Abstract:

Basic Error recovery problems between interacting processes are first discussed where the desirability of having separate recovery mechanisms for co-operation and competition is demonstrated. The paper then concentrates on the recovery mechanisms for processes competing to use the shared resources of a computer system. Appropriate programming language features are developed based on the class and inner features of SIMULA, and on the structuring concepts of recovery blocks and monitors.
BANATRE, Jean-Pierre,


Newcastle upon Tyne: University of Newcastle upon Tyne Computing Laboratory, 1977.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 99).

12"

Added entries

SHRIVASTAVA, Santosh Kumar
UNIVERSITY OF NEWCASTLE UPON TYNE,

Suggested classmarks (primary classmark underlined)

Library of Congress:
Dewey (17th):
U.D.C.: 001.64404 681.322.06

Suggested keywords
CONCURRENT PROCESSES
ERROR RECOVERY
MONITORS
OPERATING SYSTEMS
PROGRAMMING LANGUAGES
RECOVERY BLOCKS
RELIABLE PROGRAMS
RESOURCE ALLOCATION
SOFTWARE REDUNDANCY

Abstract

Basic error recovery problems between interacting processes are first discussed where the desirability of having separate recovery mechanisms for co-operation and competition is demonstrated. The paper then concentrates on the recovery mechanisms for processes competing to use the shared resources of a computer system. Appropriate programming language features are developed based on the class and inner features of SIMULA, and on the structuring concepts of recovery blocks and monitors.

About the Author

Mr. Banatre was a visiting member of the Computing Laboratory, supported by I.R.I.A., during 1975-1976. He is now with the University of Rennes, France.

Dr. Shrivastava has been a research associate in the Computing Laboratory since August 1975. Prior to that he was with the Plessey Co. Ltd.
RELIEABLE RESOURCE ALLOCATION BETWEEN UNRELIABLE PROCESSES

by

J-P. Banatre and S.K. Shrivastava

University of Newcastle upon Tyne, England.

Abstract

Basic error recovery problems between interacting processes are first discussed where the desirability of having separate recovery mechanisms for co-operation and competition is demonstrated. The paper then concentrates on the recovery mechanisms for processes competing to use the shared resources of a computer system. Appropriate programming language features are developed based on the class and inner features of SIMULA, and on the structuring concepts of recovery blocks and monitors.

Key Words and Phrases: programming languages, reliable programs, software redundancy, concurrent processes, error recovery, recovery blocks, resource allocation, monitors, operating systems.

the several research programs, particularly the Center for Corporate Information and the Engineering Research Center. The Center for Corporate Information and the Engineering Research Center are the principal research institutes at the University of California, San Diego. The Center for Corporate Information is located in the Engineering Research Center and provides services to the engineering community. The Engineering Research Center is located in the Engineering Research Center and provides services to the engineering community.

The engineering research community at the University of California, San Diego is very active and involves a large number of researchers and students. The Center for Corporate Information and the Engineering Research Center are located in the Engineering Research Center and provide services to the engineering community.

The engineering research community at the University of California, San Diego is very active and involves a large number of researchers and students. The Center for Corporate Information and the Engineering Research Center are located in the Engineering Research Center and provide services to the engineering community.

The engineering research community at the University of California, San Diego is very active and involves a large number of researchers and students. The Center for Corporate Information and the Engineering Research Center are located in the Engineering Research Center and provide services to the engineering community.

The engineering research community at the University of California, San Diego is very active and involves a large number of researchers and students. The Center for Corporate Information and the Engineering Research Center are located in the Engineering Research Center and provide services to the engineering community.

The engineering research community at the University of California, San Diego is very active and involves a large number of researchers and students. The Center for Corporate Information and the Engineering Research Center are located in the Engineering Research Center and provide services to the engineering community.

The engineering research community at the University of California, San Diego is very active and involves a large number of researchers and students. The Center for Corporate Information and the Engineering Research Center are located in the Engineering Research Center and provide services to the engineering community.

The engineering research community at the University of California, San Diego is very active and involves a large number of researchers and students. The Center for Corporate Information and the Engineering Research Center are located in the Engineering Research Center and provide services to the engineering community.
1. Introduction

The realization that even well designed and tested software is likely to contain residual errors has increasingly led designers to consider the application of redundancy techniques to software construction. Recently a program structure called a recovery block has been developed that allows redundancy, in the form of standby spares, to be added systematically and efficiently to computer programs to make them more reliable [1,2]. The essence of this scheme is that it provides a facility for a computation to backtrack to an earlier state, if an error is detected, and proceed again using a possibly different algorithm.

