Consiglio Nazionale delle Ricerche

ISTITUTO DI ELABORAZIONE DELLA INFORMAZIONE

PISA

ERROR DETECTION / FAULT TREATMENT IN THE MUTEAM SYSTEM

D. Briatico, A. Ciuffoletti, L. Simoncini
L. Strigini

Nota interna B84-06

Giugno 1984

ERROR DETECTION / FAULT TREATMENT IN THE MuTEAM SYSTEM.

D. Briatico *, A. Ciuffoletti *, L. Simoncini **, L. Strigini **

* Dipartimento di Informatica, Universita' di Pisa
** Istituto di Elaborazione dell'Informazione, CNR, Pisa

ABSTRACT

MuTEAM constitutes a continuing effort to investigate the potentiality of control decentralization in the design of multimicroprocessor systems. The architecture which has been so far designed and implemented provides a set of mechanisms, in the hardware, in the kernel an in the programming language, to help the exploitation of decentralized non-hierarchical policies for both resource management and fault treatment.

In this paper a possible ED/FT (Error Detection/Fault Treatment) policy is described, and the functionality of a precompiler, for its implementation at a very high level, is detailed. The use of such a tool allows complete transparency to the user of the redundancy which is necessary for a reliable computation.

1. INTRODUCTION.

MuTEAM [1], [2], [3], [4], is part of the Computer Science Program of the National Council of Researches MUMICRO Project for the development of multimicroprocessor systems for process control applications.

MuTEAM is an experimental prototype, to be used for the evaluation of hardware and software design concepts. It supports concurrent programming and decentralized non hierarchical policies for both resource management and fault treatment.

The exploitation of concurrency and decentralization of control has required a careful design of all the functional levels, from the programming language, to the kernel, to the physical configuration. Several constructs and mechanisms have been designed and inserted, to provide the possibility of implementing a robust system.

The system is constituted by a set of computer elements (nodes); the operating system is replicated in each node. The details of the organization are in [1], [2], [3], [4].

1.1. Programming language.

The concurrent programming model which has been chosen is based on a set of processes, each running in a local protected environment. The programming language used for the description of this environment, is the ECSP [5], an extended version of CSP [6], which is a suitable starting point for a message based concurrent language.

The communication commands in ECSP provide a variety of mechanisms and may support synchronous an asynchronous as well as symmetric and asymmetric communications.

Messages are typed and a type check is performed during each communication between processes. This check enforces a logical error detection mechanism and it is active all the times that a process need to communicate across the boundary of its local environment.

The logical communication structure is static in this version of MuTEAM, which means that it is defined at compile time an maintained in the kernel data structure for each process of the set which resides on a given node. Since care must be taken of the fact that one or more nodes can be faulty, we must be also able to selectively choose which logical channels must be active at any given time. The introduction of dynamic channels, with processname variables has a strong impact on the flexibility by which such choice can be performed.

The alternative and repetitive commands, introduced for the management of nondeterminism, and the extension in the ECSP by using input guards with priorities, are useful for allowing a flexible implementation of a backward recovery strategy.

An important point, which must be taken into account when designing a system with robustness characteristics, is that a correct exception handling mechanism must be provided. In the MuTEAM programming language, process termination is used at this aim. A process can abnormally terminate, causing the invocation of suitable exception handlers.
Moreover, the output command primitive is provided with three return points, to take into account successful termination of the command, on failing termination of the command, when e.g. an abnormal but not erroneous situation happens, and on error termination of the command, e.g. when an erroneous situation happens.

1.2. Physical configuration.

In order to have each process running in a local separated environment, a proper protection of the physical shared elements is required. In MuTEAM these are constituted by the shared memory segments distributed inside the several nodes. To enforce physical separation, a Protection Unit is provided for accesses to the shared memory.

A single node is considered as the field replaceable unit in the system and therefore these PU's protect each shared segment against the set of processes which run in each node. Error signals from the PU's and those provided by the usual addressing mechanism, based on segment limit, are used as low level error detection mechanisms.

All these constructs and mechanisms can help to build a robust system. The aim of this paper is to describe a possible ED/FT (error detection/ fault treatment) policy implementation in the MuTEAM, and to show how to use correctly the introduced constructs and mechanisms. A precompiler has been used to provide a high level implementation.

