On a scheme for backward recovery in complex systems including both client processes and data servers

Lorenzo Strigini*, Felicita Di Giandomenico**, Alexander Romanovsky***

* Centre for Software Reliability, City University
** Istituto di Elaborazione della Informazione, Pisa, Italy
*** Dept. of Computing Science, University of Newcastle-upon-Tyne, U.K.

Technical Report. October 1996. Last revised: March 1997**.**

Abstract

We discuss a design scheme for co-ordinated backward recovery in complex systems. This report concludes a series of papers on this topic. We give the rationale of our proposed approach, a complete, reasoned specification of the mechanisms, a comparison with other related research, and pointers to other papers describing proof-of-concept examples of use and implementation of this scheme..

We consider backward error recovery for complex software systems, where different subsystems may belong to essentially different application areas, like databases and process control. Examples of such systems are found in modern telecommunication, transportation, manufacturing and military applications. Such heterogeneous subsystems are naturally built according to different design "models", viz. the "object-action" model (where the long-term state of the computation is encapsulated in data objects, and active processes invoke operations on these objects), and the "process-conversation" model (where the state is contained in the processes, communicating via messages). To allow backward error recovery in these two "models" of computation, two different schemes are most appropriate. For the object-action model of computation, *atomic transactions* are now the accepted model of backward recovery. For the process-conversation model, a recovery scheme based on planned *conversations* has been widely studied. We have shown how checkpointing and roll-back can be co-ordinated between two sets of such heterogeneous subsystems, namely sets of message passing processes organised in *conversations* and data servers offering *atomic transactions*. Assuming that each of the two kinds of subsystem already has functioning mechanisms for backward error recovery, we have described the additional provisions needed for co-ordination between heterogeneous subsystems. Our additions are based on rather general models of both transactions or conversations: they could be adapted for most specific instances of either scheme. Our solution involves altering the virtual machine on which the programs run, and programming conventions which seem rather natural and can be automatically enforced.

Related papers. The ideas presented here were first outlined in [27]. A shorter version of the discussion and specification parts of this report, with an example of use of our method, is published in [29]. Material from that paper is reused here with permission from IEE. The report [28] also describes an Ada implementation of a simple version of our proposed scheme.

Author's addresses:

L. Strigini, Centre for Software Reliability, City University, Northampton Square, London EC1V OHB, UK. E-mail: strigini@csr.city.ac.uk

F. Di Giandomenico, Istituto di Elaborazione della Informazione, CNR, Via S. Maria 46, Pisa, 56126 Italy. E-mail: Di Giandomenico@iei.pi.cnr.it

A. Romanovsky, Computing Department, University of Newcastle upon Tyne, NE1 7RU, Newcastle upon Tyne, UK. E-mail: Alexander.Romanovsky@newcastle.ac.uk

1. Introduction

1.1. Backward recovery

A general means for providing fault tolerance is *backward* error recovery. This consists in "rolling back" a computation, in which an error has been detected, to a previous state (a *recovery point* or *checkpoint*), assumed to be correct, a copy of which had previously been saved for this purpose. The computation may then be restarted from this correct state, and will proceed correctly, if the cause of the previous error is not encountered again.

Backward recovery in a system of multiple processes requires co-ordination between processes to avoid inconsistencies in the overall state of the system after roll-back. Among the various schemes studied for ensuring consistency, we consider in this paper *planned conversations* (in which the internal states of a set of communicating processes are rolled back together), which also allow the recomputation after roll-back to use different code from the first computation, so that errors caused by software design faults may not be repeated at the new execution [2, 8, 20], and *atomic transactions* (in which a sequence of changes on a set of data items are undone together), which allow the designer to manage together error recovery and concurrency control in accessing data objects (external to the processes) [12, 18]. Conversations and atomic transactions are in fact *dual* models of recovery, as discussed in [26]: they are two ways of describing the same backward recovery philosophy in the two models (or design styles) which the authors of [26] call the "object-action" model (where the long-term state of the computation is encapsulated in data objects, and active processes invoke operations on these objects), and the "process-conversation" model (where the state is contained in the memory of the processes, which communicate via messages).

Conversations have been studied mainly in application scenarios of tightly coupled processes, interacting in a predictable fashion, e.g. control applications. Atomic transactions are characteristic of applications where different, independent agents access shared data at times that they decide autonomously from each other and from the data repositories. It would be wrong to conclude, from the consideration of the duality of the two models, that one of these two models, or design organisation principles, is sufficient for all applications. Design styles must first of all be natural for the people who have to use them, and there is little doubt that for each of these two programming models there is a sizeable community of designers who feel at ease with it. Requiring a designer to use an unnatural (for him or her) design style for the sake of language standardisation would be quite wrong: the designers' intelligence is best spent in making a design right, rather than in adapting to a counterintuitive notation, and high-level design errors, as could be caused by casting an application in an "unnatural" computational model, are certainly a concern even for mature software development organisations. So, both the "object-action" and the "process-conversation" models (as well as others) are here to stay, and so are the styles of backward recovery that best adapt to them.

1.2 Heterogeneous systems and our proposal

An interesting aspect of large, complex modern systems is that they may include parts that belong to traditionally separate application domains, and thus are naturally designed according to different approaches. For instance, the "intelligent network" evolution of the telephone network relies on the interaction of real-time control components with vast on-line data bases, with different timing and availability requirements [11]. Similar factors of heterogeneity may be expected in other large-scale control systems, e.g. in transportation (air traffic, vehicle-highway systems), and integrated manufacturing control. In all these cases, if backward recovery is employed, difficulties may be expected in co-ordinating such heterogeneous subsystems.

