# Error Recovery in Asynchronous Systems

ROY H. CAMPBELL, MEMBER, IEEE, AND BRIAN RANDELL

*Abstract*—**The demand for highly reliable computer systems has led to techniques for the construction of fault-tolerant software systems. A fault-tolerant system detects errors created as the effects of a fault and applies error recovery provisions in the form of abnormal or exceptional mechanisms and algorithms to continue operation and restore normal computation. Backward error recovery is intended to restore a system state which occurred prior to the manifestation of the fault. Forward error recovery is intended to correct or isolate specific errors and is accomplished in the system state containing the errors.**

**The organization and control of error recovery in asynchronous systems is very complex. Nevertheless, it is possible to limit this complexity by appropriate system structuring aids. Techniques for structuring backward error recovery are comparatively well understood. This paper proposes techniques for structuring forward error recovery measures in asynchronous systems and generalizes recent ideas of atomic actions (transactions) so as to support fault-tolerant interactions between processes.**

*Index Terms*—**Asynchronous systems, atomic actions, error recovery, exception mechanism, programming techniques, software fault tolerance, software reliability.**

## I. Introduction

THE demand for reliable computer systems has led to techniques for the construction of fault-tolerant software systems [6], [11]. These techniques are intended to ensure that a system fulfills the purpose for which it was constructed despite software faults, hardware faults, and invalid invocations of its functions. Networks of computers, distributed resources, and multiple processors introduce new problems of constructing reliable systems and involve the complex organization and control of error recovery in asynchronous systems [8], [13], [15], [20]. This paper introduces general principles and a framework for the design of reliable asynchronous systems based on fault tolerance incorporating forward and backward error recovery.

### A. Fault Tolerance and Error Recovery

A fault-tolerant system is one that is designed to function reliably despite the effects of faults (component or design faults) during normal processing. Such a system detects errors produced by faults and applies error recovery techniques in the form of exceptional mechanisms and abnormal algorithms to continue operation and resume normal computation. However, error propagation may hamper error recovery; the continued operation of a system containing an error can result in the introduction and spread of further errors. Successful fault tolerance must enable the system to function despite error propagation during the time interval, which may be lengthy, between the first manifestation of a fault and the eventual detection of an error.

So-called "forward error recovery" aims to remove or isolate specific errors so that normal computation can be resumed [17]. It is accomplished by making selective corrections to a system state containing errors. Because recovery is applied to a system state containing errors, forward error recovery techniques require accurate damage assessment (or estimation) [1] of the likely extent of the errors introduced by the fault.

In contrast, "backward error recovery" aims to restore the system to a state which occurred prior to the manifestation of the fault. Using this earlier state of the computation, the function of the system is then provided by an alternate algorithm until normal computation can be resumed [11].[1] Because backward error recovery restores a valid prior system state, recovery is possible from errors of largely unknown origin and propagation characteristics.[2] Backward error recovery may involve considerable time overhead and could require extensive testing of potentially acceptable system states.

Forward and backward error recovery techniques complement one another, forward error recovery allowing efficient handling of expected conditions and backward error recovery providing a general strategy which will cope with faults a designer did not—or chose not to—anticipate. As a special case, a forward error recovery mechanism can support the implementation of backward error recovery [7] by transforming unexpected errors into default error conditions.

### B. Asynchronous Systems

We assume that all the activities of a computer are composed of the activities of a set of primitive operations

Manuscript received September 29, 1983; revised December 28, 1984. This work was supported by NASA under Grant NSG 1471. R. Campbell was supported by the Science and Engineering Research Council of the United Kingdom with a Senior Visiting Fellowship while at the University of Newcastle upon Tyne.

R. H. Campbell is with the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801.

B. Randell is with the Computer Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne NE1 7RU, England.

IEEE Log Number 8609343.

[1] In practice, the most recent restorable system state which is free from the effects of the fault may be difficult to determine. In order to find an appropriate system state, a search technique may be used involving iteratively attempting recovery from successively earlier restorable states until recovery is successful.

[2] All that is required is that the errors have not affected the state restoration mechanism.

("atomic actions"), each of which has the property of indivisibly advancing the state of a computation. Likewise, we can also consider the activities of systems that are more abstract than a computer (for example, the execution of a system of software components) as being formed from a basic set of primitive atomic actions that have the property of indivisibly advancing the state of that system. An *asynchronous system* is one that is designed so that its activities may consist of two or more independent and simultaneously active primitive atomic actions. Of course, abstract asynchronous systems of software could be executed by a computer that is sequential in operation.

In practice, each primitive atomic action is part of a sequence of actions called a *process* which advances a particular computation (or operation on a system) from an initial state, through a set of successive intermediate states, perhaps to a final state. If atomic actions from different processes may be interleaved or active simultaneously, then the system is often described as having *concurrent processes.* Two or more processes *interact* if they include primitive atomic actions which, reciprocally,[3] modify each other's intermediate states. (Such atomic actions are shared in the sense that they advance the computation of more than one process.)

For fault tolerance to be effective, asynchronous systems require the coordination and synchronization of normal activity with any activity supporting fault tolerance. The errors generated by a fault may propagate from one process to another by interactions or interprocess communication. Moreover, faults may manifest themselves in several processes if the fault is a malfunction of a common element in their respective processors. Control of fault-tolerant mechanisms may be defined by a centralized component of the system or by the system's distributed components. The pattern of interprocess communication may permit one group of processes to recover from a particular fault while other system processes continue to perform their normal activities.

Corresponding to a spectrum of constraints that can be imposed upon interprocess communication, there is a spectrum of error recovery techniques for asynchronous systems. For example, conversations [11] so synchronize a preidentified group of interacting processes that these processes can perform error detection and error recovery before they communicate to other processes not in the group. The restriction on communication prevents possible error propagation to other processes during the conversation and simplifies state restoration.

Transactions constructed from the interactions of processes using a programmed two-phase commit protocol [9] are coordinated so as either to produce a result agreeable to all the constituent processes or to restore all information changed by the transaction to its prior state. Such transactions can have a varying number of constituent processes providing that they all obey the protocol.

If no synchronization is imposed on normal activity, processes may detect errors and attempt to perform error recovery independently of other processes. However, such processes require more complex coordination schemes for fault-tolerant provisions, as in the chase protocol [18], [26].

The chase protocol starts with a process that has detected an error. Backward error recovery will be used to restore a prior state of the process. Recovery will involve discarding the results of the recent activity of the process, including any communications. The process sends an error message to every process involved in these communications. These processes may have received erroneous data or their activity may become invalid as a consequence of the backward error recovery. Recursively, backward error recovery will be used to restore a prior state of each of these processes and may result in further error messages. Although efficient backward error recovery requires the selection of the most recent appropriate prior state for a process, subsequent error messages may require an earlier state to be chosen. This can occur because of the domino effect [20] and the concurrent detection of errors. The protocol continues until each process can select a backward error recovery that is consistent with the backward error recovery of all the other processes involved in the protocol. Within the restrictions imposed by the protocol, each process may independently and asynchronously proceed with recovery. Algorithms to implement the protocol are described in detail by Wood [26].

The construction of systems with activities that are formed from hierarchies of atomic actions provides a structure for fault tolerance in asynchronous systems [1]. Within the hierarchy, the activities of a group of components are coordinated to have the properties of an atomic action using more primitive atomic actions (these properties are described in Section III-A). For example, the components of a critical section may be coordinated to update a set of variables indivisibly by the invocation of appropriate operations on semaphores. There are two reasons why such a hierarchy is a convenient structure. If a fault, resulting error propagation, and subsequent successful error recovery all occur within a single atomic action they will not affect other system activities. Furthermore, if the activity of a system can be decomposed into atomic actions, fault tolerance measures can be constructed for each of the atomic actions independently. Thus, atomic actions provide a framework for encapsulating fault tolerance techniques within modular components.

Although atomic actions have been defined many times in different ways (for example, [8], [14], [15]) we will use the following definition [1]:

> "The activity of a group of components constitutes an *atomic action* if there are no interactions be-

[3]Note that if one process unilaterally—and not reciprocally—modifies the state of another, then the validity of the computation performed by the first process is independent of the computation performed by the second process. However, the computation of the second process does depend upon that of the first process.

tween that group and the rest of the system for the duration of the activity."

This definition precludes communication between those processes that are in the atomic action with those that are not in the action. A more formal definition and analysis using occurrence graphs formed from events and relations of causality can be found in [3]. Atomic actions are characterized by the set of events they generate. This set has the property that if any two events within that set are connected by a causal chain of events, all the events in that chain must also reside in the set. An interaction between two activities called *A* and *B* would correspond, in the event model, to a causal chain of events between two different events generated by *A* which passes through at least one event generated by *B*. For example, an atomic action may asynchronously input values or may asynchronously output values. However, if a message and acknowledgment is passed between two activities, then that exchange is an interaction and the activities do not constitute two separate atomic actions. (Notice that both definitions are more primitive and less constraining than a definition with the property "all or nothing" [14].)

A *system* has been defined as a set of components which interact under the control of a design [16]. Systems that are designed explicitly so as to synchronize the activities of their components in order to form atomic actions have *planned atomic actions*. The design also determines the way in which the components interact with the environment of the system [1]. The environment of a system is another system which provides input to and receives output from the first system. Such an exchange of information is an *operation*. The activities concerned in an atomic action may be internal to the system or may be operations.

If all the operations on a system involve only planned atomic actions, then that system is an *atomic system* (an exchange of information is not necessarily an atomic action). Such systems may be used as components in the design and construction of other, more complex, systems as if their activities were primitive atomic actions. Systems may also contain *spontaneous atomic actions* that arise fortuitously from the dynamic sequences of events occurring in a system. For the purposes of structuring fault tolerance measures, spontaneous atomic actions are of little value even if they can be easily identified as such.