In the following Sections the ED/FT strategy for the MuTEAM is described, its detailed implementation is given and final comments are provided.

2. ED/FT IN MuTEAM.

The ED/FT strategy which has been designed for MuTEAM is based on four distinct phases:

a) Error detection;

b) Diagnosis of the node(s) which is (are) faulty in the system;

c) Physical and logical reconfiguration;

d) Recovery.

These phases have the following meaning:

Error detection: it is used to notify the occurrence of an erroneous event; it is implemented through low level signaling (segment-limit violation and access right violation), through failing of communication primitives and through periodic scheduling of diagnostic processes. The invoked mechanism is the exception handling in the run-time support of the language, which constitutes the interface towards the fault treatment.

Diagnosis: it is used to identify what node(s) is (are) responsible for the erroneous event and to identify the set of processes which must be reconfigured in the system.

Reconfiguration: it is the phase in which the communication structure is modified, on the basis of a suitable redundancy, to establish a well functioning set of channels to be used as a support to the recovery. The set of processes which is connected by this set of channels is obtained by a proper replication of the application processes, by insulating the processes running on the faulty node(s) and by connecting the substituted processes.

Recovery: it consists in the identification of a correct system- wide state (for all the processes in the reconfigured set) and in the restart of the computation from a mutually agreed state.

The redundancy which is necessary for a correct functioning of ED/FT is provided by a redundant set of communication channels, by a redundant set of back-up processes in the system and by the set of mechanisms necessary to manage this redundancy.

The design hypothesis used in structuring ED/FT is that its implementation is based on the ECSP and must be transparent to the user.

The objects which have been specified are:

1) a set of ECSP processes for ED/FT and

2) a precompiler to enforce transparency of the entire policy.

The job of the precompiler is to expand or modify an ECSP program, written with no knowledge of the internal redundancy of the system, in such a way that it can reliably perform its computation. The precompiler extensively uses the set of constructs of the language to provide a clean and readable version of the original program.

In the following paragraphs we will describe in detail the single phases previously introduced and discuss the functionality of the precompiler.

2.1. Error detection.

The basic mechanism for ED, provided by the ECSP language, is the raising of exceptions on:

a) the checks on the communications;
b) the abnormal process termination;
c) the failing of commands.

In all these cases the precompiler inserts, as an exception handler, an output command toward the FT process. As an example, the precompiler transform the following output command

```
Q ::  .... P ! constr (expr)
```

into

```
Q ::  .... P ! constr (expr)
          ON ERROR FT(k) ! awake ( )
               (rollback)
```

(In the ECSP language, a channel is identified by both a type and by a constructor constr, which is a special identifier of the channel.)

Since the FT process is replicated in each node, the invocation on error detection is performed toward the particular FT process running in the same node with the invoking process.

The other error detection mechanisms, as low level signaling or periodic scheduling of diagnostic processes are treated to emulate the sending of a message invoking the FT process from a dummy process.

2.2. Diagnosis.

The invocation of the FT process determines the activation of the Diagnostic Process DP. DP is replicated on each node in the system. In terms of ECSP processes, this may be described as:

FT :: [FT(1) ! FT(2) ! ..... ! FT(i) ! ..... ! FT(n)]

where

FT(i) :: [DETECT ! DP(i) ! RECONF ! RECOV]

The DP is based on a distributed algorithm implementing a model based on the PMC Diagnosis [3]. It is described in [7]. Its aim is the identification of the set of processes which are to be reconfigured.

2.3. Reconfiguration.

As briefly said in the Introduction, the reconfiguration must rely on a redundancy in the system. This redundancy is twofold: a redundant set of logical channels must be provided and a redundant set of application processes must be used.

For sake of simplicity, the organization of redundancy is described for the case in which one single node may be faulty in the system. The extension to a higher fault multiplicity is straightforward.

2.3.1. Process replication.

The precompiler acts on the initial activation constructs:

P:: [P1 ! P2 ! .... ! Pn]

and creates "twin" processes:

P:: [P1p ! P2p ! .... ! Pnp ! P1t ! P2t ! ... ! Pnt]

The loader will allocate the "primary" Pip and its "twin" Pit on different nodes and the execution of the activation command will activate both Pip and Pit. While Pip will be responsible of performing the actual computation, Pit, after activation, will only manage the updating of its recovery structures with the data that are sent to it by Pip and wait for a message from the FT process.