In this paper we extend this idea to apply to the error recovery problems of concurrent processes of an operating system sharing the limited resources of a computer system. A very general overview of the problem we wish to tackle here can be obtained by considering the progress of a process in an operating system. Suppose during its 'forward motion' the process is generating results entirely by assignments to variables in its private space. If an error is detected, the 'reverse motion' of this process to a prior state is easily performed — the undoing of assignments is equivalent to restoration of prior values. However, the actions of a process can be quite diverse — e.g. control of a peripheral. In general then, during its forward motion this process will generate results by recording them in various resources — storage locations, input–output equipment etc. Since many of the resources involved will usually be shared between processes, the process under consideration will occasionally be interacting with other processes during its progress. If an error is detected, the reverse motion of this process is no longer as easy as before and it may become necessary to provide algorithms for undoing the effects of previously done operations. Just as the process interacts with other processes during its forward motion, it may also interact during its reverse motion. It is thus seen
that when programming for processes that are capable of backtracking, apart from programming for their normal forward progress, we must also be prepared to program for their reverse progress. The important point to note is that appropriate programming language tools must be provided to cope with this additional complexity in a systematic manner, otherwise resulting programs are likely to be even less reliable than versions with no redundancy. We believe that the programming language features developed in this paper meet the above criterion; however, the reader must be the ultimate judge.

The paper is structured into seven sections. Since recovery blocks play a prominent role in this paper, in section 2 we give a brief description of them and of the associated recovery cache mechanism. It is necessary to understand the main error recovery problems between interacting processes before appropriate programming language features can be developed. For this reason, we discuss the recovery problems in section 3 and then precisely define, in section 4, the subset of the problems for which language features are to be developed. In section 5, we describe how recovery blocks can be introduced in resource allocation algorithms implemented using monitors. A program structure called a port is developed in section 6. A port provides facilities for specifying how a resource should be used and what recovery actions a reversing process should undertake. Finally, in section 7 we briefly discuss some implementation details and summarise the work presented.

2. Recovery Blocks and Recovery Cache

A recovery block consists of a conventional block which is provided with a means of error detection (an acceptance test) and zero or more standby spares (alternates). Its structure is shown below:
ensure acceptance test by
  alternate 1 or primary block
else by
  alternate 2
...
else by
  alternate n
else error;

Figure 1.

As discussed in [1,2], the alternates represent different algorithms for producing acceptable results, with alternate 1 (primary block) representing the preferred algorithm. After the execution of the primary block, the acceptance test (a boolean expression) is evaluated to check that the results produced are acceptable. If so, the statement following the recovery block is executed; otherwise the state of the computation is restored to that just before entry to the recovery block and the second alternate is tried and so on. If during the execution, some error is detected (e.g. divide by zero) then this is also regarded as the failure of the current alternate and the same recovery actions are taken as in the case of the failure of the acceptance test. If all the alternates fail then it is regarded as the failure of the current recovery block; further recovery may be performed by the enclosing recovery block (recovery blocks can be nested to any degree).

State restoration is achieved with the help of a device called the recovery (or recursive) cache. Since recovery blocks may be nested, the cache is organised as a stack and contains entries for recovery blocks entered but not yet exited. It is not necessary to save the entire environment at each recovery block entry; all that is needed is a mechanism that treats non local variables as follows: the value of a non local variable is cached just before the first update in that recovery block is performed – subsequent updates to that variable
in that recovery block need not be cached. To undo the effects of operations other than assignments, the designers of the recovery block scheme presented their initial ideas on the concept of 'recoverable procedures' [1]. Such a procedure has a 'normal' procedure body that performs a given operation and a 'reverse' procedure body to be executed when reversal of that operation is desired. These ideas are developed further in this paper.

At this point we introduce a simple facility (not present in the original proposals [1,2]) in the recovery block scheme which will turn out to be of great use when programming reverse operations. We assume that with each process in the system a boolean flag 'errorflag' is associated that indicates whether this process is going forward or backward. It is the responsibility of the recovery cache of a process to update this flag properly. We make this flag accessible to programs in a read only mode.

Complete details of the recovery block scheme and its cache are presented in [1]. Many of the theoretical ideas on structuring complex systems using recovery blocks are given in [2] while practical details of an implementation of simple recovery blocks are presented in [3].

3. Error Recovery Problems between Interacting Processes
3.1 General

Concurrent processes in an operating system are said to be loosely coupled, i.e., most of the time their activities are independent of each other but sometimes they exchange information. Consider now the recovery problems associated with process interaction. Assume that a process passes some information to some other process(es) (i.e., sends a message or modifies the environment common to them). While this operation may be 'acceptable' to the information generating process, in general we cannot afford to throw away the recovery information associated with this operation (i.e. the cache entries corresponding to this operation which we assume is inside a recovery block) until the other concerned processes
have 'accepted' the passed information. These processes will however react in their own time; thus the recovery information may have to be preserved for a long time. What should the first process do in the meantime? On the one hand if it were allowed to proceed with its computation, during which there may be further interactions involved, then there is the risk of domino effect [2], i.e. a reversal in one process may give rise to an avalanche of reversals in other processes. On the other hand, if its progress were halted, pending acceptance of the sent information by the receivers, then there is the risk of seriously reducing the asynchronism in the system. It is thus seen that the control of processes and management of recovery information can become an extremely complex problem and a general solution does not appear to be possible. What seems possible however is to develop mechanisms for different classes of interactions. We have found that the classification of interactions into interference, co-operation and competition, as defined in [4] to be the most suitable. Interference includes those interactions that are unacceptable or unanticipated. Mechanisms which allow processes to operate on shared data without interference (e.g. P and V operations on semaphores) are simple and implemented at a very primitive level, we shall assume here that they operate reliably. Co-operation occurs when processes explicitly wish to exchange information with each other while competition occurs when processes have no explicit desire to exchange information but nevertheless they do so as they must share resources. These two forms of interaction are discussed below in some detail.