Planned and spontaneous atomic actions represent the two opposite ends of a spectrum of error recovery techniques and depend upon the extent to which explicit constraints are imposed upon interprocess communication. The conversation is an example of a planned atomic action with which backward error recovery is associated. The chase protocol associates backward error recovery with a more spontaneous form of atomic action dynamically determined by the protocol from past patterns of interprocess communication and available fault-tolerant provisions. Other error recovery techniques based on atomic actions that are more spontaneous than those of the conversation but less spontaneous than those of the chase protocol exist. For example, the two-phase commit protocol explicitly coordinates processes entering and leaving a transaction but does not specify which processes are involved.

In the present paper, we introduce principles, structure, and a framework for synchronizing and coordinating forward and backward error recovery in asynchronous systems based on atomic actions. We adopt the definitions of error, fault, and failure introduced by [17] and improved by [1]. A fault-tolerant system includes four consistent activities identified as:

1) error detection;
2) damage confinement;
3) error recovery;
4) fault treatment and continued service.

A fault-tolerant scheme must support all four activities. We first review error recovery in a single process system. Next, we propose a general error recovery scheme for asynchronous systems. Finally, we introduce specific implementation techniques for fault-tolerance in systems.

## II. Error Recovery in Single Process Systems

A framework for fault-tolerance can be provided by the notions of exception, exception condition, exception handler, and forward error recovery [1], [14], [7]. Anderson and Lee provide the diagram in Fig. 1 to illustrate the framework. A component, pursuing its normal activities, receives a request for a service from another component, performs the service, and returns an appropriate response. The request may be parameterized. The component may service a request by invoking the services of other components.

If a service provided by the component is invoked with an invalid set of parameters, the component may return an abnormal result or *interface exception*. Similarly, if a component fails because it cannot tolerate a fault that it has detected, it may return a *failure exception*. Components that explicitly return abnormal results are said to *signal an exception* to the requesting component. (The exception may have parameters.)

If a component either receives an abnormal response from an invocation of another component or detects an error or abnormal condition during normal activity, it should *raise an exception* and invoke appropriate fault tolerance measures. Recovery is an abnormal activity of the component and is continued until the component either returns to its normal activities or signals an exception. The relationship between the normal and abnormal activity of a component and the raising and signaling of exceptions is shown in Fig. 1. Note that an exception is raised within the component, but signaled between components.

The flow of control of a computation within a component should change as the result of a raised exception. Such a modified or *exceptional flow of control* is distinguished from the normal flow of control. Within a pro-

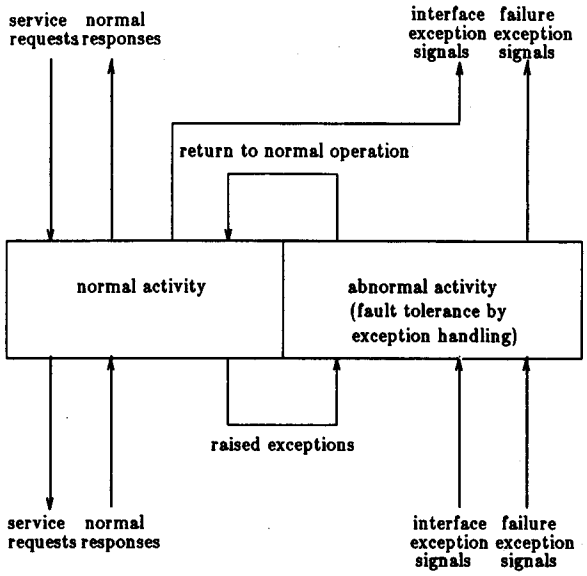



Fig. 1. Framework for an ideal fault-tolerant component.

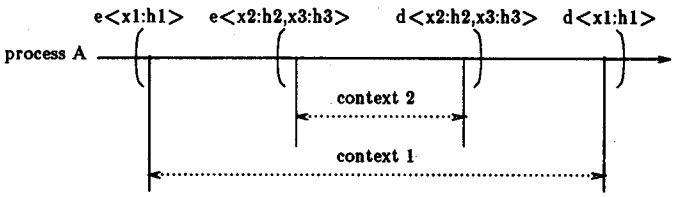



Fig. 2. Example of contexts, exceptions, and handlers.

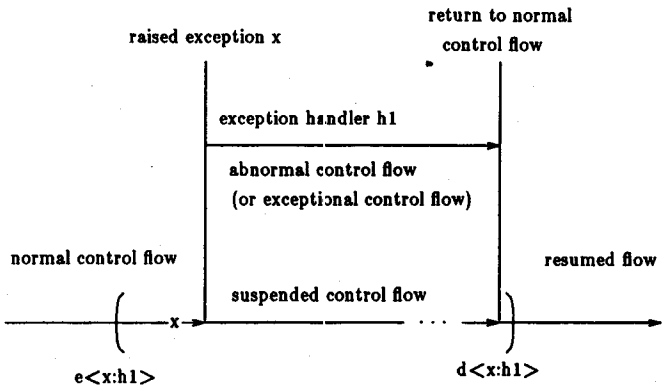



Fig. 3. Example of successful forward error recovery.

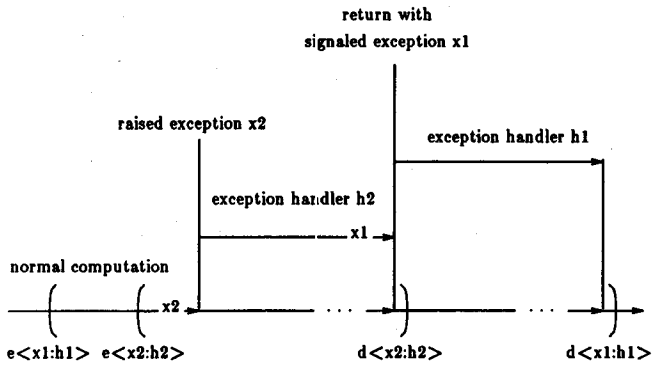



Fig. 4. Example of returning an abnormal response.

gram, exceptional flow of control is associated with code fragments that are called *exception handlers*. The exception handlers may examine any parameters associated with the exception and provide measures to deal with the exception. Exceptions, software components, and exception handlers are associated by a *handling context*. The *enable* operation creates a handling context and associates it with the current flow of control. The *disable* operation terminates the context. An example of a context nested within another context is shown in Fig. 2. The parentheses "("and")" represent the enable and disable operations, respectively. The notation

$$\text{"}e{<}x1{:}h1, \cdots, xi{:}hi{>}\text{"} \quad \text{and}$$

$$\text{"}d{<}x1{:}h1, \cdots, xi{:}hi{>}\text{"}$$

represents enable and disable operations with the exceptions $x1, \cdots, xi$ and corresponding handlers $h1, \cdots, hi$.

The measures provided by the exception handler are intended to deal with an exception occurring during the execution of the software component with which it is associated by context. The context may be determined dynamically by the control flow of the program (as in PL/1), by the data flow (as in Id), or statically by scope (as in Ada®). Many exception mechanisms use a stack to save contexts. This stack is often coupled with the procedure call mechanism. Careful structuring of the manner in which contexts, components, and exception handlers interact can simplify the provision of fault tolerance.

If the fault tolerance measures are successful, a handler may provide a normal control flow return from the component which raised the exception to the component which invoked that component. Fig. 3 shows an example of successful forward error recovery in which the relationship between control flow, context, and exception are illustrated.

If the fault tolerance measures are unsuccessful or inadequate, a handler should signal a failure exception. Abnormal control flow continues in an exception handler of the invoking component. To prevent cyclic and possibly nonterminating patterns of fault and recovery behavior when fault tolerance cannot be achieved, no means is provided whereby a component which receives a signaled exception as a result of an invocation can resume activity [14]. Fig. 4 shows an example of returning an abnormal response in which exception handler $h2$ signals exception failure $x1$. The component which invoked the failed activity raises exception $x1$ in response to the signal and invokes handler $h1$.

An exception handler is a component and may have its own context, exceptions, and exception handlers. This permits the nesting of exception handling facilities.

If an exception is raised within a component (or an exception handler) that does not have a context defining an

®Ada is a registered trademark of the U.S. Department of Defense (Ada Joint Program Office).

```
ensure acceptance_test
by primary_block
else by alternate_block
else by error;
```

Fig. 5. Example of recovery block.

appropriate handler, the component fails and a failure exception is signaled.

### A. Exception Mechanisms

Exception mechanisms implement the change in flow of control (or flow of data) implied by the signaling or raising of an exception. Many explicit tests and branches in a software component may be avoided if the exception mechanism is integrated with the interpreter that implements the activity of the component. (For example, the mechanism is often integrated with the operating system or programming language processor.) The mechanism may detect a standard set of implicit exceptions (for example, address out of range, divide by zero, invalid operation code) in addition to those raised explicitly by the component.

### B. Implementing Backward Error Recovery

The framework provided by "exceptions" can be used to implement the recovery block scheme proposed by [11]. (See also [7].) As illustrated in Fig. 5, a recovery block consists of a primary algorithm, one or more alternates, and an acceptance test. On invocation of a recovery block, the primary algorithm is performed and its results are validated by the acceptance test. If, for any reason, the algorithm fails to complete or to satisfy the acceptance test, restoration of a prior system state removes the effects of the algorithm and an alternate is attempted. Each alternate is tried in turn until either a satisfactory evaluation of the acceptance test permits a normal return or the lack of any further alternates requires the signaling of a failure exception.