The structures of Pip and Pit is then the following:

```
Pip::                Pit::
     begin                begin
     DL                   DL'
                         *[ 1; FT(k)? (RIC)-)
                            (reconfiguration)
     CL                   []0; Pip ? (recovery structures) -)
                            (update rec.struct.)
                         ]
     end                  end.
```

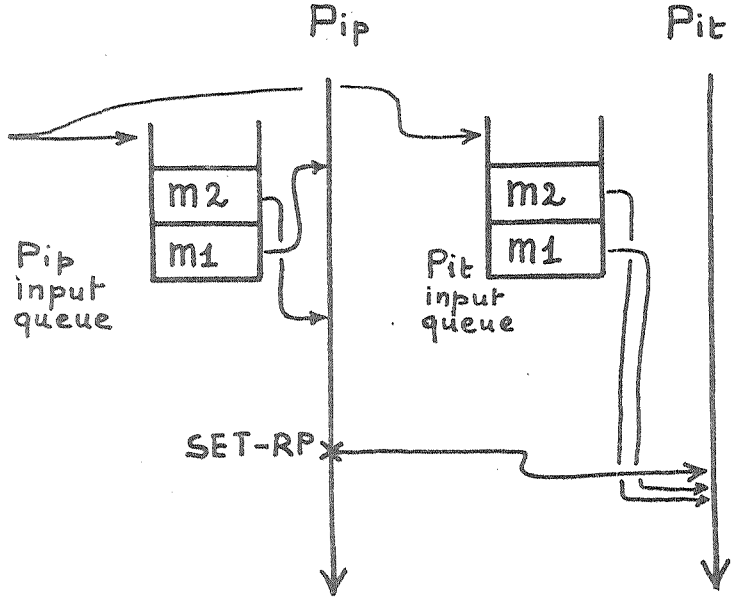
("CL" means "Command List"; "DL" means "Declaration List")

As ECSP allows the nesting of parallel activation commands, processes are structured in trees, whose roots are Pip, Pit, with i=1,2,..,n. The failure of any process in the tree with root Pip induces the substitution of all the processes in that tree; any activation command in the tree of Pip must induce an analogous activation command in the tree of Pit.

2.3.2. Redundant communications channels.

To provide the possibility of recovery, all the messages sent to Pip must be sent also to Pit.
When Pip executes an operation to save its status (SET-RP see later) this is notified to Pit which will update its recovery structures, executing the receive of all messages which Pip has received.

Fig.1 shows the mechanism by which Pit empties its input queue only when Pip notifies to it a SET-RP.



- Fig. 1 -

To implement the sending of a message to both Pip and Pit, the precompiler must transform the output command:

```
Q:: ... Pi ! constr (expr) ...
```

as it is written by the programmer into the form described in Fig. 2.

```
Q:: ....
    [ 1 ; FT(k)? (RIC) -> skip
    [] 0; true         -> skip
    ]
    [ RIC(i) = Pip and Pit fault free -> Pip! constr(expr)
                                         Pit! constr(expr)
    [] RIC(i) = Pip faulty            -> Pit! constr(expr)
    [] RIC(i) = Pit faulty            -> Pip! constr(expr)
    ]
    ......
```

- Fig. 2 -

The meaning is that to the input guard (true), which is always verified, is associated a priority (0) which is lower than to the priority associated to the presence of a message from FT.

The message incoming from FT assigns a value to the variable RIC which is a vector with one entry for each of the channels. Each entry contains the state (faulty, fault free) of the partner.

Moreover every process must be able to receive messages from either Pip or Pit.
To implement it, the precompiler must transform the input command:

```
Q:: .... Pi? constr(var)
```

as it is written by the programmer, into the form described in Fig. 3.

```
Q:: ....
    [ 1; FT(k)? (RIC) -> ( reconfiguration)
    [] 0;  true       -> skip
    ];
    XP? constr(var)
    ....
```

- Fig.3 -

(reconfiguration) correspond to the phase which, based on the examination of RIC, assigns the value Pip to the processname variable XP if Pit is faulty or Pit to XP if Pip is faulty.

