Abstract:

The programming language Concurrent Pascal has been extended to include some language features that facilitate the writing of fault-tolerant software. As a result, it is possible now to (i) write operating systems with a measure of fault-tolerance, and (ii) for such an operating system to support fault-tolerant user programs. The first part of this report describes these language features and illustrates their use with the help of a few working examples. The second part contains some of the implementation details.

Concurrent Pascal with Backward Error Recovery

By

S.K. Shrivastava

TECHNICAL REPORT SERIES

Series Editor: Dr. B. Shaw

Number 127
October, 1978

© 1978 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.
SHRIYASTAVA, Santosh Kumar.

Concurrent Pascal with backward error recovery.

Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1978.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 127.)

12"

Added entries: UNIVERSITY OF NEWCASTLE UPON TYNE.

Suggested classmarks (primary classmark underlined)

Library of congress:
Dewey (17th) 001.6424 001.6425
U.D.C. 681.322.06

Suggested keywords:
CONCURRENT PASCAL
ERROR RECOVERY
FAULT-TOLERANT SOFTWARE
OPERATING SYSTEMS
PORTS
RECOVERY BLOCKS
RECOVERY CACHE
SEQUENTIAL PASCAL

Abstract

The programming language Concurrent Pascal has been extended to include some language features that facilitate the writing of fault-tolerant software. As a result, it is possible now to (i) write operating systems with a measure of fault-tolerance, and (ii) for such an operating system to support fault-tolerant user programs. The first part of this report describes these language features and illustrates their use with the help of a few working examples. The second part contains some of the implementation details.

About the author

Dr. Shrivastava joined the Computing Laboratory of the University of Newcastle upon Tyne in August, 1975, where he is currently a Lecturer. Prior to that he was with the Plessey Co. Ltd.
Concurrent Pascal with Backward Error recovery: 
Language Features and Examples

By
S.K. Shrivastava
Computing Laboratory,
University of Newcastle upon Tyne,
England.

Summary

The programming language Concurrent Pascal has been extended to include some language features that facilitate the writing of fault-tolerant software. As a result, it is possible now to (i) write operating systems with a measure of fault-tolerance, and (ii) for such an operating system to support fault-tolerant user programs. The paper describes these language features and illustrates their use with the help of a few working examples.

Key words
Fault-tolerant software, operating systems, Concurrent Pascal, Sequential Pascal, Error recovery, recovery blocks, ports.
INTRODUCTION

A computer system may contain faults in either the hardware or the software or both. As a consequence of this, when an error is detected while executing a program, it is often not possible to accurately locate the fault that caused the erroneous situation. When writing fault-tolerant software, (defined to be software that produces acceptable results despite faults in the hardware and software) one recommended practice therefore is to classify erroneous or abnormal situations encountered into those that were anticipated and the rest as unanticipated. For example, suppose a programmer's task is to develop an input-output program module for a certain peripheral. He can classify a number of situations as 'abnormal'—these may include 'peripheral device not connected to the computer', 'parity error' etc. Suppose that his tested and debugged program still contains design faults as a result of which a 'division by zero' exception is detected during the program execution—such a situation will be considered as an unanticipated abnormal situation if no provision for coping with it has been provided. It can be appreciated that special purpose routines can be written for coping with anticipated abnormal situations, and many programming languages provide facilities for writing such 'exception handlers' (e.g. ON units in PL/1). In contrast, any technique for coping with unanticipated situations has to be sufficiently general such that it can (hopefully) deal with all of the erroneous situations that were not foreseen by the programmer. The best known technique for coping with unanticipated erroneous situations is the so called backward error recovery technique which consists of abandoning the state of the computation in progress and 'rolling it back' to a prior state which is believed to be error free. After the roll back, the computation is resumed again with the hope of avoiding the offending fault. Furthermore, if after the rollback, a different algorithm is used for producing results, then a measure of tolerance against design faults can also be obtained.
Some work has been done at Newcastle in the area of constructing fault tolerant software that makes use of backward error recovery\textsuperscript{4,5,6}. This paper describes further effort in this direction: the programming languages Concurrent and Sequential Pascal\textsuperscript{2,3} have been extended to include some backward error recovery features. As a result it is now possible to write operating systems with a measure of fault-tolerance and with the capability of supporting fault-tolerant user programs. In this paper, these language features are described and their use illustrated with the help of a few working examples. A companion paper\textsuperscript{10} describes the implementation of these features.

**LANGUAGE FEATURES FOR ERROR RECOVERY**

**Design decisions and objectives**

Backward error recovery involves the restoration of a prior state. A decision was taken that the state to which a computation should be backtracked will be specified by the programmer - the term recovery point will be used to refer to such a state - and that the task of restoring the state will be performed automatically by a recovery system that is part of the system that is supporting the computation. In order to appreciate what is involved when state restoration is performed, consider the progress of a computational process supported by an operating system. If the actions of the process, after a recovery point has been specified, is to produce its results entirely by updating its private variables, then the act of state restoration involves undoing the updates which merely consists of restoring the prior values of the updated variables - the recovery system can be designed to perform this task easily. However, the actions of a process can be arbitrarily complex; suppose the process is involved in the control of some input-output equipment. Under such a situation it may become necessary for the programmer to specify what should constitute the undoing of the input-output control operations (e.g. if the action was unwind a tape for accessing a particular file, the action needed for undoing might be to rewind the tape). A design decision was therefore taken that for all the actions other than assignments, for
which recovery is desired (actions for which recovery is provided will be termed recoverable actions) the programmer must specify the appropriate routines for undoing the effects resulting from the actions; it will be the responsibility of the recovery system to appropriately invoke these programmer specified recovery routines when state restoration is required.

So far, the state restoration of a single process has been discussed. For interacting processes, errors can propagate from one process to other; this may mean that when an error is detected, it becomes necessary to restore the state of a group of interacting processes. Many problems arise when state restoration of a group of processes is required; it is sufficient to note here that such a situation should be avoided as far as possible. There are, fortunately some important cases where a certain degree of interaction between processes can be allowed and yet the error recovery of processes can be made independent from each other. Such processes are said to belong to the class of independently recoverable processes.

Definition: A group of processes is said to be independently recoverable if (a) no interaction between them takes place; or (b) they make private use of shared resources, that is, any interaction is solely for competition; or (c) they make shared use of the resources, but any information exchange that takes place is compensatable.

Statements (b) and (c) above need some explanation. If processes are only competing and making private use of shared resources (i.e. interaction is restricted to that necessary for acquiring and releasing the resources) then if the state of a process is to be restored to that at its recovery point, it is necessary to undo any effects due to any resource use and to release these resources - these actions need not affect the rest of the processes. Assume that a process A passes a message to some process B (i.e. A and B make shared use of a resource - say a buffer) and later it is necessary to restore the state of A. Then the message from A to B is said to be compensatable if it is possible for A to undo any effects due to that message by sending a
corrective message to B; B can process this message as a part of its normal activity therefore it is not necessary to restore the state of B.

