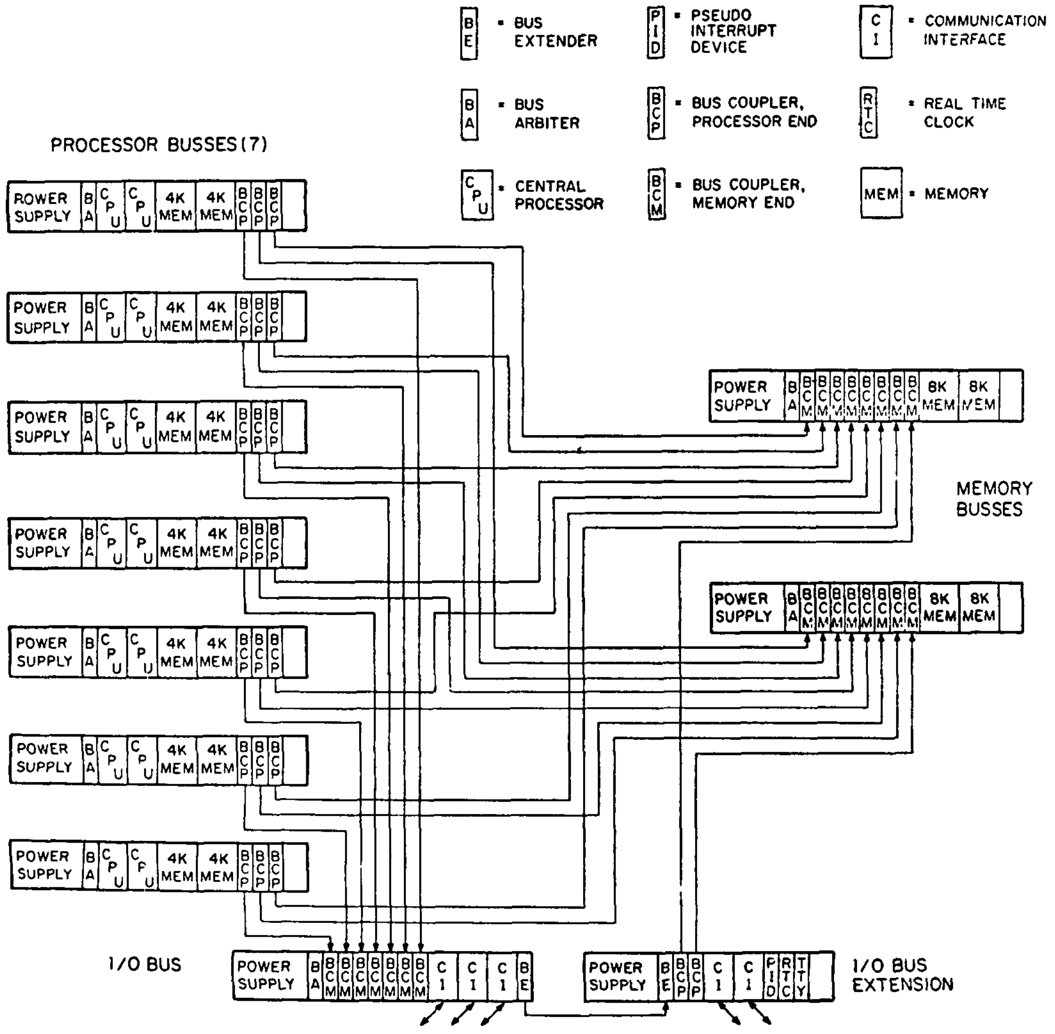

FIGURE 14. PLURIBUS hardware configuration

implemented in hardware by a special pseudo interrupt device (Figure 14). By limiting the maximum execution time of a strip to 400 msecs (the time the most urgent task can afford to wait) and priority-ordering the task queue, a number of scheduling and multiprogramming difficulties, such as saving and restoring the machine state, have been eliminated.

