The Reliability of Programming Systems

H. Gerstmann, H. Diel, and W. Witzel,
IBM Germany, Boeblingen

1. ABSTRACT

The reliability of a programming system is not only determined by the number of errors to be expected, but also by its behaviour in error situations. An error must be kept local to identify its origin and annul its effects at an tolerable expense. This paper discusses a uniform approach to the limitation of error propagation, the identification of the process in error, and the provision for error recovery.

2. THE MODEL

The concepts of reliability are described for a model of a programming system which consists of three basic types of objects:

(1) procedures, which may be nested as in higher level languages

(2) a state space, represented by the variables declared within the procedures

(3) processes, which are the units of asynchronous operations.

The notions used are taken from reference [1]. The resources of the system, also called objects, are represented by variables. All variables constitute the variables state set

$$R = \{x1, x2 \ldots, xn\}.$$

An assignment of values to all the variables in the state variable set defines a state of the system. The set of possible states is the state space. With each variable xi a type is associated which defines the set Vi of values it may assume. In these terms the state space can be written as

$$S = V1 \times V2 \times \ldots \times Vn.$$

The set of processes

$$\{P1, P2, \ldots, Pn\}$$

is partially ordered by a precedence relation

$$Pi < Pk ,$$

which can be illustrated in the form of a diagram (figure 1).

All processes Pi, such that $Pi < Pk$ must have completed before Pk can be initiated.

Each process P uses a subset Rp of the set R of resources. These objects define the subspace of the system state space in which the actions of the process take place. Resources utilized by more than one process are shared resources. Input resources Rip to P are shared resources set by other processes which are referenced by P, output resources Rop those set by P and referenced by other processes.

The input state Si of process P is defined by the state of each input resource at process initiation. Correspondingly the state of the output resources at process termination describes its output state So.

With each process a set of input states {si} is associated for which a mapping Fp to ouput states {So} is defined (figure 2).

From a functional point of view Fp is a partial function. No action is defined in case P is initiated in a state S {Si}. The next section is devoted to this exceptional situation.

3. EXCEPTION HANDLING

It has long been recognized by engineers that instructions perform partial functions. To cope with them, exceptions have been introduced. The ZERODIVIDE and OVERFLOW conditions are typical examples.

Higher level languages either ignore this property or just support exceptions on the instruction level as in the case of PL/I. There is, however, no consistent treatment of exceptions at the level of procedures. The argument that such a feature is not needed goes as follows: Exceptions at the procedure level either can be programmed or reduced to hardware exceptions.

Although this is a true statement, it expresses a narrow attitude with respect to the purpose of a language. The semantic distinction between functional and exceptional actions should also be reflected in the syntax of the language.

As indicated in figure 3 the functional action Fp expresses the function of the procedure as long as its arguments are in {Si}. If this is not the case the exceptional action Ep maps the invalid state into an exception description.

To support this property in a programming language such as PL/I, extensions of the following kind are required:

(1) The values a scalar variable may assume can be constrained by appending a range to its data attribute.

(2) The values of structured variables (including arrays) can be constrained by imposing relations between its subcomponents.

(3) A built-in-function RANGE which returns '1'B or '0'B depending on wheter the argument lies in its range or not.

(4) A built-in-function ON_ERROR_DESCR which returns an error description in the form of a structure:

```
1 ERROR_DESCR
  2 ERROR_TYPE
    3 ERROR_MAIN_TYPE
    3 ERROR_SUB_TYPE
  2 STATEMENT_NO
  2 STATEMENT_LABEL
  2 AFFECTED_VARIABLES
    3 VARIABLE1
    .
    .
    .
```

Figure 4 shows the use of these language constructs to define exceptional actions. The language elements introduced should not be considered as a proposal to extend PL/I. Its purpose is to indicate the direction in which extensions are needed to separate the functional part of a procedure from its exceptional part. A consistent solution cannot neglect type attributes as provided in PASCAL [2,3].

4. ERROR ISOLATION

The capability to isolate errors in a system does however not only depend on the realization of the partial function concept. Additional system properties are required to attribute an error unequivocally to a certain process: At each point of time only one process may update a shared resource.

To this end the use of shared resources must be restricted.

Two different cases are to be considered:

Case 1

The resource is shared by processes which lie on a path through the system (figure 5). Since $P1 < P2$ it is always possible to allocate the resource R in such a way that

$$\text{deallocate}(R,P1) < \text{allocate}(R,P2).$$

During the execution of processes at any point of time R is allocated to at most one process.

Case 2

The resource is shared by processes which do not lie on a path through the system (Figure 6).

In this situation the direct access to the resource is prevented by establishing an interface between the processes and the resource. Assuming that the processes either want to read or to update (read and write) the resource, they have to initiate separate atomic processes READ (R, Pi) or UPD (R, Pj) which are associated with R and obey the constraints indicated in figure 7.

An empty circle represents any other process including read and update. Dependent on the intended use of the resource R the processes Pi are decomposed into subprocesses. According to above constraints, disregarding symmetry, this results in one of the three types of diagrams shown in figure 8.