3.2 Co-operation and Competition

Consider a conceptually two level system in which the lower level manages all the resources to be shared by the processes of the system (see fig. 2). To be specific, we assume that the resource allocation algorithms have been implemented using monitors [6]. So the processes wishing to acquire/release
these resources invoke appropriate monitor procedure calls. In the lower level, the processes are competing for the shared resources. When these resources are

\[
\begin{align*}
\text{no interaction} & \quad \text{co-operation} \\
\begin{array}{c}
P_1 \quad \cdots \quad P_j \quad \cdots \quad P_n \\
\end{array} & \quad \begin{array}{c}
P_1 \quad \cdots \quad P_j \quad \cdots \quad P_n \\
\end{array} \\
\text{shared resources} + \quad \text{competition} & \quad \text{shared resources} + \quad \text{between } P_1..P_n \quad \text{their monitors} \quad \text{their monitors} \\
\text{their monitors} & \\
\end{align*}
\]

Fig. 2 (a) \quad \text{Private use of shared resources.} \quad \text{Fig. 2 (b) \quad Shared use of shared resources.}

used privately by processes (fig. 2(a)) then, at the higher level, these processes appear to be logically independent: the conceptual interface hides the competition. On the other hand, processes of fig. 2(b) are co-operating, as the acquired resources are explicitly being used for information exchange (as shown by the heavy arrows). We thus see that competition is a lower level activity with respect to co-operation. We consider first the recovery problems of co-operating processes.

In general, co-operating processes are capable of exchanging arbitrary information (in the sense that a process can update a shared file in a manner it thinks fit or can send any message to other processes). This implies that in general, only the receivers can verify the sent information. For this reason, in the conversation mechanism of [2], when processes enter a conversation (i.e. start exchanging information inside a common recovery block), they are allowed to exit from it only when all the concerned processes pass their acceptance tests (meaning, they are satisfied with the information exchange).

Obviously, this restriction can reduce the degree of asynchronism in the system. As an example, consider the simple case of error recovery between

6.
producing and consuming processes coupled via a bounded buffer as described in [5]. If the 'production of a message' and its 'consumption' is programmed as a conversation, then it is no longer possible for the producer to race ahead of the consumer. Whether the conversation mechanism can be complemented by other mechanisms, which would allow the above producer to race ahead, remains to be seen and we are currently investigating this area.

Considering competition next, we find that, processes using a given monitor are interacting only through the monitor's shared data. This case is much more easy to deal with than co-operation for the following reason: the only information that processes exchange is that necessary to achieve harmonious resource sharing and this is determined by the monitor procedures. Thus it is no longer possible for processes to exchange arbitrary information using a monitor's shared data. As a result, when a process performs an operation on a monitor's shared data (i.e. executes a monitor procedure body) it should be in a position to assert the 'global acceptability' of the shared data. This means that this form of interaction need not be structured as a conversation. We have exploited this property in the design of recoverable monitors to be described later.

4. Recovery Requirements for Competing Processes

In this paper we are concerned with competing processes only, i.e. processes of fig. 2(a). We can make the system of fig. 2(a) reliable by systematically introducing redundancy using the recovery block approach. Reliability is achieved in the lower level by making resource allocation algorithms reliable - this is done by using recoverable monitors (clearly, it is also necessary to have hardware redundancy, but here we ignore this aspect and concentrate only on software redundancy). In the upper level, the codes of the processes will also contain redundancy. Since at this level these processes are logically independent, their recovery actions are also independent. If a process, after acquiring and using
a resource, wishes to backtrack to an earlier state where this resource was not acquired then it is necessary to (i) release the resource and (ii) to undo any effects owing to the use of the resource. We study these recovery actions in a greater detail below.

