North-Holland
Microprocessing and Microprogramming 11 (1983) 287–294

Recovery Blocks for Communicating Systems*

Paola Velardi
Fondazione Ugo Bordoni, V. le Trastevere 108, 00153 Rome, Italy

and

Bruno Ciciani
Istituto di Automatica, Università di Roma, via Eudossiana 18, 00184 Rome, Italy

In many practical applications of real-time computing (avionics, switching systems) a message-passing inter-processes communication approach is adopted for both modularity and reliability aims.

In the present paper, the problem of adding fault-tolerance in a message passing multiprocesses environment is examined. Recovery blocks implementation schemes for both asynchronous and synchronous communications are proposed, with the aim of avoiding domino-effects and exploiting the message oriented system structure.

When a sender process produces a message, an acceptance test is performed on the message by *system procedures*, which in sequence: i) transfer the message on the receiving process working memory, ii) save present process status, or in case of error, restore some previous process status, and iii) discard no longer needed status informations.

A PURGE procedure managing recovery points deletion is also proposed and described in some detail.

Keywords: Recovery blocks, communicating systems, real-time computing, fault-tolerance, system procedures, PURGE procedure.

1. Recovery Blocks in Communicating Processes

In real-time applications of multiprocessing systems, processes can be devoted to particular tasks (task sharing) or they can cooperate by sharing the overall system load (load sharing). In either case, a communication exists between the processes, either via shared data areas, or via explicit information exchanges. Recovery block implementation in this environment presents particular problems.

Usually when an error is detected, a recovery procedure is activeted. In systems where processes communicate among them, the recovery procedure concernes every process that has cooperated with the failed process. The action for the recovery of a consistent state in the cooperating processes with the failed one is *recovery propagation*.

A typical undesired effect due to interprocesses communication is the occurence of an uncontrolled propagation of state restoration amongst the processes, during a recovery procedure; this is commonly known as the 'domino effect'. Fig. 1 shows an example of this effect; a restoration of state a_j in the process P_a leads to a restoration of state b_j in process P_b; this further involves a restoration of state a_{j-1} in P_a, etc.

To avoid this effect Randell [3] proposed the so called 'conversation structure' method. An improvement to this method was devised by Kim [4]. To obtain greater advantages from parallel processing, a higher degree of independence between processes should be pursued; this is obtained if communications take place only *between pairs of processes*. The systems adopting this communication scheme are known as 'massage-passing' systems.

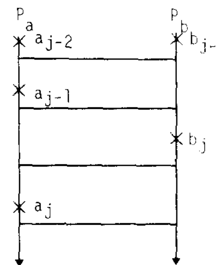

Fig. 1. Domino Effect.

* This work was carried out at the Fondazione Ugo Bordoni under agreement with the Istituto Superiore P.T. and the Fondazione Ugo Bordoni.

Message-passing systems have the following advantages:

a) no resources or common variables are shared by processes, as information is *directly* exchanged between processes. Each process operates then in a closed environment; this greatly helps *errors confinement*.
b) communications between processes occur by means of a well stated message-passing protocol [6,9]; *protocol respect control* is therefore an implicit *error detection* mechanism.

Recovery blocks implementation in a message-passing environment was dealt with by Russell [7], where solutions are proposed to avoid the domino effect for the case of *asynchronous communications*. Each process P_i keeps a record of received messages; in this way, no backword recovery is needed for a sender process P_j if the receiver process P_i fails; during restoration, P_i simply rereads messages from its 'message history' list.

Russell also proposed a domino-free recovery scheme for particular systems, summarized by the expression:

(MARK (state); RECEIVE (mess)*; SEND (mess)*)* (where MARK (state) denotes memorization command of the process state). In this scheme a recovery point is established by means of the MARK command *before* the occurence of any sequence of RECEIVE and SEND commands not mutually interleaved.