As a consequence of these considerations, we studied in [27] a scheme for co-ordinating the backward recovery of subsystems of processes using conversations with "data-server" subsystems offering an atomic transaction-based interface (we call these "TDSCs", for *transactional data server components*). This would allow a clean structuring of the overall

system into active processes, whose states are protected via conversations, and passive data, protected by the atomic transaction-based servers. Our approach imposes some further restrictions on the patterns of interaction that the application designer is allowed to program between the subsystems, and requires some additional facilities in the run-time environment and/or the language compiler[2]. The result is to hide from the application programmer some of the complexity of guaranteeing consistent recovery in such complex systems.

It is appropriate to emphasise that we do not propose a monolithic platform offering facilities for every application programmer to use both atomic transactions and conversations. Rather, we assume that the whole application system is divided into data-server subsystems (TDSCs) and client subsystems (possibly running on separate and even different platforms). The TDSCs are designed (at the application level, plausibly) to offer their clients the use of atomic transactions; this facility in the TDSCs may be largely programmed at the application level, so that the concurrency control and recovery algorithms are tailored to the semantics of the data, their usual access patterns etc. The client subsystems are built according to the "process-conversation" model, and it is therefore natural to give them interpreter-provided support for recovery via conversations. So, the programmers of client processes would use conversations to organise the recovery of groups of communicating client processes. Here, the mechanisms for checkpointing and roll-back would be mostly provided by the interpreter and the application designer's authority would be limited to designating the parts of computation forming each conversation. When, instead, the client processes have to use the data in the TDSCs, they would exploit the atomic transactions offered by the TDSCs to guarantee concurrency control and manage roll-back. This, however, would not prevent some kinds of inconsistency in backward recovery. We then propose enhancements to the interpreter mechanisms which support conversations (which, in a distributed system, may well be limited to that subset of platforms that runs client processes) to preserve consistency across roll-back operations.

Design issues for atomic transactions and conversations are well understood, and we specify our added co-ordination mechanism without reference to details of how either mechanism is implemented in the pre-existing subsystems. For similar reasons, we are not interested in which variant of either scheme is implemented, and do not propose any new variant. We do use, in our discussion, a reference specification for each of the two schemes, so as to allow us to specify our proposed additions with sufficient precision. However, our reference specifications are rather general: for instance, we would expect that adapting our proposal for use with existing transactional DBMSs would require dropping some of its features (e.g., nested transactions, or co-ordination of transactions on multiple servers) rather than changing it radically. In addition, we note that our proposal is not a simple extension to either model. There may be cases in which an application that can be implemented with our solution could also be designed by using some ad-hoc extension of existing transactional or conversational models; but such extensions would not have the general property of allowing interconnections of pre-existing subsystems of the two kinds.

1.3 Related research

There have been very few previous proposals for co-ordinating conversations with server processes. The difficulty of managing concurrent, independent accesses to shared data servers in the framework of conversations was pointed out in [13]. The problem of joint recovery of client processes (organised in recovery blocks) together with data-base management systems was independently raised in [14], which considers enhancements to the Recovery Meta Program (RMP) approach [1] - a monolithic, interpreter-level solution - to support this kind of recovery.

---

[2] We shall call these, together, the *interpreter* on which the application runs, using the terminology of [16]. For the purpose of most of the ensuing discussion, one can picture the functions of the "interpreter" as realised in a complex run-time support, either part of a kernel or of a layer overlaid above the kernel, intercepting calls from the application process. In practice, many of these functions could be realised by code and data structures at the application level, produced by the use of libraries or pre-processing before compilation.

Our work can also be seen as related to work in the field of so-called "advanced transaction models" (a survey of which can be found in [9]. There is a general difference between all those models and our proposal: these models refer to systems in which all protected interactions among activities take place through transactions on shared data; our proposal, instead, allows parts of the system to be designed according to the process-message style of interaction. Apart from this aspect, the "advanced" transaction models address essentially three problems that arise in applying the transaction models to new and, in particular, to larger-scale applications: the need for transactions to access many data items and/or to last for a long time, and therefore suffer a high risk of conflicts and/or of failures, with consequent inefficiency; the need to integrate multiple pre-existing database systems; and the need to allow *co-operation* of active entities via the shared data protected by transactions, rather than just regulating *competition* in accessing these data. The solutions proposed for these problems belong to two broad categories: allowing partially performed transactions to be tentatively committed (so that the partial results are visible outside the transaction), having programmed explicit "compensating" transactions to be invoked when the effects of the tentative commits must be undone; and defining refined application-specific consistency rules. Our approach is different and actually orthogonal to these: it allows one to combine subsystems, that individually enforce the classical ACID properties, in such a way that these properties are also guaranteed in the combined system, without requiring application-level changes to the individual subsystems[3].

Another related paper is [7], which proposes quite a different approach to combining components developed in the process and the server/client paradigms. These authors concentrate on implementing co-ordinated recovery within the Ada language [3]. Because of this, their proposal on the one hand is quite practical (though they require some changes to Ada) but on the other hand too restrictive. It implies locking all servers potentially involved in a conversation (and in all possible nested ones) for the whole duration of the conversation's execution: rather, we allow application-specific locking policies. In terms of ease of use, [7] requires the programmers to specify in each request a level of nesting, a list of all servers needed at the given level, a list of all processes in each given conversation; these requirements are absent in our proposal. (We have also explored the possibility of implementing our scheme in the Ada language. This is described in a technical report [28, Section 4 and appendices]).

