CONSIGLIO
NAZIONALE
DELLE RICERCHE

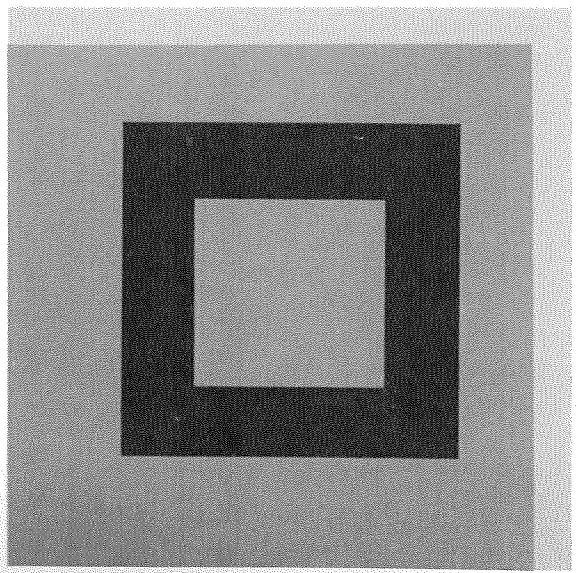
PROGETTO
FINALIZZATO
INFORMATICA

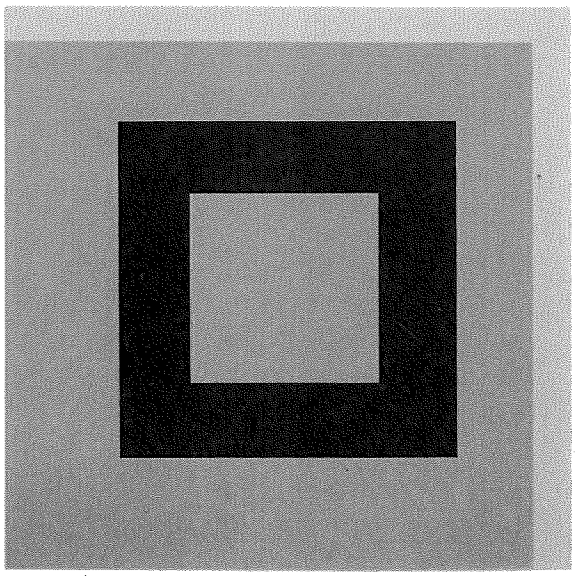
Cnet

79

Implementation Schemes for a Crash Recovery Mechanism

P. Ancilotti, M. Fusani





Collana Cnet

1. **F. Bruni**, Gestione degli errori e procedure di correzione
2. **R. Marcogliese**, Valutazione delle prestazioni di rete locale: metodologie, strumenti, proposte per Cnet
3. **C. Montangero**, La lista dei requisiti e la struttura dei tipi del linguaggio Ada
4. **M.Simi**, La rete locale del "Laboratory for Computer Science"
5. **V. Ambriola, M. Bellia, P. Degano**, Sistemi integrati per la produzione di software
6. **G. Attardi, A. Martelli, U. Montanari**, Il meccanismo dei moduli nel linguaggio di Cnet
7. **U. Montanari, C. Simonelli**, On distinguishing Concurrency from Nondeterminism
8. **G. Levi, A. Martelli, C. Montangero**, Il linguaggio dell'ambiente di programmazione di Cnet
9. **P. Degano, R. De Nicola**, Una base di dati di moduli in un ambiente integrato per la produzione di software in una rete locale
10. **E. Dameri, G. Levi**, AISS: un Ambiente Integrato per lo Sviluppo di Software
11. **G. Attardi**, L'evoluzione degli elaboratori personali: linee di tendenza
12. **F. Borgonovo, L. Fratta**, A Communication Protocol for Integration of Data and Speech in a Local Communication Network
13. **F. Borgonovo, L. Fratta**, Design and Performance Evaluation of a Local Communication Subnetwork
14. **M. Ajmone, F. Borgonovo, L. Fratta**, Considerazioni sulla scelta delle caratteristiche di base della sottorete di comunicazione per Cnet
15. **M. Martelli, F. Tarini**, Meccanismi di comunicazione "internode": analisi e proposte per una lista di requisiti
16. **N. Lijtmaer**, Cnet: proposta di ricerca 1981
17. **M. Di Santo, L. Nigro, W. Russo**, Meccanismi di astrazione del controllo: una proposta per il linguaggio di Cnet
18. **M. Ajmone Marsan**, Considerazioni preliminari sul protocollo di accesso a canale comune
19. **F. Tisato, R. Zicari**, Operating System and System Language Requirement List.
20. **F. Liguori, G. Rossi**, Criteri e meccanismi e modularizzazione
21. **E. Dameri, C. Simonelli**, Linguaggi intermedi per un ambiente integrato di sviluppo
22. **E. Astesiano, E. Zucca**, Semantics of CSP via Translation into CCS
23. **A. Celentano, P. Della Vigna**, Modularizzazione in linguaggi PASCAL-like
24. **M. Buttò**, Alcune considerazioni sulle reti locali
25. **G. Barberis, A. Luvison**, Fibre ottiche per comunicazioni ad alta velocità in sistemi ad anello
26. **D. Di Pino**, Considerazioni preliminari sulla natura e le funzioni di Gateway per una rete locale quale la Cnet
27. **S. De Micheli**, Integrated Voice, Data, Text and Fax Communications on Local Computer Networks
28. **G. Barberis, A. Luvison, G. Pellegrini, G. Pirani**, La sottorete di comunicazione nell'ambito del progetto Campus-net
29. **G. Fioretto, L. Gabrielli, C. Giuliani, M. Sposini**, Proposta per una sottorete di comunicazione in fibra ottica per sistemi distribuiti
30. **S. De Micheli**, Relazione sulle attività dell'area "Sottosistema di comunicazione"
31. **R. Barbuti, M. Bellia, E. Dameri, A. Martelli, C. Simonelli**, Gli editor guidati dalla sintassi: strumento base di un ambiente integrato di sviluppo software
32. **C. Montangero**, A finite state model of the behaviour of an Ada Task