The objective of the work to be described here was to develop and implement language features necessary for the recovery of independently recoverable processes. When the work began (late 1975), a program structure called recovery block was already been proposed by my colleagues as means for error detection and backward error recovery. Hence it was decided to develop and implement further language features within the framework of recovery blocks. An abstract data type called port was later developed by J.P. Benstre and myself. A port provides facilities for programming recoverable actions of independently recoverable processes. In what follows, an understanding of recovery blocks and the language Concurrent Pascal will be assumed; in addition, a familiarity with the ideas presented on ports will be helpful to the reader.

**Recovery blocks and ports**

The Pascal system of Brinch Hansen consists of the language Concurrent Pascal which is intended for writing operating systems supporting concurrent processes. Each of the processes of such an operating system is capable of executing a sequential program (written in either Concurrent Pascal or Sequential Pascal). The task of the concurrent program - the operating system - mainly consists of providing, to the sequential programs, appropriate 'abstract' operations on the resources of the system. The backward error recovery capability has been provided to the sequential programs of processes in the form of recovery blocks. Sequential Pascal was first extended with recovery blocks and later the similar exercise was performed for Concurrent Pascal. The notation for the recovery block as incorporated in these two languages is shown in figure 1.

The language Concurrent Pascal was further extended with data type port, making it possible for a concurrent program (operating system) to provide arbitrarily complex recoverable operations to sequential programs (e.g. recoverable updates on random access files).
ENSURE (acceptance test) BY
  (statement) "primary"
ELSE-BY
  (statement) "1st alternative"
  --
ELSE-ERROR;
  --

Figure 1. Recovery block notation for Concurrent and Sequential Pascal. The notation of port data type as embodied in Concurrent Pascal is given informally in figure 2.

TYPE (port name) = PORT(.*formal access right parameters.*);
  -- port variable declarations --
PROCEDURE ENTRY (procedure name) (.*formal parameters.*);
  BEGIN -- use of the acquired resource -- END;
  -- other procedures: --
REVERSE PROCEDURE;
  BEGIN -- undo use of the resource -- END;
BEGIN
  (statement) "prelude, concerned with resource acquisition"
  INNER;
  (statement) "postlude, concerned with the resource release"
END;

Figure 2. A port data type.

A port is a system type (in the Concurrent Pascal report\textsuperscript{3}, data types PROCESS, CLASS and MONITOR are referred to as "system type") and its properties closely match those of a class. A port differs from a class in the following ways:

1. In a port, recovery blocks can only be used within its procedure bodies.
2. A port contains a nameless and parameterless routine of a kind ‘reverse procedure’ whose body specifies undoing of actions. A
reverse procedure cannot be accessed by any program component;
only the recovery system can access it (see the following subsection entitled 'recovery semantics of ports').

3. The initial statement of a port contains an inner statement that splits the initial statement into a 'prelude' and a 'postlude'.

4. A port cannot be initialised by an init statement. Let PN be a variable of type port. Then PN can be initialised and its entry routines made use of as follows:

    USING PN(--- actual access rights parameters --) DO S;

where S is a statement that can contain calls on entry routines of PN. The above using statement defines the access rights of the port (just like the init statement). The inner statement of PN is textually replaced by S and the body of PN is executed (this has now the effect of executing the sequence 'prelude; S; postlude').

It can be appreciated that a port can specify facilities for acquiring a resource, using it, undoing the effects of the use and releasing the resource. A simple example should further clarify these ideas.

Example: Assume that it is required to construct a recoverable message sending facility between a process and an operator. The process can send a numbered message to the operator; should it be necessary to restore the state of the process such that 'unsending' of the message is required, a compensating message "ignore the message" can be sent. A port with these facilities is shown in figure 3, where two data types, a class CONSOLE (for sending a message) and a monitor RESOURCE (for acquiring the operator's terminal), have been assumed to be provided.
TYPE SENDER = PORT (CON:CONSOLE;I:INTEGER;RES:RESOURCE)
PROCEDURE ENTRY OUTPUT (L:LINE);
BEGIN CON.SEND (L,I) END;
REVERSE PROCEDURE;
BEGIN CON.SEND ("IGNORE MESSAGE NUMBER",I) END;
BEGIN
RES.ACQUIRE;"acquire the terminal"
INNER;
RES.RELEASE "release the terminal"
END;
---
MESSAGE:SENDER;
---
ENSURE "acceptance test" BY
---
USING MESSAGE (TERMINAL,J,UNIT) DO
BEGIN
---
MESSAGE.OUTPUT(L);
---
END;
---
ELSE-BY
---

Figure 3. A port example.

Recovery semantics of ports

As mentioned earlier, state restoration is carried out automatically by a recovery system. The actions of the recovery system are: (i) to carry out recovery actions arising from the use of ports, if any, and then (ii) to restore the state of global variables updated in the recovery block in which the error was detected. In this section, the recovery actions arising from the use of ports will be explained with reference to figure 4. Let an erroneous situation be detected at one of the following points while executing the program of figure 4.
(i) Error detected at point "a": An error has been detected after the execution of the prelude of

```
PN: PRT; "Port instance"
```

ENSURE -- BY
BEGIN

```
USING PN(-- --) DO "U1"
BEGIN
    "a"
    PN.PROC1 (-- --);
    "b"
    PN.PROC2 (-- --);
    "c"
END;

"d"
USING PN (-- --) DO "U2"
BEGIN
    PN.PROC1(-- --);
END;
    "e"
END ELSE-BY
```

Figure 4. A program to illustrate recovery semantics of ports.

PN, that is, after the acquisition of some resource. Since the state is to be restored to the point where this resource was not acquired, it must be released; this is done by executing the postlude (note that it becomes the programmer's responsibility to ensure that preludes and postludes are for resource acquisition and release respectively).
(ii) Error detected at point "b": The reverse procedure of PN is called and then the postlude is executed.

(iii) Error detected at point "c": The actions are the same as at "b". Note that while recovering, a reverse procedure is called as many times as the number of times the corresponding using statements that contains calls on the port procedures, have been activated in the recovery block in question.

(iv) Error detected at point "d": Let AB(Ui) represent the state of the variables and parameters of port AB after the execution of the using statement Ui. Then the recovery action is: starting with port PN in the state PN(U1), execute the sequence 'prelude; reverse procedure; postlude'.

Clearly, it is necessary to execute the reverse procedure. However, the necessary resources have been released (as the postlude has been executed). It is therefore necessary, starting with the port in the appropriate state, to re-execute the prelude and subsequently the postlude.

(v) Error detected at point "e": Starting with port PN in state PN(U2), the sequence 'prelude; reverse procedure; postlude' is executed; then starting with PN in the state PN(U1), the same sequence is executed. The same actions would be undertaken if the acceptance test of the recovery block failed.