By means of this device case 2 is reduced to case 1. The process administering the resource R and the associated set of processes $\{READ(R, Pi)\} \cup \{UPD(R, Pj)\}$ in accordance with above constraints is called a resource manager.

There is an interesting parallel to the concept of monitors introduced by Brinch Hansen [4].

Due to the use of resource managers a unique path of serial processes can be associated with the state changes of each shared resource (figure 9).

For the purpose of error isolation each process including those controlled by resource managers is requested to check its input states.

Whenever an error is detected by a process Pk it must have been caused by some process Pi < Pk on the path for the resource concerned. In any case, Pk will accuse its immediate predecessor Pk-1 of having made an error based on the following consideration:

Either Pk-1 caused the error during its execution or made an error in accepting an erroneous input state.

As indicated in figure 10, going the path backwards in this way, the process originating the error can be identified. The process Pk may wrongly accuse Pk-1 to have supplied faulty input. To settle this case, it is necessary that obligatory specifications detailing the interfaces between processes have been established before the implementation.

The language features described for input checking were introduced in the previous section. Their use is now described. Constraints imposed on the state space are either process or system specific. Process specific constraints define the admissible input states of a process. Formal parameters are to be specified with ranges. Dependencies between global variables and/or formal parameters are checked as indicated.

System specific constraints are properties of shared resources represented by global variables. To maintain their integrity ranges are appended. Since the sequence in which a shared resource will be used by the processes is undetermined, the ranges must express invariant properties, i.e., the conditions imposed on its state before and after process execution must be the same (figure 11).

The embedding of on-units in process hierarchies is the subject of the next section.

5. ERROR RECOVERY

The concepts developed for error isolation are not sufficient for the purpose of recovery. This can be shown by the following example:

Process P1 uses resource R to provide input to process P2.

Case 1

The resource R is used by the serial processes P1 and P2 (figure 12). After providing input to P2 the process P1 terminates. P2 detects an input error. Recovery must comprise process P1 which is no longer in existence.

Case 2

The resource R is used by the parallel processes P1 and P2 (figure 13). After providing input to P2, process P1 continues to exist and discovers an error affecting R. Process P1 has to return to a previous state and recall the data supplied to P2. Therefore recovery must also include process P2.

Although in this situation the resource manager is responsible for the input check and errors violating the constraints imposed on R are detected before P2 is initiated, the subprocess P12 may consider the values supplied to R as inconsistent according to the internal semantics of the program.

Thus the concept of input validation as described for the purpose of error isolation must be extended for the purpose of recovery. Two strategies which supplement eachother are discussed

- a discipline with respect to data communications (commmitment discipline)

- a hierarchical structure of processes with respect to recovery.

The intent of the commitment discipline is to enforce that no data is committed outside a process unless either it is ensured that there will never be a need to recall the data or there is a mechanism available to do it.

As described in section 2, a process makes use of input and output resources. For the purpose of commitment values submitted to output resources are classified as:

uncommitted - Data the receiving process cannot rely on. It will not be recalled in case the sending process fails.

committed - Data the sending process commits as consistent to the receiving process. Consequently it will not be recalled by the sending process.

precommitted - Data that can exist in one of three states: 'OPEN', 'RECALLED' or 'COMMITTED'. The initial state is 'OPEN'. At recall or commitment the state is changed to 'RECALLED' or 'COMMITTED'.

This distinction is introduced in reference [5], however, applying different terminology and semantics.

Output is committed by the supplying process when it is considered consistent. The term 'consistent' remains undefined. It is up to the individual process to establish appropriate criteria. They should, however, at least guarantee valid output.

The commitment implies for committed data the release to other processes and for precommitted data a state transition from 'OPEN' to 'COMMITTED'. The data must be committed before the highest level process terminates to which the output variables are non-local.

Committed and uncommitted data do not introduce dependencies between processes which have to be considered for recovery. The situation is different for precommitted data. This notion allows to extend the scope of in-process recovery, which is based on the fact that the process to be recovered is still in existence.

Figure 14 shows the state diagram for precommitted data. The sending process sets the data in the initial state 'OPEN'. The associated resource manager guarantees that the data are not changed by any receiving process as long as they are in the state 'OPEN'. Only the sending process is entitled to change the state to 'COMMITED' or 'RECALLED'. The receiving processes are not permitted to terminate before the state 'COMMITTED' is entered. In addition they must not commit output which depends on input not yet committed.

This mechanism ensures that all dependent parallel processes are still in existence in case data have to be recalled. It therefore allows to apply in-process recovery to several processes. For back out each of them can be reset to the initial state which was kept at process initiation. Figure 15 illustrates the commitment discipline. Process P1 precommits data to the resource R and sets its state to 'OPEN'. The data can be read but not updated until the end of P21. Before P2 is permitted to update R and/or terminate its execution it must wait for the commitment of R by P1.