PROGETTO FINALIZZATO INFORMATICA
C.N.R.
Progetto P_1 Cnet

P. ANCILOTTI - M. FUSANI

IMPLEMENTATION SCHEMES FOR A CRASH RECOVERY MECHANISM

Istituto di Elaborazione della Informazione
C.N.R. - PISA

ETS/PISA

IMPLEMENTATION SCHEMES FOR A CRASH RECOVERY MECHANISM

P. Ancilotti, M. Fusani - Istituto di Elaborazione della Informazione - C.N.R. - Via Santa Maria 46 - 56100 Pisa - Italy

Summary: Atomic actions represent a powerful tool for system structuring, controlling accesses to shared data and implementing backward error recovery techniques. In this paper properties of atomic actions are analyzed, specifically referring to crash recovery problems. In order to guarantee consistency of shared data in spite of crashes, programs implementing atomic actions must satisfy some constraints. Implementation schemes are developed in sequential and concurrent contexts. Different schemes are separately described, namely: single object and multiple objects actions, uniprocess and multiprocess actions. Practical problems that arise during the implementation of atomic actions are pointed out.

1. Introduction. This work is developed in the context of the CNET project /CNET-80/. The target architecture of this project is a local network composed of a collection of autonomous nodes that communicate by information exchange over a communication network.
Due to node autonomy and inherent modularity in the system, local networks offer some potential reliability characteristics, that can be made effective if techniques are developed to properly handle component failures. This paper is concerned with the design of recovery mechanisms for enabling a system to tolerate a sudden loss of resource availability, usually known as a crash. Atomic Actions (A.A.) are a powerful instrument for dealing with this kind of failure, especially when concurrent activities are involved. The purpose of the paper is to analyse both properties and structure of the A.A.'s, viewed in the various environments of CNET applications. More emphasis is given to the problem of recovering from local crashes.

The physical model is briefly presented in section 2. The basic properties of an A.A. are described in section 3. Sections 4 and 5 deal with structuring A.A.'s in various contexts, while some practical problems that arise during the implementation of A.A.'s are pointed out in section 6.

2. Basic model. In this section we discuss some assumptions that underlie our approach. Distributed applications can be structured in CNET as virtual networks. A virtual network is a collection of virtual nodes. Each virtual node provides controlled accesses to a set of resources and is allocated on a single real node. A virtual node can be conceived as two disjoint sets of components: processes and data objects.
Two different sets of inter-process communication mechanisms are provided: the first one suitable for intranode interactions, the second one dedicated to internode communications. In particular, processes local to a virtual node may interact through common objects, while remote processes communicate by sending messages.
Distributed applications can be programmed in CNET by using high level language that supports the construction of well structured programs through

an abstraction mechanism, especially data abstraction.

Objects can be partitioned according to their types. New abstract types may be defined using concrete types and previously defined abstract types. Programming with abstract data types /LIS-74/ leads to programs which are structured as a hierarchy of data abstractions. In particular the logical architecture of a virtual node can be conceived as a set of processes operating on a set of (concrete and abstract) objects defined at various levels of abstraction.

As far as the underlying physical system is concerned, our assumptions about node, network and media behaviour are the same as those discussed in /LAM-78/. In particular we assumed that a distributed system is subject to independent failure modes of its components.