Fig. 3 shows a recovery block where 'units' is a recoverable monitor (i.e. a monitor whose procedure bodies are coded as recovery blocks - precise details of which are to be discussed in section 5) managing a pool of units - a process can acquire a unit, use it and release it. The actual use of the acquired unit is performed inside a forward procedure 'q' which specifies how the resource is to be used. There is also a backward procedure 'undoq' that specifies how to undo any effects of the resource use (precise details of the program structure that provides facilities for defining such procedures need not concern us just now; they will be discussed in section 6). We see that the execution of 'q' is preceded by a 'prelude' (concerned with resource acquisition) and followed by a corresponding 'postlude' (concerned with the release of the resource).

\[
\text{forward procedure } q(...) ;
\begin{align*}
\text{begin } & \text{.. use of one unit .. end} \\
\text{backward procedure } undoq ; & \text{begin } \text{.. undo the effects of use .. end} \\
& \text{\textbf{ensure} \text{<at> by} "recovery block \text{R}"} \\
& \text{\textbf{begin} "first alternate"} \\
& \text{units.acquire(..);"prelude" } \leftarrow (a) \\
& \quad \ldots \leftarrow (b) \\
& \text{q(...); } \leftarrow (c) \\
& \quad \ldots \leftarrow (d) \\
& \text{units.release(...);"postlude" } \leftarrow (e) \\
& \quad \ldots \leftarrow (f) \\
& \text{\textbf{end else} by} "\text{second alternate}" \ldots
\end{align*}
\]

Figure 3.

Let an error occur while performing an operation. We consider recovery actions other than state restoration that might be needed.
Error at point 'a': This means that all the alternates of procedure 'acquire' of the recoverable monitor 'units' have failed and obviously this gives rise to the failure of the first alternate of recovery block R; the next alternate must now be tried - no special recovery actions are necessary (note that if 'acquire' is successful, i.e. no error at 'a', the net result is the appropriate update of the monitor variables).

Error at point 'b': the problem here is how to undo the effects of 'acquire'. Clearly it is possible to restore the state of the monitor variables but this would mean backtracking all those processes that have performed operations on 'units' after the process under consideration performed 'acquire'. This is not necessary since simple reasoning tells us that all that is required is to release the acquired unit. A formal justification for this reasoning is as follows: a monitor (or any other SIMULA-like class object) is an abstract data type providing abstract operations over an abstract space - monitor variables being a concrete representation of this space. It is the abstract state of the object that is of concern to a calling process and it is only necessary to restore the abstract state when reversing - this does not necessarily mean restoring the concrete state. Thus, by releasing the unit (i.e. by calling 'release') we ensure that the operation 'acquire' can continue to provide the abstraction 'a unit will be made available within a finite time'. To summarise, the backtracking process must execute the postlude.

Error at point 'c': A call on a forward procedure may be regarded as performing an indivisible operation that either produces the necessary side effects or does not. Thus if an error return is obtained, nothing needs to be done as far as 'q' is concerned ('q' has not produced any side effects). The recovery action is the same as at 'b' discussed previously.
Error at point 'd': The backward procedure 'undoq' must be executed to undo the effects of the use of the acquired unit. As an example, consider that forward procedure is for printing a file on the acquired line printer. Then the backward procedure for 'unprinting' might be to send a message to the operator's console to ignore that printed file. This example also illustrates that resources needed for 'undoing' may not be the same as that for 'doing'. After executing the backward procedure 'undoq', further recovery actions needed are the same as at 'b'.

Error at point 'e': This error, implying that resources cannot be returned, is fatal. This is because if the process is allowed to backtrack, it will eventually try to undo the prelude by executing the postlude (which it was unable to execute in the first place). A failure in a postlude is therefore regarded as a collapse of the recovery mechanism and the system is aborted.

Error at point 'f': Since the postlude has been executed, the only effect that needs undoing is that due to 'q' – this may be done by calling 'undoq'. Provision must be made for acquiring the necessary resources needed for undoing (in particular, it may be necessary to reacquire the released resource).

Error while backtracking: Any error while backtracking is regarded as fatal and the system is aborted.

We conclude this section by pointing out an important class of errors which can occur if proper program structuring is not used. In fig. 4(a), a buffer is acquired from a shared pool of buffers and its release is done inside a recovery block.
b:buffer;
...
pool.acquirebuffer(b);
put some information
in the acquired buffer;
ensure <at> by
begin
...
pool.releasebuffer(b);
...
end else by
begin
...
pool.releasebuffer(b);
...
end ...

b:buffer;
...
pool.acquirebuffer(b);
put some information
in the acquired buffer;
ensure <at> by
begin
...
end else by
begin
...
end ...

else error
pool.releasebuffer(b);
...

Figure 4(a)                 Figure 4(b)

If an error occurs at the point shown by the arrow (i.e. after the release),
the execution of the alternate is meaningful only if the released buffer is
acquired again, however, this cannot be guaranteed. This problem is not
present in fig. 4(b) where the acquired resource is released outside the recovery
block concerned with its use.