The commitment discipline cannot be applied to serial processes. To guarantee the existence of a process that can perform the recovery, the system should be designed as a hierarchy of processes. In cases where this hierarchy cannot be predefined measures for post-process recovery have to be introduced in the direction as described in reference [6]. Figure 16 shows an idealized system meeting above design constraints. The system is structured in three processes P1, P2 and P3. Each process Pi consists of subprocesses Pij. Recovery situations affecting only parallel processes such as P22 and P23 or P32 and P33 can be handled by means of the commitment discipline. In any other situation the process detecting the error has to escalate it to the next level in the process hierarchy. To achieve this on-units providing for recovery must also be ordered hierarchically. Figure 17 indicates a way how errors can be escalated to higher level on-units for recovery.

6. DISCUSSION

The concepts of reliability described in the preceding sections require a system partitioned into modules. Following Parnas [7] a system is considered well structured in case the interfaces between modules contain little information, where interfaces are the assumptions modules make about each other. To minimize the information being transferred, interfaces must be raised to a higher level of abstraction.

In this context the features discussed in the paper offer tools to enforce abstractions. As abstractions represent design decisions independent from the program flow, the features can only partially be provided automatically by a compiler. A compiler handles one external procedure at a time, whereas module interfaces comprise more than one external procedure. Also, not every external procedure declaration constitutes an abstract interface and an abstract interface may contain assumptions which cannot be expressed in terms of parameters.

The enforcement of interfaces requires additional effort at execution time. Sometimes performance reasons are pretended to reject an approach of this kind. There are at least two reasons which show that the argument is not stringent. PASCAL [8] has demonstrated that dynamic range checking can be implemented efficiently. Ranges, as proposed here, are more complex since they can be defined by any relation. On the other hand, they need not be checked at any reference but just at the interface. This leads to the second reason: It is the designers responsibility to minimize the information passed across interfaces.

7. SUMMARY

The preceding sections presented an attempt to handle errors systematically. Error isolation and recovery were treated under one aspect. Error isolation led to the realization of the partial function concept and the provision of resource managers. Error recovery in addition necessitated the introduction of a commitment discipline in conjunction with a hierarchical structure of processes.

REFERENCES

1. J.J. Horning and B. Randell: Process Structuring, Computing Surveys, Vol. 5, No 1, March 1973

2. N. Wirth: The Programming Language Pascal, Acta Informatica 1, 35-63 (1971)

3. N. Habermann: Critical Comments on the Programming Language Pascal, Acta Informatica 3, 47-57 (1973)

4. P. Brinch Hansen: Concurrent Programming Concepts, Computing Surveys, Vol. 5, No 4, December 1973

5. Ch. T. Davies, Jr.: Recovery Semantics for a DB/DC System, 1973 Proceedings of the ACM

6. L.A. Bjork: Recovery Scenario for a DB/DC System, 1973 Proceedings of the ACM

7. D. L. Parnas: Software Engineering or Methods for the Multi-Person Construction of Multi-Version Programs, published in these proceedings