3. Atomic Actions. Atomic Action (A.A.) were recently proposed and described by various authors /LOM-77; BES-81; CRI-79; RAN-78; LAM-78; REE-78/ as a general tool for system structuring. In particular an A.A. is conceived as a "consistency unit" that is an operation that maintains data consistency. A consistency constraint is a predicate that can be either object or application dependent. The role of an A.A. is to maintain data consistency in spite of concurrent accesses to shared objects and possible failures of system components. Thus the main property of an A.A. is that its behavior can be described and understood considering the A.A. separately. Attempts to define A.A.'s properties are present in the literature. A.A.'s are supposed to be serializable, indivisible, instantaneous, recoverable, independent, all-or-nothing and so on. We suggest that two separate, but mutually dependent, properties must be satisfied by A.A.'s in order to maintain consistency in spite of concurrency and crash occurrences, that is indivisibility and crash atomicity.

As previously described an A.A. can be defined as an operation that, when applied to some set O of objects, transforms their initial consistent state in a final consistent state, passing through a sequence of intermediate inconsistent states. The indivisibility property ensures that intermediate states are not modifiable or observable by other A.A.'s, including those running concurrently. Indivisibility requires a synchronization mechanism.

The crash atomicity property implies that a crash occurring during the execution of an A.A. does not leave O in an inconsistent state. That is, an A.A. must be defined as an operation that either produces the specified effect or leaves the objects unchanged. This property of an A.A. is also named all-or-nothing property.

As previously pointed out, a processor crash implies the loss of all the data stored in its volatile memory. Thus, to ensure crash atomicity, we assume that any processor is able to store information in a particular type of memory, called stable memory, that never fails and whose contents are unaffected by any failure of system components. Implementations of stable memory have been presented in the literature /LAM-78; SVO-80/. In this paper it is assumed that all the nodes provide stable memory and that information stored there can be changed atomically, although it does not necessarily mean that such information is updated in place. Information stored in stable memory must be changed atomically so that a crash in the middle of the change

does not leave the stable data in an inconsistent state. Thus we will assume the existence of two primitive atomic operations to manipulate stable memory: stable-write (S.W.) and stable-read (S.R.). They are executed to atomically transfer information to or from stable memory. We will assume each S.W. or S.R. transfer a set n of bytes, called a stable page.

Stable storage can be used to implement the all-or-nothing property of an A.A. storing there the initial values of the objects modified by the A.A. (stable objects) and introducing appropriate crash recovery procedures (Restart) for bringing objects back to their initial values if a crash occurs before the A.A. ends. Furthermore, processes executing A.A.'s must be restarted at the beginning of their A.A.'s. Thus, stable memory allows processes to save their states at some well-defined points in their execution (check-points). The procedure Restart is then executed after a crash to restore in volatile memory the last saved copy of any stable object and to restore each process to its most recently saved state.

To satisfy the all-or-nothing property it would be sufficient to save in the stable memory both the state of the involved stable objects and the state of the executing process at the beginning of any A.A., saving again the new values at the end of its execution. Obviously this solution would work only in the case of an atomic saving operation, e.g. a single S.W. Unfortunately we can not assume to save both stable objects and process state with a single S.W. Thus, the problem to be solved in the following sections is to program an atomic save operation in terms of sequences of S.W.'s. From now on we assume the possibility of saving the state of a process through a single S.W. We will name SET such S.W.. Then, after a crash a process is restored to the last executed SET. As a consequence of the above assumption the process state is limited to a few hundred bytes, but this restriction can be easily removed /LAM-78/.

As far as crash recovery is concerned we will structure any A.A. following the scheme of figure 3.1.

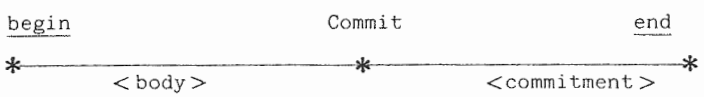


Figure 3.1. Structure of an A.A.

Any A.A. is partitioned in two parts delimited by a more primitive A.A. called a commit action (usually a SET operation). The first part, called body, includes the "normal" operation of the A.A. The second one, called commitment, is concerned with saving the new values of the involved stable objects. A crash during the body implies an abort of the action; in fact the values of the stable objects in stable memory are as they were at the beginning of the A.A. ("nothing effect"). A crash during the commitment must imply the correct completion of the A.A. The effect is the same as if a crash occurred after the end of the action ("all effect"). In other words the commit action distinguishes the undoable operations of an A.A. from the

operations that can not be undone any more.

4. Single process Atomic Actions. Let us now describe how high level A.A.'s can be structured using the only primitive A.A.'s S.W. and SET, starting from the simplest case of A.A.'s involving only one process. A.A.'s operating on a single object will be separately described from multi-object A.A.'s.