Some additional remarks on recovery features

(i) If a failure occurs while restoring the state of a process, then, conceptually this is regarded as a collapse of the recovery system since no assurance about continued service can be given.

(ii) The recovery blocks of a concurrent program and a sequential Pascal program are not regarded as nested. Consider the following typical program in Concurrent Pascal where
the concurrent program is making use of the PROGRAM statement to execute a sequential program (written in Sequential Pascal). The Sequential Pascal program can make use of recovery blocks, but they would not be regarded as nested within that of the concurrent program. Thus, Sequential Pascal programs represent independent modules from the point of view of recovery.

(iii) Two standard functions have been provided in Concurrent Pascal.

(1) boolean function ERRORFLAG. The true value indicates that the state of the calling process is being restored by the recovery system and the false value indicates that the process is normally executing its program.

(2) An integer function RLEVEL. The value returned indicates the current degree of nesting of recovery blocks in the calling process.

RLEVEL=0 indicates that the process is not executing within a recovery block.

EXAMPLES

Three complete working examples will be used to illustrate how ports may be used to provide recoverable actions. The main aim of these examples is to illustrate the basic principles involved and as such the programs have been made as simple as possible. Thus, not much effort should be needed to understand these programs and the reader can concentrate upon their recovery aspects.
Producer-consumer: recovery by compensation

This example illustrates what is known as recovery by compensation: a process has sent some messages to other processes and later it becomes necessary to restore the state of the sending process to the state where these messages were not sent; the abstraction of 'unsending a message' can be provided by sending a corrective message to the receivers. If the receivers have been programmed to cope with corrective messages, then any recovery actions of the sender can be made independent.

Assume that there is a 'producer' that is sending integer valued messages to a 'consumer'. The function of the consumer is to output the total sum of the values received from the producer once the last message has been received. The compensating action of the producer is to send negative integer valued messages so as to cancel the effect of normal messages. The complete program listing is given in figure 5. There are four data types of interest: manager (a monitor for sending-receiving messages), sender (a port for providing recoverable message sending), producer (a process for producing messages) and consumer (a process for consuming messages). As it happens in this example, the prelude and postlude of sender are sufficient to carry out the necessary actions and no procedures are needed in the port (the need for the variable compensate can be appreciated by reading the discussion on 'error detected at point a' with reference to figure 4). The producer sends three messages (values 1, 2 and 3) in the primary algorithm of the recovery block. The acceptance test has been deliberately chosen to cause its failure. The print-out resulting from the execution of this program is shown in figure 6 which shows that the consumer has produced the correct output (the producer sends three corrective messages, -3, -2 and -1, to the consumer). Note that in a Concurrent Pascal program, the initial process is always numbered as one, so in this example the producer is 'process 2' and the consumer is 'process 3'. The number of a process is included in all the run time error messages.
TYPE IODEVICE=(TYPEDEVICE, DISKDEVICE, TAPEDEVICE, PRINTDEVICE, VTRDEVICE, CARDDEVICE);
TYPE IOOPERATION=(INOUT, OUTPUT, MOVE, CONTROL);
TYPE IOARG=(WRITE, ERASE, REWRITE, UPSPACE, BACKSPACE);
TYPE IORESULT=(COMPLETE, INTERVENTION, TRANSMISSION, FAILURE, ENDFILE, ENDMEDIUM, STARTMEDIUM);
TYPE IOPARAM=RECORD
  OPERATION:IOOPERATION;
  STATUS:IORESULT;
  ARG:IOARG
END;

TYPE OUT=CLASS;
PROCEDURE ENTRY PRINT(C:CHAR);
VAR
  PR:IOPARAM; DV:IODEVICE;
  CH:CHAR;
BEGIN
  PR.OPERATION:=OUTPUT; DV:IODEVICE;
  PR.C:=C; IO(CH, PR, DV)
END;

PROCEDURE ENTRY RECEIVER(I:INTEGER; LAST:BOOLEAN);
BEGIN
  IF FULL THEN DELAY(SENDER);
  BUFFER:=I; LAST:=LAST; FULL:=TRUE;
  CONTINUE(RECEIVER)
END;

PROCEDURE ENTRY RECEIVE(VAR I:INTEGER; VAR LAST:BOOLEAN);
BEGIN
  IF NOT FULL THEN DELAY(SENDER);
  I:=BUFFER; LAST:=LAST; FULL:=FALSE;
  CONTINUE(SENDER)
END;

BEGIN FULL:=FALSE END;

TYPE PORT=METHOD(MANAGER:I:INTEGER; LAST:BOOLEAN);
VAR
  VALUE:INTEGER; COMPENSATE:BOOLEAN;
REVERSE PROCEDURE;
BEGIN END;
BEGIN
  IF ERRORFLAG THEN
    BEGIN
      COMPENSATE:=TRUE;
      MAN.SEND(VALUE, LAST)
    END ELSE
    BEGIN
      VALUE:=I; COMPENSATE:=FALSE;
      MAN.SEND(I, LAST)
    END;
    INNER;
  IF ERRORFLAG AND NOT COMPENSATE THEN
    MAN.SEND(VALUE, LAST)
END;

Figure 5. Recovery by compensation
TYPE PRODUCER=PROCESS(M:MANAGER)
67 "RECOVERY DATA SPACE= 3M1000"
68 VAR
69 SEND:SENDER; I:INTEGER; LAST:BOOLEAN;
70 BEGIN
71 LAST:=FALSE;
72 ENSURE (I=9) BY
73 BEGIN
74 FOR I:=1 TO 3 DO
75 USING SEND(M,I,LAST) DO;
76 END ELSE BY
77 BEGIN
78 I:=9;
79 USING SEND(M,I,LAST) DO;
80 END
81 ELSE ERROR;
82 LAST:=TRUE;
83 USING SEND(M,0,LAST) DO;
84 END;
85
86 TYPE CONSUMER=PROCESS(M:MANAGER)
87 VAR
88 CONSOLE; OUT; I:INTEGER; SUM:INTEGER;
89 CH:CHAR;
90 LAST:BOOLEAN;
91 BEGIN
92 LAST:=FALSE; SUM:=0; INIT CONSOLE;
93 WHILE NOT LAST DO
94 BEGIN
95 M:=RECEIVE(I,LAST);
96 IF NOT LAST THEN SUM:=SUM+I
97 ELSE
98 BEGIN
99 IF SUM > 9 THEN SUM:=0;
100 CH:=CHR(SUM+48);
101 CONSOLE; PRINT(CH)
102 END
103 END
104 END;
105 END;
106
107 VAR
108 M:MANAGER;
109 PR:PRODUCER; CR:CONSUMER;
110 BEGIN
111 INIT M,Pr(M),Cr(M)
112 END
113

Figure 5. Recovery by compensation (continued)
Figure 6. Output of the example of figure 5.
The dining and vomiting philosophers