8. N. Wirth: The Design of a PASCAL Compiler, Software, Vol 1, 309-333 (1971)

Precedence Relation between Processes

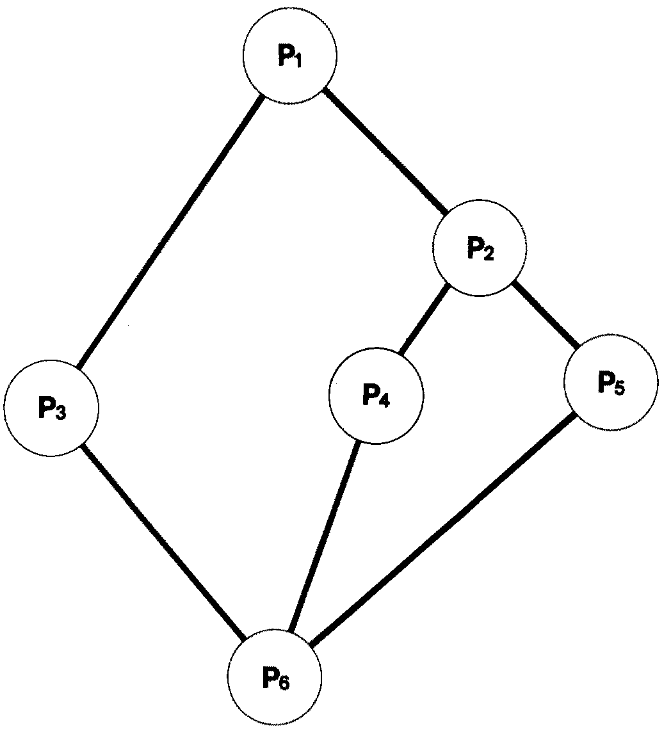


Fig. 1

Mapping of Input States to Output States defined by a Process

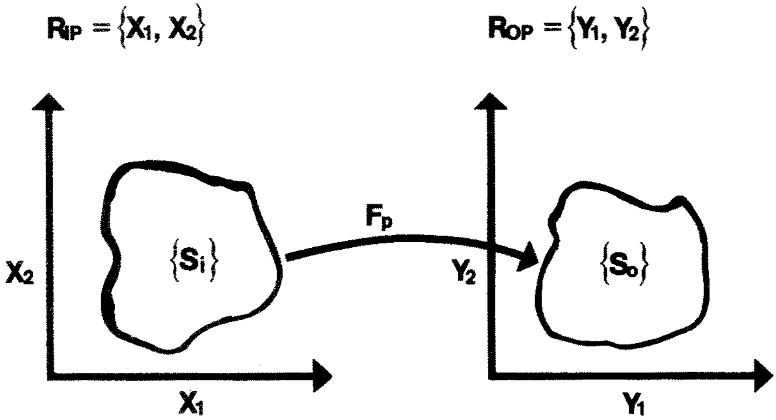


Fig. 2

Distinction between Functional and Exceptional Actions of a Process

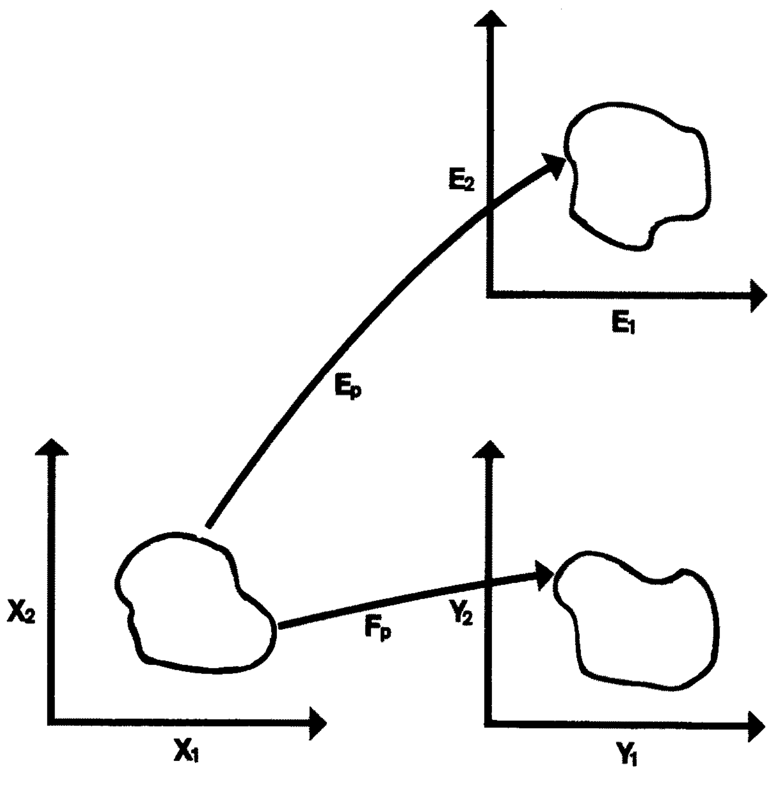


Fig. 3

Use of Ranges and Error Descriptions

Example 1:

```
1 X (I X1 < 10 * X2 I)
2 X1 INTEGER (I 0 ... 99 I)
2 X2 INTEGER (I 0 ... 10 I)

BLOCK ARRAY (10) CHAR (4) (I 'READ', 'WRTE', 'WAIT' I)
```

Example 2:

```
P: PROC;
   DCL (X, Y) INTEGER (I 0 ... 10 I) EXTERNAL, 1 E ... ;
     •
     •
     •
   ON CONDITION (INPUT) BEGIN; ... ; E =
     •                          ON_ERROR_DESCR; ... ; END;
     •
     •
   IF ¬(RANGE (X) ∧ RANGE (Y) ∧ X < Y) THEN
     •                          SIGNAL CONDITION (INPUT);
     •
     •
   END;
```