This brief investigation of the 'state of the art' leads to some considerations; all proposed solution state an upper limit to the propagation of state restoration, but in all cases, error-free process roll-back is not avoided. Moreover, the number of communications is increased, as processes need to exchange information concerning recovery points management and recovery propagation (as an example, in the solution proposed by Kim, messages must be sent between processes engaged in the same conversation, to update their 'potential recaller' list).

An increase of interprocess communications is unsuitable mainly for two reasons:

1) communication in a multiprocessor environment involves the control of shared hardware and software resources, process context switching and system procedure intervention;
2) the exchange of informations *is the main error propagation source.*

These considerations lead to the following conclusions, related respectively to the previous statements 1) and 2):

C1) a higher degree of independence should exists between processes in recovery point management;

C2) *a tight control* should be exercised *on messages* to be exchanged.

To attain requirement C1 the following rule is proposed:

R1) A recovery point on a process P_i must be created only *as a consequence* of an action performed *by the process P_i itself.* This rule involves that recovery block generation in the process P_i is *transparent* to the other processes.

To attain requirement C2 more steps are needed.

Error propagation in a message passing system is due to either an incorrect use of communication paths or to a wrong message. The first error source is avoided by adopting the following rule:

R2) Message transfers from a sender process to a receiver process are respectively accomplished by SEND and RECEIVE *system Kernel procedures.* This statement implies that a sender process simply *produces* a message and a receiver *consumes* this message, while communication paths are managed by procedures assumed to be fail safe; the Kernel is in fact a system hardcore.

The second cause of error propagation is more difficult to deal with; this problem can be solved by assuming a third rule:

R3) An acceptance test concerning the message itself is performed *before* calling the SEND procedure.

This rule implies that *only correct messages* can be sent, and is obviously very strict.

In the next sections rules R1, R2 and R3 are adopted to define a recovery blocks scheme for both asynchronous and synchronous communications; the proposed procedure matches the following characteristics:

1) recovery propagation among error-free processes is avoided, or *at least* no error-free messages are undone.
2) overhead due to process context switching and system procedures intervention is limited by

means of a suitable test point generation policy.

2. A recovery block structure for asynchronous communications

When an *asynchronous* communication takes place between two processes, the sender process P_i can restart normal activity as soon as the message has been stored in the appropriate communication buffer. No acknowledgement is needed by receiver process P_j; conversely, P_j can read a message from the buffer at any time without affecting P_i.

For these systems, the 'message history' list introduced in [7] can be implemented as follows: with each communication channel is associated a buffer list; once a message has been read by the receiver, the buffer location where the message was stored *is not released*; a pointer moves on the following buffer location, and a 'buffer free' indicator is set. Messages in a buffer are tagged with a 'recovery block identifier'. When the PURGE (x_j) procedure is called to eliminate a recovery point which is no longer needed, the buffer locations tagged with 'x_j' are set free. As shown in the previous section, since the messages related to any recovery block are memorized, no recovery propagation occurs when a RECEIVE command is revoked.

The actual cause of error propagation is then the sending operations. The problem is then to define a recovery blocks generation scheme complying with the rules R1, R2 and R3 previously stated; as rule R1 is certainly satisfied by adopting rule R3, rule R2 is considered first.

This rule can be satisfied by adopting a communication protocol as proposed in the MuTEAM operating system [6,8,9]. In this system a communication buffer is univocally identified by the triad: (name of sender process; name of receiver process; type of message). Any process P_i producer of a message specifies the communication channel it refers to; a SEND *system procedure* is then started, whose purpose is:

1) to exercise a control on protocol correctness, by verifying the existence of the specified buffer;
2) to transfer the message onto the buffer.

To attain both rules R1 and R3 an extension of the described SEND procedure is here proposed.

Let P_i be the sender process; once the message has been produced, P_i specifies the triad identifying the communication channel, and then enters a waiting state. A Kernel procedure uses the 'type' information in an extended way to vector onto a predefined *system table* enclosed in the calling *process descriptor*. This table associates each 'type' word with the address of a suitable routine, which evaluates message correctness. If this test accepts the message, steps 1) and 2) of the SEND procedure are executed. Once the message has been stored, P_i is woken up. We call this routine 'ENSURE-SEND' procedure.

The proposed structure is summarized in Table 1.

Table 1

```
              ENSURE-SEND (mess j)  ;; acceptance test on
                                       message
                                    ;; and communication
                                    ;; protocol assurance
              MARK (x_j)            ;; a new recovery
                                       point is
                                    ;; established
              PURGE                 ;; eliminates no longer
                                       needed
                                    ;; status informations
              by: begin: <statement ;; this is the recovery
                          list>     ;; block body
                          produce   (mess j)

                 end

              else by: RESTORE(x_j) ;; in case of error x_j
                                    ;; is restored and
                   <1st alternate>  ;; an alternate block
                                    ;; is executed

              else by .
                      .
                      .
                      .
                      .
              else error

<statement
 list> ::=    { {statement}_0^{i_n}; {RECEIVE (mess k)
                                   consume (mess k)}_0^{i_m};
                {statement}_0^{i_k} }_0^{i_n}
```

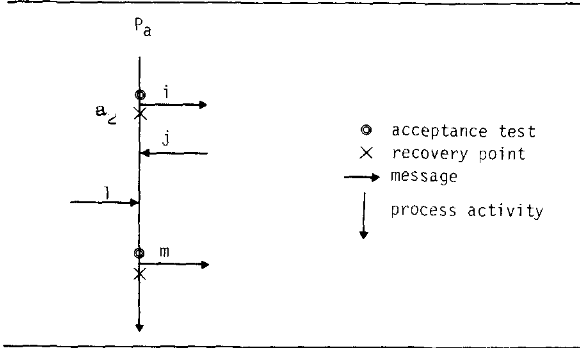

Fig. 2. Recovery Block Scheme.

As an example, we can refer to fig. 2. If the acceptance test on message m and on communication protocol correctness fails, then the process P_a rollback to recovery point a_2 and P_a executes the first alternate, reading the received messages (j and 1) from its message history. If the acceptance test is performed with success, then P_a sends the message m, else tries again with another alternate and so on.

This recovery scheme is claimed to present the following advantages:

a) checks on communication protocol correctness and recovery blocks execution are united in one procedure thus limiting the overhead due to process context switching and external procedures intervention;
b) no recovery propagation occurs, if the acceptance test is assumed to be reliable;
c) when a new recovery block is opened, the preceding one can be purged.

A peculiarity of this recovery scheme is that SEND and RECEIVE procedures *do not belong to the process recovery block body*; they are external procedures supposed to be error-free.

However, a disadvantage to this procedure can be found. When a process P_i produces a list of N consecutive messages, the overhead due to recovery blocks creation and delation can be unacceptable. In this case, the acceptance test can be activated when the last message of the list has been produced; this test must concern the *overall message sequence correctness.*

Since only communication protocol has been controlled for the first $(N-1)$ messages, error propagation may occur, during the time which elapses between the dispatching (SEND procedure) of a mistaken message and the detection of the error (ENSURE procedure).

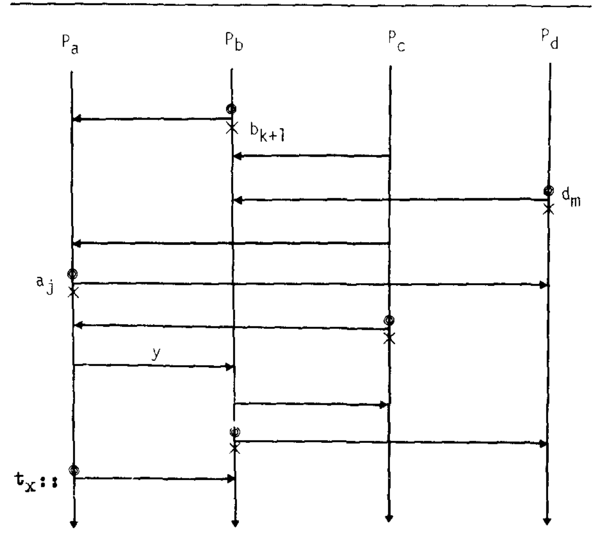

Fig. 3. Recovery Block Scheme with Recovery Propagation Amongst Processes.

Nevertheless if sending actions are not interleaved with receiving actions, the system lies under conditions stated in [7], and is therefore domino-free. This can be seen by observing Fig. 3. Let P_a be a failed process; at time t_x an error is detected and RESTORE (a_j) is issued. All the actions following a_j must be revoked by P_a. Let y be the first message sent by P_a to be rescinded. If P_b is the receiver of the message y, P_b must roll back to the first recovery point, say b_{k+1} preceding the command RECEIVE (y). All the message-exchange commands included between MARK (b_{k+1}) and RECEIVE (y) are assumed to be RECEIVE commands. Recovery propagation takes place only for messages sent by P_b in a time after t_y. As a general rule, recovery propagation is caused only by messages sent *after* the message y. The state restoration for all system processes will then stop at the first recovery point immediately preceding (or at the same time as) the recovery point a_j restored by the failed process P_a. The system is thus domino-free.

In this last recovery scheme rule R3 is neglected as far as overhead limitation requirements are concerned; however, only process activity involved with some wrong received or sent message has to be

undone. The restoration then *does not affect fail-free program blocks.*

In this section a recovery scheme for asynchronous communications has been proposed; this scheme adopts a particular view of the message-list concept proposed by Russell, and accepts the message passing scheme of the MuTEAM operating-system. The proposed structure is mainly based on a suitable test point generation policy which avoid error propagation between processes.

3. A recovery block structure for synchronous communications

Synchronous communications are now taken into account. Two types of synchronism can be considered [9]. In the *'tight rendez-vous'* communication, once the message has been stored in the buffer, the sender process P_i remains in a waiting state *until* the conversation partner P_j executes a receive command. In the case of *'extended rendez-vous'* communication, the waiting time for the sender process is extended until the receiver process produces and sends an answering message; a complete symmetry is then obtained.

When synchronism is required between processes, the revocation of a RECEIVE command *necessarily* involves the revocation of the corresponding SEND command; keeping a message history has then no practical use in such a system.

However in the case of 'tight rendez-vous' the recovery blocks structure proposed in the previous section can still be used by modifying the 'statement list' executed in the body of the block, as follows

<statement list> :: { {statement}$_0^{j_n}$;
{RECEIVE(mess K)
MARK (K)
consume (mess K)}$_0^{j_m}$;
{statement}$_0^{j_l}$}$_0^{j_r}$

In this structure, the process state is marked *after* any RECEIVE command takes place.

As an example, we can refer to fig. 4. If the acceptance test of P_a on message m and on communcation protocol correctness fails, then the process

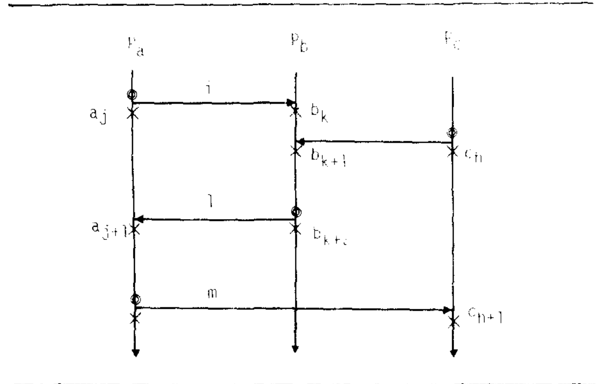

Fig. 4. Recovery Propagation in Synchronous Systems.

P_a rollbacks to recovery point a_j, P_a executes another alternate and the process P_b rollbacks to recovery point b_{k+1}. The two processes synchronize each other through the new message exchange. The system is still domino-free, as shown in fig. 5. Let P_a be a failed processor; at time t_x P_a returns to the first controlled fail-free state, sai a_j. Let y be the first message action to be rescinded and P_b the producer of y. P_b must roll back to the first recovery point preceding SEND(y) which is located immediately *after* the first message exchanged before SEND(y). No messages exchanged in a time preceding y can then be undone because of y revocation. The system is thus domino-free.

In this scheme the propagation of restoration could have been avoided by setting up an acceptance test *after* any received message. There are two arguments against this solution:

- an acceptance test on a received message can at best ensure that some expectation is met; a

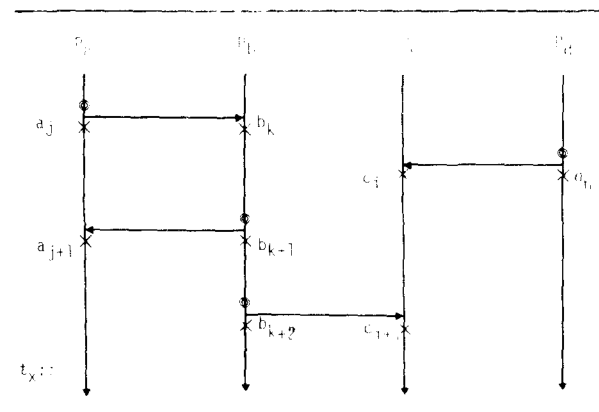

Fig. 5. Recovery Block in Synchronous Systems.

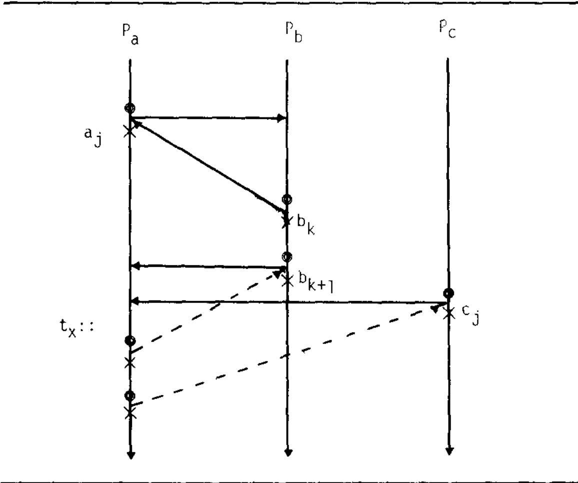

Fig. 6. Recovery Blocks for Extended Rendez-Vous Synchronous Systems.

stronger test on the message could be accomplished only when a mutual knowledge of respective activities exists between processes.
- the number of acceptance tests would increase unacceptably. ,

A solution if found by adopting the 'extended rendez-vous' communication. The scheme proposed for asynchronous communications can still be used without modification, as shown in fig. 6.

Let's suppose that the process P_a issues a RESTORE (a_j) command at time t_x, before sending answers to processes P_b and P_c. The revocation of the message x and y does not lead to a recovery propagation, because the processes P_b and P_c are still blocked in states b_{k+1} and c_j. Therefore, no recovery propagation takes place.

A comparison between different message-exchange techniques is not the aim of this paper; nevertheless the higher fault-tolerance degree is clearly achieved by adopting an extended rendez-vous synchronous communication. Major deadlock risks and interdependence between processes are the obvious disadvantages. A suitable choice is therefore required of the designer.