4.1. Atomic Actions on a single private object. This case concerns an A.A. consisting of an operation op a process P executes on an stable abstract object O, where O is private to P: Concurrent accesses to O are not possible. Then the first property of A.A.'s (indivisibility) is automatically satisfied for op (sequential case). We must now specify how op must be programmed in order to satisfy the all-or-nothing property when crashes are involved. First of all, let us consider the very simple case in which the stable copy of O resides in only one stable page. To make atomic the operation op it is sufficient to terminate its execution with a S.W. that saves the new computed value of O. Since S.W. is atomic, then when a crash occurs the final S.W. has either happened (in which case the situation is equal to a crash just after the end of the operation) or not (in which case it is identical to a crash just before the beginning of op). In fact, in the second case the executing process is restored back to the last executed SET (check-point), while the stable copy of O is not modified. Then op will be executed again with the same values.

With respect to the general structure of an A.A. given in section 3, the body corresponds to the complete operation op, the commit action to the final S.W. and the commitment phase is null.

Note that this solution works only if the A.A. is a restartable operation, (an operation op is defined as a restartable operation if executing any number of different prefixes of op and then executing op has the same effect of executing op). In fact, a crash occurring after the final S.W. and before the following SET implies a new execution of op with O having the new value saved by the final S.W.

A more general solution, without the previous constraints and valid also for objects residing in more than one stable page, consists in the implementation of an atomic operation that saves both the state of O and of the executing process. We will call SAVE that operation. Let $P_1 \ldots\ldots P_n$ be the n stable pages containing the value of O and $P'_1 \ldots\ldots P'_n$ be an n-tuple of stable pages, called intention pages, used as a work area in stable memory. The SAVE operation can be structured as in Figure 4.1.:

```
SAVE: begin
        <sequence of S.W. to store the new value of O in the intention pages>
          SET;
        <sequence of transfers from the intention pages to the object pages>
         SET;

      end
```

Figure 4.1. General structure of SAVE

The task of verifying the SAVE atomicity is a trival one and is left to the reader.

The A.A. assumes then the following structure:

begin <op> ; SAVE; end

The commit action corresponds to the first SET of the SAVE operation and the commitment phase to the sequence of transfers from the intention pages.

4.2. Atomic Actions on a single shared object. This case regards an A.A. consisting of an operation op a process P executes on a stable shared object M. Since the object is shared, concurrent accesses to M are now possible. In order to satisfy indivisibility, it is sufficient to execute op in mutual exclusion with other accesses to M. Then, we define M as a monitor /HOA-74/, programming op as a monitor procedure.

As far as the all-or-nothing property is concerned, the atomicity of op can be obtained by saving the new value of M at the operation end (through single S.W. or SAVE, depending on both object size and restartability of op, as in the case of a private object). However, the mutual exclusion of op now creates a problem. In fact, mutual exclusion is provided by a monitor lock which is part of the monitor data. A monitor procedure op initially acquires this lock by executing a primitive LOCK operation, then operates on the monitor data and finally releases the monitor lock through a primitive UNLOCK operation. As pointed out in /LAM-78/ the stable copy of a monitor M may or may not include the lock information. In the first simpler case LOCK and UNLOCK operations manipulate only volatile memory. The second case is more complex and expensive, since stable memory is also involved.

Unfortunately the alternative of keeping the lock information only in volatile memory can be adopted only when op terminates with a single S.W. Such a solution presents, however, the limitations pointed out in section 4.1. When op terminates with a SAVE, the lock information must be kept also in stable memory. In fact, if a crash occurs during the commitment phase, the executing process is restored to the last executed SET (the first SET in SAVE, see Figure 4.1.), that is, it is restarted inside the monitor procedure op. Then, if the lock information is only in volatile memory, such information would be lost at crash time and the process would be restarted inside a free-locked monitor.

Since lock information is maintained also in stable memory, LOCK and UNLOCK operations must be modified in order to record in stable memory the lock state. We will denote these operations as stable-LOCK and stable-UNLOCK. Figure 4.2. shows the structure of a stable monitor procedure.

begin; stable-LOCK; <op>; SAVE; stable-UNLOCK; end

Figure 4.2. Single process, single object A.A.

From the structure of this solution the important remark should be made, that stable-LOCK and stable-UNLOCK must be programmed as restartable operations.

From a logical point of view, the lock information and the two operations, stable-LOCK and stable-UNLOCK can be viewed, on the whole, as a primitive monitor used to implement more general monitors. In particular, this primitive monitor has all the characteristics of the simpler case in which the lock information is kept only in volatile memory. In Figure 4.3. a possible solution for the primitive monitor is shown. In this solution, lock-indicator=0 denotes that the controlled general monitor is free, while lock-indicator = i denotes that the process Pi is executing a monitor procedure. The integer parameter, passed to stable-LOCK and stable-UNLOCK, denotes the name of the calling process. It is used to obtain restartable operations.

```
Monitor mutual-exclusion=
     var lock-indicator:integer;
         c : condition;
     Procedure stable-LOCK (p: integer);
         begin
            if lock-indicator=0 then begin
                                        lock-indicator :=p;
                                        S.W. (lock-indicator);

                                     end
                                else if lock-indicator ≠ p then begin
                                                               c.wait;
                                                               lock-indicator:=p;

                                                                    end
         end
     Procedure stable-UNLOCK (p: integer);
         begin
            if lock-indicator = p then begin
                                              lock-indicator:=0;
                                              S.W. (lock-indicator);
                                              c.signal;

                                           end

         end
     begin lock-indicator:=0; S.W. (lock-indicator); end
```

Figure 4.3. Stable-LOCK and stable-UNLOCK as monitor procedures

4.3. Multiple objects Atomic Actions. Let us examine the case of an A.A. op, consisting of a sequence of operations a process executes on a set O= $\{o_1 \ldots o_n\}$ of objects like a transaction on a data base system.