The well known five dining philosopher's problem (due to Dijkstra) has been modified to include recovery capability. (In what follows, it will be assumed that the reader is familiar with this problem and its solution.) A philosopher must first acquire two forks before eating. Eating in this example consists of printing a message on the console, indicating which philosopher is eating. If after eating but before releasing the forks, a philosopher detects an error and has to be restored to a state where eating was not performed then the philosopher must 'vomit' (vomit!) which in this example consists of printing a message to that effect on the console. If the philosopher detects an error after eating and releasing the forks, then he has only to vomit, no forks are needed for this purpose. It is thus seen that processes (philosophers) are competing for two sets of resources - forks and the console. The program listing is given in figure 7. In order to illustrate some of the possible recovery actions, provisions have been made in this program to force the occurrence of errors. These include (a) failure of the acceptance test after the execution of the primary of the recovery block in the body of a philosopher process, and (b) range error in the primary of the recovery block in procedure EAT. In this primary, philosopher after calling MANG:EAT, acquires the console to receive an input from the user. If the user types in a string beginning with 'F' then the primary will pass the acceptance test; any other input will cause a range error and the recovery action would involve vomiting (note that in this particular recovery situation a philosopher retains his forks). A sample console printout is presented in figure 8 where strings 'FFF' and 'YYY' are user supplied inputs to the philosophers.

Recoverable file updates

The last example illustrates one way whereby ports can be used quite elegantly to provide recoverable operations to sequential Pascal programs. In most multi-user operating systems, a job control language is used, among other things, for specifying resource requirements for a job; these resources are first acquired before executing the job. A simplified version of such a scheme can be implemented using ports as follows:
CONST NL="(;10;)"; LENGTH=72;

TYPE FORK=MONITOR
VAR A: ARRAY (0..4) OF INTEGER;
WT: ARRAY(0..4) OF QUEUE; J: INTEGER;
PROCEDURE ENTRY GET(I: INTEGER);
BEGIN
  IF AVG(J) <= 2 THEN
    DELAY(WT(I,J));
    AV(J(I+4) MOD 5)=AV(J(I+4) MOD 5)+1;
    AV(J(I+1) MOD 5)=AV(J(I+1) MOD 5)+1;
    IF AV(J(I+3) MOD 5)=2 THEN
      CONTINUE(WT(J(I+3) MOD 5));
  END;
PROCEDURE ENTRY GIVE(I: INTEGER);
BEGIN
  AV(J(I+1) MOD 5)=AV(J(I+1) MOD 5)+1;
  AV(J(I+4) MOD 5)=AV(J(I+4) MOD 5)+1;
  IF AV(J(I+3) MOD 5)=2 THEN
    CONTINUE(WT(J(I+3) MOD 5));
END;
BEGIN FOR J=0 TO 4 DO AV(J)=2 END;

TYPE FIFO=CLASS(LIMIT:INTEGER)
VAR HEAD, TAIL, LENGTH: INTEGER;
FUNCTION ENTRY ARRIVAL:INTEGER;
BEGIN
  ARRIVAL:=TAIL+1;
  TAIL:=TAIL MOD LIMIT +1;
  LENGTH:=LENGTH+1;
END;
FUNCTION ENTRY DEPARTURE: INTEGER;
BEGIN
  DEPARTURE:=HEAD;
  HEAD:=HEAD MOD LIMIT +1;
  LENGTH:=LENGTH-1;
END;
FUNCTION ENTRY EMPTY:BOOLEAN;
BEGIN EMPTY:=LENGTH=0 END;
BEGIN HEAD=1; TAIL=1; LENGTH=0 END;

TYPE RESOURCE=MONITOR
CONST NUMB=5;
VAR FREE:BOOLEAN;
Q: ARRAY (1..5) OF QUEUE; NEXT: FIFO;
PROCEDURE ENTRY REQUEST;
BEGIN
  IF FREE THEN FREE:=FALSE ELSE
    DELAY(Q(NEXT, ARRIVAL));
END;
PROCEDURE ENTRY RELEASE;
BEGIN
  IF NEXT. EMPTY THEN FREE:=TRUE ELSE
    CONTINUE(Q(NEXT, DEPARTURE));
END;
BEGIN FREE:=TRUE; INIT NEXT(NUMB) END;

TYPE LINE= ARRAY (0..1, LENGTH) OF CHAR;
TYPE IOOPERATION=(INPUT, OUTPUT, MOVE, CONTROL);

Figure 7. Dining and vomiting philosophers.
70 TYPE IOPARAM = RECORD
89   OPERATION: IOPERATION;
90   STATUS: IORESULT; ARG: IOARG
91   END;
92
93 TYPE INOUT = CLASS(ACCESS: RESOURCE)
94 PROCEDURE ENTRY WRITE(TEXT: LINE);
95   VAR PR: IOPARAM; I: INTEGER; C: CHAR; DV: IODEVICE;
96 BEGIN
97   ACCESS: REQUEST;
98   PR.OPERATION = OUTPUT;
99   DV = TYPEDEVICE;
100  I := 0;
101  REPEAT
102     I:=I+1; C:=TEXT(I); IO(C, PR, DV);
103     UNTIL (C=NL) OR (I=LENGTH);
104   ACCESS: RELEASE;
105  END;
106 PROCEDURE ENTRY READ(VAR TEXT: LINE);
107   VAR PR: IOPARAM; I: INTEGER; C: CHAR; DV: IODEVICE;
108 BEGIN
109   ACCESS: REQUEST;
110  PR.OPERATION = INPUT;
111  DV = TYPEDEVICE;
112  I := 0;
113  REPEAT
114     I:=I+1; IO(C, PR, DV); TEXT(I) := C;
115     UNTIL (C=NL) OR (I=LENGTH);
116   ACCESS: RELEASE;
117  END;
118 BEGIN
119  END;
120
121 TYPE MANAGER=PORT(CONS:INOUT; FRKS: FORKS; I: INTEGER)
122 VAR ACQ: BOOLEAN;
123 PROCEDURE ENTRY EAT;
124 BEGIN
125   CASE I OF
126     0: CONS. WRITE(" PHIL0 EATING (10l")");
127     1: CONS. WRITE(" PHIL1 EATING (10l")");
128     2: CONS. WRITE(" PHIL2 EATING (10l")");
129     3: CONS. WRITE(" PHIL3 EATING (10l")");
130     4: CONS. WRITE(" PHIL4 EATING (10l")");
131   END;
132 REVERSE PROCEDURE;
133 BEGIN
134   CASE I OF
135     0: CONS. WRITE(" PHIL0 VOMITING (10l")");
136     1: CONS. WRITE(" PHIL1 VOMITING (10l")");
137     2: CONS. WRITE(" PHIL2 VOMITING (10l")");
138     3: CONS. WRITE(" PHIL3 VOMITING (10l")");
139     4: CONS. WRITE(" PHIL4 VOMITING (10l")");
140   END;
141 END;
142 IF ERRORFLAG=FALSE THEN
143 BEGIN ACQ:=TRUE; FRKS, GET(I) END
144 ELSE ACQ:=FALSE;
145   INNER;
146   IF ACQ THEN FRKS, GIVE(I)
147 END;
148
149 Figure 7. Dining and vomiting philosophers (continued)
150 17
Figure 7. Dining and vomiting philosophers (continued)
Figure 8. An output of the example of figure 7.
Let A, B and C represent the ports that provide the operations on resources that a user program requires. In this situation the following program suggests itself.

```
---
PROGRAM JOB (-- --);
---
USING A(-- --) DO
USING B(-- --) DO
USING C(-- --) DO
BEGIN
---
JOB (-- --); "execute a sequential program"
---
END;
---
```

The necessary resources are automatically acquired and released around the execution of a sequential program.