The organization of this paper is as follows. In Section 2, we specify the variety of conversations and atomic transactions which we consider. In Section 3, we describe our scheme for their co-ordination, first as a high-level specification and then in detail. Section 4 contains a general discussion.

2. Conversations and Atomic Transactions. General model

2.1 Backward Recovery

We shall briefly justify here our choice of planned conversations and atomic transactions as the two models of backward recovery which we are trying to integrate.

As stated before, backward recovery is defined as "rolling back" a computation, in which an error has been detected, to a previous, correct state. By contrast, *forward* error recovery involves correcting errors so as to move the computation to a new correct state, which it could have reached had the error not happened in the first place. Backward recovery has the appeal that it does not require redundant computation (except the saving of checkpoints) during error-free processing (as opposed, for instance, to running multiple copies of a computation to mask errors by voting), and that it is a *general* recovery mechanism, so that one general set of tools for backward recovery could be used in many different applications. It has thus been the subject of much research work in the last twenty years or so. The main complication with backward error recovery is that roll-back must be *consistent*: if the computation is made up of communicating processes, rolling back only some of them will make them ignore, during re-

[3] The transaction model that we assume actually allows (but does not mandate) some of the features that [9] classifies as belonging to "advanced" models, like nesting of transactions and co-operation of processes not only via messages but also via shared access to the same data in the same transaction.

execution, communications that took place between the saving of the recovery point and the detection of the error: a rolled-back process might wait forever for a message, or read a new value of a shared variable when it should read a previous, now overwritten value, etc. Furthermore, the new execution may produce different outputs from the one that was aborted by the error, e.g. because the error is not produced, or the state of the external world has changed, or the designer explicitly provided for different code to be used in the re-execution: the correct processes, if acting upon output from the execution that was interrupted, may find themselves in states inconsistent with that of the rolled-back process.

Inconsistency may be avoided by "propagating" the roll-back operation to more processes (those that communicated with the process that is now to be rolled back, after its last checkpoint). This may unfortunately produce a "domino effect" [21]: rolling back these additional processes requires further propagation to those processes with which they communicated. Due to the domino effect, one error may cause a large amount of error-free computation to be undone. When providing what we called "general tools" for backward recovery (in the form of interpreter-provided mechanisms, or even of programming conventions and templates), guaranteeing consistency is thus an essential requirement, and limiting the domino effect is an important one.

Researchers have suggested various ways of organising the use of backward recovery so as to avoid inconsistency, simplify the task of application programmers and limit the domino effect. Among these techniques we can quote *planned conversations*, [20] and *atomic transactions* [12, 18], which, together, are the topic of this paper and will be detailed in the next two subsections. We briefly discuss here the other alternatives: avoiding roll-back propagation altogether and making it transparent to the programmer.

The problem of propagating roll-back may be avoided altogether by requiring effective confinement of errors inside individual components, and guaranteeing (in the run-time support) that all the messages received or sent by a failed component (in the example above) after a checkpoint and before an error are respectively replayed to it, and not delivered a second time to their recipients, during its re-execution [6, 19]. This solution is attractive but requires, besides perfect confinement of errors, deterministic processes: running from a given initial state, and receiving a given sequence of messages, the retry of a process must produce exactly the same message exchanges produced by the previous (aborted) run. These assumptions are often used in the world of distributed data-bases, for designing algorithms that can cope with simple (crash) hardware failures. In more critical applications, error confinement may be improved by special design of the hardware. However, it is difficult to make them hold for errors or failures caused by software faults.

Another possibility is given by *programmer-transparent schemes* (examples of which are found in [4, 16] and [30]), where the application programmer may be given the illusion of a completely reliable underlying machine. The limitations of these schemes are that they risk being less efficient (e.g. in the amount of checkpoint information to be stored) and less effective (in terms of achieved reliability) than those requiring the intervention of the programmer, like conversations and atomic transactions. A programmer can design more effective checks of correctness than those possible for an application-independent interpreter, and can plan for the checkpointing to take place at times when the state information is less and/or easier to verify.

Discarding the programmer-transparent schemes implies requiring the programmers to specify directives for checkpointing and recovery (or commit of changes) and this does allow higher effectiveness or efficiency, but also complicates their task. Schemes like conversations and atomic transactions seem to facilitate the programmers' task, compared to programmers developing backward recovery on an ad-hoc basis, by giving them a sound template to follow, and standard mechanisms to use.

2.2. Conversations

[20] proposed the use of a programming construct called a *recovery block* which combined checkpointing and backward recovery with retry by a diverse variant of the code. This allowed redundancy and design diversity to be hidden inside program blocks (Fig. 1).

ensure Acceptance Test
by $Alternate_1$
else by $Alternate_2$
. . . . . .
else by $Alternate_n$
else error.

Fig. 1. The recovery block construct.