Above the hardware architecture interface, the first software level views the processors and stores as distinct virtual computers for each of which it creates a context, so that the computer can function reliably. To achieve this reliability, two components are used within each computer, called the "code-tester" and the "individual." The code-tester monitors the operation of the computer. It performs such tasks as checking all local memory code and safeguards all control and "lock" mechanisms. With the help of the code-tester, the individual provides an abstraction (multi-computer) interface with the second software level. Each individual finds all memories and processors it considers usable and attempts to form a dialog with the other individuals so they can work together as reliable virtual computers above the multi-computer interface.

The second software level is concerned with the problem of forming the separate virtual computers provided at the multi-computer interface into a single reliable computer system. This task is performed by three components, called the "consensus," the "IMP-system-reliability," and the "IMP-system." The consensus monitors the interactions of the separate virtual computers and provides a single computer system at the communications interface with the IMP-system-reliability. Specific tasks performed by the consensus include checking all common memory code, finding all usable hardware resources (processors, memories, etc.) from information supplied by the individuals, testing each resource and creating a table of operable ones.

The IMP-system-reliability monitors the operation and, in particular, the data structures of the computer system (the IMP), and helps to assure its reliability. It is assisted in this task by the last component at this level, the IMP-system, which monitors the behavior of the ARPA Network and will not allow the IMP to cooperate with seemingly irresponsible network behavior. Above the abstraction (IMP) interface the application program can view the IMP as a reliable, fast sequential computer with a large store.

Reliability Strategies

For the purpose of ensuring fault tolerance the hardware and software components of PLURIBUS are organized into special functional units called *reliability subsystems.* These reliability subsystems are the components (e.g., code-tester, consensus, etc.) outlined above in the discussion of levels and interfaces. The notion of reliability subsystems seems to assume that all faults are masked above the interface and that the subsystem on the other side of the interface has no responsibility for detecting and coping with errors in the lower subsystem.

A reliability subsystem is some grouping of system resources (hardware, software and/or data structures) whose integrity is verified as a unit by a self-contained test mechanism. Moreover, each subsystem contains a reset mechanism which will return it to some specified initial recovery point. [ORNS75] describes the entire PLURIBUS system as being made up of such subsystems, which communicate via data structures, and which appear conceptually to operate asynchronously. Furthermore these subsystems are organized into a "chain" (Figure 15) in which each member monitors the behavior of the next member of the chain, and may externally activate the reset mechanism of that system, if it detects some malfunction.

The monitoring of subsystems is carried out using watchdog timers, which ensure that each subsystem passes through a predefined cycle of activity. This is done by including code in the cycle to restart the timer, so that if the cycle is not executed properly the timer will run out and cause the monitoring subsystem to detect an error. The defined cycle must also contain an execution of a self-test mechanism, so that correct passage through the cycle provides strong evidence of the reliability of the subsystem. Another aspect of the "chain" structure is that subsystems low in the chain attempt to provide and "guarantee" the reliability of some components used by higher subsystems.

Error detection——Strategies used by the reliability subsystems of PLURIBUS for error detection are mainly based on watchdog timers, supported at the hardware level by the 60 Hz interrupt. These facilities are used within level 2 (the first software level) by the code-tester to monitor the operation of each processor and store. First, it sumchecks all low level code (including itself); second, it ensures that all subsystems are receiving a share of the processor's attention; and finally it safeguards that "locks" to critical resources do not hang up. Also at level 2 is the individual that performs error detection at the interface with level 3. An individual running from the local store of each processor performs the task of locating all usable resources. This involves addressing every de-