Fig. 4

Shared Resource used by Sequential Processes

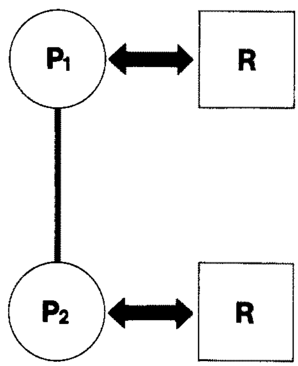


Fig. 5

Shared Resource used by Parallel Processes

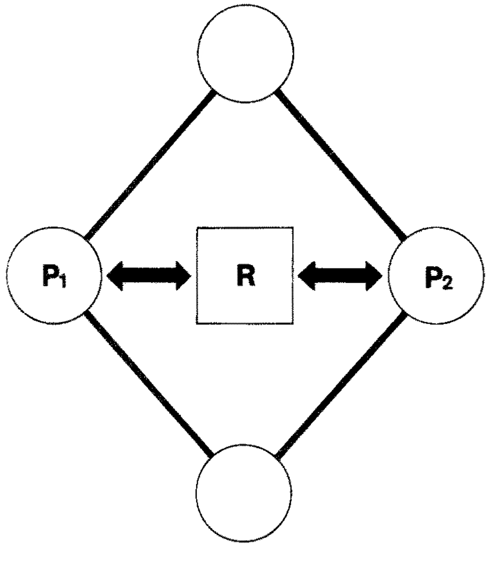
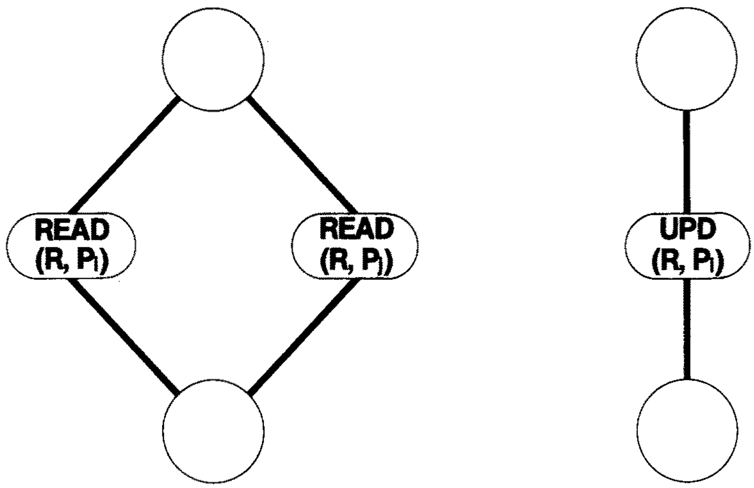


Fig. 6

Processes generated by Resource Manager



Decomposition of Processes into Subprocesses to access Shared Resource through Resource Manager