The $Alternate_i$ modules are diverse implementations of the function of the whole recovery block, so that if an execution (by $Alternate_1$, for instance) fails the Acceptance Test, the following retry, by, for instance $Alternate_2$, may not repeat the same error. Each retry executes a different $Alternate_i$ (and is subject to the same Acceptance Test), and exchanges messages that differ (in their sequence, or in their contents) from those exchanged by the previous alternate. So, co-ordinated roll-back is necessary. To this end, the same paper proposed *conversations*. A *conversation* (Fig. 2) can be described as a multi-process recovery block: when two or more processes enter a conversation, each must checkpoint its state, and they may only leave the conversation (thus committing the results computed during the conversation, and discarding their checkpoints) by consensus that all their acceptance tests are satisfied. Processes can asynchronously enter a conversation, but all must leave it at the same time. During the conversation, they must not communicate with any process outside the conversation itself (violations of this rule are called *information smuggling*). So, the occurrence of an error in a process inside a conversation requires the roll-back of all and only the processes in the conversation, to the checkpoint established upon entering the conversation. Conversations may be nested freely, meaning that any subset of the processes involved in a conversation of nesting level *i* may enter a conversation of nesting level *i* + 1.

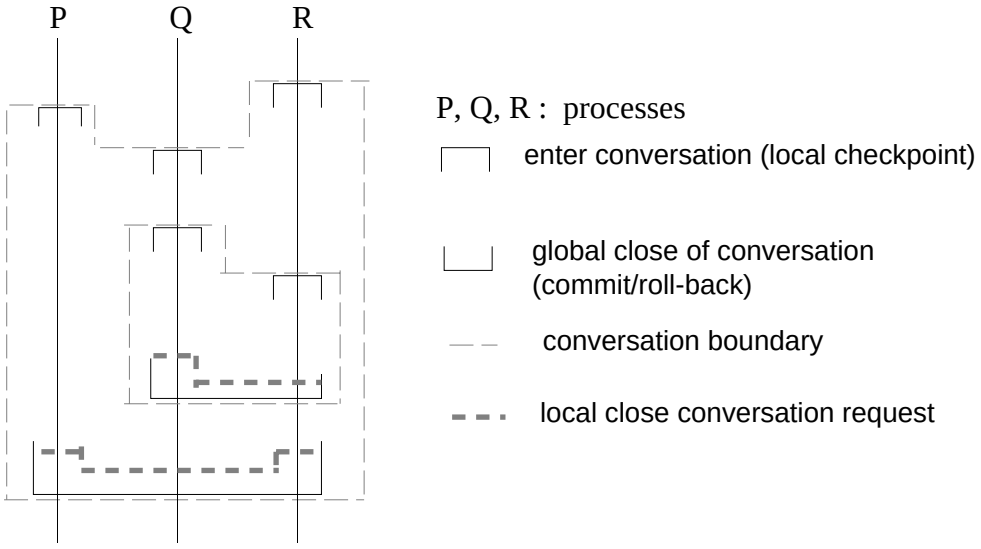

Fig. 2. Two nested conversations

[20] did not specify an implementation language construct for the conversation scheme. Several proposals have later appeared of language constructs and implementations, differing in the resulting semantics of conversations. A critical survey can be found in [8]. This latter paper concludes that the simpler implementations of conversations, based on static membership and restricted communication mechanisms, appear useful, albeit for restricted classes of applications (e.g., cyclic control systems); extensions to accommodate the needs of a wider class of applications are paid in terms of a less controllable execution and added complexity in the run-time support, and their usefulness appears questionable.

The results developed in the present paper apply to a rather wide class of implementations of conversations. We only assume that support is provided for *some* kind of conversation. A simple model is as follows. The interpreter provides application processes with an

enter_conversation, a *close_conversation* and an *abort_conversation* calls. The *enter_conversation* call may, in some implementations, carry as a parameter a conversation name or an explicit list of participants; additional controls on the legality of a process entering a given conversation may exist ([8] discusses in some detail the different options for naming and protection). Both *local acceptance tests* (inside the participating processes) and a *global acceptance test* could exist; the latter would allow checks on conditions that only hold after all the participants have concluded their respective parts of the conversation, or that involve the contents of variables which are not all visible by any one participant (how and whether this could be practically implemented depends on details of the operating system architecture and the computational model presented to the application programmer). For an application process P, the interpreter executes these calls as follows:

enter_conversation(conversation_name) :

- check that it is legal for P to participate in conversation *conversation_name*;
- checkpoint the state of P;
- record P among the participants in *conversation_name* (this information is also used in preventing communication between the participants in the conversations and processes outside it);
- activate the first alternate of P in *conversation_name*.

close_conversation (it is not necessary to specify the conversation name: it is the innermost conversation P is in; let us call it C):

- record that P agrees to close the conversation C;
- suspend P until all processes in C have agreed to close C, or at least one has issued an *abort_conversation* call for C (see below), then
- run the global acceptance test, if any; if it is passed, then atomically commit the results computed by all participants during the conversation and discard their checkpoints (the conversation is successfully closed), returning control to P after the *close_conversation* statement; if the global acceptance test fails, proceed as for *abort_conversation* below.

abort_conversation (as for *close_conversation*, it is not necessary to specify the conversation name; again let us call it C):

- record that each participant in C has tried its current alternate and failed;
- rollback all the variables modified by all the processes in the current alternate of C;
- if each participant has an alternate that it has not yet tried, activate it, otherwise raise an appropriate exception (or return an appropriate error message) for all participant processes.

This is a fairly general model of conversations, though it may exclude some of the more complex variations.

2.3. General description of atomic transactions

Atomic transactions [12, 18] are a useful means for structuring access to shared data by multiple independent agents or clients. Atomic transactions help in solving the two problems of concurrency control and error recovery, by letting the programmer of each client see only consistent states of the data, as they evolve through a strictly sequential execution of transactions, and by not letting him/her see the effects of faults. A crash failure of a data server, which interrupts the processing of transactions, is guaranteed to leave non-volatile data in such a state that, after restart, the data will be either in the consistent state preceding, or in the consistent state resulting from, the transactions.