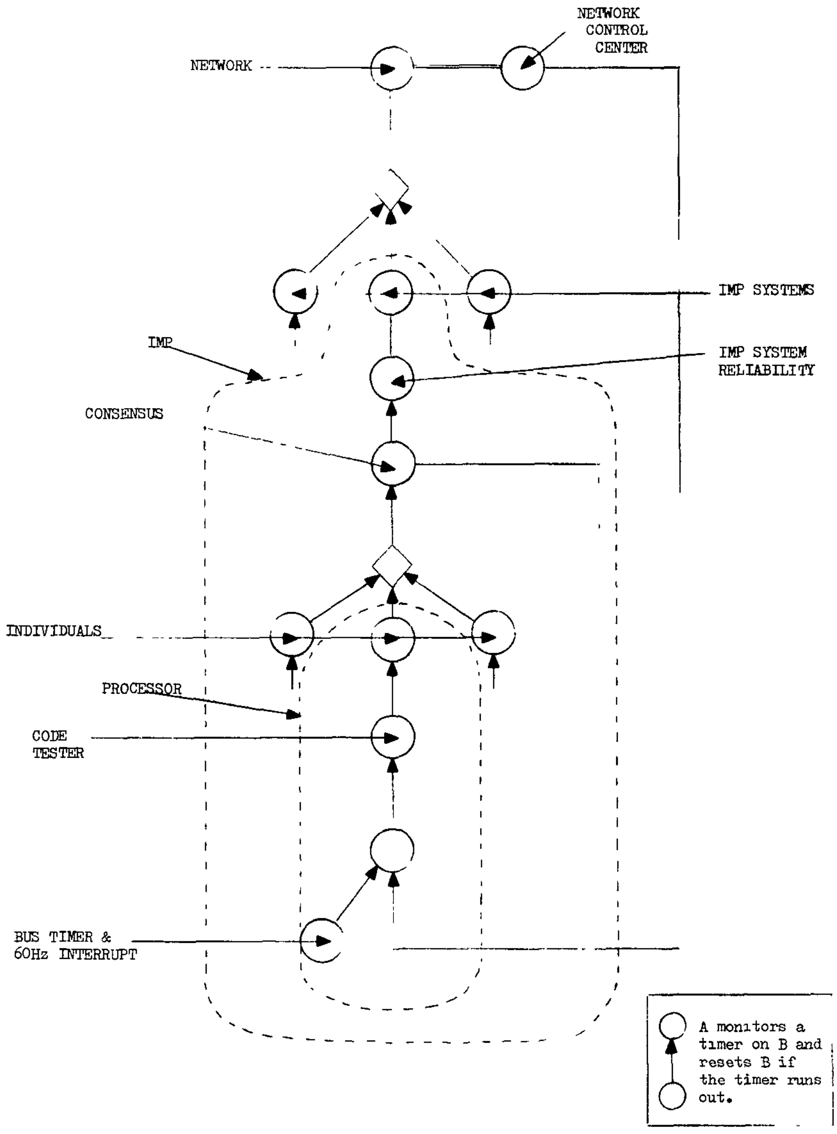

FIGURE 15. PLURIBUS reliability structure.

vice in the system and every page in memory and listing those which are operable.

At level 3, error detection is handled by the consensus, performing such tasks as sumchecking all common code and maintaining a timer for each processor. The consensus will count down these timers in order to detect uncooperative or dead processors. The IMP-system-reliability, also at level 3, detects errors in the IMP-system by monitoring its cycle of behavior and watching its data structures. Finally at level 3 the IMP-system monitors the behavior of the other IMPs in the ARPA Network and will not respond to erroneous operations. In return the ARPA Network, through the Network Control Center, monitors the behavior of the IMP and if it detects a dead IMP trys to restart it remotely.

Fault treatment——Given that an error has been detected in a subsystem of the PLURIBUS, another subsystem will attempt to locate and remove the fault. For example, at the hardware level the bus timer will reset an inactive bus after 1 sec. which clears any hung device commands. To allow for faults in hardware resources, all bus couplers have a program-controllable switch that inhibits transactions via the coupler. The individuals at level 2 use these switches to effectively "amputate" a bus by turning off all couplers from that bus. This mechanism is protected from capricious use by requiring a particular data word (password) to be stored in a control register of the bus coupler before use.

Fault treatment data is built up by each individual (indicating the set of devices considered operable) in regions in common memory. This information is used by the consensus at level 3 to determine the "true" set of usable hardware resources, by a form of replicated voting. The consensus can also run tests on resources to determine the extent of a fault.

Damage assessment——Within PLURIBUS, damage assessment seems to be based entirely on a single strategy—it is assumed that everything that the subsystem has done since starting a task is damaged and hence the task being performed is abandoned. This strategy is adequate in PLURIBUS, as discussed below, because of the fault tolerant environment in which an IMP operates.

Error recovery——When an error is detected in a reliability subsystem, the monitoring subsystem performs backward error recovery by re-initializing the reliability subsystem, and treating any information flow as insignificant. Each processor in the PLURIBUS system, for example, can reset or reload any other processor by means of password protected paths in the hardware. The notion of reliability subsystems is vulnerable to a simultaneous transient fault (e.g., loss of power) of all processors. However the Network Control Center has access to the above recovery facilities and can therefore restart an IMP if necessary.

Reliability Evaluation

Although a number of PLURIBUS systems have been constructed and are in operation, the referenced papers do not include any evaluations of the reliability strategies employed in the system.

SUMMARY

Clearly, all three of the systems we have discussed are notable examples of the use of fault tolerance in order to achieve high levels of system reliability. The success of the ESS No.1A system is particularly impressive, given the complexity of the task which it has to perform—very detailed descriptions and analyses of this system have been published, and this is reflected in the length of our account of the system compared to the others.

The system designs differ greatly, due in no small measure to the differences between the environments for which they were designed. Thus, the JPL-STAR, intended for spacecraft guidance, and hence unable to rely on manual repair, employs spontaneous replacement (though at a level which would perhaps not be appropriate for modern technology). In contrast, the ESS No.1A and PLURIBUS depend on manual repair to achieve their reliability objectives. The designers of the ESS No.1A have put immense effort into the design of diagnostic programs aimed at assisting the activities of maintenance personnel. The intended environments of the PLURIBUS and ESS No.1A are somewhat similar, but the designers of PLURIBUS have chosen to use dynamic rather than static redundancy.

The tasks of the designers of these two systems were obviously considerably aided by experience gained from predecessor systems. This is particularly the case with the ESS No.1A processor, whose design relies heavily on the accuracy and completeness with which faults and their consequences have been predicted, in order to make use of numerous forward error recovery techniques. Equally, both systems also rely on the tolerance exhibited by their environment—callers can be expected to redial in the case of ESS, and the ARPANET protocols have been designed to cope with a failing node processor.

The three systems differ greatly with respect to provisions for coping with software faults. The design of the JPL-STAR makes few such provisions, and issues concerning the reliability of application programs are largely ignored in the papers that we have referenced. The PLURIBUS system is intended to run with a fixed program, which together with the basic software, is expected to be fault free. Software issues figure much larger in the work of the ESS No.1A designers; despite the use of the (well-tested) programs from earlier ESS systems, it has been recognized that the complexity of the programs and the need for their modification will result in "software deficiencies." Thus, reliance is placed mainly on forward error recovery strategies to correct errors introduced by both software and hardware faults. Again, the experience with earlier systems has influenced the design of such strategies.

CONCLUSIONS

This paper has attempted to bring out all of the assumptions (justified or not) which are always present in any discussion of the reliability of a complex computing system, and in system designs aimed at providing very high reliability. It has shown how reliability concepts are inextricably intermingled with system structuring concepts, and how a concern for system structuring can help (and hinder) the rational design of fault-tolerant systems.

The aim has been to deal with all possible types of fault, including design inadequacies (both hardware and software) and valid input data which is belatedly identified as incorrect, as well as more obvious faults such as those due to hardware component wear and ageing.

Various basic concepts, such as those of atomic action, level, recovery line, commitment and compensation have been defined and described. As an aid to further explication of these ideas, and to appreciating their manifestations and significance in actual systems, overall descriptions of a small number of different approaches to the design of highly reliable systems were given in Section 5.

The one major moral to be drawn from this paper is the prime importance of the choice and design of interfaces, both communication and abstraction, and of ensuring that they are matched as closely as possible by the "actual" structure of the system. Each interface that is specified allows the system designers to achieve what Dijkstra [DIJK76] terms "a separation of concerns"—this separation may be beneficial, but can also be misleading. Particular types of interface are to be preferred (when the cost of providing them can be justified). These are ones which enable all, or at any rate a large proportion of, faults to be masked, and those which enable the existence of complete backward error recovery to be assumed, even in situations involving parallelism. Both types simplify the interface specification, and hence the task of the designer who is trying to provide or make use of the interface. Ideally the design task will be so simplified by appropriate choice of interfaces that it can be carried out faultlessly—if there is reason to doubt this (and there nearly always is), and the needs for high reliability justify attempting to achieve design fault tolerance, we would argue that this should be based on backward error recovery (or possibly replication and voting) rather than forward error recovery.

