# ERROR RESYNCHRONIZATION IN PRODUCER-CONSUMER SYSTEMS

David L. Russell and Thomas H. Bredt
Stanford University

This paper is concerned with error processing for parallel producer-consumer interactions such as encountered in the desing of multi-process operating systems. Solutions to resynchronization problems that occur when a consumer process detects errors in information received from a producer process are presented. Fundamental properties of this error processing are discussed. It is shown that explicit error processing results in an increase in program complexity and a decrease in the ease of understanding a program.

Key Words and Phrases: software reliability, error detection and recovery, fault tolerance, message facilities, operating systems, asynchronous programming, interprocess communication

CR Categories: 4.22, 4.29, 4.32, 4.35

## 1. INTRODUCTION

A substantial portion of the software in most systems is devoted to error processing of some form; therefore fundamental properties of error processing need to be clearly understood. Wulf [1] has discussed the techniques being used for treating errors in the HYDRA operating system. Some general conventions that are useful in writing collections of sequential programs have also been proposed [2,3]. In particular, Parnas has said [3]:

> The interfaces between the modules must enable the communication of information about external errors. For example, it should be possible for a module to be informed that information, given to it earlier, was incorrect, or that a request, which it had issued some time ago, was executed incorrectly. It should be possible for a module, which detects inconsistancies in incoming information, to inform the supplying module about those inconsistancies. The module supplying those data should be designed to respond meaningfully to such a notification.

In this paper it is shown that, for parallel systems, error recovery of the type proposed above can be difficult to perform correctly. A particular example of parallel process interaction, the producer-consumer problem, is considered. Known solutions are extended to handle errors of a particular type. Analysis of the program solutions shows that fundamental properties of the error situations cause the programs to be difficult to design correctly and to understand.

This work was supported in part by the National Science Foundation under Grant No. GJ41644.

## 2. THE PRODUCER-CONSUMER SYSTEM

To motivate the discussion that follows, brief descriptions of two situations that require some kind of error recovery are presented.

WYLBUR [4] is a text editor system developed at Stanford University. In normal use, the WYLBUR system accepts line images of commands that are to be performed; these include commands to obtain files for editing, the actual editing commands, and miscellaneous system commands. Thus, for instance, the command INSERT 3.5 allows a text line to be inserted into the active file, between lines 3 and 4, and assigns the line number 3.5 to it. If however, a line numbered 3.5 already existed, WYLBUR replies with an error message and the command must be corrected and repeated. Similarly, if a command is misspelled or is unacceptable for some other reason, WYLBUR refuses the command and issues an error message.

In some versions of WYLBUR, a preprocessor allows commands to be batched and/or generated by macro substitution. Thus if a sequence of commands is stored in the active file, the command EXEC ACTIVE submits these commands as a block to the text editor. A sequence of frequently used commands can be efficiently submitted to the editor using this facility. If there is an error in one of these commands, however, the editor stops at the point that the incorrect command is scanned and the block submission of commands is terminated. Any commands remaining in the block are automatically thrown away.

In some cases, this flushing of the remainder of the block is desirable; in other cases, it would be preferable for the editor to accept a corrected command and then continue with the rest of the block of submitted commands.

A second example occurred in the development of a small operating system. An input driver for a terminal was to be written. The driver was to accept text from the terminal, buffer it, and pass it on to the remainder of the system. In addition, for increased efficiency, the terminal was to be operated in a read-ahead mode; that is, an entire page of text and command words was to be input at one time.

When the processors of the system found an error in the command stream of the input text, they were to return an error message to the terminal. However, by this time, subsequent commands had already been entered into the input driver buffer; some means had to be found to remove these commands from the input stream and retry a corrected version of the offending text.

In both of these examples, the following elements are present: a source of data that is unaware of the syntactical or semantic correctness of the data, a sink for that data that is sensitive to the correctness of the data, and multiple buffering of the data stream that allows the source of data to get ahead of the sink.