Since the ECSP language provides many different types of

communications the precompiler will apply suitable transformations to each of these types.

The two example shown correspond to the simplest of these transformations. In Appendix A, the transformations for all type of communications are provided.

Each FT process maintains the data structure related to the activation of the application processes and their allocation to the nodes. This is available since the process allocation is static in the MuTEAM. On the basis of the identification of which node is faulty, each FT process defines the new set of processes which must recover computation and the new set of channels which are to be used. FT sends reconfiguration messages to this set of processes.

2.4. Recovery.

The set of reconfigured processes is responsible for the execution of all the recovery actions in the system. During normal functioning each primary process must save its virtual state executing set-recovery-point (SET-RP) operations. The SET-RP can be explicitly issued by the programmer or can be autonomously inserted by the protocol for the maintenance of consistent recovery lines in the system.

The tools provided to the processes to implement recovery are:

a. the SET-RP operation;

b. the protocol for maintaining consistent recovery lines;

c. the rollback algorithm.

All these tools are added to the application programs by the precompiler and are therefore transparent to the user.

2.4.1. SET-RP operation.

This instruction is provided to the programmer who is in charge of using it for checkpointing in suitable points his program. The SET-RP operation saves the state of the process: this state is composed by the value of the variables declared either by the programmer or by the precompiler. The variables which are declared by the precompiler are used to maintain a) information about the flow of control of the process, b) information about the consistency of the recovery lines.

The SET-RP operation must send this state to the twin

process, to provide him with the information necessary in case of recovery.

The precompiler transforms the program shown in Fig. 4a into the one shown in Fig. 4b, plus the twin process shown in Fig. 4c.

```
P ::          Pp ::             Pt ::
  begin           begin             begin
    DL            DL'               DL'
    CL1           CL1              *[ 1; FT(k) ? (RIC) -> CL0
                  Pt ! (DL')        [] 0; Pp ? (DL')    -> CL0'
    SET-RP        store (DL')       ]

    CL2           CL2
     .             .                 .
     .             .                 .
  end             end               end

   a)              b)                c)
```

- Fig. 4 -

2.4.2. Protocol for consistent recovery lines.

It is well known [8] that in a set C of concurrent communicating processes, a "domino effect" may be caused by an improper organization of recovery points for C. We have investigated and implemented a policy which prevents the "domino effect" by dynamic planning of recovery lines in C.

A formal model has been developed and is shown in [9]. In the following an informal presentation and the details for implementation are provided.

Dynamic planning is provided whenever an event which may alter the consistency of a recovery line happens in C. The events which are to be considered are those related to the exchange of messages among processes in C (send and receive events) and SET-RP events.

For each of these events the recovery information must be updated in the partner processes.

Each process P in C associates to each recovery line an ordering number Tp related to the SET-RP operations in P. The ordering is on the basis of time t local to P, and is such that