Fig. 6 shows how the recovery block is implemented by a context that includes the primary block, a set of exceptions, and an exception handler. Recursively, the exception handler may have a context, a set of exceptions, and an exception handler. Each recursion implements a particular alternate block. The primary block activates a recovery cache for the preservation of the initial state, executes the primary algorithm, and then applies the acceptance test. If errors are detected, an exception is raised and the exception handler of the primary algorithm is invoked. The exception handler restores the initial state using the cache, attempts an alternate algorithm, and applies the acceptance test. If the acceptance test indicates the presence of errors, the exception handler raises an exception and thus activates its own exception handler. The most deeply nested exception handler signals a failure exception. If any application of the acceptance test indicates a satisfactory result, a normal return is made from the primary or alternate block (exception handler) and hence

```
(* ensure correct operation *)
system_component primary_block:
    { initialise_cache
    (* start primary *)
    enable(other_exceptions, alternate_1)
    do_primary_algorithm
    if not (acceptance_test) then signal(alternate_exception)
    disable(other_exceptions, alternate_1)
    discard_cache
    return }
(* end primary *)


exception_handler alternate_block_1:
    { restore_cache
    enable(other_exceptions, alternate_2)
    do_alternate_algorithm
    if not (acceptance_test) then signal(alternate_exception)
    disable(other_exceptions, alternate_2)
    discard_cache
    return }
(* end alternate *)


exception_handler alternate_block_2:
    { restore_cache
    signal(failure_exception) }
(* end alternate *)
```

Fig. 6. Equivalent recovery implemented using exception handlers.

```
exception_handler engine_failure:
    { avoid_stall                          (* damage confinement                          *)
    lower_flaps                            (* damage confinement- slower descent          *)
    select_emergency_landing_site          (* damage confinement                          *)
    switch_fuel_tanks                      (* in case of blocked fuel-lines/empty tank    *)
    switch_magnetos                        (* in case of ignition system fault            *)
    open_de-icers                          (* in case of fault in iced throttle           *)
    ...
    } (*end handler*)
```

Fig. 7. Emergency procedure for light aircraft.

from the recovery block. This is, in effect, a stylized use of the exception framework to provide backward error recovery. Unexpected exceptions are transformed into the default "other_exceptions" and errors are removed by restoring a prior state using the cache.

However, in general, exceptions are used to support forward error recovery schemes which assume detailed knowledge of the erroneous state and attempt to isolate errors. For example, the forward error recovery scheme shown in Fig. 7 is implemented using the exception framework and is taken from recommended takeoff emergency procedures for light aircraft. The forward error recovery strategy attempts to land the aircraft safely and thus confine any damage to the engine. Various recovery strategies are attempted to clear possible faults within the aircraft engine.

## III. Forward Error Recovery in Asynchronous Systems

The exception handling described in the previous section of this paper provides a framework for the implementation of fault tolerance in systems with only a single sequential process. Fault tolerance provisions for systems of concurrent processes are complicated by the possibility of communication of erroneous information and the need to coordinate processes engaged in recovery. Generalizing the exception handling framework to support fault tolerance in asynchronous systems requires additional system structure concerning the cooperation and coordination of the individual processes.

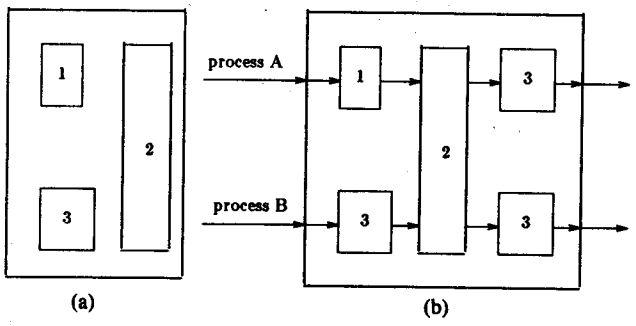



Fig. 8. An example of a system, its components, and processes. (a) Structure of a system with planned atomic action components 1, 2, and 3. (b) The invocation of atomic actions within a system by two processes *A* and *B* showing their control flows and entry and exit points.

### A. Structuring Systems of Concurrent Processes

The construction of systems out of components whose activities form atomic actions provides a structure for fault tolerance in asynchronous systems. A system of concurrent processes contains many separate flows of control. Each flow of control represents the sequential activity of one of the processes of the system. Atomic actions, however, involve concurrent activities in which processes communicate in order to cooperate or to coordinate their use of shared resources. The flow of control of a process joins or leaves an atomic action at any *entry* or *exit point* of a component, respectively. The system shown in Fig. 8(a) contains three components each of which is designed so that its activity constitutes a planned atomic action. Fig. 8(b) shows the control flows of two processes that participate in the planned atomic actions. Synchronization is associated with the process entry and exit points of each component in order to ensure that an atomic action occurs.

### B. Fault Tolerance Structuring Principles

If successful, any fault tolerance measures employed within an atomic action are invisible to the rest of the system. This provides a framework for encapsulating such measures into modular components.

The notion of reliability requires that a system have a specification against which the actual results of invoking its operations can be assessed. When an atomic action is executed, a well-defined state exists at the beginning and termination of its activity (although these states may not necessarily be instantaneously observable). The intended relationship between these states constitutes a specification for the atomic action which is independent of any asynchronous activity inside or outside the atomic action.

The reliability of an atomic action depends upon the reliability of each of its components. (Atomic actions which do not contain measures for handling possible faults have been described as being "out of control" [4].) An initial and final state can be associated with the flow of control of each process joining and leaving the atomic action. Pre- and postconditions associated with such initial and final states can specify the results of the activity of each process. These pre- and postconditions constitute a decomposition of the specification of the atomic action. The specifications and the encapsulation associated with an atomic action provide a context for the application of error detection and damage assessment techniques. Because atomic actions delimit any error propagation caused by interprocess communication, they also support error confinement.

We propose the following two principles for structuring fault tolerance within asynchronous systems:

1) The operations provided by a fault-tolerant asynchronous system should be implemented by atomic actions.

2) Each fault tolerance measure should be associated with a particular atomic action and should involve all of the processes that share (enter and leave) the action.

A fault-tolerant system is reliable, even though it may suffer from internal faults and contain internal errors, as long as its operations provide services which are in accordance with the system specification. Any fault tolerance measures that the system invokes as a result of detecting such errors should be invisible when that system is used as a component of another system. Hence, system services must be atomic actions. Although this principle appears to restrict the applications for which our techniques are appropriate, in fact this is not the case. Computer hardware and software are often merely components in much larger systems which can be regarded as fault-tolerant, perhaps partly or wholly through the efforts of people and the safety devices of other equipment. Of course, error recovery in such systems must be coordinated between components having very different characteristics.

### C. Exception Handling in Atomic Actions

If a component of an atomic action raises an exception, it indicates the detection of an abnormal condition, or error. The error may have been produced as a result of the activity of this component and/or one (or more) of the other components of the atomic action. Alternatively, the original fault may have occurred prior to the atomic action. The raising of an exception within a fault-tolerant atomic action requires the application of abnormal computation and mechanisms to implement the fault tolerance measures. If the recovery measures succeed, the atomic action should produce the results that are normally expected from its activation. Atomic actions that explicitly return an abnormal result have components that cooperatively signal an exception.

An atomic action may contain internal atomic actions. If an exception is raised within an internal atomic action, then the fault tolerance measures of that internal atomic action should be applied. However, an internal atomic action may signal an exception. This exception is raised in the containing atomic action. An *atomic action failure exception* signifies the failure of one or more of the components of an internal atomic action. In particular, a failure exception should be used to indicate that an internal atomic action did not have an appropriate exception han-

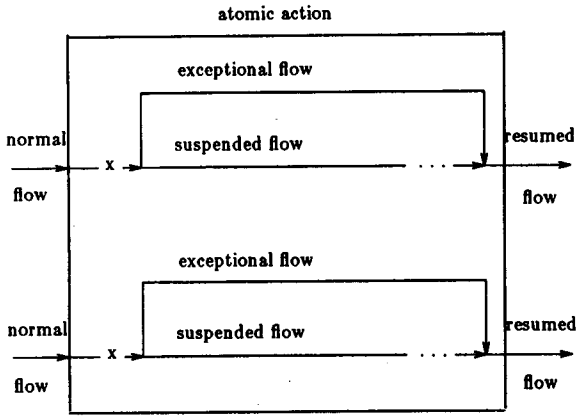



Fig. 9. An example of successful error recovery in an atomic action.

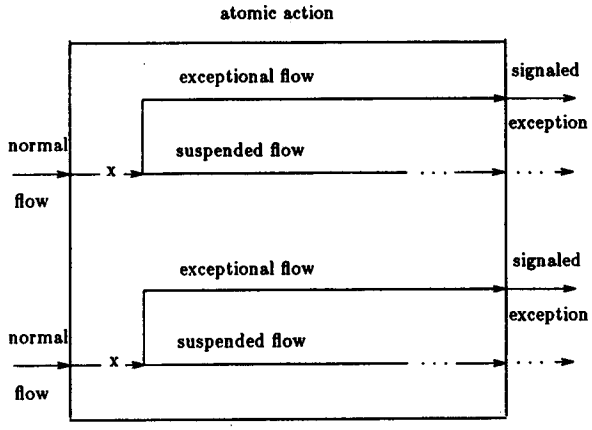



Fig. 10. An example of returning an abnormal response or failure from an atomic action.

dler for an exception that was raised by one of its components.

We propose the following exception handling scheme for atomic actions:

Whether one or several components of the atomic action raise an exception, *the fault tolerance measures necessarily involve all of the processes of that atomic action.* (The fact that an exception has been detected elsewhere among the processes in an atomic action invalidates the assumptions that any of the processes can terminate normally and provide the appropriate results. If some of the processes are not required to change their flow of control to execute fault tolerance measures, they do not interact with the other processes and hence should participate in a separate atomic action.) Examples of an atomic action in which a component raises an exception and each process of the atomic action changes its flow of control are shown in Figs. 9 and 10.