As an idealized version of these problems, the producer-consumer problem [5] is used in this paper to study error recovery techniques. In this problem, two relatively independent processes, a producer and a consumer, execute concurrently and communicate through an interprocess communication mechanism that ensures synchronization of the two processes and mutual exclusion between their critical sections. An abstract view of the data flow between the two processes is shown in Figure 1. The producer generates data and this data is used by the consumer. It is assumed that the producer and consumer processes execute in an environment with common shared memory and one or more processor units. Each processor may have some local memory.

When an error is detected by the consumer, the consumer takes action to inform the producer, so that the producer can correct and retransmit the corrected data. This exchange of error messages and producer responses acts to resynchronize the two processes, and to allow their continued cooperation. In this paper, solutions to the producer-consumer problem that include explicit resynchronization of the processes in the event of an error are considered. The program representation, exact failure situations, assumptions, and restrictions are discussed below.

## Program Representation

The programs of this paper are expressed in a PASCAL-like language [6] augmented by the parallel statement proposed by Dijkstra [5]:

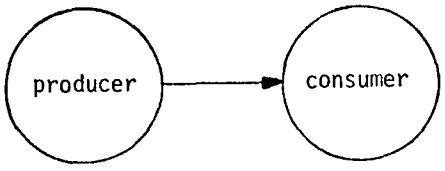


Figure 1. Producer-consumer data flow

parbegin $S_1$; $S_2$; ... $S_n$ parend. The parallel statement indicates that the statements $S_1$, $S_2$, ... $S_n$ are to be executed concurrently. In addition, the language is augmented by messagelist declarations and SEND-RECEIVE primitives similar to those discussed by Brinch Hansen [7,8]. A declaration var m: messagelist(n) of T, for example, declares m to be a messagelist whose capacity is n messages, all of type T. The operation SEND(m,x) where m is a messagelist of some type and x is a variable of that same type, will put the value of x into the messagelist named m; further, that operation will be a primitive operation and therefore a protected critical region. The complementary operation, RECEIVE(m,x), takes a message from the messagelist m and places it in the variable x of the same type; it is also a primitive operation. If a process attempts a RECEIVE operation but the messagelist is empty, the process is blocked (placed in a wait state). If a process attempts a SEND operation and the messagelist is full (it has a finite capacity), the process is also blocked. In addition, messages are assumed to be received in the same order as they are sent.

With these features for interprocess commuication, one possible solution to the producer-consumer problem is given in Figure 2. In this solution, a messagelist of capacity n is used to transmit data between the producer and the consumer. The multiple buffering of data between the two processes contributes to the overall efficiency of the system by increasing the potential system throughput. (Note that these processes loop forever; no provision is made for handling end-of-data situations.)

```
type buff = <<information to be transmitted>>;
var mlist: messagelist(n) of buff;

procedure producer;
  var buffer: buff;
  begin
    while true do
      begin
        fill buffer;
        send(mlist,buffer);
      end;
  end producer;

procedure consumer;
  var buffer: buff;
  begin
    while true do
      begin
        receive(mlist,buffer);
        empty buffer;
      end;
  end consumer;

begin
  parbegin producer; consumer parend;
end.
```

Figure 2. Access by value

## Type of Failure Situations

Two types of failure situations are considered in this paper. In both, the producer produces data for the consumer. When the consumer obtains this data it is able to tell if the data is invalid before another message is received. If the message is invalid, the consumer initiates corrective action of some kind by sending an error indication to the producer. Between the time that the producer sends the message that is invalid and the time that it receives the error indication for that invalid buffer, the producer may have generated and sent several messages to the consumer. The two failure situations considered differ in the properties of these messages. In failures of type $F_d$ the records produced by the producer are assumed to be dependent upon the preceding records; the messages that have been sent but not received must not be allowed to be consumed by the consumer. In failures of type $F_i$ the messages produced by the producer are assumed to be independent; although the sequence order of the messages is important, the fact that one record may be incorrect does not necessarily invalidate subsequent records.

In this paper, we seek solutions to the resynchronization problem of returning the producer and consumer processes to normal interaction following detection of a failure situation of type $F_d$ or $F_i$ by the consumer. The following assumptions and restrictions are made in considering solutions to the resynchronization problem.

## Assumptions

1. The consumer detects an invalid buffer immediately, as soon as it is received.
2. The producer is able to correct and retransmit a buffer that is invalid when obtained by the consumer. (This may involve consulting with an external oracle or the keeping redundant information by the producer).
3. End-of-data conditions are ignored.
4. The message system works correctly.
5. The hardware system works correctly.

## Restrictions

1. The multiple buffering of the original solutions must be retained as long as the transmissions are successful. (Efficiency should not be drastically reduced because of reliability requirements.)
2. The transmitted buffers must be received in sequence.
3. The resynchronization of the producer and consumer must be performed using SEND-RECEIVE primitives and global data.

## 3. SOLUTIONS TO THE RESYNCHRONIZATION PROBLEM

The program of Figure 3 solves the producer-consumer problem in the absence of error. The SEND-RECEIVE primitives are used to transmit the location of the data that is the real object to be transferred. Thus this is essentially an access by reference. Furthermore, although the data flow is in only one direction (from producer to consumer), the addresses of empty and full buffers travel in both directions. This gives the two processes of the solution a pleasant symmetry; it is not difficult to be convinced that this program is a correct solution.

Three solutions to the resynchronization problem that access the message buffers by reference are now discussed in detail. The first two examples are solutions to problems of type $F_i$; messages that have been produced are allowed to stay in the global buffers and are used after the invalid message has been corrected. The third solution discards messages that were produced after the incorrect buffer; this solution applies to failures of type $F_d$.

This section concludes with a short description of some other methods of providing error recovery in the producer-consumer system, followed by a brief discussion of the delay problem associated with many of the available solutions.

## Access by Reference

How can the system of Figure 3 be extended to provide for retransmission of buffers that were incorrect when obtained by the consumer?

In the first place, if the multiple buffering of the original program is to be maintained,

```
type buff = <<information to be transmitted>>;
     index = 1..n;
var buffer: array(index) of buff;
    empty, full: messagelist(n) of index;

procedure producer;
  var emptyid: index;
  begin
    while true do
      begin
        receive(empty,emptyid);
        fill buffer(emptyid);
        send(full,emptyid);
      end;
  end producer;

procedure consumer;
  var fullid: index;
  begin
    while true do
      begin
        receive(full,fullid);
        empty buffer(fullid);
        send(empty,fullid);
      end;
  end consumer;

begin
  for i := 1 to n do send(empty,i);
  parbegin producer; consumer parend;
end.
```

Figure 3. Access by reference

then there are some constraints on the ways that the consumer can acknowledge the correctness of the data. In particular, the consumer cannot be required to send an explicit "ok" back to the producer after each buffer and before the producer is allowed to fill the next buffer. This would cause lock-step synchronization between the two processes, and the multiple buffering would be lost.

If the producer is to be able to get 0, 1, or more buffers ahead of the consumer, then the consumer cannot just return a simple error indication, since the producer would not know which buffer was invalid. The consumer must indicate to the producer which buffer needs to be corrected and retransmitted. Either the consumer must explicitly indicate the address of the buffer that needs to be retransmitted, or the contents of the buffer must be self-identifying. The former case is considered first.

When the producer is informed of a contaminated buffer, it can correct the buffer using redundant information of some form. It then must signal the consumer in some way that the buffer is ready to be obtained again. One way to signal the consumer is by sending a message in a message list. But the same messagelist full that is used to transmit the addresses of the original buffers cannot be used since this may allow other buffers to be obtained out-of-order. A different messagelist must be used.

The solution of Figure 4 seems to satisfy all these requirements. Messagelist err is used to tell the producer if there was an invalid buffer, and if so, which one it was. Messagelist checked is used to let the consumer know when the buffer has been corrected. (Checked is a messagelist of type null, and has no data passed through it; SEND and RECEIVE then act like Dijkstra's V and P synchronizing primitives [5].)

In this solution an extra SEND-RECEIVE pair is generated for every buffer that is transmitted from the producer to the consumer. If the overhead is large for messagelist operations then it may be desirable to avoid some of these operations by encoding some of the error information in global shared data. An example of a solution that uses global shared data to pass error information is given in Figure 5.

The solution in Figure 5 illustrates a problem that often arises because there are two different actions the consumer must take, depending on whether the buffer was in error or not.

(1) If the buffer was not invalid, then the consumer must first
   (a) empty the buffer (consume it)
and then (b) return the buffer (SEND(empty, fullid) or equivalent).

(2) If the buffer was invalid, then the consumer must first
   (a) return an error indication,
   (b) wait for a correct version of the buffer contents to arrive,
and then (c) empty the buffer.

In case of an invalid buffer (case 2), some kind of synchronization is necessary so that the producer will know when it can examine the error indicator. If busy waiting is to be avoided, this must be done by a RECEIVE command. The error indication may be passed in the messagelist of the RECEIVE command itself, or it may be in a shared global variable. Further, if this RECEIVE command is not to add too much overhead to the solution, it must be the same command that obtains the empty buffer.

But now there are two different orders of the operations <return empty buffer> and <empty the buffer> and the consumer must deal explicitly with the two different cases. The two cases may be combined in the consumer but only at the expense of clarity. Recall that the producer must use two different messagelists to receive error indications, or out-of-order problems could occur. Therefore the consumer must synchronize the error handling by using these two different messagelists. This further complicates combination of the error and no-error

In Figure 5, the error response is initially sent through the global buffer, and the messagelists retry and retryok are subsequently used to synchronize the retransmission of the corrected buffer.

All of the solutions so far have assumed that the processes themselves explicitly indicate the addresses of the buffers to be retransmitted and that the buffers contain no self-identifying information. If, for example, the records are numbered when sent, then retransmission can be synchronized by indicating the number of the buffer to be resent. An example of this type of solution is shown in Figure 6.

If the consumer detects an error, it returns an "empty" buffer with the message "error - please start over with message #_". The producer then uses its redundant information to start over at the appropriate record. It may use the same messagelist for retransmission as well, since the consumer can just ignore records until the proper # record arrives. Thus the out-of-order problem is solved.

## Other Solutions

Solutions to the error recovery problem for the producer-consumer system can be characterized by at least five factors: the failure situation that is solved, the type of solution that is used, the location where the error indication is passed, the way that the incorrect message is identified, and finally the particular version of synchronizing primitives that is used. Of course, these factors are highly interdependent. Some of the possibilities are discussed below.

Two relevant failure situations have already been discussed: independent failures, $F_i$, and dependent failures, $F_d$.

Two basic solution types also exist: retry and purge. In retry solutions the invalid buffer is repeated until accepted by the consumer; any already-generated but not-yet-consumed buffers are

```
type buff = <<information to be transmitted>>;
     index = 1..n;
     etype = 0..n;
var buffer: array(index) of buff;
    empty, full: messagelist(n) of index;
    err: messagelist(n) of etype;
    checked: messagelist(1) of null;

procedure producer;
  var emptyid: index;
      etyp: etype;
  begin
    while true do
      begin
        receive(err,etyp);
        while etyp ¬= 0 <<error occurred>> do
          begin
            correct buffer(etyp);
            send(checked);
            receive(err,etyp);
          end;
        receive(empty,emptyid);
        fill buffer(emptyid);
        send(full,emptyid);
      end;
  end producer;
procedure consumer;
  var fullid: index;
  begin
    while true do
      begin
        receive(full,fullid);
        while buffer(fullid) is in error do
          begin
            send(err,fullid);
            receive(checked);
          end;
        <<now buffer(fullid) is ok>>
        send(err,0);
        empty buffer(fullid);
        send(empty,fullid);
      end;
  end consumer;

begin
  for i := 1 to n do
    begin send(empty,i); send(err,0) end;
  parbegin producer; consumer parend;
end.
```

Figure 4. Retry solution
(error location in messagelist)

```
type buff = <<information to be transmitted>>;
     index = 1..n;
var buffer: array(index) of buff;
    empty, full: messagelist(n) of index;
    retry: messagelist(1) of null;
    retryok: messagelist(1) of Boolean;

procedure producer;
  var emptyid: index;
      errpresent: Boolean;
  begin
    while true do
      begin
        receive(empty,emptyid);
        errpresent := buffer(emptyid) = "error";
        while errpresent do
          begin
            correct buffer(emptyid);
            send(retry);
            receive(retryok,errpresent);
          end;
        fill buffer(emptyid);
        send(full,emptyid);
      end;
  end producer;

procedure consumer;
  var fullid: index;
      errpresent: Boolean;
  begin
    while true do
      begin
        receive(full,fullid);
        if buffer(fullid) in error then
          begin
            buffer(fullid) := "error";
            errpresent := true;
            send(empty,fullid);
            while errpresent do
              begin
                receive(retry);
                errpresent := buffer in error;
                if ¬errpresent then
                  empty buffer(fullid);
                send(retryok,errpresent);
              end;
          end
        else
          begin
            empty buffer(fullid);
            send(empty,fullid);
          end;
      end;
  end consumer;

begin
  for i := 1 to n do send(empty,i);
  parbegin producer; consumer parend;
end.
```

Figure 5. Retry solution
(error location in buffer)

```
type buff = <<information to be transmitted>>;
     index = 1..n;
var buffer: array(index) of
              record serno: integer; info: buff end;
    empty, full: messagelist(n) of index;

procedure producer;
  var emptyid: index;
      next, lastgen: integer;
  begin
    next := 0;
    lastgen := 0;
    while true do
      begin
        receive(empty,emptyid);
        with buffer(emptyid) do
          begin
            if info = "error" then next := serno
              else next := next + 1;
            if next = lastgen + 1 then
              begin
                produce record # next;
                lastgen := next;
              end;
            serno := next;
            info := record # next;
          end;
        send(full,emptyid);
      end;
  end producer;

procedure consumer;
  var fullid: index;
      lastrec: integer;
  begin
    lastrec := 0;
    while true do
      begin
        receive(full,fullid);
        with buffer(fullid) do
          begin
            if info is in error then
              begin
                serno := lastrec + 1;
                info := "error";
              end
            else
              if serno ¬= lastrec + 1 then
                ignore it
              else
                begin
                  lastrec := lastrec + 1;
                  empty info;
                end;
          end;
        send(empty,fullid);
      end;
  end consumer;

begin
  for i := 1 to n do
    begin
      send(empty,i);
      buffer(i).info := "ok";
    end;
  parbegin producer; consumer parend;
end.
```

Figure 6. Purge solution

left alone, and are received when the consumer resumes normal operation. In purge solutions the producer starts over at the incorrect buffer; any already-generated but not-yet-consumed buffers are thrown away. (The solutions of Figures 4 and 5 are retry solutions; the solution of Figure 6 is a purge solution.) In order to handle a failure of type $F_d$ a purge solution must be used. On the other hand, failures of type $F_i$ may be handled by either retry or purge solutions. The risk is that perfectly good buffers (which might be very expensive to generate) may be discarded needlessly.

The actual indication of error may be passed in several locations. In the solution of Figure 4, the error message was passed in a messagelist. In the other two examples the error messages were in the global shared message buffers; there was one error location per buffer. Other solutions might have one error location per buffer, where the error locations are in global shared memory but separate from the message buffers, or they might have one error location which applies to all buffers at once.

The buffer that is invalid and must be corrected can be identified by giving each message a serial number, as in Figure 6. An alternate method, which is applicable when the buffers are in global shared memory, is to indicate the address or the array index of the bad buffer. If an array of error indications is used, the value of the array element can specify the correctness of the corresponding buffer. The exact method of specifying the incorrect message is clearly related to where the error indication is passed and other possibilities than those mentioned here may be found.

The exact form of the SEND-RECEIVE primitives used also characterizes a particular solution. In this section messages were passed by reference. By adding a return error path to Figure 2, a solution could be found that passed the messages by value. But passing messages by value in a messagelist means that individual buffers are not addressable; thus, some of the ways to identify incorrect buffers are not possible. In particular, the messages must contain their own identification numbers.

If all information is removed from the messagelists, they become messagelists of type null and are equivalent to Dijkstra's P-V primitives. If only this type of primitives is used, then a messagelist may not be used to convey the error indication and identify the erroneous buffer, and global variables must be used. (Of course, the primitives may be used to define a critical region and therefore to simulate a messagelist.)

It is possible to combine several of these techniques in a single solution. For example, messages could be passed efficiently by reference until an error was detected. Then the two processes could communicate the attempts to correct the buffer contents by value until an acceptable transmission was completed.

A diagram showing the relationships among solutions characterized in various ways is given in Figure 7. An arrow from A to B indicates that the problem or characterization of A may be resolved or implemented using the characterization of B. (Note that the diagram does not include characterizations based on the location of the error indications.) The diagram represents the principal approaches suggested to handle failure situations; however, other designs may be possible.

Delay in Error Response

All of the solutions presented so far suffer from a peculiar delay problem caused in part by the requirement that multiple buffering be retained in the resulting programs.

When the consumer sends an error indication to the producer, the producer may not obtain this error message for some time. In fact, if the error message is sent in a messagelist, in a global array of buffers, or in a global array of error flags, the error message will not be seen until the entire set of error flags has been examined. If the messagelist or global array has a capacity of n error messages, up to n buffers may be produced before the error message is received. In the case of a retry solution, the consumer must wait for the corrected buffer and the entire buffer will be filled with produced, but unconsumed messages. In the case of purge solutions, n messages are produced and then thrown away by the consumer when the producer restarts at the corrected buffer; if this message is again invalid, another n messages will be generated and thrown away.

This delay in seeing an error message is due directly to the fact that n different locations for error indications are provided and that an immediate response is therefore not always possible. One possible solution is to look at all n global error flags every time through the producer loop; the error indications can not therefore be sent in a messagelist. However, a single global location can act as an error flag and identify the invalid message. This is because only one buffer at a time will be flagged as invalid, since retry solutions wait for the corrected version of that buffer, and purge solutions ignore other buffers until the corrected version is received.