Forward error recovery can be very effective for predictable faults inside a system; the actual choice of interface will again be important because of the effect it has on the ability of the system to detect errors. (For example, results which can have a reversal check applied to them are better in this regard than a single bit result which must either be trusted, or recalculated.) However we would suggest that because of the complexity it engenders, forward error recovery should be used sparingly, and regarded as an optimization of backward error recovery, which in any case might still be needed to deal with unanticipated faults. However, forward error recovery, in the form of compensation, may be necessary (as opposed to a mere optimization) when dealing with environments that cannot be forced to back up, although it is much better to prevent incorrect information flow than to have to compensate for it later.

Ideally all these various design issues would be decided upon, in a particular case, by mainly quantitative methods, based on relative probabilities of faults, the costs and performance characteristics of different strategies, and interfaces, etc. Certainly, conventional reliability engineering calculations can and should be used in those parts of the system design task which are sufficiently well understood and codified, such as the construction of simple subsystems from tried and tested standard hardware components. However it would seem that many of the design tasks involved in achieving high levels of overall reliability from large and complex hardware/software systems will continue for a long time to require large measures of creative skill and experience on the part of the designers.

ACKNOWLEDGMENTS

The ideas presented in this paper, and a number of sizable portions of the text, owe much to the work of other past and present members of the UK Science Research Council-sponsored project, at the University of Newcastle upon Tyne, on the design of highly reliable computing systems. In particular the preparation of the actual paper has been greatly aided in particular by discussions with Tom Anderson, Ellis Cohen and Santosh Shrivastava, and by the referees' comments on an earlier version. Earlier discussions with Jim Gray, of IBM San Jose, and with members of IFIP Working Group 2.3 on Programming Methodology have also had a strong influence on the paper, as has the pioneering work by Davies and Bjork on spheres of control. Nevertheless, the detailed contents and any inadequacies of this attempt at an overall analysis of reliability issues and the role of system structuring in complex computing systems remain the responsibility of the authors.

REFERENCES

[ANDE76] ANDERSON, T.; AND KERR, R. "Recovery blocks in action: a system supporting high reliability," in *Proc. Int Conf. Software Engineering*, 1976.

[ANDE77] ANDERSON, T.; LEE, P. A.; AND SHRIVASTAVA, S. K. *A model of recoverability in multi-level systems*, Tech. Rep 115, Computing Laboratory, Univ. Newcastle upon Tyne, UK, Nov. 1977. To appear in *IEE Trans. Softw Eng*

[AVIZ71] AVIZIENIS, A. et al. "The STAR (self testing and repairing) computer· an investigation of the theory and practice of fault tolerant computer design," *IEEE Trans. Comput.* **C-20,** 11 (Nov. 1971), 1312-1321.

[AVIZ72] AVIZIENIS, A.; AND RENNELS, D. A. "Fault tolerance experiments with the JPL-STAR computer," *IEEE COMPCON 72,* IEEE, New York, 1972, pp. 321-324.

[AVIZ76] AVIZIENIS, A "Fault-tolerant systems." *IEEE Trans. Comput* **C-25,** 12 (Dec. 1976), 1304-1312.

[BASK72] BASKIN, H. B.; BORGERSON, B. R.; AND ROBERTS, R. "PRIME—a modular architecture for terminal-orientated systems," in *Proc. 1972 AFIPS Spring Jt. Computer Conf*, Vol. 40, AFIPS Press, Montvale, N.J., pp. 431-437.

[BELL64] *Bell Syst. Tech. J.*, (Sept. 1964).

[BELL77] *Bell Syst. Tech. J.*, (Feb. 1977).

[BJOR72] BJORK, L. A.; AND DAVIES, C. T. *The semantics of the preservation and recovery of integrity in a data system,* Rep. TR 02.540, IBM, San Jose, Calif., Dec. 1972.