*Every component of the atomic action responds to the raised exception by changing to an abnormal activity.* Each process whose normal control flow is within one of the components changes to an exceptional control flow which executes a handler for that exception. This handler either returns the component to normal activity or signals a further exception. (The change in control flow of a process that occurs as a result of a raised exception in a sequential system is a special case of the changes in control flow that should occur in any asynchronous system.) Figs. 9 and 10 show the possible control flows of two processes participating in an atomic actions following an exception. In Fig. 9, the recovery measures implemented by the exception handlers succeed and the normal control flow of the processes is resumed. Fig. 10 shows the control flow of the processes of an atomic action when the exception handlers for the components cannot recover.

It is convenient to restrict signaled exceptions so that *each component (or exception handler) of an atomic action returns the same exception.*[4] The signaling of the same exception ensures that the components agree on the abnormal result that should be returned to indicate the failure of the atomic action. An exception is raised in an atomic action if one of its internal atomic actions signals an exception. Signaling a single exception from an internal atomic action simplifies the selection of the appropriate exception handlers and recovery measures.

If *any of the components of an atomic action do not have a handler for a raised exception then all of the components should signal an atomic action failure.*

The raising of an exception in an atomic action indicates that the computation has reached an erroneous state. The strongest postcondition on the state of an action after an exception is detected is a *damage assessment* predicate. (This definition differs somewhat from [1] because we have chosen to assert what is known about the state of the system containing the errors and faults rather than specify the errors and faults.) The predicate should imply the (preferably weakest) precondition for the measures introduced to implement fault tolerance for a given exception. The postcondition for the measures is identical to the postcondition of the atomic action because we have adopted a termination model [14] for the semantics of exception handling.

### D. An Example Banking Service

The principles and framework we have introduced provide a basis for a notation with which to design particular systems. However, language design issues are very complex and we will not attempt to provide a specific syntax or semantics for such a notation in this paper. Instead, we will discuss how an existing practical system might be specified in an informal, and possibly imprecise, example notation. Implementation of the system might include software, computers, and other components such as people. Consider the use of exception handling in the simplified banking service shown in Fig. 11.

In Fig. 11, processes are represented by solid lines, interactions by dotted lines, and atomic actions by boxes. A *client*, represented as a process, presents a check (represented by the interaction labeled a) to its bank *first_fed* and receives a receipt (b). To clear the check, the bank sends the check (c) to the bank *nat'nl* which has the pay-

[4] An exception should be raised if two or more components try to signal different exceptions. The exception handlers for this exception should signal a failure exception.

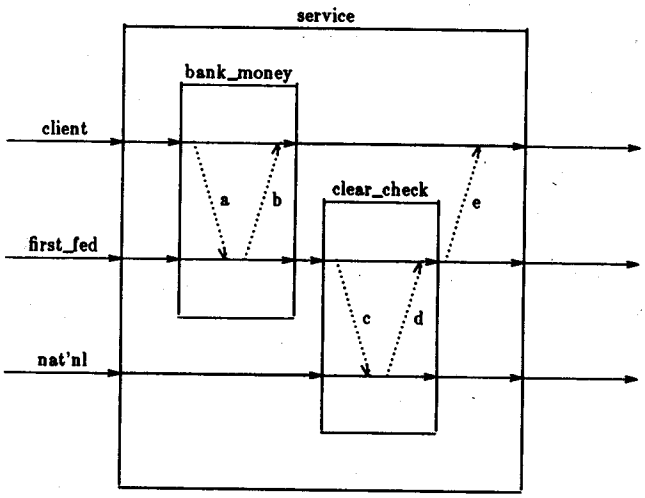



Fig. 11. Atomic actions in a banking service.

```
service = atomic action
    { processes (customer,bank1,bank2);
    exceptions
        { bank_money_failure,
        clear_check_failure,
        check_bounced,
        check_not_valid }
    }

bank_money = atomic action
    { processes (customer, bank);
    exceptions
        { no_signature,
        teller_away_from_desk,
        deposit_slip_incorrect,
        check_on_non_existent_bank }
    }

clear_check = atomic action
    { processes (bank1,bank2);
    exceptions
        { inadequate_balance,
        incomplete_check =
            { inconsistent_values,
            wrong_date},
        theft =
            { forged_signature,
            stolen_check, ... }
    }
```

Fig. 12. Declarations of banking service atomic actions.

er's account. Once *first__fed* receives the cash for the check (d), it sends *client* a new statement of its account (e).

Each atomic action in Fig. 11 identifies an activity which might have faults for which error recovery is desired. Although it might be useful to make the transmission of the statement (e) reliable, it is not necessary and we have chosen to omit any fault-tolerant measures for this interaction. Recovery measures within *bank__money* and *clear__check* can provide fault-tolerant operation for certain localized faults concerning just the pairs of processes involved. However, if the check bounces (that is, does not clear), then *clear__check* must return an abnormal response and invoke the recovery measures of *service*.

An example set of declarations for the atomic actions is shown in Fig. 12 and includes some sample exceptions that might be detected during their execution. The atomic action notation is based on an example syntax given in [12]. Each atomic action declaration has an identifier, a formal parameter list of processes which may take part in the action and a list of possible exceptions. For example, atomic action *clear__check* includes exceptions *inadequate__balance*, *incomplete__check*, *wrong__date*. Exceptions may be structured; that is, an exception may be a composite of several exceptions. The structured exception scheme permits the specification of the exception tree discussed in Section IV.

```
process client;
  { satisfy: service(first_fed,nat'nl);
    exception_handler
    check_bounced: { legal_action }; . . ;
    check_not_valid: { signal failure };
    { deposit: bank_money(first_fed);
      exception_handler
      no_signature: ... ;
      { present(first_fed,check);
      get(first_fed,receipt) }; (*end deposit*)
    receive(first_fed,statement); } (*end satisfy*)
  } (*end client*)

process nat'nl;
  { satisfy: service(client,first_fed);
    exception_handler
    check_bounced: {supply_evidence}; ... ;
    { collect: clear_check(first_fed);
      exception_handler
      inadequate_balance: { update(credit_rating);
        signal check_bounced };
      incomplete_check: ... ;
      { receive(first_fed,check);
      process(check);
      send(first_fed,funds) }; (*end collect*)
    } (*end satisfy*)
  } (*end nat'nl*)

process first_fed;
  { satisfy: service(client,nat'nl);
    exception_handler
    check_bounced: { supply_evidence }; ... ;
    { deposit: bank_money(client);
      exception_handler
      no_signature: ... ;
      { receive(client,check);
      give(client,receipt) }; (*end deposit*)
    collect: clear_check(nat'nl);
    exception_handler
    inadequate_balance:
      { signal check_bounced };
    incomplete_check: ... ;
    { send(nat'nl,check);
    receive(nat'nl,funds) }; (*end collect*)
    send(client,statement) } (*end satisfy*)
  } (*end first_fed*)
```

Fig. 13. Processes within banking service.

An example set of process definitions for the processes in Fig. 11 is shown in Fig. 13. Associated with each process is a block of sequential statements enclosed by brackets { and }. Processes enter an atomic action by executing an atomic action statement. For example, in Fig. 13, *client* enters the atomic action *service* labeled *satisfy*. The label of an atomic action uniquely distinguishes a particular activity. Also, the atomic action statement identifies the other processes involved in the activity by means of its actual parameters. Thus *satisfy* is entered by and involves all three processes.

The atomic action statement includes exception handlers and a block of statements to be executed by the process as its contribution to the action. For example, *satisfy* in *client* defines a handler for the exception *check__bounced* and has a block that begins with the statement labeled *deposit*. Other processes may define different handlers for the same exception. For example, *satisfy* in *first__fed* has a different exception handler for *check__bounced* even though both statements refer to the same activity. The exception handlers defined for an atomic action are enabled on entry to and disabled on exit from the atomic action.

Many possible exceptions and handlers could be defined for the banking service. Exceptions such as *teller__away__from__desk* or *deposit__slip__incorrect* declared in atomic action *bank__money* in Fig. 12 can be handled locally within that action. Such exceptions would involve processes *client* and *first__fed* and should not interfere with the activity of *nat'nl*.

Consider the exception *inadequate__balance* declared in atomic action *clear__check* in Fig. 12. If either *first__fed* or *nat'nl* raise *inadequate__balance* in *collect* in Fig. 13, both banks will execute their corresponding exception handlers. *First__fed* will signal *bounced__check*. *Nat'nl* will also signal *bounced__check* after updating the credit rating for its customer. The atomic action

*collect* will terminate and raise the *bounced_check* exception in atomic action *satisfy*. *Satisfy* involves the three processes *client*, *first_fed*, and *nat'nl*. All three processes will proceed to execute their exception handlers for *bounced_check* and this results in a coordinated effort to get the *nat'nl* customer to pay his debt.

This example banking service could be programmed using standard programming techniques. However, these techniques would involve the use of messages, control constructs, and synchronization schemes which would complicate the specification of the service. We believe that, while the example notation used to discuss the service lacks sophistication and refinement, the framework and principles we propose simplify the construction of fault-tolerant asynchronous systems.

## IV. Exception Resolution

The fault tolerance structuring principles guide the design of a synchronization and coordination framework for forward error recovery in asynchronous systems. However, some aspects of the design of the framework are not obvious and need detailed consideration. The concurrent and potentially parallel nature of the execution of the processes within an atomic action may introduce ambiguity in the choice of fault tolerance measures to handle a particular exception condition. Coordination between the fault tolerance measures provided by an atomic action and the fault tolerance measures provided by any internal atomic actions also requires careful consideration.