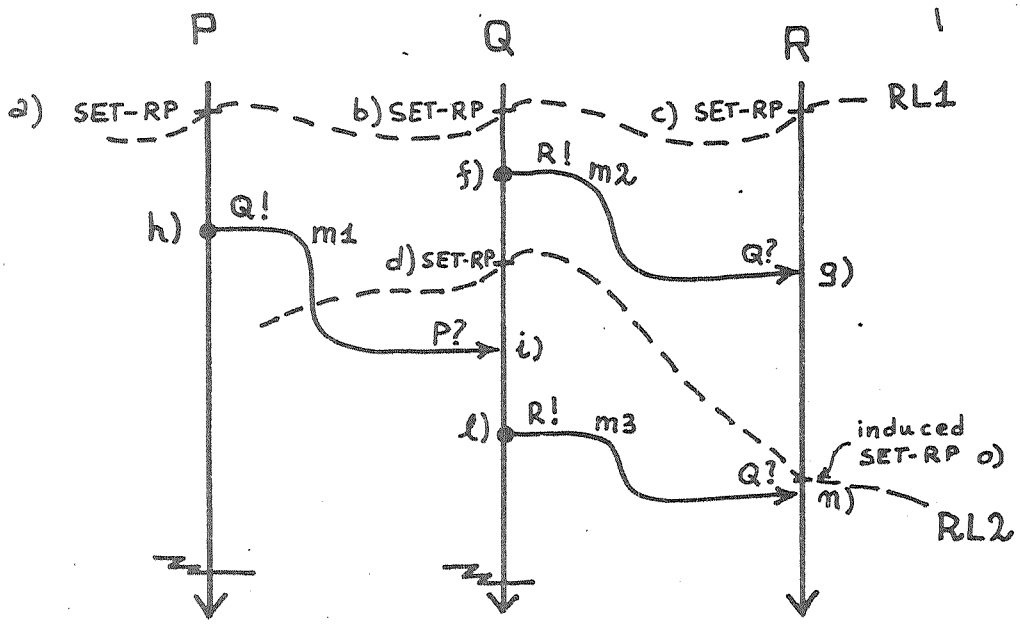
Tp(t) > Tp(t-a),

if two operation SET-RP have been performed by P at time t-a and t. The values Tp and Tq are compared between two partners P and Q in a communication, in such a way that a logical partial ordering is induced among the set of events in different processes in C.

Each process P in C maintains also the information related to which processes it has communicated with. This set is called PE. P maintains also a subset ME of the messages exchanged in C. This subset is constituted by the messages which have been sent to P before the recovery line maintained in C and which P received after that recovery line.

Example:

Let us consider a set C of three processes P, Q and R as shown in Fig. 5.



	T	PE	MSG
a)	1	∅	∅
h)	1	Q	∅

	T	PE	MSG
b)	1	∅	∅
f)	1	R	∅
d)	2	∅	∅
i)	1	P,R	∅
	2	∅	m1
l)	1	P,R	∅
	2	R	m1

	T	PE	MSG
c)	1	∅	∅
g)	1	Q	∅
n)	1	Q	∅
	2	Q	∅

- Fig. 5 -

The events a), b), c) in P, Q and R determines only the increment in Tp, Tq and Tr. Event f): the sending of message m2 from Q to R has the following consequences: - Q adds the name R to PEq; - R compares Tr with Tq: since Tr = Tq, R adds the name Q to PEr. Event d) in Q determines: - Q increases the value of Tq; - Q empties PEq and MEq. Event h): the sending of message m1 from P to Q determines the following consequences: - P adds the name Q to PEp; - in i) Q compares Tq with Tp; Q adds m1 to MEq in the recovery points such that Tq > Tp and adds the name P to PEq in the recovery points such that Tq < Tp. In n) m2 is not saved since Tr < Tq, but one SET-RP is induced in o) to provide the correct partial ordering among Q and R.

The planned recovery lines which are present in P,Q,R are indicated by T=1 and T=2. A crash in P will determine a rollback to RL1 which involves P, Q and R. A crash in Q will determine a rollback to RL2 which involves only Q and R. If the induced SET-RP o) had not been set, the attempt of Q to rollback on RL2 would have been unsatisfactory for R, which would have propagated a request of rolling back to RL1 and this would have caused a "domino effect" on Q.

The previous example has informally described the operation of the protocol to provide consistent recovery lines.
The formal proof of correctness of the protocol and of these operations is shown in [9].

The implementation of this protocol is based on the following data structure: record PRP (planned recovery point)

```
type PRP = record
           T: integer;
           PE: set of processname
           ME: set of messages
           D: descriptor of recovery points
           end
```

Each process P maintains an array PRPSET of PRPs; the actual number of PRPs in PRPSET is K.

For each SET-RP event a new element is created, with T = k+1, PE = 0, ME =0 and D = descriptor of the recovery point.

Associated with a send event from P to Q is the modification of the field PE for each PRP, with the addition of Q to PE.

Associated with a receive event in Q of a message m from P is the following modifications:

if Kq < Kp, the process Q creates a number Kp - Kq of SET-RP, and adds the name P to PEq in all its PRPs;

if Kp < Kq, the process Q, for each PRP such that Tq =< Kp, adds the name P to PEq, while for each PRP such that Tq > Kp, adds the message m to MEq.

The precompiler will add to the code of Pp the code performing these operations, and to the code of Pt the code performing similar operations upon reception of a SET-RP message from Pp.

The transformations which the precompiler must perform on the code of a process P is the following.

a. In a send command:

```
P::                    Pp::
   begin                   begin
   ...                     ...
   ...                     type PRP
   ...                     ...
   ...                     var  PRPSET: array (1..maxlength) of
PRP;
   ...                     K: integer;
   ...                     ...
   ...                  ! [ i; FT(h)? (RIC) ->
                                  (reconfiguration)
   ...                  ! [] O;  true       -> skip
   ...                  ! ]
   Q! constr(expr)      ! [ RIC(i) = Qp and Qt fault free ->
                                  Qp! constr(expr)
   ...                  !          Qt! constr(expr)
   ...                  ! [] RIC(i) = Qp faulty  ->
                                  Qt! constr(expr)
   ...                  ! [] RIC(i) = Qt faulty  ->
                                  Qp! constr(expr)
   ...                  !  ]
   end                     end
```

b. in a receive command:

```
P::                  Pp::
  begin                 begin
  ...                   ...
  ...                   type PRP
  ...                   ...
  ...                   var PRPSET: array (1..maxlength) of PRP;
  ...                   K: integer;
  ...                   ...
  ...                 | [ 1; FT(h)? (RIC) -) (reconfiguration)
  ...                 | [] 0;  true       -) skip
  ...                 | ]
  Q? constr(VAR)      | XQ? constr(Kq)
  ...                 | (update PRPSET.PE, PRPSET.ME
  ...                 |  if needed generate SET-RP)
  ...                 | XQ ? constr(VAR)
  ...                  ...
  end                  end
```

c. in a SET-RP command:

```
P::          Pp::          Pt::
   begin        begin         begin
   DL           DL'           DL'
   ...          ...
   ...          ...
   ...        | Pt! (DL')    | [ 1; FT(h)? (RIC) -) ( reconf.)
   SET-RP     | (st(DL'))    | [] 0; Pp? (DL')   -) CL2
   ...                       | ]
   ...          ...
   ...           ...
   end           end
```

where CL2 stores DL, receives the waiting messages and updates its recovery structures.

2.4.3. Rollback.

The rollback protocol is generated by the precompiler, which adds it to each application process. Each application process in the reconfigured set, when the rollback protocol is invoked, will execute a consensus algorithm to search a recovery line which is accepted by the processes in the set. Each process P maintains for rollback the following data structure:

```
type OFFER = record
             FLAG: (termination, rollback, confirmation);
             NO: PRP number;
             LPA: set-of-processes;
                 (* list of processes who accept *)
             LPI: set-of-processes;
                (* list of invited processes *)
             RL:  set-of-processes;
                (* list containing the recovery
                   line *)
             end;
```

A variable of this type, OFFER, is sent as the body of messages involved in the rollback. Any process Q, in the reconfigured set, may originate this message, initializing OFFER as follows:

NO contains the value of T related to PRP chosen by Q;

LPA contains the name of Q

LPI contains the names of the processes in PEq in the chosen PRP.

RL is initially empty. A process P, whose name is in LPI, will receive OFFER. If P agrees with OFFER.NO it adds its name into OFFER.LPA (deleting its name from OFFER.LPI) and possibly inserts in OFFER.LPI (if not present) the names of the processes to which propagate the offer. These names are contained in PRPSET(offer number).PE in P. Then P propagates the OFFER to other processes. If P cannot accept the OFFER, it resets the OFFER by specifying a new OFFER.NO, canceling OFFER.LPA, setting OFFER.LPI to the value it has in PRPSET(new offer number).PE. Then it propagates OFFER.

The OFFER may be rejected when it is related to a recovery line involving a terminated process.

The rollback protocol is complicated by the parallel-hierarchic structure of ECSP processes. A comprehensive analysis of the several rejection conditions is in [10].

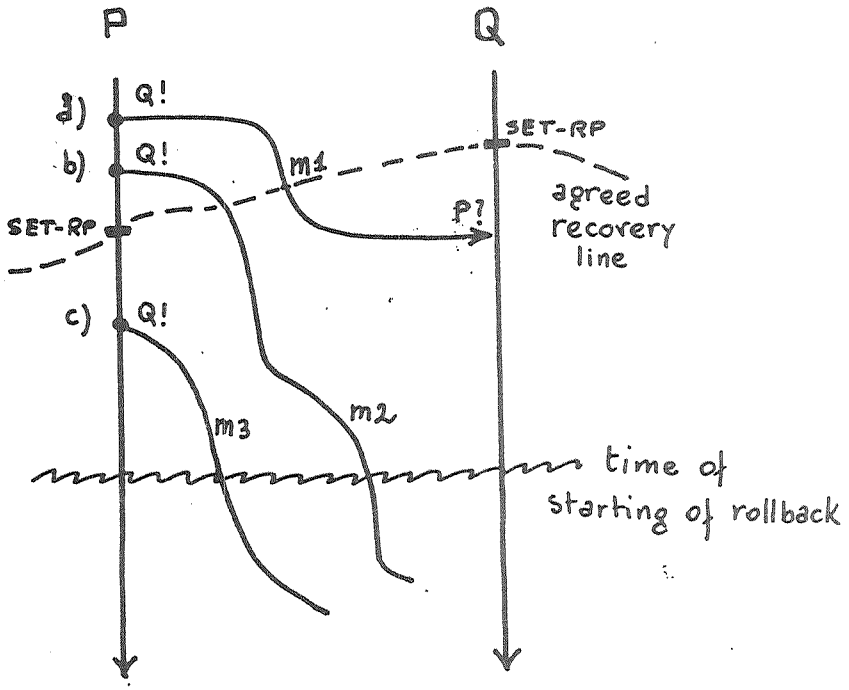
The first phase of the rollback protocol, in which OFFER is manipulated and circulated by the processes in the reconfigured set, terminates when all these processes have agreed on a common recovery line. This is guaranteed by the fact that during normal functioning the processes maintain consistency among PRPs, as seen in paragraph 2.4.2. Therefore a "domino effect" cannot be present; the choice of an older recovery line may only be induced by the particular process termination rule used in ECSP.

At the end of the first phase, OFFER.LPA contains the names of the processes in the reconfigured set, and OFFER.LPI is

empty. The last process which has manipulated OFFER will copy OFFER.LPA into OFFER.RL and will start the second phase of rollback.

The second phase of the rollback protocol consists in a compensation action on the input buffers of each process in the reconfigured set.

Consider Fig.6.



- Fig. 6 -

The communication a) was initiated and was terminated before the starting of rollback; therefore the message m1 is saved in the PRP of Q. The communication b) was initiated before the agreed recovery line (message m2 has associated timestamp T (PRP.T); m2 is in the input buffer of Q, but it has not been received and therefore is not saved in PRPq. This communication will not be reexecuted and therefore Q must, during the second phase of rollback, reexecute the receive of these messages.
The communication c) was initiated after the agreed recovery line (message m3 has an associated T = PRP.T); m3 is in the input buffer of Q, but it has not been received and is not saved in PRPq. This communication will not be reexecuted and therefore Q must, during the second phase of rollback, discard these messages.

The process which initiates the second phase of rollback will: - compensate its input buffers; - delete its name from

OFFER.LPA ; - propagate OFFER to other processes in OFFER.LPA. The second phase terminates when OFFER.LPA is empty. The set of involved processes is maintained in OFFER.RL. To implement the restart from the chosen recovery line, the precompiler will translate the body of a process into repetitive structure as shown in Fig. 7.

```
P::          Pp::
   begin        begin
   DL'          DL'
   !CL0          IND:= 0
   !SET-RP       *[(IND=0) -> CL0; IND:=1
   !CL1          [](IND=1) -> CL1; IND:=2
   !SET-RP       ....
 CL!...          ....
   !CL(n-1)      ....
   !SET-RP       [](IND=n) -> CLn; IND=exit
   !CLn          ]
   end           end

    a)            b)
```

- Fig.7 -

Fig. 7a shows the outline of a program as written by the programmer, Fig.7b shows the modified version by the precompiler. This shows the simplest transformation. The full set of transformations is shown in [10].

The main idea is that the body of each process is transformed into a cycle, and the termination of the roll-back protocol will assign the proper value to the variable IND which will point to the proper restart point in the process.