Let us assume that the facility of recoverable file updates is to be provided to Sequential Pascal programs so that such programs may update files from within a recovery block. Minor changes were made to the SOLO operating system for this purpose. The following simplifying assumptions were made: (i) make use of the existing SOLO filing system, (ii) only one file may be opened for recoverable updates, (iii) Sequential Pascal programs do not make use of nested recovery blocks, and (iv) recovery is to be provided at a page level, that is, sequential programs are given the facility of modifying file pages.

The changed part of the SOLO concurrent program is shown in figure 9 (an understanding of the SOLO operating system is required to appreciate this example). A port FILEMANAGER has been programmed that makes use of the DATAPILE data type for file manipulations (read, write, open and close). In the prelude of the port, a file called RECOVERY is first opened (this file will record the previous contents of updated pages) followed by the opening of the file whose name is supplied by the user; in the postlude,
Figure 9. Recoverable file updates
TYPE JOBPROCESS =
PROCESS
(TYPEUSE: TYPERESOURCE; DISKUSE: RESOURSE;
CATALOG: DISKCATALOG; INBUFFER, OUTBUFFER; PAGEBUFFER;
INREQUEST, INRESPONSE, OUTREQUEST, OUTRESPONSE; ARGBUFFER;
STACK; PROGSTACK);
"PROGRAM DATA SPACE ="+14000
CONST MAXFILE = 21;
TYPE FILE = 1..MAXFILE;
VAR
FILES: ARRAY(1..FILE.) OF DATAFILE;
FN: FILENAME;
FILEMAN: FILEMANAGER;
OPERATOR: TERMINAL; OPSTREAM; TERMINALSTREAM;
INSTREAM, OUTSTREAM: CHARSTREAM;

CODE: PROGFILE1;

PROGRAM JOB(VAR PARAM: ARGLIST; STORE: PROGSTORE1);
ENTRY READ, WRITE, OPEN, CLOSE, GET, PUT, LENGTH,
MARK, RELEASE, IDENTIFY, ACCEPT, DISPLAY, READPAGE,
WRITEPAGE, READLINE, WRTLINE, READARG, WRTARG,
LOOKUP, IOTRANSFER, IOMOVE, TASK, RUN;
PROCEDURE CALL(ID: IDENTIFIER; VAR PARAM: ARGLIST;
VAR LINE: INTEGER; VAR RESULT: RESULTTYPE);
VAR STATE: PROGSTATE; LASTID: IDENTIFIER;
BEGIN
WITH CODE, STACK DO
BEGIN
LINE = 0;
OPEN(ID, STATE);
IF (STATE = READY) & SPACE THEN
BEGIN
   PUSH(ID);
   IF ID <> "DO" THEN
   BEGIN
      FK, SPECIFY(PARAM(2), ARG);
      USING FILEMAN(FILES(1..FILE), FILES(2..FILE), FN) DO
      JOB(PARAM, STORE)
   END ELSE JOB(PARAM, STORE);
   POP(LINE, RESULT);
   END ELSE
   IF STATE = TOOBIG THEN RESULT = CODELIMIT
   ELSE RESULT = CALLERROR;
   IF ANY THEN
   BEGIN GET(LASTID); OPEN(LASTID, STATE) END;
   END;
END;
PROCEDURE ENTRY READ(VAR C: CHAR);
BEGIN INSTREAM, READ(C) END;

Figure 9. Recoverable file updates (continued)
Figure 10. A Sequential Pascal test program for the example of figure 9.
these files are closed. The WRITE procedure of the port records the
previous contents of the page to be updated in file RECOVERY before
performing the update. The reverse procedures makes use of RECOVERY
file for restoring the page contents.

Figure 10 shows a simple test program with a recovery
block that updates a file in its primary algorithm and prints a modified
page on the console. After the termination of this program, the file
should be in its original state; this was verified by printing the file
on the line printer.

Many interesting issues are raised when the design of a multiuser
filing system providing recoverable file updates and with a 'crash
resistance' property (that is the property that no matter where an
error is detected, the system is always recovered to a consistent state)
is considered; such issues are discussed elsewhere. The reader wishing
to pursue the subject of recoverability further may find the ideas
presented in ref. 12 of interest, where many of the fundamental aspects
of recovery are discussed.

It is hoped that these illustrative examples will have convinced the
reader that ports provide a systematic method for designing recoverable
operations and that the extended version of Concurrent and Sequential
Pascal can be used for programming fault tolerant systems.

Acknowledgements

This work was supported by the Science Research Council (UK) and was
carried out as a part of the Highly Reliable Computing Systems project at
Newcastle University.

Appendix

Implementation restrictions

The following restrictions (in addition to those given elsewhere) in
the implemented version of concurrent and Sequential Pascal with
recovery should be noted:
(i) Real and pointer variables are not supported.

(ii) For a given port, its prelude, postlude and procedures represent the smallest unit of recovery such that no recovery is possible should an error be detected while executing any one of them. As a result of this restriction, it is not meaningful to use recovery blocks in the bodies of port procedures.

(iii) The variables of the initial process of a concurrent program are unrecoverable.

(iv) The body of the initial process of a concurrent program can not contain recovery blocks.

References:


10. S.K. Shrivastava, 'Concurrent Pascal with backward error recovery: implementation', This issue.


Concurrent Pascal with Backward Error Recovery: Implementation

by

S.K. Shrivastava
Computing Laboratory
University of Newcastle upon Tyne
England

Summary

The implementation of backward error recovery features requires the support of a run time subsystem (called the recovery system) that is responsible for performing the task of state restoration. The recovery system implemented to support the recovery features of Concurrent Pascal includes, for each process, a recovery cache for recording appropriate recovery data. This paper describes the details of the recovery system that was implemented as a part of the interpreter of Brinch Hansen's Concurrent Pascal system.