### A. Resolution of Concurrently Raised Exceptions

Two or more components of an atomic action may concurrently raise different exceptions. This event is likely if the errors resulting from one or more faults cannot be identified with a unique exception by the error detection facilities of each component in the action. Suppose the two processes $A$ and $C$ shown in Fig. 14 raise exceptions $x$ and $y$, and that these are different. Two different fault tolerance measures could exist to provide recovery for $x$ and $y$, respectively, each consisting of a set of handlers (that is, a handler for each of $A$, $B$, and $C$). However, the two exceptions, in conjunction, constitute a third exception $z$: the condition that *both* exception $x$ and $y$ have occurred. A resolution scheme is required to determine the correct recovery strategy. The introduction of an *exception hierarchy* allows resolution of concurrent exceptions within the same atomic action.

*Exception Hierarchy:* One simple method of providing an exception hierarchy to resolve the ambiguity arising from exceptions that are raised simultaneously is to order the exceptions. From among the exceptions raised within the atomic action, the resolution scheme would select the exception with the highest priority in the order. A resolution mechanism would ensure that all the processes in the atomic action change control flow to execute the appropriate handlers associated with this chosen exception. This scheme is frequently used in sequential systems

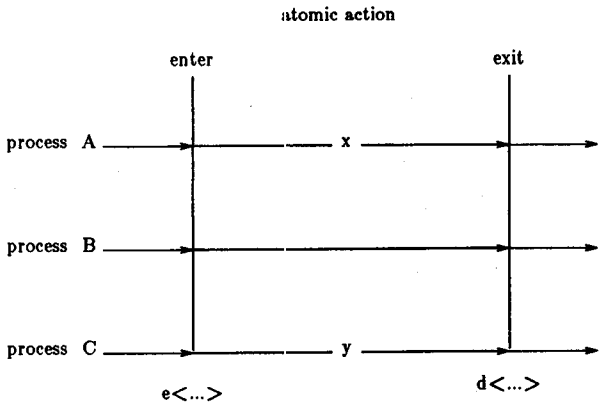



Fig. 14. Exceptions $x$ and $y$ in processes $A$ and $C$.

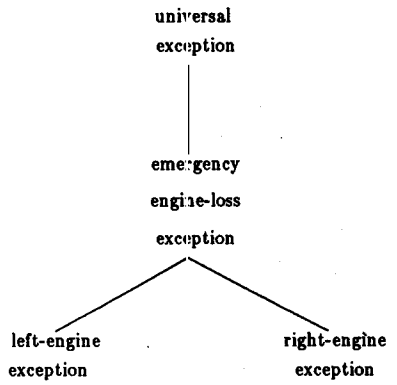



Fig. 15. Example exception tree for twin-engine aircraft.

where the state of only one process is involved in error detection within a component but several exceptions may be raised simultaneously. (For example, a power failure interrupt may take precedence over the detection of an invalid operation code which may take precedence over a page fault for an operand address.) The disadvantage of this simple scheme is that the presence of two or more concurrent exceptions might be symptomatic of a different, more complicated, erroneous state.

We therefore instead propose the use of an *exception tree*. If several exceptions are concurrently raised, the exception used to activate the fault tolerance measures is the exception that is the root of the smallest subtree containing all of the exceptions. This hierarchy permits the occurrence of various exceptions (and thus the detection of erroneous states by several components) to be categorized appropriately by the resolution mechanism.

For example, consider the exception tree for a twin-engine aircraft shown in Fig. 15. If the left (or right) engine fails, the pilot can adjust the controls appropriately to compensate for the loss of the left (right) engine in order to fly the aircraft to the nearest airport. If both the right and left engine fail, the pilot must follow the emergency landing procedures. Even with the complete loss of both engines other exceptions could occur that would endanger the emergency landing procedure (for example, fire). All such further exceptions, if not explicitly listed individually within the exception tree, are categorized as the universal exception.

Each atomic action will have its own tree of exceptions. At the root of each tree is, in principle, *the universal exception*. (In practice, the possibility of a "universal exception" is often ignored.) The universal exception cannot be explicitly raised or signaled by a component; rather it can only be signaled or raised by the underlying exception mechanism. In general, the damage assessment for an exception in the tree will imply the intersection of the damage assessments for the exceptions in each of its subtrees. The greater the number of different exceptions raised within an atomic action, the less the damage assessment predicate may assert about the current state of the system. The damage assessment for the universal exception must assume that any and perhaps all of the state variables and even the representation of the process may have been corrupted. Only the underlying exception mechanism itself should be presumed undamaged. In contrast, the leaves of the exception tree may have very detailed damage assessments corresponding to the failure of particular internal atomic actions or the detection of specific abnormal conditions and errors.

The exception handlers invoked as a result of several exceptions raised concurrently should have weak enough preconditions (that is, equal or weaker than the damage assessment) to allow them to provide the appropriate fault tolerance measures. If the universal exception is raised, its handler should signal a failure exception. The exception mechanism raises the universal exception for any exception that is raised which does not have a handler. However, if there is no handler for the universal exception, the exception mechanism must act on behalf of the atomic action and signal the universal exception to avoid circularity. Even backward error recovery measures require a stronger precondition than that provided by the universal exception. The damage assessment predicate for the other__exceptions exception (introduced in Section II-B to help specify backward error recovery using the exception framework) assumes that the cache correctly holds the prior state of the system. In general, exceptions that invoke backward error recovery measures will be descendents of the universal exception and ancestors of any exceptions invoking forward error recovery measures. Some of the leaves of the exception tree may be failure exceptions of internal atomic actions.

It is convenient to provide *default* exceptions and handlers for specific exceptions or classes of exceptions that may occur within the atomic actions of a system. The exception mechanism must ensure that the correct default handlers are enabled during the activation of each atomic action. Examples of default exception handlers are backward error recovery for the "other__exceptions" exception, forward error recovery that signals a distinguished failure exception for the universal exception and diagnostics for the class of failure exceptions.

The example of nested atomic actions shown in Fig. 16 implies the set of exception trees shown in Fig. 17. The default other__exceptions exception might be associated with backward recovery measures in the form of a conversation. Exceptions $x1$, $x2$, $x1$ and $x2$ (written $x1 \cap x2$), $x3$, $x4$, $x5$, Failure__Atomic__Action__2, and Failure__Atomic__Action__3 might be associated with specific forward error recovery measures.

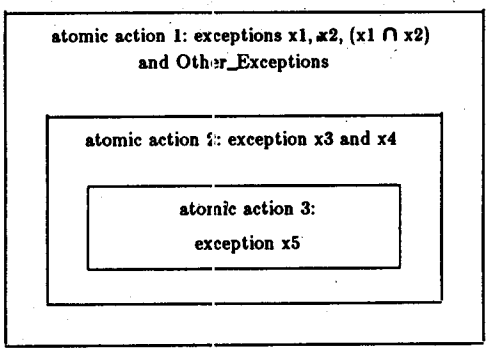



Fig. 16. Nested contexts of three atomic actions.

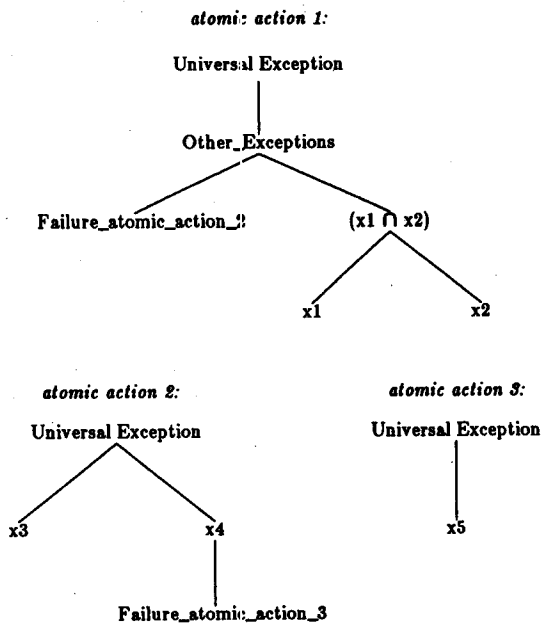



Fig. 17. The three exception trees of the atomic actions.

The exception tree could be generalized to a complete lattice [19]. The lattice would represent a partial ordering of the exceptions. The resolution mechanism would resolve concurrently raised exceptions by selecting the exception that is their least upper bound within the lattice. The least upper bound of all the exceptions in the lattice would be, of course, the universal exception. Whether such a general structure is desirable for constructing reliable systems can only be determined from future practical experience.

### B. Nested Atomic Actions

The forward error recovery framework for asynchronous systems must synchronize and coordinate recovery from exceptions within fault-tolerant systems that are themselves constructed from fault-tolerant components. In particular:

1) A component of the atomic action may raise an ex-

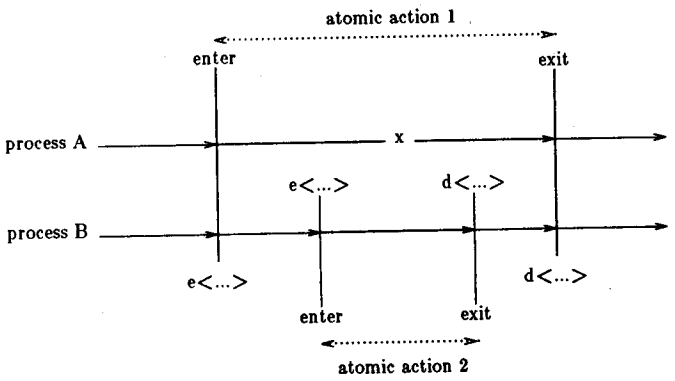



Fig. 18. Example of an exception $x$ in an atomic action with an internal atomic action.

ception while other components of the atomic action are involved in internal atomic actions.

2) The fault tolerance measures for an atomic action may require that internal atomic actions be aborted.

*1) Exceptions and Internal Atomic Actions:* Suppose a component of an atomic action raises an exception while other components are involved in internal atomic actions. Then, in principle, all the components of the atomic action must invoke fault tolerance measures even if the atomic action includes internal atomic actions. However, the definition of atomicity makes internal atomic actions indivisible. In addition, out of the large number of possible exceptions that might be raised within an atomic action, many will have no meaning within an internal atomic action. An example of a nested atomic action is shown in Fig. 18. The fault tolerance measures implemented for exception $x$ in atomic action 1 will assume that either the atomic action 2 has not yet started or that it has already completed. Further, exception $x$ may have no meaning within atomic action 2.

Thus, after the detection of an exception, any active internal atomic action must be completed before the fault tolerance measures of the containing atomic action are invoked. (This also implies that if components of an atomic action and components of an internal atomic action concurrently raise different exceptions, the fault tolerance measures of the internal atomic action will be completed first.) However, in certain circumstances it may be desirable to abort an internal atomic action and this situation is examined next.

*2) Aborting Internal Atomic Actions:* Although, in theory, a containing atomic action can always compensate for interior atomic actions by masking their effects, these fault tolerance measures cannot be commenced until the internal atomic actions terminate. Thus, if an exception associated with real-time concerns is raised in the containing atomic action, the delay caused because internal atomic actions are active may prevent a timely recovery. Alternatively, the containing atomic action may detect an exception which indicates that its internal atomic actions will not terminate (for example, a deadlock condition). In this case, it would never be able to invoke its recovery measures. The fault tolerance framework must therefore permit the abortion of internal atomic actions. We thus propose the following solution.

An internal atomic action may be aborted if it is defined to have a distinguished abortion exception. An *abortion exception is raised by the exception mechanism to indicate that an exception has been raised in the containing atomic action.* The abortion exception is, to all intents and purposes, a special interface exception that is raised automatically to indicate that the preconditions under which the internal atomic action was invoked are invalid. If an abortion exception is raised, the internal atomic action should proceed to apply fault tolerance measures to abort itself. When the internal atomic action has completed its fault tolerance measures and terminates, the containing atomic action may invoke its own fault tolerance measures.

If an internal atomic action cannot abort itself correctly and return normally, it may signal an exception. (If the atomic action can neither return normally nor signal an exception it is not fault-tolerant.) The proposed scheme does not distinguish between a signaled exception returned from an aborted atomic action and one returned from the completion of an ordinary internal atomic action. In both cases, the signaled exception is raised in the containing atomic action and may influence the choice of the fault tolerance measures that are subsequently invoked.

The principle of recovery occurring within an atomic action is not contravened by the abortion scheme because:

1) the abortion exception is included as part of the specification of the internal atomic action;

2) any recovery instituted as part of the abortion of the internal atomic action is accomplished within that atomic action;

3) any recovery instituted by the internal atomic action is indivisible with respect to the containing environment. (The processes in the containing environment are suspended, if necessary, until the internal atomic action completes its recovery.)

4) only one abortion exception is allowed for each internal atomic action and it is raised if any exception occurs in the containing atomic action. *Complete* damage assessment of an atomic action can only be made when all of its processes suspend their normal control flow (and all internal atomic actions have completed). If multiple abortion exceptions were permitted and were bound to different exception conditions in the containing atomic action then:

a) an aborted internal atomic action could signal an exception which aborts another internal atomic action. (This complicates the exception mechanism and may impose undesirable delays upon error recovery while internal atomic actions abort each other.)

b) an abortion exception could be raised in an internal atomic action that is already trying to abort itself. (That is, the initial abortion of an internal atomic action is based on an *incomplete* damage assessment.)

The abortion scheme we propose requires an internal atomic action to be defined with an explicit abortion ex-

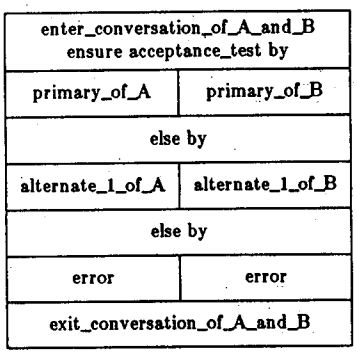



Fig. 19. A conversation.

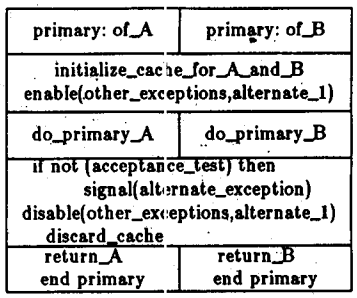



Fig. 20. Primary implementations.

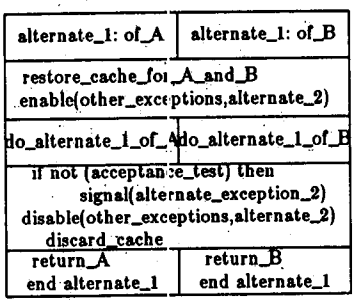



Fig. 21. Alternate implementations.

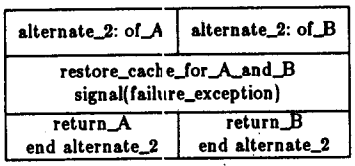



Fig. 22. Default alternate implementations.

ception. However, sometimes it may be more convenient to make the abortion exception implicit and to associate a default handler with the exception. For example, backward error recovery schemes often assume a default abortion exception and handler. If an exception is raised within a conversation, provided a recovery cache permits restoration of prior system states, any internal conversations may be immediately aborted.

### C. Implementing Backward Error Recovery

The framework we propose permits the implementation of the conversation scheme proposed by [21]. Each conversation is an atomic action composed of several components which have contexts, exceptions, and exception handlers. The fault tolerance measures require the activation of a cache for the initial state of the atomic action. The acceptance test provides error detection and raises an exception if it fails. Should an exception be raised in one or more components, normal control flow is suspended in all the processes and exceptional control flow is commenced. The handlers restore changed cached values and attempt the alternate algorithms. If all the alternates fail, a failure exception is signaled to a containing atomic action. We assume that the processes invoking the conversation are appropriately synchronized to execute the conversation correctly, although we do not discuss the mechanisms concerned.

*Example of a Conversation:* The interaction of two processes *A* and *B* in a conversation is shown in Fig. 19. Individual actions of each process and combined actions of both processes are distinguished by enclosing them in boxes. Synchronization and coordination between the processes is implied by the structuring of the diagram. The conversation can be implemented using the exception handling scheme shown in Figs. 20, 21, and 22.

It is trivial to generalize this implementation of the conversation so that internal conversations can be aborted by an abortion exception.

### D. Recovery Schemes Supported by the Framework

The exception framework supports both backward and forward error recovery schemes. Within a particular atomic action, it is possible to employ both schemes. For example, particular exceptions may have forward error recovery handlers and any other exception may have a default handler that implements a backward error recovery scheme. Thus, the framework generalizes the scheme of combining forward and backward error recovery for a sequential process described in [17] to the case of asynchronous systems.

### E. Structured Exceptions in the Banking Service Example

In the example shown in Fig. 11, an exception tree is associated with each atomic action. The root of each tree is the default universal exception. The subtrees of the trees are specified by the exceptions declared in Fig. 12. Thus, for the atomic action *clear_check*, the universal exception has three subtrees, *inadequate_balance*, *incomplete_check*, and *theft*. The exception *incomplete_check* is a structured exception and contains further exception declarations enclosed by { and }. The exceptions declared within a structured exception form subtrees of the structured exception. Thus, *incomplete_check* has subtrees *wrong_date* and *inconsistent_values*.

Multiple exceptions are resolved using the exception tree. If, in *clear_check*, the check has *inconsistent_values*, the exception could be handled by assuming that the value specified alphabetically is valid and the value specified numerically is incorrect. Similarly, if the check

has the *wrong_date*, the exception could be handled by correcting the date. However, if both *wrong_date* and *inconsistent_values* are detected, the check would be handled as an *incomplete_check*.

## V. A Spectrum of Error Recovery Techniques

Depending upon the extent to which atomic actions are planned or spontaneous, the framework of fault tolerance based on atomic actions supports a spectrum of error recovery techniques (described in Section I-B) for asynchronous systems.

The *resolution mechanism* automates the propagation of an exception occurring in one process to all other processes in the atomic action while resolving any ambiguities caused by several components raising different exceptions concurrently. It separates the provision of the underlying resolution and exception facilities from the user and simplifies the construction of recovery measures from the exception handling framework. The method by which the processes of an atomic action may be identified depends upon the way in which atomic actions are implemented and the degree to which their constituent activities are parameterized in the definition of the atomic action. Essentially, the mechanism:

1) suspends all the processes engaged in the atomic action.[5]

2) aborts any internal atomic actions requiring abortion because of the exception condition.

3) chooses the appropriate exception that reflects the erroneous state of the atomic action.[6]

4) raises the chosen exception in all of the processes, causing them to change control flow and execute the appropriate exception handlers.

We will examine an implementation of the resolution mechanism for two different forms of planned atomic action. The first form of planned atomic action assumes that the identity of the atomic action is explicitly defined (for example, as in a conversation). The second form of planned atomic action, which provides more spontaneity, allows atomic actions to be implicitly created during the lifetime of a system (for example, as in the chase protocol scheme).

### A. Explicitly Defined Atomic Actions

Explicitly defined planned atomic actions have predetermined components that are designed to synchronize to form a particular atomic action. Examples include 1) the checkpointing of a large distributed database, 2) the organization of compiled code for a multiprocessor, and 3) computer architectures with uninterruptible instructions.

The synchronization required to implement a planned atomic action may impose an overhead on the efficiency with which internal actions are executed, or may restrict the degree of parallelism between those actions. For example, if concurrent access to a set of resources is infrequent, any synchronization imposed to achieve atomicity is largely a performance overhead. Similarly, mutually exclusive access to a set of resources provides atomicity but eliminates parallel processing.

However, planned atomic actions simplify certain implementation concerns when they are used as a framework for fault tolerance. Prior knowledge of the components of a planned atomic action aids:

1) identification of the components within the atomic action.

2) communication required to coordinate invocation of fault tolerance measures.

3) rigorous specification of the function implemented by the atomic action. Rigorous specifications support error detection and damage assessment.

A range of synchronization and coordination techniques may be used to construct planned atomic actions. One component of the action may provide a centralized control for synchronizing the other components. Such schemes are similar to the monitor [10] and to similar synchronization schemes for accessing shared data. Alternatively, each component may support a distributed control scheme perhaps based upon transmitting messages and employing a two-phase protocol [9]. Such a mechanism is outlined in the Appendix.

### B. Implicitly Defined Atomic Actions

Atomic actions which are implicitly defined are less easy to use to support forward error recovery although they have been proposed to support backward error recovery in the chase protocol scheme [18]. The difficulty in using such atomic actions arises from the problems of specifying their correct behavior for the purposes of measuring reliability. Despite the problems of using spontaneous atomic actions in practice, we shall briefly examine the consequences of spontaneity upon the proposed exception handling scheme and resolution mechanism.

The group of components constituting an implicit planned atomic action is recognized *by the interactions that have occurred* and the handling contexts established by the components. The boundaries of the atomic action will coincide with the boundaries of the individual components and contexts that provide fault tolerance measures. Implicit atomic actions introduce a new ambiguity in the choice of fault tolerance measures to handle a particular exception condition. The ambiguity arises because there may be no predetermined relation between a particular exception and a given atomic action. Consider the interaction between two processes shown in Fig. 23. The contexts for the handlers $h1$, $h2$, and $h3$ define the boundaries of possible implicit atomic actions that may be used in recovery schemes. Process $A$ and $B$ exchange information at interaction $I1$ and, by definition, at this point they must be in the same atomic action. Both processes have handlers for an exception $x$ and the enable and dis-

[5]Atomicity prevents the suspension of a process engaged in an internal atomic action until that action terminates.

[6]Exceptions may be raised concurrently during the interval between the occurrence of the first raised exception and the completion of the last internal atomic action.

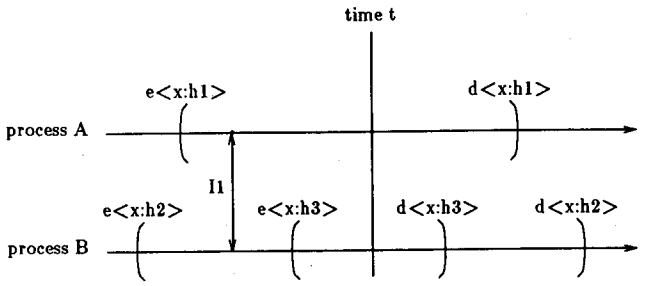



Fig. 23. Exception in implicit atomic action.

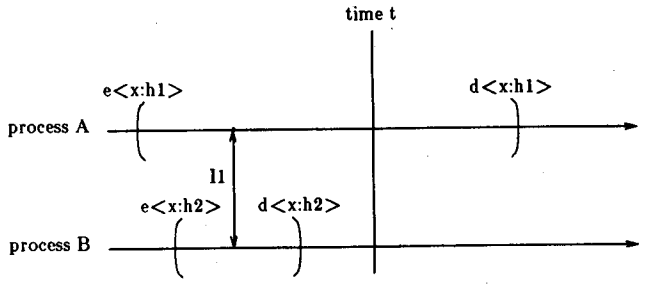



Fig. 24. Example of a commitment exception.

able operations for this exception now delimit an implicit atomic action. If process $A$ raises an exception $x$ at time $t$, the appropriate context associates handler $h1$ with the recovery to be invoked. Although exception $x$ occurs when process $B$ is executing within the context associated with handler $h3$, there is no interaction of that context with process $A$ and it can be regarded as a different implicit atomic action. Therefore, that internal atomic action should be completed and then handler $h2$ of the enclosing context of process $B$ invoked for the exception $x$. (If the exception $x$ had been detected in process $B$ instead of process $A$, only handler $h3$ would be invoked and process $A$ would be unaffected.)

Any resolution mechanism for implicit atomic actions must compute a recovery strategy from the known set of interactions and the contexts in which they occurred. Such a resolution mechanism will be similar, in many ways, to the mechanism underlying the chase protocol [26].

*Commitment Exceptions and Failures:* A direct result of the implicit formation of atomic actions is that a process may discard provisions for forward error recovery prematurely. For example, it may disable a handler for a context containing an interaction with another process even though that other process might still raise an exception. Fig. 24 illustrates such a situation. Exception $x$ raised in process $A$ at time $t$ will generate a failure exception in process $B$ which has discarded its exception handler ($h2$) for $x$. (This assumes $B$ does not have a handler for $x$ in a containing context.) Such exceptions result from process $B$ committing itself too early to the results of a computation [23] that were formed in a cooperative atomic action. Because implicit atomic actions are involved, it is very difficult to devise a practical forward error recovery strategy for commitment errors.

## VI. Conclusion

We have introduced a framework for the provision of fault tolerance in asynchronous systems. The proposal generalizes the form of simple recovery facilities supported by nested atomic actions in which the exception mechanisms only permit backward error recovery, as has been proposed for databases [4]. It allows the construction of systems employing both forward and backward error recovery and thus allows the exploitation of the complementary benefits of the two schemes. Backward recovery, forward recovery, and normal processing activities can occur concurrently within the organization proposed.

We believe that a reduction in the complexity of the design of fault-tolerant software for an asynchronous system can be achieved by using atomic actions to structure the activity of the system. Although many notations have been devised for error recovery which include explicit definitions of atomic actions [2], [13], [15], [16], [22], [23], [25] most of the notations are either inadequate or too restricted to permit their use as the basis for the exception scheme we have described. Practical systems can only be constructed if suitable notations are developed to express the concept of an atomic action [5].

We have generalized exception handling to provide a uniform basis for fault tolerance schemes within the atomic action structure. The generalization included a resolution scheme for concurrently raised exceptions based on an exception tree and an abortion scheme to permit the termination of internal atomic actions.

Finally, we have outlined an automatic resolution mechanism for exceptions in atomic actions which allows users to separate their recovery schemes from the details of the underlying algorithms. While we have not discussed implementation in any detail in this paper, the mechanism can be implemented with distributed control by means of a simple message passing system.

## Appendix
## Resolution Algorithm and Mechanism for Planned Atomic Actions

We will assume a distributed system in which processes can exchange messages. Three kinds of messages are employed:

1) Raise exception $x$ in atomic action $aa$.
2) Acknowledge exception in atomic action $aa$.
3) Commit components involved in atomic action $aa$ to invoke the exception handlers.

The message passing system includes timeout, checksum, and other facilities to ensure reliable transmission. If the message passing system fails to transmit or receive a message for a process attempting recovery, an undefined exception is raised in that process. Each process, message, context, and exception can be uniquely identified. Some (unspecified) mechanism provides a mapping from an atomic action into the processes that are engaged in that atomic action.

The distributed algorithm, shown in Fig. 25, implements the resolution mechanism and is executed by each process in an atomic action. The algorithm consists of four parts corresponding to 1) the detection of an exception, 2) the completion of internal atomic actions, 3) the receipt of an acknowledgment, and 4) the receipt of a "commit" message. The algorithm terminates after one or more processes receive a complete set of replies to a broadcast. Each of these processes then broadcasts a commit message to all the other processes in the atomic action permitting them to begin recovery. Separate copies of the algorithm are instantiated for each of the processes involved in the atomic actions. Each copy of the algorithm has its own set of variables. The four parts of each instantiation of the algorithm are mutually exclusive.

A measure of the complexity of the algorithm is the total number of messages required to establish exception handling. The minimum number of messages occurs when only one process detects an error and transmits an exception message to the other processes engaged in an atomic action. The minimum is:

$$3(n - 1)$$

where $n$ is the number of processes involved in the atomic action. The maximum number of messages required to establish recovery within a particular atomic action occurs when:

1) Every process detects an error and sends exception messages to the other processes in the atomic action concurrently. This contributes $n(n - 1)$ messages.

2) Every process receives one exception message and sends new exception messages to the other processes in the atomic action concurrently.

3) Step 2) is repeated the maximum of $n - 2$ times after which every exception has been received by every process. This contributes $n(n - 1)(n - 2)$ messages.

4) Every process replies to the last $n - 1$ exception messages. This contributes $n(n - 1)$ messages.

5) Every process broadcasts a commit. This contributes $n(n - 1)$ messages.

The maximum number of messages is:

$$n(n - 1) + n(n - 1)(n - 2) + n(n - 1) + n(n - 1)$$

or

$$n(n - 1)(n + 1).$$

This assumes that the height of the exception tree is at least $n$.

Although the resolution mechanism requires communication among the processes in an atomic action there is no overhead if an exception is not raised. Moreover, the overhead can be much reduced by centralizing the control of the resolution mechanism, minimizing the height of the exception tree (or number of exceptions), or minimizing the number of processes in each atomic action.

```
(*When an exception is detected:*)

when receive("raise",x:exception; aa:atomic_action) or raised(x:exception; aa:atomic_action) do
    {
    (*Save current pending exception for broadcast condition.          *)
    previous_pending[aa] := pending[aa];

    (*Resolve exception within tree and save in 'pending'.    *)
    if raised(x) then
        {
        pending[aa] := if node(x,exception_tree[aa]) then x
                       else universal_exception
        }
    else pending[aa] := root_smallest_subtree(exception_tree[aa],pending[aa],x )

    (*Check for an exception handler for the exception.*)
    if handler[aa,pending[aa]]=nil then pending[aa] := universal_exception

    (*Let the other processes know which exception is pending          *)
    (*in this process if a raised exception changes the pending        *)
    (*exception or a raise message results in a new exception.*)
    if      pending[aa] <> x or
            (raised(x) and (previous_pending[aa] <> pending[aa]) then
        {
        broadcast("raise",pending[aa],aa) to other_processes[aa]
        replies_needed[aa] := number(other_processes[aa])
        }
    else
        {
        replies_needed[aa] := 0
        (*Save acknowledgments until can suspend process.*)
        enqueue_ack("acknowledgement",aa,source_process)
        }

    (*Finish any internal atomic actions.*)
    if internal_atomic_action[aa].active then
        {
        (*If permitted abort internal atomic action.*)
        if internal_atomic_action[aa].aborts and
              not internal_atomic_action[aa].aborted then
              {
              (*Abort only on the first raised exception  *)
              (*in the containing atomic action.          *)
              internal_atomic_action[aa].aborted := true
              raise(abort,internal_atomic_action[aa])
              }
        (*Let the internal atomic action finish     *)
        (*using a co-routine resume.                *)
        resume_process
        }
    else
        {
        (*If there are no internal atomic actions                    *)

              (*send any acknowledgments and suspend the process.    *)
              while queued_acks do send(dequeue_ack)
              suspend_process
              }
        }


    (*When a process returns from an internal atomic action          *)
    (*to a containing atomic action with a pending                    *)
    (*exception:                                                      *)

    when process_returns(aa:atomic_action) do
          if pending[aa] <> nil then
              {
              (*Send any acknowledgements.*)
              while queued_acks do send(dequeue_ack)
              suspend_process
              }


    (*When an acknowledgement is received for the *)
    (*last broadcast made, check to see if every                      *)
    (*acknowledgment has been received:                               *)

    when receive("acknowledgement",aa:atomic_action) do
          if acknowledge_last_broadcast and (replies_needed[aa] > 0) then
              {
              replies_needed[aa] := replies_needed[aa]-1
              if replies_needed[aa = 0 then
                    broadcast("commit",aa) to other_processes[aa]
              execute(handler[aa,pending[aa]])
              }
          else ignore


    (*When resolution has finished, one or more of the processes     *)
    (*will have received a complete set of acknowledgments.          *)
    (*These processes broadcast a commit, and every process          *)
    (*which receives a commit may commence recovery.                 *)
    (*Note that processes cannot commit until all internal           *)
    (*atomic actions are terminated and acknowledgements made.  *)

    when receive("commit",aa) do
          execute(handler[aa,pending[aa]])
```

Fig. 25. A resolution mechanism algorithm.

## REFERENCES


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

[2] T. Anderson and M. R. Moulding, "Dialogues for recovery coordination in concurrent systems," Comput. Lab., Univ. Newcastle upon Tyne, Tech. Rep., in preparation, 1983.

[3] E. Best and B. Randell, "A formal model of atomicity in asynchronous systems," Comput. Lab., Univ. Newcastle Upon Tyne, Tech. Rep. 130, Dec. 1980.

[4] L. A. Bjork and C. T. Davies, "The semantics of the preservation and recovery of integrity in a data system," IBM Tech. Rep. TR02.540, Dec. 1972.

[5] R. H. Campbell and T. Anderson, "Practical fault tolerant software for asynchronous systems," in *SAFECOMP 83, Third Int. IFAC Workshop Achieving Safe Real-time Comput. Syst.* Oxford, England: Pergamon, 1983.

[6] L. Chen and A. Avizienis, "*N*-version programming: A fault-tolerance approach to reliability of software operation," in *Dig. of Papers FTCS-8: Eighth Annu. Int. Symp. Fault-Tolerant Comput.*, Toulouse, June 1978, pp. 3-9.

[7] F. Cristian, "Exception handling and software fault tolerance," *IEEE Trans. Comput.*, vol. C-31, pp. 531-540, June 1982.

[8] C. T. Davies, "Data processing spheres of control," *IBM Syst. J.*, vol. 17, no. 2, pp. 179-198, 1978.

[9] J. N. Gray, "Notes on data base operating systems," in *Lecture Notes in Computer Science, Vol. 60*, R. Bayer, R. M. Graham and G. Seegmuller, Eds. Berlin, Germany: Springer-Verlag, 1978, pp. 393-481.

[10] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Commun. ACM*, vol. 17, no. 10, pp. 549-557, Oct. 1974.

[11] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell, "A program structure for error detection and recovery," in *Lecture Notes in Computer Science, Vol. 16*, E. Gelenbe and C. Kaiser, Eds. Berlin, Germany: Springer-Verlag, 1974, pp. 171-187.

[12] P. Jalote and R. H. Campbell, "Fault tolerance using communicating sequential processes," in *14th Int. Conf. Fault-Tolerant Comput. (FTCS-14)*, Orlando, FL, June 1984, pp. 347-352.

[13] K. H. Kim, "Approaches to mechanization of the conversation scheme based on monitors," *IEEE Trans. Software Eng.*, vol. SE-8, pp. 189-197, May 1982.

[14] B. H. Liskov and A. Snyder, "Exception handling in CLU," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 546-558, Nov. 1979.

[15] B. Liskov, "On linguistic support for distributed programs," *IEEE Trans. Software Eng.*, vol. SE-8, pp. 203-210, May 1982.

[16] D. B. Lomet, "Process synchronization, communication and recovery using atomic actions," *SIGPLAN Notices*, vol. 12, no. 3, pp. 128-137, Mar. 1977.

[17] P. M. Melliar-Smith and B. Randell, "Software reliability: The role of programmed exception handling," *SIGPLAN Notices*, vol. 12, no. 3, pp. 95-100, Mar. 1977.

[18] P. M. Merlin and B. Randell, "Consistent state restoration in distributed systems," in *Dig. Papers FTCS-8: Eighth Annu. Int. Symp. Fault-Tolerant Comput.*, Toulouse, France, June 1978, pp. 129-134.

[19] R. Milne and C. Strachey, *A Theory of Programming Language Semantics*. London, England: Chapman and Hall, 1976.

[20] B. Randell, P. A. Lee, and P. C. Treleaven, "Reliability issues in computing system design," *ACM Comput. Surveys*, vol. 10, no. 2, pp. 123-165, June 1978.

[21] B. Randell, "System structure for fault tolerance," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 220-232, Mar. 1975.

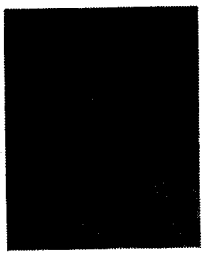
[22] D. L. Russell and M. J. Tiedeman, "Multiprocess recovery using conversations," in *Dig. Papers FTCS-9: Ninth Annu. Int. Symp. Fault-Tolerant Comput.*, Madison, WI, June 1979, pp. 106-109.

[23] S. K. Shrivastava, "A dependency, commitment and recovery model for atomic actions," in *Proc. Second Symp. Reliability in Distributed Software and Database Systems*, Pittsburgh, PA, July 1982, pp. 112-119.

[24] S. K. Shrivastava and J-P. Banatre, "Reliable resource allocation between unreliable processes," *IEEE Trans. Software Eng.*, vol. SE-4, pp. 230-241, May 1978.

[25] A. Z. Spector and P. M. Schwarz, "Transactions: A construct for reliable distributed computing," *Operat. Syst. Rev.*, vol. 17, no. 2, pp. 18-35, Apr. 1983.

[26] W. G. Wood, "A decentralised recovery control protocol," in *Dig. Papers FTCS-11: Eleventh Annu. Int. Conf. Fault-Tolerant Computing*, Portland, OR, June 1981, pp. 159-164.

**Roy H. Campbell** (M'84) received the B.Sc. degree in mathematics from the University of Sussex, Sussex, England, in 1969 and the M.Sc. and Ph.D. degrees in computer science from the University of Newcastle upon Tyne, Newcastle upon Tyne, England, in 1972 and 1977, respectively.

Since 1976, he has been with the Department of Computer Science at the University of Illinois, Urbana-Champaign, and was promoted to Full Professor in 1985. His research interests include distributed systems, operating systems, programming language design, and software engineering. He is principal investigator of the EOS project to develop techniques and methodologies for programming embedded real-time operating systems (NASA) and the SAGA project which is building a prototype software development environment (NASA).

Dr. Campbell is a member of the IEEE Computer Society and the Association for Computing Machinery.

**Brian Randell** received the degree in mathematics from Imperial College, London, England, in 1957.

He then joined the English Electric Company where he led a team which implemented a number of compilers, including the Whetstone KDF9 Algol Compiler. From 1964 to 1969 he was with IBM, mainly at the IBM Research Center in the United States, working on operating systems, the design of ultrahigh-speed computers, and system design methodology. He then became Professor of Computing Science at the University of Newcastle upon Tyne, Newcastle upon Tyne, England, where in 1972 he initiated a program which now encompasses several major research projects sponsored by the Science and Engineering Research Council and the Ministry of Defence of England.