Atomic transactions come in many flavours, which share the general concept but vary in details of their specification as well as their implementation. For our co-ordination scheme, we have assumed a quite general model of atomic transaction, as follows. A transactional data server component , or "TDSC", encapsulates a set of data, with a set of specific access methods which can be called by client processes by "remote procedure call" (RPC); and it has the

additional, special methods, *start_transaction*, *abort_transaction* and *commit_transaction* (we will later see that, if transactions on multiple TDSCs are assumed, a fourth method, *prepare_to_commit*, may be necessary). Any call to methods other than these three must be bracketed between a *start_transaction* and an *abort_transaction* or *commit_transaction*; the whole sequence, including the bracket statements, is called a transaction. *start_transaction* returns a *transaction_identifier*; all the subsequent calls in the same transaction must carry this identifier as a parameter.

We also assume that the client process which started a transaction can pass the transaction identifier to other processes, which can then operate on the TDSC concurrently with the initiator process. The transaction identifier is effectively a *capability* [25] for access to the protected data in the current transaction[4]. Thus, when a process in a conversation starts a transaction, the other processes in the conversation do not automatically become parties to this transaction, but become legally entitled to do so. The decision on whether and how to do so is left to the application programmer of the processes in the conversation.

This is a more general model of transaction than usually considered, and is motivated by the consideration that programmers of tightly interacting groups of processes may well wish to distribute the operations that are part of the same transaction among multiple processes. In this case, the programmer of the client processes is in practice seeing the set of client processes as a multi-thread process performing the transaction and must take responsibility for the co-ordination of the threads and consistency of the data after the transaction[5]. Our proposal for co-ordinated recovery in heterogeneous systems, which can deal with this generalised model of transaction, can of course deal with single-process transactions as well.

Guaranteeing serialisability is the responsibility of the programmer of the TDSC, who must implement a policy for accepting or rejecting *start_transaction* and *commit_transaction* calls (and for honouring the calls to other methods) that satisfies the consistency invariants established for the data it protects.

We assume the classical "ACID" properties of transactions: atomicity; consistency; isolation (or serialisability); and durability (or permanence). Our proposal is limited to a scheme for co-ordination with conversations, so we do not assume any particular way of implementing these properties. In a typical scenario, a run-time support may take care of atomicity and provide mechanisms for isolation and durability, and the application programmer (of the clients and of the TDSC) is probably responsible for consistency and any non-standard policies for isolation (application-specific concurrency control policies).

Nested transactions are allowed: the *start_transaction* method can be called from within a transaction. The TDSC guarantees serialisability between transactions at the same level of nesting and between nested transactions and other atomic operations performed within the same parent (enclosing) transaction. TDSCs that do *not* allow nested transactions (as is common in existing DBMSs) would allow one to use a simpler version of our co-ordination scheme. In this paper, we only describe the more general case allowing nested transactions.

We do not assume any special mechanism for co-ordinating transactions involving multiple TDSCs: if desired, they can be organised by the application programmer, possibly through an "umbrella" TDSC which takes care of starting and aborting/committing the transactions on all the other TDSCs involved. We will show that our co-ordination mechanism also offers a solution for multi-TDSC transactions.

---

4 Choosing the identifier in pseudo-random fashion from a sparsely populated name space can make this protection as effective as required. Of course the designers of the TDSC may also implement additional protection policies at the application level, e.g. to protect data confidentiality.

5 So, different client processes share the data encapsulated in the TDSC either by performing separate transactions, in which case serialisability is guaranteed by the TDSC itself, or by sharing the same transaction, in which case the correct co-ordination of their accesses is their responsibility. The TDSC may provide help in the form of internal consistency-check methods, and an ability to autonomously detect some of its internal errors, in which case it may abort on-going transactions and/or recover from the errors transparently.

3. Our proposal for co-ordinated recovery

3.1. Principles

We now assume a system made up of "conversational" components (processes, tasks, or threads) which synchronise their checkpointing/recovery via conversation-like mechanisms, and of server components (TDSCs) which implement an atomic transaction mechanism: they accept requests (in the form of messages or remote procedure calls) to start, abort and commit transactions and serve them according to some policy that ensures proper concurrency control among competing transactions. We assume that the interpreter on which the "conversational" components run accepts the *enter_conversation*, *close_conversation* and *abort_conversation* requests defined in Section 2.2, and that the division of components into conversational components and TDSCs is known to the interpreter. In addition, we assume that the interpreter is able to recognise and intercept transaction control requests (open, abort, commit) issued by the conversational components towards the TDSCs.

Our proposal is then to realise programmer-transparent co-ordination between the recovery of the conversation-based and transaction-based subsystems. Our rationale is that, while programmers of an individual subsystem should be trusted to use the mechanisms available there properly, it seems excessive to ask them to plan the co-ordination of heterogeneous components, presumably developed separately, which implement different models of computation and interact in possibly complex ways.