3. CONCLUSIONS.

In this paper an implementation of an ED/FT strategy for the MuTEAM prototype has been presented.
This policy is based on an innovative distributed recovery protocol for the planning and retrieving of recovery lines.
Some problems have arisen in the implementation due to the process termination rule used in the ECSP, with respect to the quick convergence of the searching for a recovery line.
Studies are under development to overcome these drawbacks either by recreation of terminated processes or by moving PRPs, related to nested processes in the hierarchy, into the root's space.

From the examples it should be evident that the use of a powerful tool like a precompiler, if the language provides suitable mechanisms, is sufficient to implement at a very

high level a system function like ED/FT.

Appendix A

Transformations by the precompiler on input and output commands.

The parallel-hierarchical structure of ECSP processes has the following visibility rules.
A process P can refer by name, in an input or output command:

1. Processes in the same activation command

   [P ¦ P01 ¦ P02 ¦ P03 ¦ P04]

2. Processes which can be referred by name by the process P0 father of

   [P ¦ P01 ¦ P02 ¦ P03 ¦ P04].

Therefore in a command list the following three cases may be present for an output command:

a. P! constr(expr), P is the real partner;

b. Q! constr(expr), Q is the visible partner;

c. X! constr(expr), X is a processname variable.

The transformation related to a. has been shown in Fig.2.

b) The precompiler transforms it in a form which is shown in Fig. 1.

c)
```
[RIC(i) = Xp e Xt fault free => Xp ! constr(expr)
                               Xt ! constr(expr)
[]RIC(i) = Xp faulty => Xt ! constr(expr)
[]RIC(i) = Xt faulty => Xp ! constr(expr)
]
```

where an assignment of a value to x corresponds to assignments to Xp e Xt.

In a command list the following input commands may be present:

a. P ? constr(var) P is the real partner

b. Q ? constr(var) Q is the visible partner

c. X ? constr(var) X is a processname variable

d. (X:Any) ? constr(var)

d'. (X:ANY FROM (lev1) TO (lev2)) ? constr(var)

d". (X:A1, A2, ... An) ? constr(var)

d., d'. and d". are three forms of asymmetric communication.

The transformations are based on the use of processname variables as in the case a) shown in Fig. 3. Case b) is identical to a).

c. The syntax is the same as for a) and b), but in case of reconfiguration, the range of the processname variable X is reduced to either Pp or Pt.

In the last three cases the syntax becomes:

(X = XA1, XA2, ... XAn) ? constr(var)

where (A1 ... An) is the set of processes indicated either by ANY, are ANY-FROM or by enumeration; in case of reconfiguration the value which XAi may assume is restricted to either Aip or Ait.

4. REFERENCES.

[1] F. Grandoni, et al. "The MuTEAM System: General Guidelines", Proceedings FTCS-11, June 1981, pp. 15-16.

[2] G. Cioffi, et al. "MuTEAM: Architectural Insights of a Distributed Multi-Microprocessor System", Proceedings FTCS-11, June 1981, pp.17-19.

[3] F. Baiardi, et al. "Mechanisms for a Robust Multiprocessing Environment in the MuTEAM Kernel", Proceedings FTCS-11, June 1981, pp.20-24.

[4] P. Ciompi, F. Grandoni, L. Simoncini "Distributed Diagnosis in Multiprocessor Systems: the MuTEAM Approach", Proceedings FTCS-11, June 1981, pp.25-30.

[5] F. Baiardi, et al. "The Operating System Kernel of a Message Passing Distributed System" submitted for publication to IEEE Trans. on Computers.

[6] C.A.R. Hoare "Communicating Sequential Processes", Communications of the ACM,21, 8, August 1978.

[7] P. Corsini, L. Simoncini, L. Strigini "MuTEAM: A Multimicroprocessor Architecture with Decentralized Fault Treatment" submitted for publication to IEEE Trans. on

Computers.

[8] T. Anderson, B. Randell edts. "Computing Systems Reliability: an Advanced Course", Cambridge Univ. Press, Cambridge, 1979.

[9] D. Briatico, A. Ciuffoletti, L. Simoncini "A Domino Free Recovery Algorithm: Formal Specification", submitted for publication to FTCS-14.

[10] D. Briatico "Trattamento dei Guasti Decentralizzato per Sistemi Multimicroprocessori: Aspetti Formali e Implementativi per il Prototipo MuTEAM", Thesis Dissertation (in Italian), October 1983.