We have identified the required recovery actions for a process using the
shared resources of the system. From this discussion it is clear that if we can
syntactically specify 'prelude', 'use', 'unuse' and 'postlude' then it is
possible to design an appropriate recovery strategy. It is then only necessary
to extend the recovery cache mechanism such that cache processing includes this
strategy. A program structure called port is developed in section 6 that provides
the necessary language features while the extension of recovery cache is
discussed in section 7. But first we study a method of making resource allo-
cation algorithms reliable.
5. **Recoverable Monitors**

Recoverable monitors are a means of adding redundancy to the resource allocation algorithms. Bearing in mind the discussion on competition in section 3.2, we can say that the acceptance test of a monitor entry procedure (i.e., a procedure which may be called from outside the monitor) should test for the 'global' acceptability of that operation, i.e., test that not only is the result of that operation acceptable to the calling process, but also that it will be acceptable to subsequent calling processes. The state of a monitor after the acceptance test of one of its operations has been passed is called its consistent state. Briefly stated then, a recoverable monitor is used by processes as follows: the calling process will expect to find the monitor in a consistent state (say \( \delta_1 \)); after the execution of the appropriate procedure body, assuming it passes the acceptance test, it will leave the monitor in a (possibly different) consistent state (say \( \delta_2 \)). In case the test fails (or an error is detected during execution), \( \delta_1 \) is restored and an alternate is tried; if none exists then the process will get an error return and will take its own appropriate recovery action. We can ensure that a calling process finds a monitor in a consistent state by making sure that (1) a monitor is initialised to a consistent state, and (2) the executing process leaves the monitor in a consistent state before any other process is given entry. Some care is needed in order to observe the second condition. Fig. 5 shows an alternate of a monitor procedure body with a synchronising operation such as a 'wait' or 'resume process'. Assume also that \( s1 \) and \( s2 \) are statements updating monitor variables.

```plaintext
ensure <at> by
... else by
begin s1;
    wait/resume operation;
    s2;
end else by
...
```

**Figure 5.**
The danger now is that, as a result of the synchronising operation, control may be given to some other process before the state of the monitor has been validated by the acceptance test. For example, if the synchronising operation is a 'signal' [6], then the awakened process (if any) will immediately be allowed to enter the monitor. It is thus clear that such operations must not be performed from inside a recovery block. A recovery block used in a procedure with synchronising operations must be placed as shown in fig. 6 where, operations inside square brackets are optional and 's1' is an operation without side effects.

```
entry procedure proctype(...); begin
  <variable declarations>;
  [s1];
  ['wait operation']
  <recovery block>;
  ['resume process operation']
end;
```

Figure 6.

Fortunately, in a majority of cases, as the examples of [6,7] show,'wait' operation is the first operation (before side effects are produced) and 'resume' operations are the last operations in a procedure, so the above structuring should not prove to be too restrictive. The proof guided methodology presented in [6,7] also indicates how acceptance tests may be constructed. In section 7 we briefly discuss recovery cache details to support these monitors.

6. Ports

A program structure called port is developed here which, as the name suggests, acts as a gate through which one or more ways of using a resource are made available to a process. In this section we shall see how the language features for recovery actions, discussed in section 4, are incorporated in this program structure.
A systematic method of allowing processes to use the shared resources of
the system is to create virtual resource objects out of real resources; a
process can then create a 'local instance' of a given virtual resource object
when it wishes to use that resource. Hoare has shown that the SIMULA language's
class and inner concepts [8] can be elegantly used for the above purposes [9].
The port structure to be described in this sub-section follows directly from
his ideas; however, the application of these ideas to recoverable processes is
believed to be new.

Let us suppose that we wish to create a virtual disk; we will consider
the write operation of this disk in more detail (a Concurrent Pascal [10] like
language is assumed). Let 'diskinout' be a class that defines all the control
operations on a disk:

```pascal
    type diskinout class;
    begin
        machine code routines to perform disk head
        movement, reading, writing etc.
    end
```

For this disk, a resource allocation algorithm is implemented that allocates
diskpages (where a diskpage is a sector on a track) to requesting processes and
also controls the movement of diskheads as suggested in [6] (recovery blocks in
the procedure bodies are not shown):

```pascal
    type diskresource = recoverable monitor;
    begin
        var pageset: set of diskpage; "pool of free pages"
        ... other variables ...
        entry procedure move (p:diskpage);
            {use algorithm of [6] to queue the request
             to move heads to the track of 'p'}
        entry procedure move to write(var p:diskpage; var
            found: boolean);
            {if pageset not empty then {acquire a free page and queue
             the request for head movement; found := true} else found := false}
        entry procedure releasepage (p:diskpage)
            {pageset := [pageset] + [p]}
        entry procedure releasehead;
            {use algorithm of [6] to service the next request}
    end diskresource;
```
A class 'diskcontrol' can now be written that provides virtual operations through ports (entry means that instances of that type can be created outside). Here, 'writepage' provides a 'write' operation for writing on the acquired page; 'readpage' provides a 'read' operation; 'releasepage' is a port for releasing a page (this is needed, for example, when a process wishes to delete a file) and 'update' provides a 'rewrite' operation for updating a previously acquired page. Only the 'writepage' port is programmed, the rest are similar.

```pascal
begin type diskcontrol=class;
    begin var inout:diskinout;
        resource:diskresource;
        entry type writepage=port;
            begin var p:diskpage;found:boolean;
                entry procedure write(c:corepage;var possible:boolean;
                                          var pp:diskpage);
                    begin
                        possible:=found;if found then begin
                            use 'inout' to write from 'c' to 'p';
                        end
                        pp:=p;end;
                    end write
                    resource.move to write (p,found);"prelude"
                    inner;
                    resource.releasehead;"postlude"
                    end writepage;
                entry type readpage=port(p:diskpage);
                    begin .... end readpage;
                entry type releasepage=port;
                    begin .... end releasepage;
                entry type update=port(p:diskpage)
                    begin .... end update;
            end diskcontrol;
end;
```

Assuming there is one disk unit, an instance of type 'diskcontrol' will be declared global to all user processes:

```pascal
diskuse : diskcontrol ;
```

and a process wishing to perform a write operation will proceed as follows (where 'c' has been declared as a corepage and 'f' a boolean and 'p' as a diskpage):

```pascal
... using diskwrite: diskuse.writepage do
begin ... diskwrite.write(c,f,p); ... end ; ...
```

15.
A local instance of port 'writepage' is created and then its 'write' operation is called. The term using emphasises the fact that, because of the inner statement, the execution of the statement following do will be enclosed by the prelude-postlude of the port. We thus see that a port can be regarded as the 'modus operandi' of a resource, taking on the responsibility of acquiring and releasing that resource and providing appropriate operations for the use of the resource. A clear separation between resource acquisition and resource use now allows us to introduce the kind of error recovery discussed in section 4.

6.1 Recovery Features in Ports

We shall now see how to make the 'writepage' port recoverable. The 'use of the resource' is programmed as a forward entry procedure ('forward' prefix says that only a forward going process can call that procedure) and 'unuse of the resource' is programmed as a backward entry procedure ('backward' prefix says that a process, only while recovering, can call that procedure). In our example we can assume that 'unuse' consists of clearing the written disk page (say, writing all zeros in it). We note also that this 'unuse' requires the disk resource. The port 'writepage' with recovery features is now shown below (to make this example more interesting, we have also put recovery blocks in the bodies of the procedures):
entry type writepage=port;
begin var p:diskpage;found:boolean;
forward entry procedure write (c:corepage;var possible:boolean;
   var pp:diskpage);
begin possible:=found;if found then
   begin pp:=p;ensure good writing by
      begin write from 'c' to 'p' end else by
         retry'execute again first alternate" else error;
   end write;
end write;
backward entry procedure unwrite;
begin var ct:corepage;if found then
   begin initialise ct with zeros;
      ensure good writing by
         begin write from 'ct' to 'p' end else by
            retry else by retry
   end else error;
end unwrite;
resource handling:resource.move to write(p,found);"prelude1"
inner;
   if errorflag & found then resource.releasepage(p);
   resource.releasehead;"the two statements form
   postlude1"
resource rehandling:if found then resource.move(p);"prelude2"
inner;
   if found then begin resource.releasepage(p);
      resource.releasehead;
end "postlude2"
end writepage;

The meaning of the various new constructs is explained with the help of the
program shown in fig. 7 (the particular recovery structure of this figure has
been chosen deliberately to explain the semantics of port, it is not intended
to show a typical use of 'writepage').

ensure AT1 by "recovery block R1"
begin
  ensure AT2 by "recovery block R2"
  begin var c:corepage;found:boolean;p:diskpage;
  using diskwrite:diskuse.writepage do
  ensure AT3 by "recovery block R3"
  begin
    diskwrite.write(c,found,p);
  ⇐ (i)
  end else by
  begin
    ⇐ (j)
    diskwrite.write(c,found,p); ⇐ (k)
  ⇐ (l)
  end else error;
  ⇐ (m)
end else by

17.
begin
  using diskwrite:diskuse:writepage do
  end
  end else by
begin
  end else by

Figure 7

We now consider various possible execution sequences (for failures, only recovery actions other than state restoration are described):

(1) AT3 passed: This means that one of the two alternates of R3 has produced acceptable results. The execution steps involved were: prelude1 followed by the alternate followed by postlude1. As the process is going forward (i.e. errorflag=false), only the diskheads will be released in postlude1 (we assume that a process after acquiring a page, retains it).

(2) Error at point 'i': The second alternate of R3 must now be tried. Before that, the backtracking process calls the backward procedure 'unwrite' of port 'diskwrite' to undo the affects of write (a port contains only one backward procedure, see the appendix).

(3) Error at point 'j': This implies the failure of R3 and the next alternate of R2 must be tried. Before this is done, the process, as it backtracks, releases the resources by executing postlude1 (as errorflag=true, the acquired diskpage will also be released).

(4) Error at point 'k': The process was unable to write properly (from the code of 'write' we see that two attempts were made to write); the recovery action is the same as at 'j'.

18.
(5) **Error at point 'l':** Before the next alternate of R2 can be tried, the backtracking process must undo the effects of 'write' and also release the resources. This is done by calling 'unwrite' and then executing postlude1.

(6) **Error at point 'm':** Before the next alternate of R2 is tried, 'unwrite' is executed between prelude2 and postlude2. The reasoning for this action is as follows: clearly it is necessary to undo the effects of 'write'; but the required resources have been released. The reacquisition of resources (and their release) is specified by the code labelled resource rehandling.

(7) **Error at point 'n':** Recovery actions are the same as at 'm'.

The 'writepage' example also illustrates how the local variables of a port may be used to record information that might be needed when reversing.

6.2 **Some Remarks on the Remaining Ports of 'diskcontrol'**

Rather than programming the remaining ports of diskcontrol (readpage, releasepage and update) we discuss here some of their interesting features and leave the task of programming to the interested reader. The port 'readpage' does not need a backward procedure for undoing the effects of a diskread operation. This is because the recovery cache will automatically store the previous contents of the 'corepage' as it is updated by the diskread operation. An interesting problem arises while programming 'update' port. If we assume that it has a 'rewrite' procedure for overwriting the contents of a disk page, then clearly the previous contents of the disk page must be stored in a local port variable so that the effect of 'rewrite' can
be undone, if desired. From the cache storage's point of view, this may prove to be very costly (especially when updating random access files) and a practical solution is as follows: the operation of 'update' port is made unrecoverable (i.e. no attempt is made to save the previous contents of the disk page; no backward procedure is needed) and recovery is provided at the file level rather than at the disk page level. File level recovery can be provided by ensuring that the filing system always creates a backup file before a user is allowed to update that file. The same arguments hold for the 'releasepage' port.

6.3 Another Example

As another example, consider that we want to make printing of files reliable (especially tolerant against transient printer faults). A port 'printtop' is programmed as shown. It provides a 'printpage' operation for printing a page. A process can acquire the printer, print the desired number of pages (by repeatedly calling 'printpage') and then release the printer. If during printing, the process detects an error (we assume that this activity is being done inside a recovery block) and backtracks, the effect of the printing is 'undone' by the operator message to ignore the number of pages printed for the user (note that backward procedure of a port is called only once while recovering - see the appendix for more details). To guard against unanticipated faults (e.g. accidental switching off of the printer while printing) a timer is used in the 'printpage' and 'newpage' procedures. A 'timeout' exception will cause the failure of the current alternate of the current recovery block of the calling process. Note that 'printpage' and 'newpage' can make use of the usual exception handling techniques to handle anticipated faults (e.g. printer not online).
type printer=class(number:integer)
begin var pc:pcontrol;"a class providing primitive printer operations"
    pa:paccess;"a recoverable monitor for exclusive access
to the printer"

    entry type printop=port(id:name);
    begin var count:integer
    forward entry procedure newpage;
    begin set timer;use 'pc' to prepare the printer to
    start on a new page;clear timer;
    end;
    forward entry procedure printpage(c:corepage);
    begin set timer;use 'pc' to print the contents of 'c'
    on the printer;count=count+1;clear timer;
    end;
    backward entry procedure undo;
    begin using operator:console.output do
    operator.send('ignore any current output of
    user',id,'on printer',number,
    'consisting of count,'pages');
    end;
    resource handling:count:=0;pa.acquire;inner:pa.release;
    resource rehandling:null;inner:null;
    end printop;
end printer;

Let us assume a 'printerprocess' whose job is to print files. We
assume two printers:

    mainprinter:printer(1);auxiliaryprinter:printer(2);

The printer process prints on the 'mainprinter' but if unable to do so,
prints on the 'auxiliaryprinter'. The code of printerprocess is as shown
below. It is assumed that the most likely cause of error is the printer, so
the second printer is tried in the second alternate. If this fails as well,
then a message is sent to the operator (who could ask the user to try later).
The above process should manage to print files despite faults in (i) a
printer, (ii) disk or (iii) the program itself. Assume for example that
the main printer becomes temporarily faulty. Then the printer process will
(i) release that printer, (ii) inform the operator to ignore the partially
printed file and (iii) start printing the file on the auxiliary printer.