The entire sequence is an A.A. if intermediate states assumed by the objects in O are not visible to other processes. In /GRA-78/ Gray proves that, to obtain this goal, the executing process must follow the so called

two-phase lock protocol. This protocol requires that the executing process requests and acquires the exclusive rights to manipulate the objects in O before using them and releases these rights when the objects are no more needed; furthermore, no release must precede any request. A typical structure assumed by the A.A. may be as follows: first a sequence of requests, then the sequence of operations and finally a sequence of releases (see Figure 4.4.).

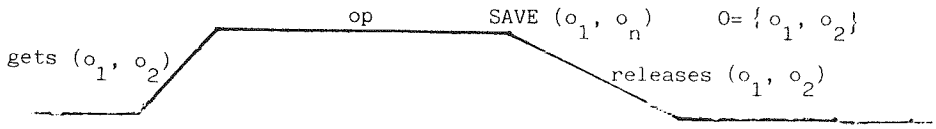


Figure 4.4. Single process, multi-object A.A.

For these reasons, objects in O must be dinamically allocated to processes. That implies the need of a set M of managers to allocate the corresponding objects to processes. A manager m_i for the object o_i is structured as a monitor whose local data contain information about the allocation state of the managed object, and whose procedures are called to get or release the exclusive right to manipulate the object /ANC-81/.

As far as the all-or-nothing property is concerned, it is again sufficient to terminate the whole sequence op with a SAVE operation. Obviously, SAVE must be executed before the releasing phase (see Figure 4.4.).

From the same considerations shown in section 4.2. it follows that the allocation state of any manager must be kept in stable memory (like the lock information in the case of a single monitor). Then, monitors implementing object managers must be programmed as stable monitors. For this purpose we can adopt the simpler solution, as shown in Figure 4.3., if get and release procedures are restartable operations.

5. Multiprocess Atomic Actions

Now let us examine the case of a set P of processes, designed to work on the same set O of objects for a certain part of their execution. We want to investigate the possibility that, during such a period, the operations of the whole set P may be given the characteristics of an unique A.A. op /RAN-75/.

Let us assume first both P and O be resident on the same node of the network. Thus, processes share the same physical memory and, according to our model, communicate via common objects (monitors). Let us see now if the basic A.A. properties hold when each process follows a two-phase lock protocol with the objects of O (see section 4.3.). Looking at Figure 5.1., it is easy to show that unwanted effects may occur.

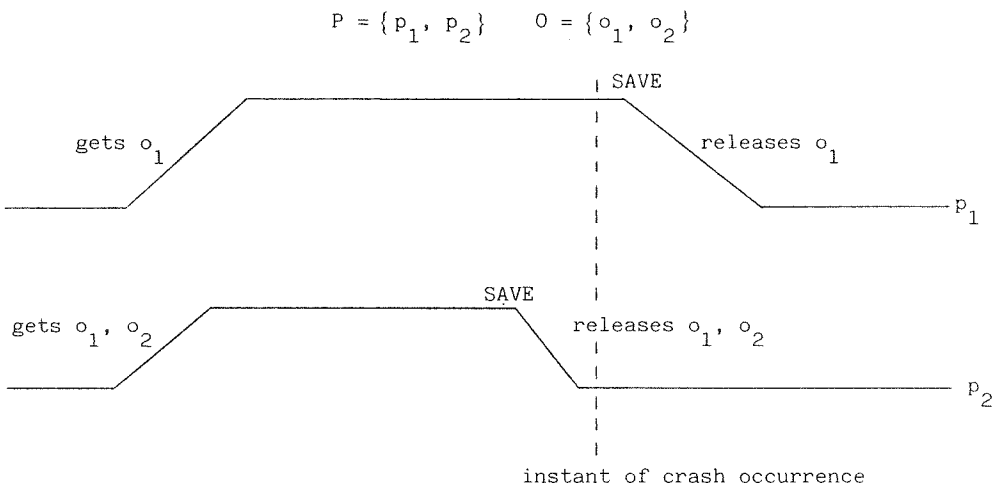


Figure 5.1. Two process performing each a two-phase lock protocol

If a crash occurs as shown, p_1 restarts finding o_1 in a state that is possibly different from the old one, since a new version of o_1 has been saved (in stable memory) by p_2. Moreover, if p_1 and p_2 must exchange information, p_1 can be blocked and deadlock may occur.