The co-ordination mechanism should preserve the fault tolerance properties of the component subsystems in the composite system. Specifically, the TDSCs are normally meant to preserve long-lived data of the computation, and will thus guarantee (through the ACID properties) survival through crash failures, software exceptions, and possibly subtler failure modes, depending on the error-detection mechanisms implemented in the servers. The conversation interpreter supporting the "conversational" client components may be implemented to tolerate simply software exceptions in the client (application) processes themselves, or transient errors in the processors supporting them and/or (through some form of reconfiguration) crash processor failures. Our co-ordination mechanism must allow the combined system to tolerate the union of the failure modes tolerated by the subsystems. To this aim, it is sufficient to guarantee that the means for recovery, i.e., the roll-back mechanisms, do not themselves cause inconsistencies[6].

If a component requests operations on a transactional component while engaged in a conversation, it must be ensured that these operations can be undone if the conversation were to be rolled back. The TDSC only provides an undo operation (the *abort_transaction*) for sequences of operations starting with a *start_transaction* (and not including the corresponding *commit*). So, a first rule for the programmer of the client component is:

RULE 1: *Any operation on a TDSC that is requested by the client process while participating in a conversation must be part of a transaction started within the innermost current conversation (we say that that conversation creates the transaction).*

The interpreter must block, and signal as errors, all calls that are not so bracketed. We assumed that TDSCs always reject requests which are not part of a transaction (notice that transactions may also be started by a process which is not involved in any conversation); what we are adding now is a prohibition of sequences like:

```
transaction_id := start_transaction(TDSC1); ...
open_conversation; ...
operation (transaction_id, TDSC1); ...
close_conversation;
```

6 Our add-on mechanism does not attempt to tolerate more failure modes than the mechanisms already existing in the subsystems. For instance, if the conversation interpreter is only meant to tolerate software exceptions, but does not tolerate crash failures in the processors running the client processes, then the composite system will still tolerate crash failures *only* in the TDSC processors.

Rule 1 requires the components in a conversation to start a transaction in the TDSC before any operation on its data can be performed (by components in the conversation considered). In practice, it requires the TDSC to "remember" the state of those data before they are changed by these operations, a necessary condition for these operations to be "undoable" if the conversation in which they were performed has then to be rolled back. This ability must be preserved so long as the possibility of the conversation being rolled back exists. So, the interpreter must also delay the commit of transactions, lest a subsequent roll-back of a conversation leave the effects of the transaction standing. A transaction can only be committed (i.e., the changes it made can be made permanent) at the same time as a conversation in which it was created is committed (that is, after it has passed its acceptance test[s]), and must be aborted if the conversation does not pass its acceptance tests and therefore aborts (rolls back) the current alternate. We now consider how long the commit of the transaction is to be delayed:

1. If the transaction is nested inside another transaction on the same TDSC, but both were created by processes in the same conversation, the inner-nested transaction can be committed without delay: its effects can still be undone, if necessary, by aborting its parent transaction.
2. Considering instead the outermost transaction opened inside a conversation (on a given TDSC), the rule for delaying its commit is more complex. If the conversation that created the transaction is the outermost conversation currently open, then the commit of the transaction must be delayed until the conversation itself commits: at that point, it is legal to make all the effects of that conversation visible to components outside it. However, more complex situations may exist. In Fig. 3, transaction T2 is created by conversation C3. If T2 were committed at the end of C3, its results would be visible to component P, outside C2. If then C2 rolled-back, this visibility would become a case of information smuggling through the borders of C2. So, T2 must only be committed when C2 commits. Likewise, if T1 did not exist (that is, if T2 were the outermost transaction on component Q), T2 should not be committed until C1 commits. We can say that:

RULE 2: *the commit of a transaction T must be delayed until the commit of its associated conversation, which can be found as follows.*

DEFINITION: *The conversation C <u>associated</u> with transaction T is the outermost conversation that contains the opening of T, but not the opening of T's parent.*

Notice that at run-time the associations between conversations and transactions are easily determined as transactions are opened: an appropriate naming scheme for conversations can provide the nesting information needed.

In the case that multiple processes can take part in the same transaction, as allowed by our generalised model, a second rule must be imposed on the programmers of client components to prevent information smuggling between conversations:

RULE 3: *If a transaction has multiple participants, these must belong to the same enclosing (innermost) conversation.*

In other words, the participants in a conversation can thus use the data protected by TDSCs as part of the data modifiable (but in an "undoable" way) by the conversation, but no process outside the conversation can. This rule can easily be enforced by the interpreter: the transaction must be created when the creator process is already in the conversation (Rule 1), and thereafter, due to the very protection mechanism of conversations, it can only communicate the transaction identifier to other participants in the same conversation.

Last, the fact that the request to commit a transaction is not executed until the commit of its associated conversation implies a third rule for the programmer (not enforceable by the interpreter):

RULE 4: *Two transactions at the same level of nesting can be opened within the same conversation only if they are known not to interfere with each other, otherwise deadlock may ensue.*

To clarify this rule, we point out that:

- in this context, "two transactions interfere" means "the TDSC would want (based on its concurrency control policy) to postpone the commitment of one until the other is done", rather than "their order affects the final state of the TDSC" (often called *dependency* between transactions). Two interfering transactions would cause deadlock, as the TDSC would never commit the first transaction until the conversation were finished (due to our requirement that the actual commit be deferred), and the second transaction could not be processed until the first is committed, but the initiator of the second transaction would not be likely to agree to commit the conversation until the second transaction had been processed. Only a time-out could break the deadlock. Whether two transactions interfere is thus a matter of how the TDSC is implemented internally;
- in general, transactions on a TDSC may deadlock by requesting access to data that are locked by other transactions, if these requests form a cyclic pattern. Rule 4 *does not* address this kind of deadlock, which is the responsibility of the TDSC designer (typically, a TDSC would let deadlock happen and then resolve it by aborting one of the deadlocked transactions);
- Rule 4 only deals with deadlock arising from the forced delay in committing transactions due to the mechanism for co-ordination with conversation-based components. The TDSCs could not detect this special kind of deadlock, except as a long, unexplained delay in committing a transaction. Since, on the other hand, this kind of deadlock may only affect processes within the same conversation, it is reasonable to leave the responsibility of dealing with it to the designer of the conversation (sets of co-operating client processes);
- for the designer of the client processes to be able to apply Rule 4, he/she should be notified of conditions for interference, for instance in the documentation of the TDSC, or by providing a special method for this purpose, or a special return code if the parameters of a *start_transaction* allow the TDSC to forecast possible interference;
- Rule 4 and the considerations above apply in the general case, allowing for the TDSCs to have elaborate concurrency control policies and for the client processes to access TDSC data through multiple concurrent transactions. In practice, most application contexts will be simpler than this. For instance, some DBMSs avoid interference altogether by an optimistic policy: they accept all *start_transaction* calls, but as soon as they perceive that two on-going transactions are interfering, they abort one of them. In this case, the programmers of the clients would never have to worry about Rule 4.

Fig. 3 shows some applications of these rules. Rules 1, 2 and 3 are easy to enforce automatically.

Our model is *asymmetric*. Conversations (being collections of processes) consist of (and could themselves be seen as) active, or client, entities. Transactions are sets of operations performed by these entities on passive TDSCs. The asymmetry of our model is manifest in that a transaction created by a conversation is affected (e.g., by delaying its commit) by the behaviour of the processes in that conversation. Instead, if a process P starts a conversation while engaged in a transaction on a TDSC, the processes in the conversation are not affected by that TDSC, except insofar as they attempt to access the TDSC. We shall see that it makes sense to visualise, in our model, conversations as *creating* or *owning* transactions, but not vice versa. This asymmetry implies, for us, that the organisation of recovery between clients and TDSCs sees the client as the caller of an "idealised fault-tolerant component" [17]. Thus, a TDSC detecting an internal error may either treat it internally, hiding it from the client, or abort the transaction and signal the client. Thus, for instance, we shall specify that a transaction abort, generated by causes internal to the TDSC, should not automatically abort the conversation of the client. The client is assumed to be better equipped for dealing with errors in the server according to the client's own goals.