[BJOR74] BJORK, L. A. *Generalised audit trail (ledger) concepts for data base applications,* Rep. TR 02.641, IBM, San Jose, Calif., Sept. 1974

[BORG72] BORGERSON, B. R. "A fail-softly system for timesharing use," *Digest of papers FTC-2,* 1972, pp. 89-93.

[BORG73] BORGERSON, B. R "Spontaneous reconfiguration in a fail-softly computer utility," *Datafair* (1973), 326-331

[CLEM74] CLEMENT, C. F., AND TOYER, R. D. "Recovery from faults in the No. 1A processor," *FTC-4,* 1974, pp 5.2-5 7

[DAVI72] DAVIES, C. T. *A recovery/integrity architecture for a data system,* Rep. TR 02.528, IBM, San Jose, Calif., May 1972.

[DENN76] DENNING, P. J. "Fault-tolerant operating systems," *Comput. Surv.* **8,** 4 (Dec. 1976), 359-389.

[DIJK68] DIJKSTRA, E. W. "The structure of the 'THE'-multiprogramming system," *Commun. ACM* **11,** 5 (May 1968), 341-346.

[DIJK76] DIJKSTRA, E. W *A discipline of programming,* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1976.

[EDEL74] EDELBERG, M. "Data base contamination and recovery," in *Proc. ACM SIGMOD Workshop Data Description, Access and Control,* 1974, ACM, New York, pp. 419-430.

[ELSP72] ELSPAS, B.; LEVITT, K. N ; WALDINGER, R J.; AND WAKSMAN, A "An assessment of techniques for proving program correctness," *Comput. Surv.* **4,** 2 (June 1972), 97-147

[ESWA76] ESWARAN, K P.; GRAY, J. N , LORIE, R. A , AND TRAIGER, I. L. "The notions of consistency and predicate locks in a database system," *Commun. ACM* **19,** 11 (Nov. 1976), 624-633.

[FABR73] FABRY, R S "Dynamic verification of operating system decisions," *Commun. ACM* **16,** 11 (Nov 1973), 659-668.

[FOSD76] FOSDICK, L. D., AND OSTERWEIL, L. J "Data flow analysis in software reliability," *Comput. Surv.* **8,** 3 (Sept. 1976), 305-330.

[GOOD75] GOODENOUGH, J. B. "Exception handling· issues and a proposed notation," *Commun. ACM* **18,** 12 (Dec. 1975), 683-696.

[GRAY75] GRAY, J. N.; LORIE, R. A.; PUTZOLU, G. R.; AND TRAIGER, L. L. *Granularity of locks and degrees of consistency in a shared database,* IBM Research Rep. RJ1654, Sept. 1975

[GRAY77] GRAY, J. N. Private communication.

[HANT76] HANTLER, S. L., AND KING, J. C. "An introduction to proving the correctness of programs," *Comput. Surv.* **8,** 3 (Sept. 1976), 331-353.

[HEAR73] HEART, F. E.; ORNSTEIN, S. M., CROWTHER, W. R; AND BARKER, W. B "A new minicomputer/multiprocessor for the ARPA network," in *Proc. 1973 AFIPS Natl. Computer Conf,* Vol 42, AFIPS Press, Montvale, N.J., pp. 529-537

[HORN74] HORNING, J.; LAUER, H C.; MELLIAR-SMITH, P. M.; AND RANDELL, B. "A program structure for error detection and recovery," in *Proc. Conf. Operating Systems; Theoretical and Practical Aspects.* IRIA, 1974, pp. 177-193. (Reprinted in *Lecture notes in computer science,* Vol. 16, Springer-Verlag, New York.)

[LAMP76] LAMPSON, B.; AND STURGIS, H. *Crash recovery in a distributed data storage system,* Computer Science Laboratory, Xerox Palo Alto Research Center, Palo Alto, Calif., 1976.

[LEVI77] LEVIN, R. "Program structures for exception condition handling," PhD Thesis, Carnegie-Mellon Univ., Pittsburgh, Pa , June 1977.

[LOME77] LOMET, D. B. "Process structuring, synchronisation and recovery using atomic actions," in *Proc. ACM Conf. Language Design for Reliable Software, SIGPLAN Notices* **12,** 3 (March 1977), 128-137.