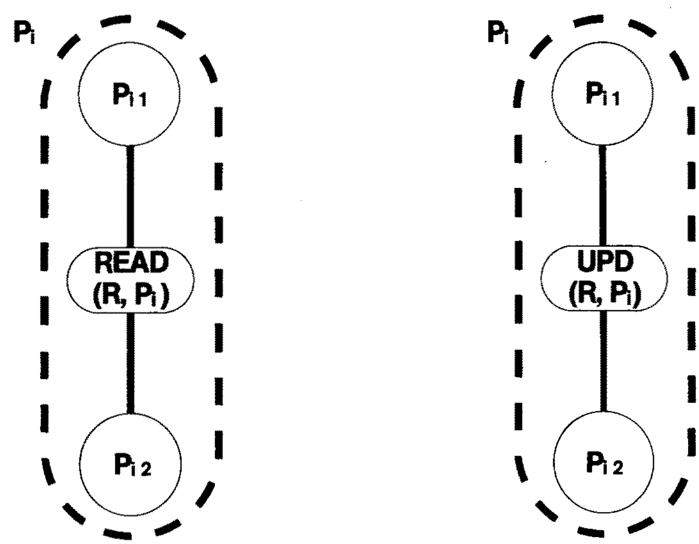


Fig. 7

Sequentialization of Accesses to Shared Resource by means of Resource Manager

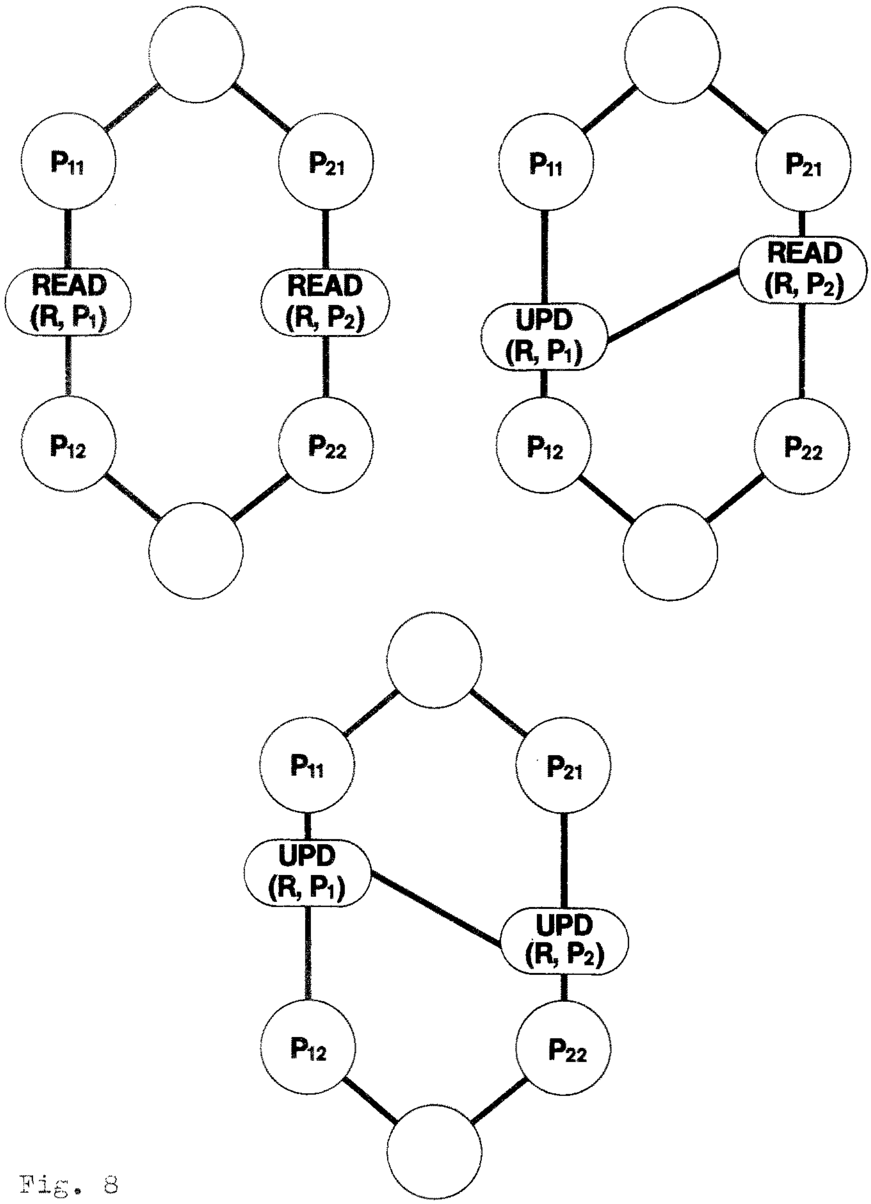


Fig. 8

Unique Correspondence between State Changes of Resources and Sequential Processes