Keywords

Recovery blocks, recovery cache, Concurrent and Sequential Pascal, error recovery, ports.
INTRODUCTION

As has been discussed in a companion paper, the backward error recovery capability, by making use of recovery blocks, has been made available to the sequential programs of concurrent processes that are programmed in Concurrent Pascal. The environment of a sequential program includes the 'abstract' objects that have been made available by the corresponding process. Those objects that are intended to be operated on from within a recovery block must be made recoverable such that if recovery is invoked, their states are automatically restored. This state restoration is carried out by a recovery system that is part of the run time system that supports the processes. If an operation on an abstract object in effect amounts to the updating of store locations within the private address space of a process, then the recoverability of such an operation is easily obtainable: all that is necessary is for the recovery system to record, in some data structure, the addresses of the locations and the prior values in those locations; the recovery is carried out by appropriately restoring these values. This is how the originally proposed recovery cache (the data structure to record the addresses and values) is intended to work in conjunction with recovery blocks. However, if an operation on an abstract object amounts in effect to operations other than update of private store locations then, in general, recovery of such an operation can be difficult to automate. The solution adopted in the extended version of Concurrent Pascal is to program such operations by making use of the abstract data type port; a port contains a reverse procedure for specifying the 'undoing' of operations. The recovery cache can be extended such that it can be made to contain enough information about ports used within a recovery block so that if recovery is invoked, this information can be made use of, for automatically executing reverse procedures and related programs. The extensions that were made to recovery cache and their associated processing algorithms to support recovery as required by port objects is the main subject of this paper.
The recovery system which maintains a recovery cache for each process has been implemented as a part of the interpreter of Brinch Hansen's Concurrent Pascal system available on a PDP11/45. The implementation exercise was carried out in two parts. In the first part, the language Sequential Pascal was extended with recovery blocks and the interpreter was modified so as to make assignments recoverable. In the second part, Concurrent Pascal was extended with recovery blocks and ports and the interpreter was further modified in a manner to be described below. In the description that follows, I shall assume that the reader is familiar with recovery cache details described elsewhere and the way caches were implemented for Sequential Pascal with recovery blocks.

PRELIMINARY DETAILS

To set the scene, the Concurrent Pascal system, as developed by Brinch Hansen's group together with the initial modifications that were made to it for maintaining recovery caches supporting recoverable assignments as required by Sequential Pascal programs with recovery blocks will be briefly described.

The system consists of a kernel, and an interpreter that executes both sequential and concurrent programs. The kernel implements processors, synchronising primitives, basic input and output and a virtual storage system. The overall run time system is depicted in figure 1. The virtual address space of a process is divided into a private segment and a segment that is common to all the processes.

This common segment contains the data necessary for resource sharing (monitor variables), virtual code (the code produced by the Concurrent Pascal compiler), the interpreter code, the current process head (containing the state vector of the current process) and the interpreter table (that has, for all the interpreter instructions, appropriate pointers to the interpreter code).
When the kernel selects a process for running, its state vector is copied into the current process head and the control is handed over to the interpreter. The current process head also acts as the interface between the kernel and interpreter. The interpreter implements a simple stack machine; the stack is maintained by the interpreter in the private segment of the selected process (see figure 1). The interpreter can also execute the virtual code produced for a Sequential Pascal program - such code is stored as 'private data' of the process.

Figure 1. The Concurrent Pascal System
It was a relatively straightforward task to modify the interpreter so as to support a recovery cache for each of the processes in the system; figure 1 shows how recovery caches were placed in the private segments. The interpreter was extended with virtual instructions 'enter recovery block', 'acceptance test pass' and 'acceptance test fail', and the virtual instructions that performed assignments (e.g. copyword) were modified to record in the cache, if necessary, prior values. The interpreter incorporates a number of consistency checks for detecting abnormal situations (e.g. range error, stack limit); when such a situation is detected by the interpreter, a jump is made to an 'exception handler' which copes with the situation. The original system had no fault tolerance capability, hence the action of the exception handler was: if the program being interpreted is a Sequential Pascal program then abort the execution of that program; however, if the program is the Concurrent Pascal program then stop the system. The logic of the exception handler was modified to include cache processing for state restoration of a Sequential Pascal program (see figure 2, which presents a simplified picture of the exception handler described more fully elsewhere).
exception: var recovery: boolean;
    recovery:=false;
    while nest>0 & recovery do
    begin
        with the recovery data of the current recovery block do
        begin
            restore;
            if an alternative exists then
            begin
                interpreter instruction counter:=start
                of the code of the alternative;
                recovery:=true
            end else
            begin
                nest:=nest-1; prepare recovery cache
                for the processing of the recovery data of
                the enclosing recovery block
            end
        end;
    end; if ~ recovery then
    begin
        if abnormal termination & program=concurrent then
            stop;
        restore stack; return
    end;

Figure 2 Exception handling by the interpreter

The variable 'nest' counts the nesting of recovery blocks for
the process (nest, together with a few other recovery block book-
keeping variables are maintained by the interpreter as a part of the
state vector of each process). If 'nest>0' is true then recovery
capability exists and state restoration is carried out by the
procedure 'restore'. The action of 'restore' is simply to appropriately
copy back the values recorded in the current region of the recovery
cache. When ports were added to Concurrent Pascal, it was necessary
to modify the procedure 'restore' to include the execution of reverse
procedures and other related programs. It was also necessary to modify
the organisation of the recovery cache, as described next.
THE RECOVERY CACHE STRUCTURE

In order to appreciate what recovery information should be recorded in the recovery cache of a process, assume that the primary of recovery block R2 (see figure 3) fails the acceptance test. Then, (a) the reverse procedure of A must be executed, (b) starting with B in state B(U2), the sequence 'prelude; reverse procedure; postlude' must be executed* and (c) the state of updated global variables must be restored. Thus the cache must record for each recovery block that has been entered but not yet exited: (i) addresses of any reverse procedures to be executed, (ii) sufficient information about ports that have been used so that actions of type (b) above can be performed, and finally, (iii) the usual address-value pairs for restoring updated 'words' (all the variables are mapped by the interpreter onto store words, to restoring a variable is equivalent to restoring the appropriate words).

With the above requirements in mind, the cache structure which was used originally was modified to that depicted in figure 4(a). As before, for all the recovery blocks entered but not yet exited, there are 'barriers' in the cache with which the appropriate recovery data can be associated; all of the barriers are linked together and the 'cachhr' entry in the process head points to the top most barrier (corresponding to the innermost recovery block). The variables 'next' and 'number' indicate the alternative to be executed and the number of alternatives in that recovery block. The variables Q, G, b and s are pointers used by the interpreter (next instruction, global variables, local variables and stack top respectively), these values are stored when a recovery block is entered. The variable 'type' records the type of the recovery block (see below). A linked list of 'port recovery data' is maintained and the variable 'ports' points to the first entry. For every 'using' statement executed in the recovery block, there will be a corresponding 'port recovery data' recorded in the cache; the contents of this data will be discussed shortly.

* B(U2) stands for the state of the variables and parameters of port B after the execution of the using statement U2.
ENSURE ... BY "R1"
BEGIN ...
    ...(1)
    USING A(...) DO "U1"
BEGIN ...
    ...(2)
    ENSURE ...
    BY "R2"
BEGIN ...
    ...(3)
    USING B(...) DO "U2"
BEGIN ...
    ...(4)
    B.PROC(...);
...
END;
    ...(5)
A.PROC(...);
    ...(6)
END ELSE BY ...
ELSE ERROR:"END OF R2"
    ...(7)
END;
    ...(8)
END ELSE BY ...

Figure 3 A skeleton program with recovery blocks and ports

Finally the variables 'reve1' to 'reve4' are pointers to any reverse procedures that need executing, as mentioned at the beginning of this section (a fixed number of entries was chosen for efficient lookup during cache processing; for all practical purposes, these four entries should be sufficient). The following two points about this organisation should be noted:

(1) When a recovery block is entered, 'barrier' to 'reve4' entries are put in the recovery cache with type=0, ports=0, and reve1 to 4=0 (a 'c' for ports and revei is taken to mean a null value).
Figure 4. Recovery cache organisation:
(a) recovery data for a normal recovery block
(b) recovery data for a using recovery block
(2) The value-address pairs and port recovery data are recorded incrementally - as assignments are performed and 'using' statements are executed.

The state vector (process head) of each process was extended to contain a few book keeping variables necessary for the maintenance of recovery blocks. The starred entries in the current process head of figure 4 shows these variables. As already stated, 'cachbr' points to the top most barrier. The other variables are used as follows: 'critic' equal to zero means that port operations are to be recoverable (this will be explained in the next section); 'nest' indicates the nesting of recovery blocks and 'direction' equal to zero means that the process is executing normally (direction>0 means the process is in the recovery mode, that is, 'going backwards').

The 'using' recovery block

The method by which the port recovery data is recorded will now be described. For a given 'using' statement, it was thought better to record any recovery data about it at one place rather than it being scattered over the enclosing recovery block and in the recovery blocks (if any) in the using statement. After many trials, a satisfactory way to record the data was found to be the technique that treated a using statement as a special recovery block with no alternatives.

The structure of the recovery data for a using recovery block is shown in figure 4(b). The portion below the barrier is exactly like the 'normal' recovery block except that type=1, indicating that it is a 'using' recovery block. On top of the barrier, the actual port recovery data is maintained as shown, where 'postlude', 'reverse' and 'prelude' record the addresses of the appropriate code fragments and 'code' has the following meaning when processing this recovery data for state restoration:
code=0 : take no action on the port recovery data

=1 : execute the postlude (as the forward going process had
 executed only the prelude)

=2 : executed the reverse procedure and postlude (as the
 forward going process had executed prelude plus the
 port procedures)

=3 : execute the reverse procedure only (the process failed
 while executing the postlude – this implies a "collapse"
 of the recovery mechanism", as resources cannot be
 returned – so provide a degraded service by merely
 executing the reverse procedure).

Two actions are undertaken when the execution of a using statement
finishes (e.g., the postlude has been executed): (i) the variables and
parameters of the port are copied in the recovery data and (ii) it is
also taken to mean the exit from the using recovery block and the
recovery data of this 'recovery block' is merged with that of the
enclosing recovery block. In particular, the port recovery data is
chained to enclosing recovery block's chain as indicated in figure 4(a).

The dynamics of recovery cache management

In order that the reader can get an overall view of how the
recovery cache of a process is dynamically managed, a number of snap-
shots of a recovery cache are presented in figure 5 for the execution
of the program of figure 3. Figure 5(1) through to figure 5(4) show
how barriers are placed and recovery data maintained for statements R1,
U1, R2 and U2. From the recovery cache's viewpoint, R1 and R2 correspond
to type o recovery block and U1 and U2 correspond to type 1 recovery
block. At point 5 in figure 3, U2 has been successfully executed, so
the corresponding cache diagram, figure 5(5), shows that U2 recovery
data has been merged to the recovery data of R2; in particular, 'ports'
cell of R2 now points to U2 recovery data. At point 6 in figure 3, a
procedure of U1 has been called, so the cache (figure 5(6)) shows
that the 'reveal' entry contains a pointer to the barrier of U1
recovery data. Thus, should an error be detected at point 6 of
figure 3 (or R2 primary fails the acceptance test), there is enough
information recorded in the cache about R2 to help reconstruct the
prior abstract state. Figures 5(7) and 5(8) show further how recovery
data is merged. The details of the merging algorithm are presented
in the next section.

INTERPRETER VIRTUAL INSTRUCTIONS AND CACHE PROCESSING ALGORITHMS

The details of recovery block virtual instructions added to the
interpreter and the virtual code produced by the Sequential Pascal
compiler for a recovery block have been described elsewhere. The
Concurrent Pascal compiler was modified to generate the similar virtual
code for recovery blocks. Since the structure of a recovery cache has
been changed from that used before, it was necessary to appropriately
change the code of recovery block virtual instructions. These changes
will not be described here.

As the port data type closely resembles the class data type, all
of the port virtual instructions provided by the interpreter have
been based upon those provided for classes. Before presenting the
details of these port virtual instructions, the following three points
should be noted:

(i) The code of an instruction can be divided into two parts
(a) the first part concerned with the manipulation of the
stack of the process in question (e.g., store return link
before a procedure call) and (b) the second part concerned
with the manipulation of the recovery cache of the process.
Only the details of the second part will be described here
since it is this second part that makes this interpreter
somewhat novel.

(ii) To keep this experimental implementation of the interpreter
as simple as possible, an implementation restriction on ports
was imposed: the prelude, postlude and procedures of a port
represent the smallest unit of recovery, in that recovery may
not be possible should an error be detected while executing
any one of them. Thus, no recovery data is generated for ports

12
used in these program parts. This considerably simplifies the management of recovery data (at the expense of providing a coarser 'grain of recovery'). A variable 'critic' was maintained in each of the process heads of processes with the following meaning: 'critic<>0' implies generate no recovery data for a port and 'critic=0' implies the opposite.

(iii) For the same reason as above, variables of the initial process were treated as unrecoverable, in that no recovery data is generated for them.

Figure 6 shows the port virtual instructions and how they are planted by the Concurrent Pascal compiler around appropriate program bodies. For the sake of completeness, all the parameters of the instructions have been shown, however only the underlined parameters are used explicitly for cache management, the rest are for stack management (stacklength indicates storage to be reserved on the stack, poplength indicates the storage to be reclaimed and so on).