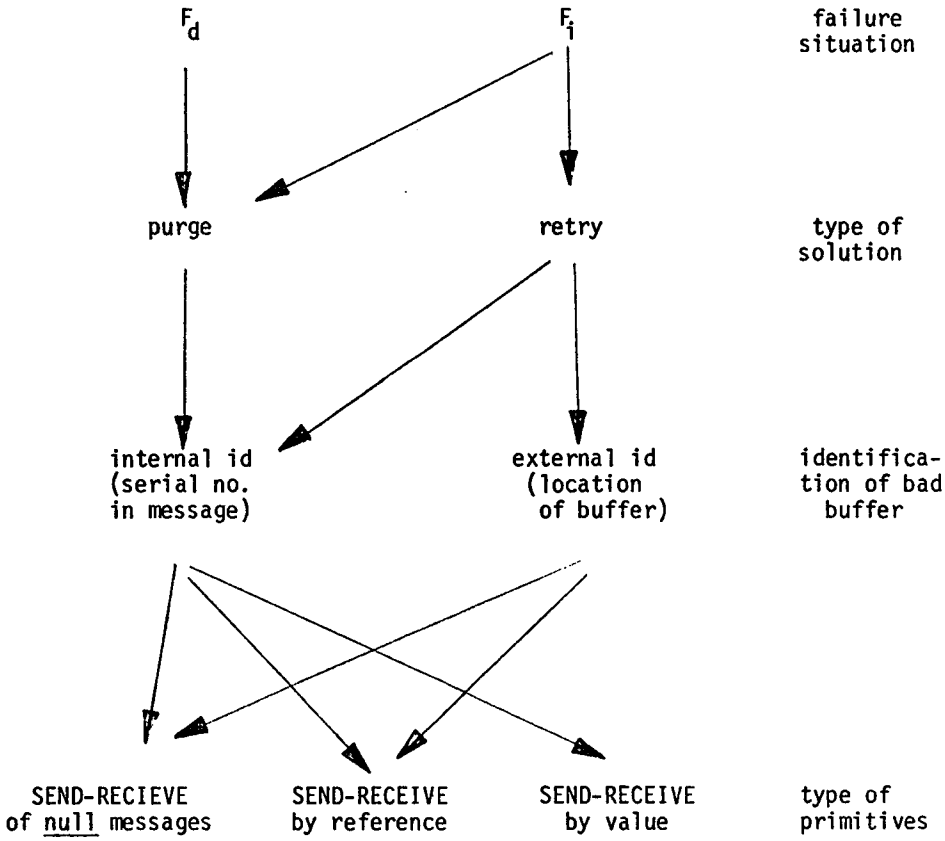


Figure 7. Characterization of resynchronization solutions.

Using a single global error flag can lead to difficulties. Since an immediate response is desired, the producer must look at the error flag on every execution of its main loop. If the producer sees that there are no outstanding errors to be corrected, it gets an empty buffer and proceeds to fill it; if there are no empty buffers, the producer will be blocked. In the retry case, however, the consumer must release the buffer to avoid deadlock. This is similar to the situation in the solution of Figure 5, where the buffer containing the error message must be returned, and leads to the same complications of having two separate operation sequences in the error and no-error cases.

## 4. CONCLUSIONS

An analysis of solutions to a resynchronization problem for producer-consumers systems has been made. A few of the many possible solutions were considered in detail. Several factors that are difficult to avoid appear in each of these solutions. These factors are typical of situations that exist in asynchronous computation and include the following:

- -- unwanted messages are left in messagelists and buffers when an error is detected, and it is difficult to get rid of this extraneous data;
- -- multiple buffering causes difficulties in relaying immediate error messages to the process that is responsible for correcting the error, and thus undesirable delays may be generated;
- -- when the sequence of messages is significant, the difficulty of error processing is increased because either two data streams must be provided (one for regular transmission and one for corrected and retransmitted buffers), or the consumer must be able to purge unwanted out-of-order records;
- -- the error and no-error cases must often be processed differently.

The solutions presented for the resynchronization problem are unsatisfying due to the increase in complexity and decrease in understandability of the resultant programs. However, by presenting these solutions we hope to point out the difficult nature of error processing and also to delimit some of the fundamental problems that must be solved in developing simpler approaches to asynchronous error handling.

## 5. ACKNOWLEDGEMENTS

The authors would like to thank Hector Garcia for the solution presented in Figure 4.

## 6. REFERENCES

1. Wulf, W. A. Reliable Hardware-Software Architecture. Proceedings 1975 International Conference on Reliable Software, SIGPLAN Notices 9,6 (June 1975), 122-130.

2. Parnas, D. L. Response to detected errors in well-structured programs. Technical Report, Department of Computer Science, Carnegie-Mellon University, July 1972.

3. Parnas, D. L. The influence of software structure on reliability. Proceedings 1975 International Conference on Reliable Software, SIGPLAN Notices 9,6 (June 1975), 358-362.

4. Fajman, R. and Borgelt, J. WYLBUR: An interactive text editing and remote job entry system. Comm. ACM 16,5 (May 1973), 314-322.

5. Dijkstra, E. W. Cooperating Sequential Processes. Technological University, Eindhoven, The Netherlands, 1965. (Reprinted in Programming Languages, Genuys (ed.), Academic Press, New York, New York, 1968.)

6. Wirth, N. The programming language PASCAL. Acta Informatica 1, 1 (1971), 35-63.

7. Brinch Hansen, P. The nucleus of a multiprogramming system. Comm. ACM 13, 4 (April, 1970), 238-241,250.

8. Brinch Hansen, P. Operating System Principles. Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1973, ch. 3.