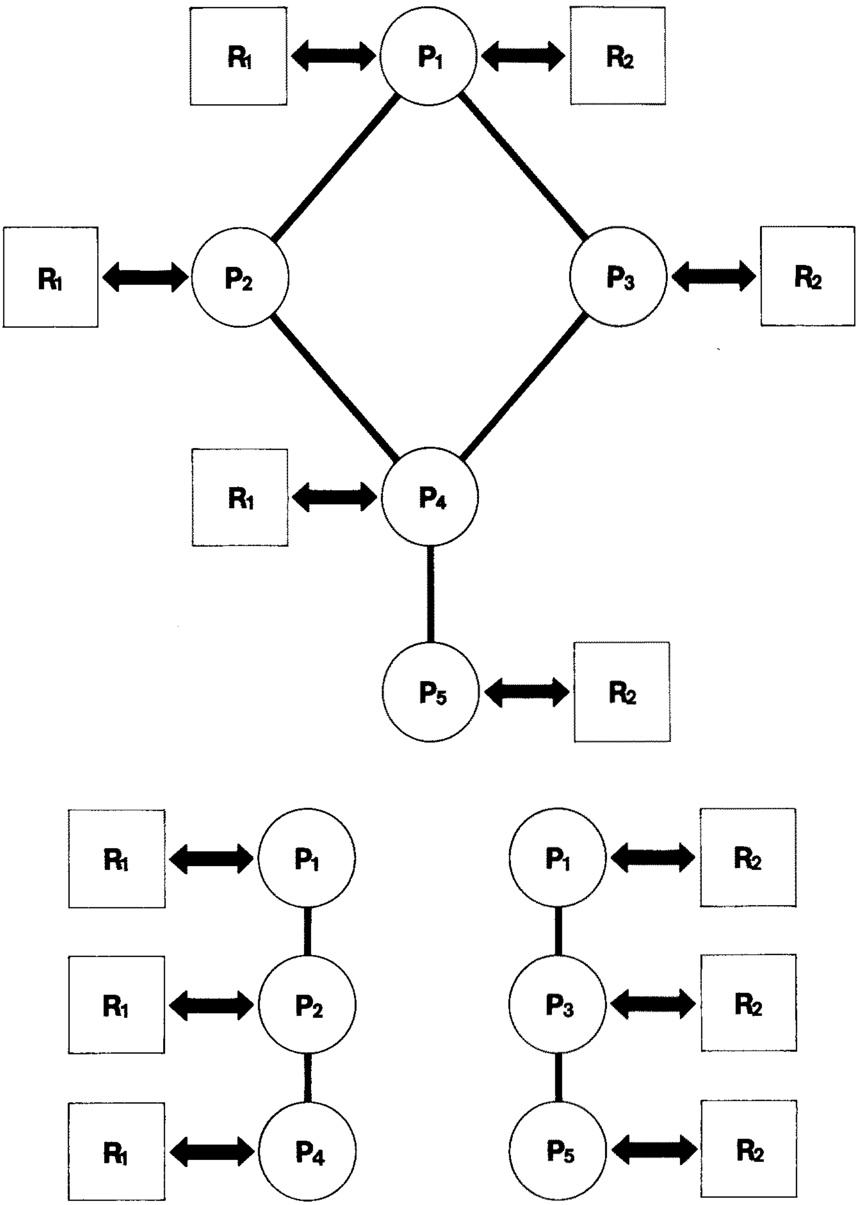


Fig. 9

Backtracking to locate Error

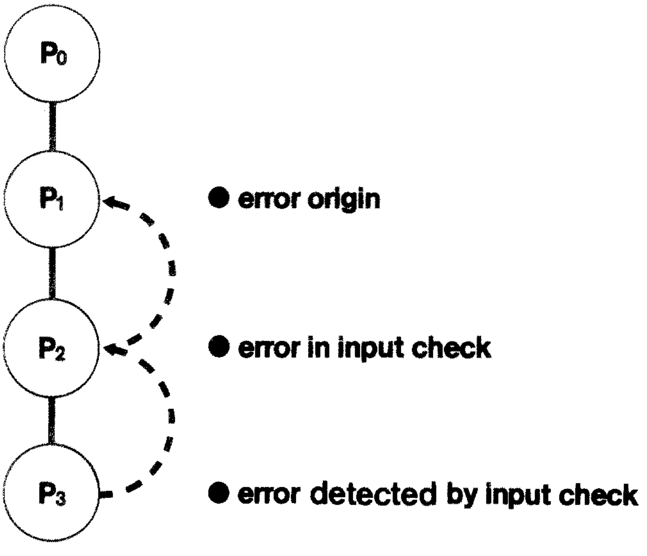


Fig. 10

INVARIANT RESOURCE CONSTRAINTS

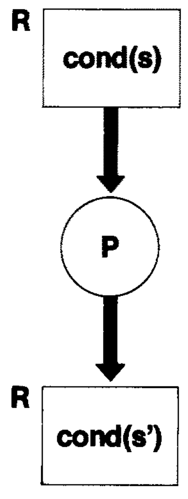


Fig. 11

Scope of Recovery for Sequential Processes

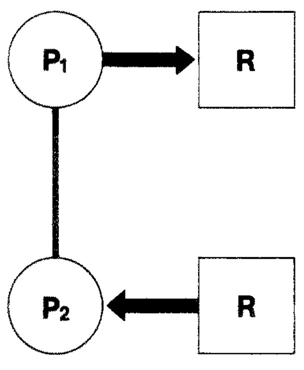


Fig. 12

Scope of Recovery for Parallel Processes

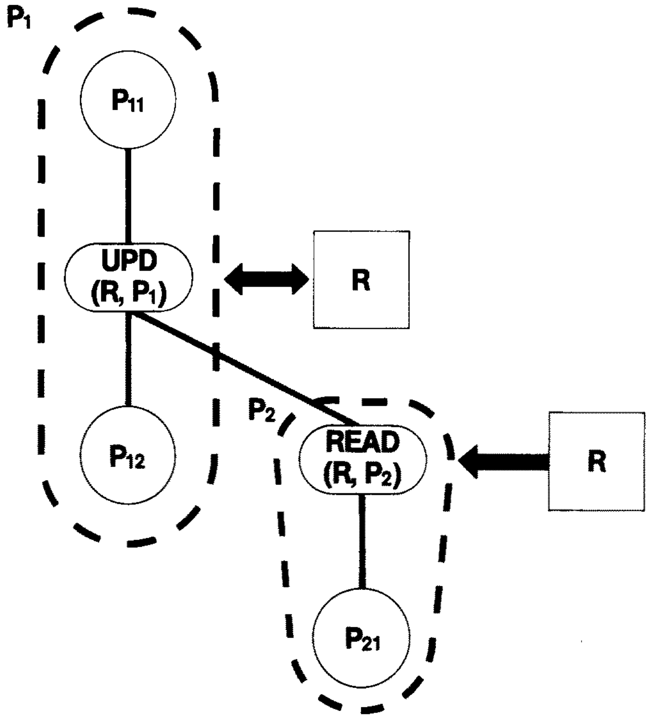


Fig. 13

State Diagram for Precommitted Data

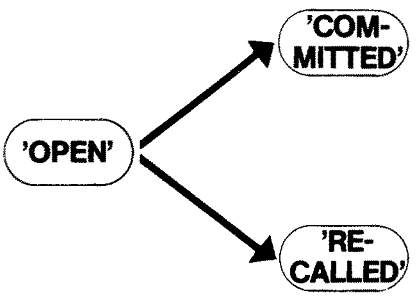


Fig. 14

Use of Precommitted Data

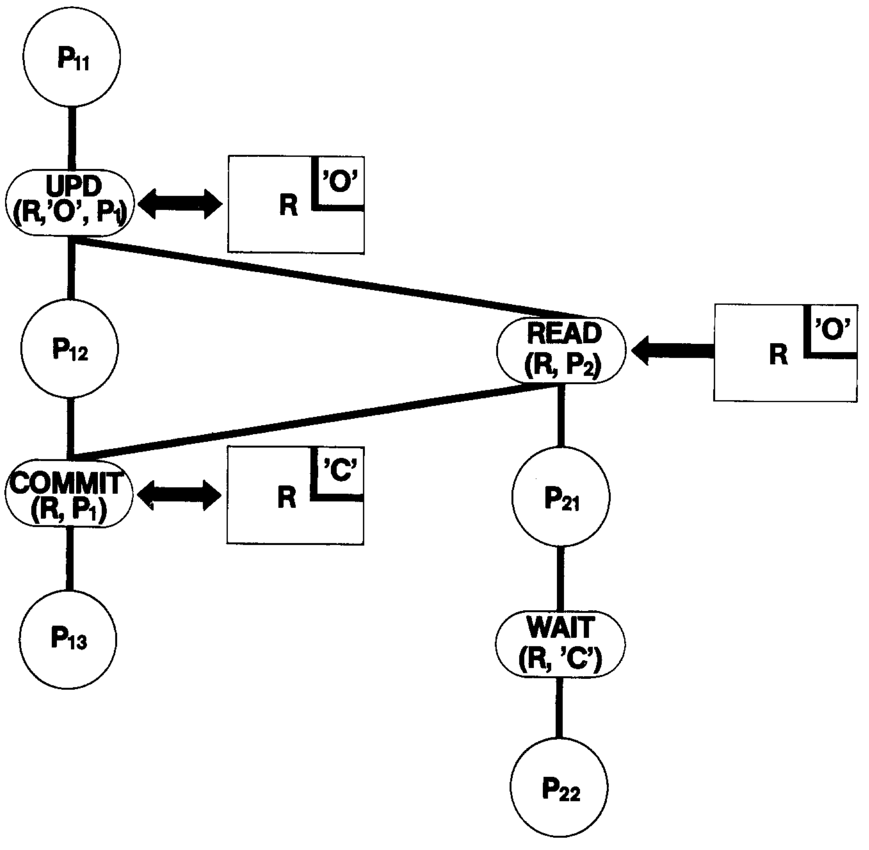


Fig. 15

Hierarchy of Processes

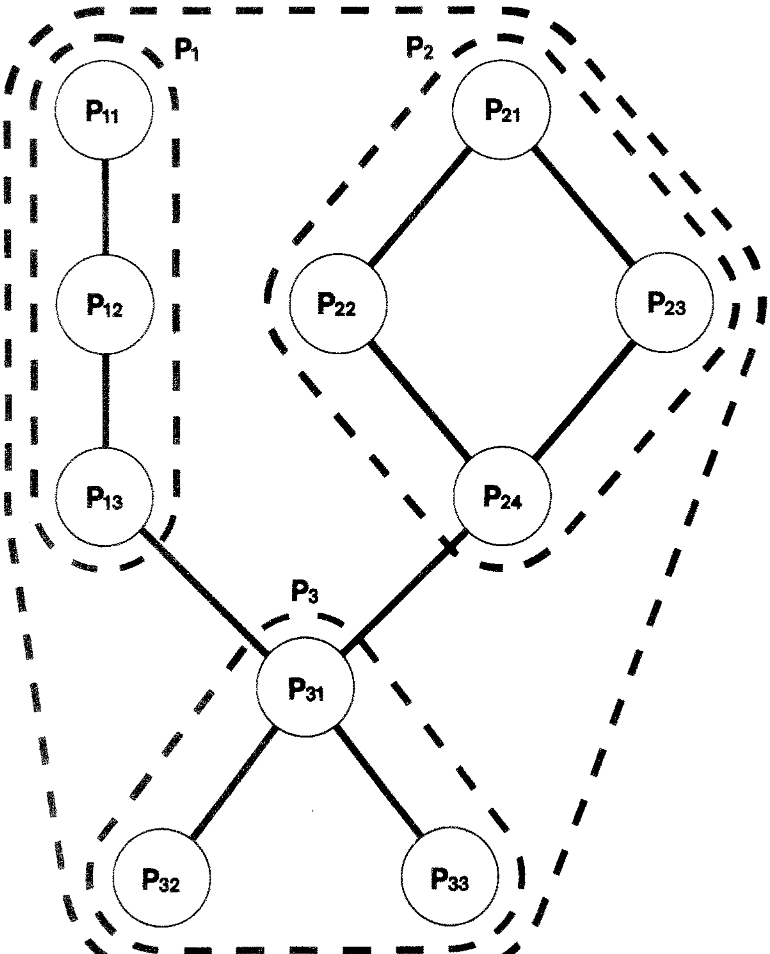


Fig. 16

Escalation to Higher Level On-Units

```
P1: PROC;
     .
     .
     .
    P11: PROC; ... END P11;
    P12: PROC;
          .
          .
          .
      ON CONDITION (INPUT 12) BEGIN;
                               .
                               .
                               .
                          E = ON_ERROR_DESCR;
                               .
                               .
                               .
                          SIGNAL CONDITION (ESCALATE);
                               .
                               .
                               .
                             END;
          .
          .
          .
     IF input-error THEN SIGNAL CONDITION (INPUT 12);
          .
          .
          .
     END P12;
     .
     .
     .
    ON CONDITION (ESCALATE) BEGIN; ... ; END;
     .
     .
     .
    CALL P11;
     .
     .
     .
    CALL P12;
     .
     .
     .
    END P1;
```

Fig. 17