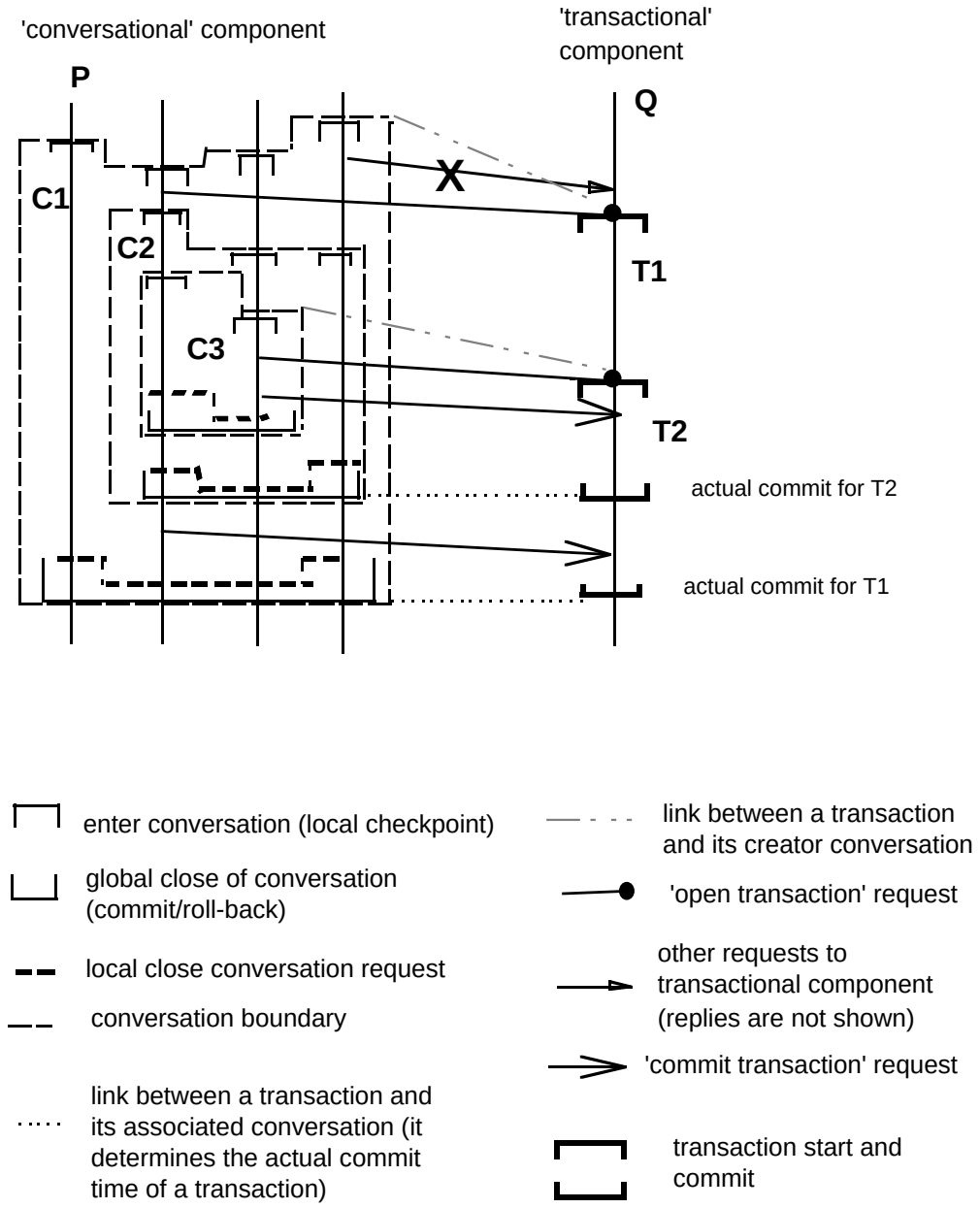

Fig. 3. Scenarios of co-ordinated conversations and transactions

The main implementation problem for our co-ordination scheme is that the closing of a conversation must be implemented as an atomic commit of the conversation and all its associated transactions. There are two solutions: restricting each conversation to work on one TDSC only (more precisely, to be associated with only one transaction[7]), or having TDSCs organised for supporting atomic commit, viz. two-phase commit. We assume in the following discussion that TDSCs export a *prepare_to_commit* operation which can be used for two-phase commit. In addition to the atomic commit on multiple TDSCs, this allows an improvement in efficiency: when the client process issues a *commit_transaction*, the interpreter translates it into an immediate *prepare_to_commit* request to the TDSC. This: 1) allows the TDSC to signal

[7] This would not eliminate the need for both the transaction and the conversation to be one atomic operation. It would simply reduce the duration of the window of vulnerability during which a fault could cause one of the two to abort and the other to commit. Any fault-tolerant protocol must of course be designed to tolerate the fault situations which are expected to arise with non-negligible probability. With multiple TDCSs involved, in a distributed environment, the two-phase commit ability which we describe seems absolutely necessary.

back to the client any failure to commit[8]; 2) notifies the TDSC that the transaction in question will contain no more operations before its actual commit, which is useful information for the TDSC's concurrency control management; 3) possibly simplifies and speeds up the atomic commit phase for multiple transactions required at the time of the *close_conversation* call, so recovering some of the performance loss due to the forced delay of transaction commit operations requested by the client processes.

3.2. Detailed specification of the operations

We specify here in more detail how the interpreter supporting the conversation model can be extended for the co-ordination of recovery between TDSCs and processes in conversations. This implies treating all the operation requests (and the corresponding replies) involved in the interaction between conversational processes and TDSCs. These include those already examined when treating pure conversations in Section 2.2. In addition, the communication primitives regarding calls to TDSCs (remember that we assume communications with TDSCs by RPC) are also intercepted and processed by this extended interpreter. Inter-process communication calls would be intercepted by the conversation interpreter in any case to prevent information smuggling; here this processing is extended with the actions peculiar to the co-ordination of recovery.

A conversation interpreter is usually required to tolerate failures of processors involved in a conversation. This degree of fault tolerance may be implemented in various ways, and we do not assume any specific implementation. The co-ordination mechanism we propose does not pose additional requirements of fault tolerance, and we do not see any special difficulty in extending a fault-tolerant conversation interpreter with the additional functions that we require.

The state information needed by this interpreter is logically described as an acyclic directed graph, that we call *co-ordination graph*, CG, containing two kinds of nodes and three kinds of oriented arcs. There is one CG for each active top-level conversation. The nodes are:

- *conversation nodes*, each labelled with the name of a conversation, and corresponding to currently *active* conversations, i.e., conversations for which a process had issued an *enter_conversation* and the corresponding *close_conversation* or *abort_conversation* has not yet been executed; and
- *transaction nodes*, each labelled with a pair {TDSC name, transaction identifier} (we shall use the notation "Serv:Trans" to denote transaction Trans on TDSC Serv), and representing *active* transactions, i.e. transactions for which a *start_transaction* has been accepted by the TDSC and a *commit_transaction* or *abort_transaction* has not yet been accepted.

An arc in the co-ordination graph may be:

- a *parent-child arc*, connecting nodes of the same type: the child node denotes a conversation (or a transaction) directly nested inside the conversation (or transaction) corresponding to the parent node. Parent-child arcs group nodes of the same kind into trees (*conversation trees* and *transaction trees*);

---

8 If at the time of committing the conversation, a TDSC found itself unable to commit a transaction (for any reason: internal failure, an optimistic concurrency control policy depending on aborting conflicting transactions, ..), the client's conversation would need to abort (roll-back) the current alternate. This may be too drastic a reaction, and a waste of computation. The client's programmer may prefer to be able to specify an alternative action locally to the alternate in execution, without aborting it. This reaction could be purely local to the client process, e.g. attempting a new, identical transaction, or a different, equivalent one, or, if the transaction was for desirable but non-essential data updates, the client could just give up on this update).

If the problem in the TDSC is only detected after invoking the *close_conversation* statement, which does not allow an error return code (either the alternate commits or it aborts), the programmer would not have this option.

- a *creation arc*. A creation arc connecting a conversation node Conv to a transaction node Trans means that Trans is the *first* transaction started on a given TDSC by any process among those currently executing Conv (note that in our asymmetric model a conversation may create a transaction, but a transaction cannot create a conversation). Clearly, transactions may exist without a creation arc;
- an *association arc*, which connects a transaction node to its associated conversation node, found according to our definition in Section 3.1. