Abstract:

The demand for highly reliable computer systems has led to techniques for the construction of fault-tolerant software (Chen and Avizienis, 1978; Horning and colleagues, 1974). Networks of computers, distributed resources, and multiple CPUs introduce new problems of constructing reliable systems and involve the organization and control of error recovery in complex asynchronous systems (Davies, 1978; Kim, 1982; Liskov, 1982; Randell and colleagues, 1978). This paper reviews the general principles and frameworks proposed for the design of fault-tolerant asynchronous systems and discusses the pragmatic issues that must be resolved before such systems become practicable.

Practical Fault Tolerant Software for Asynchronous Systems

By

R.H. Campbell, T. Anderson and B. Randell

TECHNICAL REPORT SERIES

Editor: Mr. M.J. Elphick

Number 187
August, 1983

© 1983 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU, England.
Practical fault tolerant software for asynchronous systems. [By] R.H. Campbell, T. Anderson and B. Randell

Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1983.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 187.)

Added entries

ANDERSON, Thomas.
RANDELL, Brian.
UNIVERSITY OF NEWCASTLE UPON TYNE.

Suggested classmarks (primary classmark underlined)

Dewey (18th):
U.D.C.
001.6425 001.64404
519.687 681.322.06

Suggested keywords

ATOMIC ACTIONS
BACKWARD ERROR RECOVERY
COMPUTER SOFTWARE
FAULT TOLERANCE
RELIABILITY
SYSTEM FAILURE AND RECOVERY

Abstract

The demand for highly reliable computer systems has led to techniques for the construction of fault-tolerant software (Chen and Avizienis, 1978; Horning and colleagues, 1974). Networks of computers, distributed resources, and multiple CPUs introduce new problems of constructing reliable systems and involve the organization and control of error recovery in complex asynchronous systems (Davies, 1978; Kim, 1982; Liskov, 1982; Randell and colleagues, 1978). This paper reviews the general principles and frameworks proposed for the design of fault-tolerant asynchronous systems and discusses the pragmatic issues that must be resolved before such systems become practicable.

About the author

Professor Roy H. Campbell is now at the Department of Computer Science at the University of Illinois, U.S.A.
Dr. T. Anderson is currently a lecturer in the Computing Laboratory at the University of Newcastle upon Tyne.
Professor B. Randell has been a Professor of Computing Science in the Computing Laboratory since 1969.
PRACTICAL FAULT TOLERANT SOFTWARE FOR ASYNCHRONOUS SYSTEMS

R. H. Campbell
Department of Computer Science, University of Illinois, 1304 W. Springfield Av., Urbana IL 61801, USA.

T. Anderson, H. Randell
Computing Laboratory, University of Newcastle upon Tyne, Claremont Road, Newcastle upon Tyne NE1 7RU, UK.

Abstract. Networks of computers, distributed resources, and multiple CPUs introduce new problems of constructing reliable systems and involve the organization and control of error recovery in complex asynchronous systems. Recent research has provided evidence that fault-tolerant asynchronous systems are both necessary and feasible. In this paper, we review and discuss several of the pragmatic issues that need to be resolved before the results of this research can be applied in practice.

Keywords. System failure and recovery; computer software; parallel processing; reliability; fault tolerance; atomic actions; backward error recovery.

INTRODUCTION

The demand for highly reliable computer systems has led to techniques for the construction of fault-tolerant software (Chen and Avizienis, 1978; Horning and colleagues, 1974). Networks of computers, distributed resources, and multiple CPUs introduce new problems of constructing reliable systems and involve the organization and control of error recovery in complex asynchronous systems (Davies, 1978; Kim, 1982; Liskov, 1982; Randell and colleagues, 1978). This paper reviews the general principles and frameworks proposed for the design of fault-tolerant asynchronous systems and discusses the pragmatic issues that must be resolved before such systems become practicable.

FAULT-TOLERANT SOFTWARE SYSTEMS

A fault-tolerant system is one that is designed to function reliably despite the effects caused by component or design faults during normal processing. Such a system detects the 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. The task of error recovery is hampered by the propagation of errors—the continued valid 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 continue to function despite error propagation during the, perhaps lengthy, time interval between the first manifestation of a fault and the eventual detection of an error.

So called "forward error recovery" is accomplished by making selective corrections to a system state containing errors. It aims to remove or isolate specific errors so that normal computation can be resumed (Randell and colleagues, 1978). Because recovery is applied to a system state containing errors, forward error recovery techniques require accurate damage assessment (or estimation) 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 (Horning and colleagues, 1974). (In practice, the most recent restorable system state which is free from the effects of the fault may be difficult to determine. It may be necessary to restore a sequence of successively earlier states until recovery is successful.) Because backward error recovery restores a valid prior system state, recovery is possible from errors of largely unknown origin and propagation characteristics. (All that is required is that the errors have not affected the state restoration mechanism.) Backward error recovery may involve a considerable time penalty in overhead and could require tests for acceptable system states.
Forward and backward error recovery techniques complement one another, forward recovery allowing efficient handling of expected conditions and backward recovery providing a general strategy which can 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 (Cristian, 1982) by transforming unexpected errors into default exception conditions.

Exception handling provides a very convenient framework for the implementation of fault tolerance in systems having only a single sequential process (Anderson and Lee, 1981). Both forward and backward error recovery can be supported within such a framework (Cristian, 1982). Fault tolerance provisions for systems of asynchronous processes are complicated by the possibility of communication of erroneous information and the need to co-ordinate processes engaged in recovery. Generalizing exception handling to support fault tolerance in asynchronous systems requires additional system structure concerning the cooperation and co-ordination of the individual processes.

STRUCTURING FAULT-TOLERANT ASYNCHRONOUS SYSTEMS

The construction of systems with activities that are formed from atomic actions provides a structure for fault tolerance in asynchronous systems. Although atomic actions have been defined many times in different ways (for example, (Davies, 1978; Liskov, 1982; Lomnitz, 1977)) we will use the following definition (Anderson and Lee, 1981):

"The activity of a group of components constitutes an atomic action if there are no interactions between that group and the rest of the system for the duration of the activity."

A more rigorous definition based on occurrence graphs formed from events and causality relations enables formal analysis of nested atomic actions (Pest and Randell, 1980).

The design of many recovery schemes has been based on a mechanism which supports atomic actions. For example, conversations (Randell, 1975), recoverable monitors (Kim, 1978), chase protocol recovery (Merlin and Randell, 1978), transactions (Spector and Schwartz, 1983) and two phase commit protocols (Gray, 1976) all provide atomic actions for interacting processes. There are two reasons why atomic actions provide the basis for so many different approaches. If a fault, resulting error propagation, and subsequent successful error recovery all occur within a single atomic action they will not affect any 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.

The notion of reliability requires that a system has 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. An initial and final state can be associated with each component joining and leaving the atomic action. Pre- and post-conditions at the entry and exit points of the components can specify the results of the activity of each component. These pre- and post-conditions 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.

The following two principles have been proposed for structuring fault tolerance within asynchronous systems (Campbell and Randell, 1983):

1) The services 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 its components.

A fault-tolerant system is reliable as long as it provides services which meet its specification, even though it may suffer from internal faults and contain internal errors. 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 a few components in much larger fault-tolerant systems involving people and process control equipment. Of course, error recovery in such systems must be co-ordinated between components having very different characteristics.

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 co-operatively 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. A distinguished atomic action failure exception signifies the failure of one or more of the components of an atomic action. The failure exception may be signalled explicitly by the components of an atomic action. Alternatively, the failure exception may be signalled implicitly as a default recovery measure for an exception that is raised within a component of the atomic action which has no appropriate exception handler.

The following exception handling scheme for atomic actions has been proposed by Campbell and Randell (1983):

1) If one or more components of the atomic action raise an exception then the fault tolerance measures necessarily involve all of the components of that atomic action. (If some of the components do not require any fault tolerance measures, they do not interact with the other components and hence can form a separate atomic action.) Figures 1 and 2 show example atomic actions in which an exception X has been raised.

2) Every component of the atomic action responds to the raised exception by changing to an exceptional control flow which executes the handler for that exception. (Thus, exception handling in a sequential system is a special case.) Figures 1 and 2 show the changed control flows of the components of two atomic actions following an exception. In Fig. 1, the recovery measures implemented by the exception handlers succeed and the normal control flow of the components is resumed.

```
<table>
<thead>
<tr>
<th>Atomic Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>+--------+---+---+-----------------+</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>+--------+---+---+-----------------+</td>
</tr>
<tr>
<td>+--------+---+---+-----------------+</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>+--------+---+---+-----------------+</td>
</tr>
</tbody>
</table>
```

Fig. 1. An example of successful error recovery in an atomic action.

3) It is convenient to restrict signalled exceptions so that each component (or exception handler) of an atomic action returns the same exception. The signalling of multiple exceptions would only serve to confuse the selection of the appropriate recovery measure within any enclosing atomic action. Figure 2 shows the control flow of the components of an atomic action when the exception handlers for the components cannot recover. A signalled exception ensures that the exceptional control flow is continued by the components that invoked the atomic action.

4) In particular, if any of the components of the atomic action do not have an handler for the exception then those components raise an atomic action failure. (The fact that an exception has been detected elsewhere amongst the processes in an atomic action invalidates the assumptions that any of the processes can terminate normally and provide the appropriate results.) A failure exception could have been signalled explicitly by the exception handlers shown in Fig. 2. Alternatively, the exception handlers might be the default recovery measure which signals a failure exception in response to detecting an exception for which there are no explicit exception handlers.
Atomic Action

<table>
<thead>
<tr>
<th>exceptional flow</th>
<th>signalled exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>normal</td>
<td>suspended flow</td>
</tr>
</tbody>
</table>

flow

<table>
<thead>
<tr>
<th>exceptional flow</th>
<th>signalled exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>normal</td>
<td>suspended flow</td>
</tr>
</tbody>
</table>

flow

Fig. 2. An example of returning an abnormal response or failure from an atomic action.

BACKWARD RECOVERY
IN ATOMIC ACTIONS

Any notation for specifying backward error recovery should define an atomic action within which the recovery, if necessary, should occur. Atomic actions have been represented by programming notations in many ways (Kim, 1982; Liskov, 1982; Lomel, 1977; Shrivastava and Banatre, 1978). We suggest that the key property of an atomic action is the fact that it restricts the sharing of state information between concurrent processes. The activity and results of the atomic action are isolated from the rest of the system for the duration of the action. It is this isolation which is the essence of an atomic action. We can extend the framework of exception handling to support backward error recovery in asynchronous systems by encapsulating it within a structure derived from atomic actions.

The conversation (Randell, 1975) is an extension of the recovery block (Horning and colleagues, 1974) which co-ordinates backward error recovery for concurrent processes by only permitting interprocess communication within an atomic action. For the duration of a conversation, it must be possible to restore the state of any participating process to that which was current on entry to the conversation. Exiting from a conversation is synchronized; all processes must leave simultaneously. No state information is retained after the conversation has successfully terminated. Linguistic frameworks for conversations have been developed and these may impose further restrictions (Anderson and Knight, 1983; Kim, 1982; Russell and Tiedeman, 1979). In particular, it can be argued that in order to simplify the organization of recovery the structures defined by conversations should be completely predetermined, rather than established dynamically by the processes when they need to communicate. Conversations restricted in this way are known as "dialogues" (Anderson and Moulding, 1983) and have been used in the implementation of a naval command and control system.

An alternative basis for a notation for atomic actions is the concept of a shared instance of an abstract data type. Such a shared instance would retain state information between atomic actions. We will refer to the realization of abstract data types in a concurrent environment as "encapsulated data". Note that the operations on encapsulated data can be executed concurrently. The activity of processes executing these operations should be structured to form an atomic action. Thus, processes would be isolated from the rest of the system while they operate on the encapsulated data. This approach is consistent with several existing proposals (Kim, 1982; Liskov, 1982; Shrivastava and Banatre, 1978).

Fault tolerance, in the form of backward error recovery, can then be associated with the operations on encapsulated data. An abstract data type can be specified by a data type invariant (Jones, 1980) together with the pre- and post-conditions of the operations on the data type. The invariant is a predicate on the state of the data type between operations which is true for all valid internal states and false otherwise. This invariant could be evaluated at the completion of the operations on the data type in order to detect errors. Should an error be detected during the execution of the operations, any fault tolerance measures which are invoked will be invisible to the system of which the encapsulated data is a part. The fault tolerance measures should, in principle, involve all of the operations whose activities constitute the atomic action. Thus, the encapsulated data and its operations form a fault-tolerant asynchronous subsystem.

RECOVERABLE OBJECTS

For the purposes of discussion, we shall describe a notation which defines a "recoverable object". The notation is based on the concept of shared encapsulated data and has been implemented experimentally in Distributed Path Pascal (Campbell, 1983) (a programming language which supports concurrent processes, shared encapsulated data, and distributed processing over a local area network). Although the recoverable object (Schmidt, 1983) is an extension to a Path Pascal object, it can also be thought of as a "recoverable" abstract data type. The state of a recoverable object is represented by the internal variables of that object (which may themselves be recoverable objects). Entry procedures and functions constitute its operations. To
detect errors in the object, each recoverable object contains a boolean function which evaluates the invariant for the object. This function serves as part of an acceptance test which is applied after the execution of any operation to determine whether its results are correct. Since the invariant should only test the state of the object to determine whether it is valid, the function is constrained so that it cannot modify that state. Any operation may also incorporate a specific acceptance test for the values of any arguments which it receives or returns.

The structure of a recoverable object is, in a Simula class-like notation:

```
object (*Recoverable Object*)

  ensure

  synchronization_of_ops;
  (*path expression*)
  defn_of_local_variables;
  list_of_ops_and_their_params;
  initialization_for_local_vars;

  invariant boolean_function_defn;
  by (*a routine for each op*)

  list_of_routine_defns;

  else by (*an alt. routine for each op*)

  list_of_alternate_routine_defns;

  else by

      ...
      list_of_alternate_routine_defns;

  else error;(*signal failure exception*)

end (*Recoverable Object*);
```

An object is considered to be idle when none of that object's operations are being executed. (Execution of the operations on an object may be synchronized by an Open Path Expression (Campbell and Kolstad, 1979); both concurrent and sequential execution of the operations can be specified. Operations updating primitive objects must be sequential.) Prior to the execution of an entry routine of an idle object, a recovery cache (Hornig and colleagues, 1974) is established. Any variables that are changed during execution of the operation have their original values stored in the cache. Once the recovery cache is established, other routines of the object may be executed concurrently (subject to the constraints of the path expression) and the prior values of any variables changed by these routines are also recorded in the cache. Routines in a recoverable object are not allowed to return values to the calling environment until all routines have finished executing.

When all the routines have completed their execution, the invariant is evaluated as well as any individual acceptance tests of the routines. If no errors are detected by the acceptance tests or the invariant, then the recovery cache is discarded and the routines return with their results. If, however, the invariant or an acceptance test fails, the cached values of the internal variables are restored and alternate routines for each executed routine are invoked. Any routine of the object can also invoke recovery by executing a standard procedure "error", by attempting to perform an invalid instruction such as divide by zero, or by invoking an operation on another recoverable object which signals a failure exception.

Recovery commences by suspending the activities of all of the routines performing operations on the recoverable object. If the alternate operations fail to satisfy the invariant or raise further errors, recovery is invoked again. This time, the second alternates will be attempted. This process continues until either all the operations finish normally and the invariant does not detect an error, or one or more routines run out of alternates. In the former case, all the operations return normally; in the latter case, all operations will signal a failure exception to their respective calling routines. If a new operation is to be performed while alternate routines are being executed, then the alternate routine for that operation must be executed.

To enforce atomicity, the passage of information in and out of the object is prevented during the execution of its operations. Information can only enter the recoverable object via the parameters of an operation before the routine which performs that operation starts executing. Information can only leave the recoverable object if all the operations have successfully completed. In this way, only validated results are passed out of the object. Thus, the conversation and recoverable object are based on similar forms of atomic action.

**DISCUSSION**

Recovery blocks, dialogues, recoverable monitors, and recoverable objects are all particular examples of associating backward error recovery with a programming mechanism for defining atomic actions. The recovery block provides error recovery for a sequential process while the recoverable monitor provides error recovery for a sequential operation on an encapsulated set of variables. A dialogue provides recovery for a fixed set of concurrent processes while a recoverable object provides recovery for a variable number of concurrent processes.
manipulating encapsulated data. All of the techniques isolate the effects of an activity for the duration of the activity. The backward error recovery mechanism in all the approaches is provided by the use of a caching scheme. The major differences between the techniques are the degree and form of the constraints they impose on information exchange between processes.

Most existing backward and forward error recovery notations restrict concurrency (for example, dialogues) or even enforce sequentiality (for example, in monitors). Mechanisms that do not constrain concurrency unduly but allow the construction of atomic actions are often difficult to integrate directly into existing programming languages. For example, programming language notations have yet to be devised to take advantage of the concurrency permitted by two-phase commit protocols or chase protocols.

The selection and design of acceptance tests and invariants is of great importance to the successful construction of fault-tolerant systems because of the crucial role they play in error detection. Ideally, adequate error detection facilities should ensure the detection of every detrimental consequence of any fault in the system ( Pest and Cristian, 1981). Further research may enhance verification techniques to allow a mechanical confirmation of adequate detection facilities. However, there are numerous difficulties that must be overcome before it becomes feasible to formally verify such properties of asynchronous systems. The simplest schemes for specifying atomic actions may well be the best.

Techniques for designing forward error recovery in asynchronous systems should exhibit the same fundamental dependence on atomicity as does backward error recovery. Indeed, the two principles described above were derived from an examination of the use of forward error recovery in concurrent systems (Campbell and Randall, 1983). Several attempts at providing exception handling in a concurrent programming language suffer from the inadequacy of their mechanisms to enforce isolation (for example, Ada and MPSA).

CONCLUSIONS

The design of fault tolerant software for an asynchronous system can be a complex and difficult task. A reduction in complexity can be expected if atomic actions are used to structure the activity of the system. However, atomic actions are merely a concept for system structuring; they can only be used to build practical systems when a suitable notation is available.

Ideally, such a notation should:
* make apparent the structures which it defines;
* clearly delineate the constraints imposed upon communication;
* enable forward and backward error recovery measures to be easily incorporated;
* be convenient for use by system implementors and facilitate inspection of the system design;
* integrate well with existing concurrent system environments;
* be amenable to formal verification techniques.

We argue for an effective notation in which to express atomic actions. We are convinced that this would be a major contribution to the development of fault tolerant software.

Acknowledgments. This paper was written while F. H. Campbell was a Visiting Senior Research Fellow of the Science and Engineering Council of Great Britain at the Computing Laboratory, University of Newcastle upon Tyne. The authors are very grateful to George Schmidt for his contributions to the development of this paper.

REFERENCES