4. A procedure for recovery point delation

The previous sections show that recovery propaga-

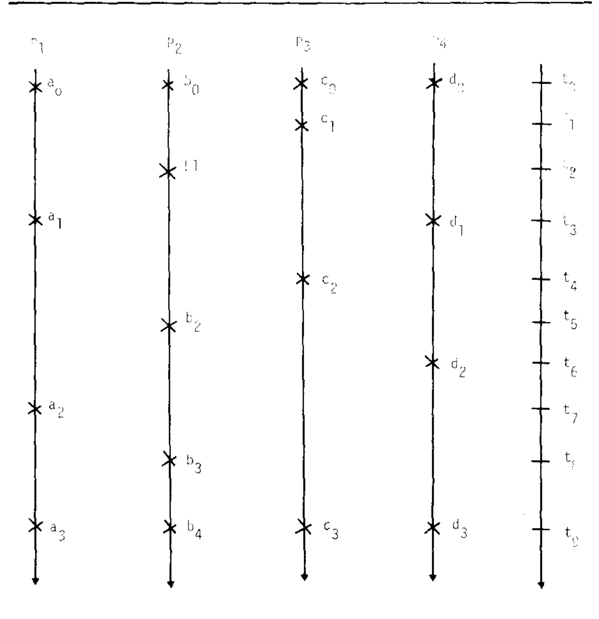

Fig. 7. Recovery Points Deletion.

tion between processes takes place only in the following cases:

i) *for asynchronous communication*, if recovery points are generated only when the last message of a list has been sent;

ii) *for synchronous communication*, when tight rendéz-vous communications between processes are established.

In both cases, if a_j is the recovery point restored by the failed process P_i, roll-back propagation for system processes stops at *worst* in the first recovery point preceding (or at the same time as) a_j. In fig. 7 an example is shown; at time t_x the worst that could occur would be a failure of the process with the earliest still open recovery block, say P_2. At worst, this would lead P_1, P_3 and P_4 to return to point a_0, c_1 and d_0 respectively. At time t_x, only recovery points b_0 and c_0 can therefore be discarded.

In this section, a PURGE procedure that manages recovery points deletion for recovery schemes i) and ii) is proposed. Let S be an N-processes system, and let P and Q be two arrays $(2 \times N)$, defined as follows:

1) $P(1,i)$ $(i = 1, \cdots, N)$ represents the latest recovery point established by P_i, expressed as a time function;

2) $P(2,i)$ $(i=1,\cdots,N)$ represents a progressive number associated with the recovery point $P(1,i)$;
3) $Q(1,i)$ $(i=1,\cdots,N)$ is the earliest recovery point of P_i not yet discarded;
4) $Q(2,i)$ $(i=1,\cdots,N)$ is the progressive number associated with $Q(1,i)$.

When a proces $P_j (j=1,\cdots,N)$ opens or new recovery block at time t_x, a PURGE procedure performing the following action is called:

a) transfer the value t_x into $P(1,j)$;
b) find the minimum value $t_{\min}$ amongst all elements $P(1,i) i=1,\cdots,N$;
c) for *all* integers j that verifies the condition $P(1,j)=t_{\min}$, execute this operation:
   α) write the value $t_{\min}$ into $Q(1,j)$
   β) discard recovery points of P_j included between $Q(2,j)$ and $P(2,j)$.

This procedure can be expressed as in Table 2.

Table 2

```
procedure PURGE:
P(2,j) = n                   ;; the nth recovery point
P(1,j) = t_x                 ;; is stated for P_j at time t_x
t_min = min P(1,i)
i = 1,···,N                  ;; find the earliest recovery
for i = 1,···,N do:          ;; block still open t_min
begin: if P(1,i) = t_min
then
 begin < Q(1,i) = t_min
        M = Q(2,i)
        L = P(2,i)
        K = L - M
        if K≠0 then:
  begin: Q(2,i) = L
    for j = 1,···,K
    begin: < PURGE
(M)                          ;; erase information related
      M = M + 1 > end        ;; with recovery point M
    end
   else continue > end
  else continue
 end
```

As an example, we can still refer to fig. 7; when process P_2 reaches recovery point b_2 arrays $\boldsymbol{P}$ and $\boldsymbol{Q}$ have the following values:

$$P \triangleq \begin{pmatrix} t_3 & t_2 & t_4 & t_3 \\ 1 & 1 & 2 & 1 \end{pmatrix}$$

$$Q \triangleq \begin{pmatrix} t_0 & t_2 & t_1 & t_0 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

At time t_5 the worst that could occur would be a failure of P_1 or P_4, owners of the earliest recovery blocks still open. This would cause at worst a restoration of P_2 and P_3 in b_1 and c_1 respectively. The recovery points stated by P_1, P_2, P_3 and P_4 at times t_3, t_2, t_1 and t_3 can still be used, while all preceding points can be discarted.

The PURGE procedure previously introduced will therefore perform the following actions:

1) write the value t_5 in $P(1,2)$ and the value 2 in $P(2,2)$
2) identify t_3 as the minimum time $t_{\min}$ amongst the elements of the first row of P
3) write the value t_3 in $Q(1,1)$ and in $Q(1,4)$, and write the value 1 in $Q(2,1)$ and $Q(2,4)$
4) discard recovery points a_0 and d_0 belonging to P_1 and P_4

Final array situation will therefore be:

$$P \triangleq \begin{pmatrix} t_3 & t_5 & t_4 & t_3 \\ 1 & 2 & 2 & 1 \end{pmatrix}$$

$$Q \triangleq \begin{pmatrix} t_3 & t_2 & t_1 & t_3 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

5. Concluding remarks

Recovery block implementation in a communicating environment can increase interdependence between processes. The solutions proposed in the literature generally involve exchange of information during the establishment of recovery points between processes.

In this paper, a recovery block scheme is proposed, with the following features:

- *independence amongst processes* in the establishment of recovery points;
- *recovery propagation* between fail-free processes is avoided by means of a suitable recovery point establishment policy;
- communication management is enthrusted to *system hardcore procedures*.

Message-passing operating systems are here recommended as the best environment for a suitable recovery block implementation. Both syn-

chronous and asynchronous message-passing techniques are dealt with. A recovery delation PURGE procedure is also proposed.

More detailed considerations on testing techniques and overhead evaluation are intended to be subject of further study.

References

[1] J.J. Horning et al., 'A program structure for error detection and recovery' Lecture Notes in Computing Science vol. 16, 1974

[2] B. Randell, 'System structure for software fault-tolerance' IEEE trans. on Software Engeneering. vol. SE-1, June 1975.

[3] B. Randell et al., 'Reliability issues in computing system design' Computing Surveys vol. 10 June 1978.

[4] K.H. Kim, 'An approach to programmer-transparent coordination of recovering parallel processes and its efficient implementation' Proceedings of International Conference on Parallel Processing 1978.

[5] P.B. Hansen, 'The Nucleus of a Multiprogramming System' Communication of ACM vol. 13 April 1970.

[6] M. Vanneschi, F. Baiardi et al., 'The MuTEAM Kernel Guidelines for a Message-Passing Multiprocessor'. ISI Internal Report S-80-23. University of Pisa – 1980.

[7] D.L. Russell, State restoration in Systems of Communicating processes'. IEEE Transactions on Software Engeneering vol. SE-6 March 1980.

[8] M. Vanneschi, F. Baiardi et al., 'Protection and Error Confinement in a Message-Passing Environment: the MuTEAM Kernel' Fault Tolerant Systems and Diagnostics – BRNO, 1981.

[9] M. Boari et al., 'Message-passing models for cooperating processes: analysis and comparison of proposed solutions' In Italian, MUMICRO Internal Report n. 16, 1979.

Paola Velardi was born in Rome (Italy) on April 26, 1955. She received the Engineering degree in Electronic Engineering, summa cum laude, from the University of Rome in the 1978. From 1979 she works as researcher in the 'Fondazione Ugo Bordoni'. Her main fields of interest are distributed architectures and fault-tolerant systems.

Bruno Ciciani was born in Rome (Italy) on February 15, 1955. He received the Engineering degree in Electronic Engineering, summa cum laude, from the University of Rome in the 1980. From 1981 he works at the Institute of Automatica of the University of Rome, his main fields of interest are distributed architectures and fault-tolerant systems.