If no crash occurs, o_2 can be totally free while o_1 is not yet in its final state, thus disclosing part of the O information before op is terminated.

These unwanted effects can be removed if some mechanism is provided that guarantees all the processes of P to release together and "instantaneously" the objects of O.

5.1. Requirements of a multiprocess commit

In order to perform a multiprocess commit, each participant will have to execute a special operation, analogous to SAVE, before releasing the common objects. Let us denote it as COSAVE. Properties required for COSAVE must be such that the mentioned drawbacks can be avoided:

i) No process may finish executing COSAVE before another starts executing it. This can be obtained by means of a "rendez-vous" mechanism.

ii) A strategy for handling restarts must be established for any occurrence of a crash with respect to the execution of COSAVE. COSAVE must be implemented in such a way that the ultimate effect of a crash should be the same as if it occurred either before or after all COSAVE executions in P. That expresses an "all or nothing" property for COSAVE with respect to crashes.

5.2. Structure of a multiprocess commit

Let us see how COSAVE must be implemented in order to meet the above

requirements.

As far as the rendez-vous is concerned, processes must be able to communicate via common objects providing both information about the state of the A.A. and synchronization facilities. The monitor utilized for this purpose should enable a process to communicate that it is ready to release the common objects, and check for the same condition in the other processes.

Regarding the behaviour at crash occurrence, both crash detection and possible properly handled abort of the entire A.A. must be taken care of in COSAVE. This also can be done by means of monitor procedures.

An example of such a monitor is presented in Figure 5.3. Its procedures are called by COSAVE, listed in Figure 5.2.

N is the number of processes in P; Flag /p/ is a variable used to exchange the ready-to-release information for process p; c is a shared variable, updated in volatile memory, used detect crash occurrences.

By looking at COSAVE (Fig. 5.2.) it is clear that the A.A. must be aborted if a crash occurred when only $K < N$ processes executed the first SET (abort condition).

If a crash occurs before any SET in COSAVE, then no process can execute the Commit procedure, and all the processes restart from the beginning of the A.A. with the stable copy of O unchanged. In this case, the abort of the A.A. is managed by the restart procedure itself.

The procedure Attempt is called before SET to update the crash indicator c. In procedure Commit each participant first tests the all-Flags-true condition. If it holds, then the A.A. can be committed. The (only) process executing S.W. (Flags) in Commit is the one that commits the A.A.. If the condition does not hold, c is tested. At this point of execution it can be shown that $c < N$ is a necessary condition for the abort condition.

In fact, if no crash occurred before c is tested, then $c = N$, because of the rendez-vous in Attempt. If a crash occurred after at least a SET, then $0 \leqslant c \leqslant N-K$ holds, since c is set to zero by restart, and $0 < K \leqslant N$ holds too, since at least the testing process executed SET.

Then, the abort condition (crash and $0 < K < N$) gives $0 \leqslant c < N$.

As mentioned above, this condition is not sufficient for the abort condition, since $c = 0$ includes the case: crash and $K = N$, in which the A.A. does not need to be aborted.

The effects of Restore are to set the control of the process back to the last SET preceding COSAVE.

In the case some of the participant processes reside on different nodes of the network, the remarks made at the beginning of this section still hold. Furthermore, the functionality of COSAVE can be made identical, in spite of the different communication mechanisms involved /LAU-79/. Local crash detection may be accomplished by using timeout features. The COSAVE solution leads to the implementation of the well known two-phase commit protocol, proposed in the field of the distributed atomic transactions on data bases /GRA-78, SHR-81, LAM-78/, whereas the proposed scheme in the case of an intra-node A.A. can be viewed as a way to implement Randell's "Conversation" /RAN-75/.

```
COSAVE: begin

                              <write intentions>;
                               Attempt;
                               SET;
                               Commit (Res, p);
                               if Res = Abort then Restore;
                              <set objects value to intentions>;
        end
```

Figure 5.2. General structure of COSAVE

```
Monitor Commit_record (N: integer)=
     var c : 1..N; Flag: array /1..N/ of boolean;
         CONDC, CONDF : condition;
Procedure Attempt;
     var j : 1..N-1;
     begin
         c: c+1;
         if c<N then CONDC.wait;
                else for j:=1 to N-1 do CONDC.signal;
     end

Procedure Commit (Res: (OK, ABORT), p: 1..N);
     var j: 1..N-1;
     begin
         Flag /p/:=true;
         if<all Flags true>then begin
                                S.W. (Flags);          {commit}
                                for j:=1 to N-1 do CONDF.signal;
                                Res:=OK;
                                end
                           else if c<N then begin
                                            Flag /P/:=false;
                                            Res:=ABORT;
                                            end
                                       else begin
                                            CONDF.wait;
                                            Res:=OK;
                                            end
         end

     begin
         c:=0; for i:=1 to N do Flag /i/:=false; S.W. (c); S.W. (Flags); end
```

Figure 5.3. A monitor used to perform a multiprocess commit

6. Conclusions

In the previous sections various schemes have been examined in order to point out both properties and abstract structures of A.A.'s. To make this analysis easier practical implementation problems have not been taken into account. To get deeper insights of such problems, while waiting for the first Cnet running version, an implementation activity is going to start in a simulated distributed environment. This work also has the purpose of verifying the fault-tolerance properties of a system implemented in terms of A.A.'s.

Some problems are rising during the design phase:

i) References to the stable storage should be kept as few as possible in order to enhance efficiency. For this purpose we have decided to limit stable objects to "long-lived" objects, as records, files and so on.

ii) Both for efficiency and for actual implementability of SET as a single S.W., the use of SET has been restricted to the other level of procedure nesting. For the same reason, stable objects must be declared in the order nesting level. In other words, stable objects are static variables.

iii) Many practical expedients may be followed in order to improve system efficiency and to save stable memory space. For instance, in the implementation of SAVE two n-tuples of stable pages are used to maintain both the old value of the object to be saved and the new value (intention pages). Furthermore, a stable pointer is used to denote the tuple containing the current (old) value. In this way, the second part of the SAVE (see fig. 4.2.) is simply the binding of the stable pointer to the tuple containing the new value.

7. References

/ADA-80/ Reference Manual for ADA programming language. U.S. Department of Defense. July 1980.

/ANC-81/ P. Ancilotti, M. Boari, N. Lijtmaer "Linguistic Mechanisms for Resource Management Strategies" Software Practice and Experience vol.11. March 1981.

/BES-81/ E. Best, B. Randell "A formal model of atomicity in Asynchronous Systems" Acta Informatica 1981.

/BRI-75/ P. Brinch Hansen "The Programming Language Concurrent Pascal" IEEE Trans. on Soft. Eng. vol. SE-1. 1975.

/CNET-80/ Cnet Project, a Status Report, Pisa 1981.

/CRI-79/ F. Cristian "A recovery mechanisms for modular software" Proceedings of the 4th Inter. Conf. on Softw. Eng. Munich 1979.

/GRA-78/ J.N. Gray "Notes on Data Base Operating Systems" Lectures Notes in Computer Science, Springer Verlag, vol. 60.

/HOA-74/ C.A.R. Hoare "Monitors: an operating system structuring concept" Comm ACM vol.10, October 1974.

/LAM-78/ B.W. Lampson, H.E. Sturgis, "Crash recovery in a distributed data storage system" XEROX PARC, Palo Alto California. (to appear on Comm. ACM).

/LAU-79/ H.C. Lauer, R.M. Needham "On the duality of Operating System Structures ACM Operating Systems Review vol. 13, n. 2, April 1979.

/LIS-74/ B.N. Liskov, S. Zilles, "Programming with abstract data types" Sigplan Notices vol. 14, n.4, 1974.

/LOM-77/ D.B. Lomet, "Process Structuring, Synchronization and Recovery Using Atomic Actions" Proceedings of ACM Conference on Language Design for Reliable Software ACM SIGPLAN Notices vol. 12, n. 3, March 1977.

/RAN-75/ B. Randell, "System structure for Software Fault Tolerance" IEEE Trans. on Soft. Eng., June 1975.

/REE-78/ D.P. Reed "Naming and synchronization in a decentralized Computer System" Lab. for Comp. Science, M.I.T. Tecnichal Report TR-205, September 1978.

/SCH-80/ F.B. Schneider, R.D. Schlichting, "Toward fault tolerant process control software" Proceeding of the FTCS81, June 1981.

/SHR-81/ S.K. Shrivastava, "Structuring Distributed System for Recoverability and Crash Resistance" IEEE Trans. on Soft. Eng., July 1981.

/SVO-80/ L. Svobodova, "Management of Object Histories in the SWALLOW Repository" MIT Laboratory for Computer Science Technical Report 243, Cambridge, Mass, July 1980.