21.
"assume a monitor 'r' that maintains a queue of print requests"
filename, username: name;
map: corepage; "to store file map" found: boolean;
length: integer; "indicates file length"

cycle r.getrequest(filename, username);
... using f: filemaster.fileop(username) do
  f.open(filename, map, length, found);
  if found then
    begin ensure <good printing> by
      using pr: mainprinter.printop(username) do
        begin var c: corepage;
          pr.newpage; c:= standard header;
          pr.printpage(c);
          for i:=1 to length do
            begin using dr: diskuse.readpage(map(i)) do
              dr.read(c);
              pr.printpage(c);
            end
        end
        c:= standard tail; pr.printpage(c);
      end else by using pr: auxiliaryprinter.printop(username) do
        <similar code as the primary>
      else by using operator: console.output do
        operator.send('unable to print file', filename, 'of user', username)
      end else error;
  end; ....
end "cycle"

We hope that the reader will now share our belief that ports are a
systematic method of providing reliable, recoverable operations on the shared
resources of a system. Programming language rules for constructing ports are
stated in detail in the appendix. Finally we note that, thanks to the inner
mechanism, resource acquisition - release is performed automatically around
its use - this eliminates the problem illustrated by fig. 4(a) since programs
are restricted to the structure of fig. 4(b).

7. Implementation Notes and Concluding Remarks

We will assume that each process in the system has its own recovery
cache. We have described briefly, in section 2, the cache mechanism (i.e.
how the recovery information is stored in the cache) for the case of
assignments to private variables. The extensions needed to support recoverable
monitors and ports are now described.
(i) **Recoverable monitors:** Upon entry to the recovery block of a procedure (see fig. 6) of such a monitor, copies of monitor's variables are stored in the cache of the calling process in the usual manner and in case the current alternate fails, the cache is used to restore the state of the monitor - again in the usual manner. If the acceptance test is passed, these variables are popped off the cache - they are no longer needed.

(ii) **Ports:** When an object of type port is created inside a recovery block by a process, the cache of the process should record details of its use (e.g. prelude executed, prelude plus a forward procedure executed etc). This can be done by reserving a block of storage on the cache (as soon as a port object is created) where the above mentioned details can be recorded. (Precise rules for caching of port variables and calling of the backward procedure are given in the appendix.) When an error is detected, the port information can be processed to provide the necessary recovery actions (as discussed with referenced to fig. 7).

The ideas presented in this paper are currently being implemented on our PDP11/45; the language chosen to incorporate these ideas is Concurrent Pascal [10]. In this experimental version, recovery caches will be purely software implemented.

To conclude: after discussing the basic error recovery problems between interacting processes, we have developed recovery mechanisms to solve a subset of these problems - that of concurrent processes competing to use the shared resources of a system. Recoverable monitors were developed to make resource allocation algorithms reliable (i.e. to make process interaction for competition reliable). A program structure called port was developed that provided facilities for specifying (i) how a process should call on a recoverable monitor to acquire and release resources (when going forward or backward) and (ii) how should a process 'use' and 'unuse' a given resource. Recoverable
monitors and ports both impose a strict (but conceptually simple) discipline on a programmer on how he should think about resource allocation and recovery; this is essential if the recovery problems are to be kept manageable.

Acknowledgements

This work was carried out as a part of the 'Highly Reliable Computing Systems' project which is supported by the Science Research Council of Great Britain. Discussions with the members of the project during the course of this work were of great value. Our special thanks go to P.A. Lee and Joost Verhofstad. One of us (J.P.B.) was a visiting member of the project during 1975-1976 and was supported by l'Institut de Recherche en Informatique et Automatique (I.R.I.A., France).
APPENDIX

Rules for constructing and using ports

The port schema is shown below for which the following rules apply:

```
[entry] type<name>=[multiple]port(formal parameters);
"multiple and entry are optional features"
begin...local variable declarations...
...procedures/forward entry procedures, one such
forward entry procedure is shown below...
forward entry procedure<name> (formal parameters);
   begin .........end;
...other procedures/forward entry procedures...
[backward entry procedure<name>;] "this procedure is optional"
   begin ..................end;
   resource handling:S1;inner:S2;
   resource rehandling:S3;inner:S4;
"S1 are statements; where, S1 is prelude 1, S2 is postlude 1,
S3 is prelude 2, S4 is postlude 2"
end"of port definition"
```

(1) Only a port is allowed to contain forward procedures and a backward procedure.

(2) The backward procedure is parameterless and can only be called by a process while recovering. Its role is purely to undo the effects due to the calls on the forward procedures (the local port variables may be used to record the information needed by the backward procedure, see the 'writepage' or 'printop' example).

(3) None of the procedures of a port are allowed explicit access to a monitor*. Only the prelude/postlude can contain monitor calls.

(4) A port has explicit access to only one monitor and it can only be accessed as in rule (3).

(5) If p1;p2;...;pn is a prelude then only pn may be a monitor call.

Operations p1;...;pn-1 either produce no side effects or are calls on port operations. Three points may now be noted: (a) the prelude structuring of this rule means that a recovery can be made if an error occurs while executing a prelude; (b) once a prelude has been

* A procedure can of course, declare a local instance of some port and thus have implicit access to the monitor of that port.
REFERENCES