[LOND75] LONDON, R. L. "A view of program verification," in *Proc. Int. Conf. Reliable Software,* 1975, ACM, New York, pp. 534-545.

[MCPH74] MCPHEE, W S "Operating system integrity in OS/VS2," *IBM Syst. J.* **13,** 3 (1974), 230-252.

[MELL77] MELLIAR-SMITH, P. M.; AND RANDELL, B. "Software reliability: the role of programmed exception handling," in *Proc. ACM Conf. Language Design for Reliable Software, SIGPLAN Notices* **12,** 3 (March 1977), 95-100.

[MERL77] MERLIN, P. M.; AND RANDELL, B. *Consistent state restoration in distributed systems,* Tech. Rep. 113, Computing Laboratory, Univ. Newcastle upon Tyne, UK, Oct. 1977.

[NAUR77] NAUR, P. "Software reliability," in *Infotech State of the Art Conf. Reliable Software,* 1977, Infotech International Ltd., Maidenhead, UK, pp. 7-13.

[NEUM73] NEUMANN, P. G.; GOLDBERG, J.; LEVITT, K. N.; AND WENSLEY, J. H. *A study of fault-tolerant computing,* Stanford Research Inst., Menlo Park, Calif., July 1973.

[ORNS75] ORNSTEIN, S. M.; CROWTHER, W. R.; KRALEY, M. F.; BRESSLER, R D.; MICHAEL, A., AND HEART, F. E. "Pluribus—a reliable multi-processor," in *Proc 1975 AFIPS Natl. Computer Conf.,* Vol. 44, AFIPS Press, Montvale, N.J., pp. 551-559.

[PARN76] PARNAS, D. L.; AND WURGES, H. "Response to undesired events in software systems," in *Proc. Int. Conf. Software Engineering,* 1976, pp. 437-446.

[RAND75] RANDELL, B. "System structure for software fault tolerance," *IEEE Trans. Softw Eng.* **SE-1,** 2 (June 1975), 220-232.

[RAND78] RANDELL, B., LEE, P. A.; AND TRELEAVEN, P. C "Reliable computing systems," to appear in *Lecture notes in computer science,* Springer-Verlag, New York.

[ROHR73] ROHR, J. A. "Starex self-repair routines software recovery in the JPL-STAR computer", *Digest of papers FTC-3,* 1973, pp. 11-16.

[RUSS76] RUSSEL, D. L. *State restoration amongst communicating processes,* Tech. Rep. 112, Digital Systems Laboratory, Stanford Univ., Stanford, Calif., June 1976

[SHOO68] SHOOMAN, M. L. *Probabilistic reliability; an engineering approach,* McGraw-Hill, Inc , New York, 1968

[SHRI78] SHRIVASTAVA, S. K.; AND BANATRE, J-P. "Reliable resource allocation between unreliable processes," to appear in *IEEE Trans. Softw Eng.* (Also published as Tech. Rep. 99, Computing Laboratory, Univ Newcastle upon Tyne, UK, June 1977.

[VERH77a] VERHOFSTAD, J. S. M. "Recovery and crash resistance in a filing system," in *Proc. SIGMOD Conf.,* 1977, ACM, New York, pp. 158-167.

[VERH77b] VERHOFSTAD J. S. M. "The construction of recoverable multi-level systems," PhD Thesis, Univ. Newcastle upon Tyne, UK, Aug. 1977

[WASS76] WASSERMAN, A. I. *Procedure-oriented exception handling,* Medical Information Science, Univ. California, San Francisco, Calif., 1976.

[WENS72] WENSLEY, J. H. "SIFT—software implemented fault tolerance," in *Proc. 1972 AFIPS Fall Jt. Computer Conf.,* Vol. 41, Part I, AFIPS Press, Montvale, N.J., pp. 243-253.

[WULF75] WULF, W. A. "Reliable hardware-software architecture." in *Proc. Int. Conf. Reliable Software; SIGPLAN Notices* **10,** 6 (June 1975), 122-130.

RECEIVED SEPTEMBER 13, 1977; FINAL REVISION ACCEPTED FEBRUARY 27, 1978.