Finito di stampare nel dicembre 1982
dall'ETS PISA

33. **R. Marcogliese, R. Novarese**, The Cnet Monitoring Center and its Performance Monitoring Function
34. **G.F. Rossi**, Meccanismi di comunicazione tra processi concorrenti nel linguaggio Ada
35. **C. Ghezzi, D. Mandrioli, F. Tisato, R. Zicari**, A Critical Analysis of the Ada Task System with Respect to Real Time Programming
36. **N. Cocco, C. Ghezzi, D. Mandrioli, F. Tisato**, Mechanisms for Building Abstract Machines
37. **G.F. Rossi**, Bibliografia sul linguaggio Ada (per argomenti, parzialmente commentata)
38. **C. Ghezzi, D. Mandrioli, F. Tisato**, A Machine Architecture Supporting Asynchronous Activities
39. **M. Coppo, S. Ronchi Della Rocca**, Analisi di alcuni modelli di sistemi distribuiti
40. **M. Venturini Zilli**, Nondeterminism and Computability
41. **P. Ancilotti, M. Fusani**, Azioni atomiche: strumenti e proprietà
42. **P. Ancilotti**, Relazione sull'attività del gruppo di lavoro su Fault-Tolerance
43. **I. Castellani, P. Franceschi, U. Montanari**, Labeled Event Structures: a Model for Observable Concurrency
44. **M. Bellia, P. Degano**, The Tool of AISS for Program Editing, Abstract Representation Construction and Program Displaying
45. **R. Barbuti, A. Martelli**, Execution and Debugging Tools in AISS Based on Denotational Semantics Concepts
46. **E. Dameri, C. Simonelli**, Rappresentazione intermedia di programmi in AISS
47. **G. Attardi**, Office Information Systems design and Implementation
48. **R. Barbuti, M. Bellia, A. Martelli, E. Dameri, C. Simonelli, P. Degano, G. Levi**, Towards the Derivation of an Experimental Programming Environment from Language Formal Specifications
49. **P. Amatruda**, Considerazioni sulle eccezioni e loro gestione: conseguenze sui requisiti di un linguaggio di programmazione
50. **F. Bruni**, Sull'inadeguatezza di qualsiasi insieme finito di strutture per rappresentare reti a livelli di astrazione successivi
51. **C. Dionisio, R. Vannini**, Survey sul trattamento digitale del segnale vocale
52. **F. Giannessi**, Tool Composition and Interprogram Communication Mechanisms
53. **E. Lazzari, F. Nicodemi**, Proposta per il Debugger
54. **F. Gallo**, Kapse Database Functional Specification
55. **F. Gallo**, Kapse Database Overview
56. **P. Inverardi, G. Vallario**, Alcune considerazioni sul KAPSE
57. **M. Ajmone Marsan, G. Albertengo**, Map: an Insertion Protocol for an Unidirectional Bus Local Network
58. **M. Martelli, F. Tarini**, Primitive di comunicazione
59. **F. Borgonovo, L. Fratta, F.A. Tobagi**, The Express Net: A Local Area Communication Network Integrating Voice and Data
60. **P. Inverardi, G.N. Vallario**, KAPSE: Un costrutto limitato di acquisizione e riconfigurazione dinamica per ADA
61. **E. Astesiano, G. Reggio, E. Zucca**, Operational Frameworks for Semantic Description of Concurrent Languages. With an application to ADA-like languages
62. **M. Di Santo, L. Nigro, W. Russo**, Impiego dei regimi di controllo per Forward e Backward Error Recovery

63. **A. Pistorello**, Aspetti di configurabilità di un sistema di elaborazione in un ambiente distribuito

64. **R. Martucci**, Introduzione al linguaggio CHILL

65. **S. Belli, R. Impagnatiello**, Elementi di Fault Tolerance nei sistemi di elaborazione e nelle loro applicazioni nel campo delle telecomunicazioni

66. **P. Colombo, C. Ghezzi, D. Mandrioli, A. Tecchio, F. Tisato, R. Zicari**, X–CODE

67. **R. Barbuti, M. Bellia, P. Degano, G. Levi, A. Martelli**, Programming Environments: Deriving Language Dependent Tools From Structured Denotational Semantics

68. **P. Inverardi, U. Montanari, G.N. Vallario**, How to program an APSE (almost) completely in ADA

69. **F. Giannessi, F. Nicodemi**, Architettura e funzionalità del sistema di Debugging di programmi ADA

70. **P. Inverardi, G.N. Vallario**, Architettura di ambiente e struttura del KAPSE

71. **M. Chesi, E. Dameri, P. Franceschi, G. Mainetto, C. Simonelli**, Specifiche e architettura di un sistema per la manipolazione di alberi di sintassi astratta in un ambiente integrato di sviluppo software.

72. **M. Chesi, P. Franceschi**, Il Pascal e gli strumenti di programmazione di UNIX(TM) per la realizzazione del prototipo AISS

73. **N. Lijtmaer**, Cnet: stato della ricerca e proposta 1982

74. **I. Castellani, U. Montanari**, Graph Grammars for Distributed Systems

75. **E. Astesiano, E. Zucca**, Parametric Channels in CCS and Their Applications

76. **O. Martini, G. Pacini, G. Pierini**, Automazione d'ufficio e manipolazione di documenti

77. **A. Brosio, A. Moncalvo, P. Morra**, Sottosistema di comunicazione a 34 Mbit/s in fibra ottica.

78. **M. Battù, M. Coppo, P. Gouthier**, Verso la descrizione formale di un costrutto di riconfigurazione dinamica per ADA.