The execution of a 'using' statement is started by the interpreter executing the 'initport' instruction. The recovery cache manipulation part of this instruction is shown below.

```pascal
procedure initport (paramlength, varlength, distance);
begin
  if critic=0 then
    {create on the cache, recovery data for type 1 recovery block; code:=0; nest:=nest+1}
end;
```

The next instruction to be executed after this is the 'beginprelude' instruction (note that 'revprocedaddr' means the address where 'enterroutine' instruction is stored, a similar meaning is attached to 'preludeaddr', 'postludeaddr'):

```pascal
procedure beginprelude (-,-,-, revprocedaddr, postludeaddr);
begin
  if critic=0 then
    {store prelude address, reverse procedure address and postlude address in the port recovery data; critic:=critic+1}
end;
```
PORTNAME = PORT(...parameters..)
  ... variables ...
PROCEDURE ENTRY PROC(...parameters..);
    \[ \text{body} \]
    \[ \text{enterport (stacklength, poplength, line, varlength)} \]
    \[ \text{exitport "return"} \]
  ...
REVERSE PROCEDURE;
    \[ \text{body} \]
    \[ \text{enterreverse (stacklength, poplength, line, varlength)} \]
    \[ \text{exitreverse "return"} \]
BEGIN
    \[ \text{prelude} \]
    \[ \text{INNER;} \]
    \[ \text{endprelude "return"} \]
    \[ \text{postlude} \]
    \[ \text{beginpostlude (stacklength, poplength, line, varlength)} \]
    \[ \text{endpostlude "return"} \]
END; "of port type"

PN: PORTNAME; "allocate space on the stack"
USING PN(...)
BEGIN ...
    \[ \text{PROC}() \]
    \[ \text{call (distance) "jump to enterport"} \]
END; \[ \text{call (distance) "jump to beginpostlude"} \]

FIGURE 6 Port virtual instructions
The last instruction of the prelude is the 'endprelude' instruction:

```plaintext
text
procedure endprelude;
    begin
        ....
        critic:=critic-1;
        if critic=0 then code:=1
    end;
```

Calling a port procedure has the effect of executing 'enterport' followed by the code for the procedure body followed by 'exitport' which returns the control back to the caller:

```plaintext
text
procedure enterport (-,-,-,-);
    begin
        ....
        critic:=critic+1
    end;

procedure exitport;
    begin
        ....
        critic:=critic-1;
        if critic=0 then
            if the current recovery data is of type 1 and refers to the port whose procedure has been called then code:=2
            else [select an empty receive cell and put in it a pointer to the barrier of appropriate type 1 recovery data; make 'code' of this type 1 recovery data=2]
        end;
```

The algorithms for the instructions 'enterreverse' and 'exitreverse' are the same as those of 'enterport' and 'exitport' respectively. Finally, the algorithms for the instructions 'beginpostlude' and 'endpostlude' are given:

```plaintext
text
procedure beginpostlude;
    begin
        ....
        if critic=0 then
            [code:=3; critic:=critic+1]
    end;
```
procedure endpostlude;
    begin
        critic:=critic-1;
        if critic=0 then
            {copy from the stack, the actual port
            parameters and variables to the port
            data area in the cache; codei:=4;
            if next>0 then merge else discard}
        end;
    
As noted earlier, successful execution of a using statement is treated
as the using recovery block passing its 'acceptance test'. Hence in
the instruction 'endpostlude', a call is made either to procedure
merge (for merging recovery data to the enclosing recovery block's
recovery data) or to procedure discard for throwing away the recovery
data. The algorithm for 'merge' is given below:

procedure merge;
    begin
        with all reve i of the recovery data of the just executed
        recovery block do
            {if the enclosing recovery block's recovery data has no reve j with
            the same value as reve i and reve i is not pointing to the barrier
            of the enclosing recovery block's recovery data then copy reve i to
            any unused reve j};
        with all the entries of the just executed
        recovery block's recovery data do
            {if the entry is a port recovery data then add it and link it
            to the port recovery data of the enclosing recovery block's
            recovery data;}
            {if the entry is a value-address pair and the variable is
            global and there is no entry for it in the recovery data of
            the enclosing recovery block then add it to the recovery data
            else throw it away;}
            update heaptop and cachbr;
    end;
The last algorithm to be described here is that of procedure \texttt{restore} which performs the task of state restoration. This procedure is called from within the exception handler of the interpreter (see figure 2). The procedure processes the current recovery data as follows:

\begin{verbatim}
procedure restore;
    begin
        critic:=critic+1;
        while any port data left do
            (select the first unprocessed port data; copy the
            parameters and variables on the stack; execute
            the sequence 'prelude; reverse procedure; postlude';
            mark this port data as 'processed'; restore stack);
        with all receive do
            (if receive i \neq null then extract the reverse procedure
            address and execute the procedure);
        if the recovery data is of type 1 then
            case code of
                0:
                1: execute the postlude;
                2: execute 'reverse procedure; postlude';
                3: execute reverse procedure;
            end "case";
        end;
        appropriately select 'address-value' pairs and
        restore the prior values; restore the interpreter
        registers S, E, G and Q;
        critic:=critic-1
    end;
\end{verbatim}

A few remarks on this algorithm are in order.

(1) By incrementing \texttt{critic} at the beginning, it is made certain that no recovery data will be generated while executing any programs.

(2) The order in which the various 'undo effects' programs are executed is not necessarily the reverse of the order in which the 'produce effects' programs were executed (this may turn out to be a weakness of this implementation and a revision of the method of recording recovery data will be needed to introduce the necessary changes).
A few changes were made to the error reporting routines of the kernel such that the identity of a process is appended at the beginning when an error message from the interpreter is printed on the console. Examples of such messages appear in a companion paper.

**Concluding Remarks**

It can be seen that the structure of a recovery cache and its processing algorithms have become much more complicated than the scheme presented earlier. This must be judged against the fact that it is now relatively straightforward for a concurrent program—an operating system—to offer abstract recoverable operations on arbitrary resources to user programs. In the absence of such a facility, ad hoc measures will have to be incorporated in concurrent and user programs. The resulting increase in complexity is likely to adversely affect the reliability of the system.

The size of the original interpreter was about 1K words. The size of the modified interpreter is about 2K words. This modest increase in the size of the interpreter was made possible by (a) making maximum use of the already existing 'class' virtual instruction codes for port virtual instructions, and (b) relying more on the use of subroutines as against macros. The modifications to the interpreter have been performed in such a manner that programs that do not use recovery blocks and ports are not affected; in particular, the SOLO operating system and the supporting programs can be used for developing concurrent and sequential programs with recovery. The system that has been described here must be regarded as experimental since, for example, any realistic fault-tolerant system will include a high degree of hardware redundancy (this system has none). Nevertheless this experimental system can be used as a basis for the design of more appropriate hardware; in particular, there is no reason why the interpreter described here could not be implemented by microprogramming.
Acknowledgements

Acknowledgement is due to P.C. Treleaven who made the necessary changes to the Concurrent Pascal compiler and to P.A. Lee whose criticisms on all aspects of this work proved to be of a great value. This work was supported by the Science Research Council (UK) as a part of the 'Highly Reliable Computing Systems' project at the Newcastle University.

References

1. S.K. Shrivastava, 'Concurrent Pascal with backward error recovery: language features and examples', this issue.


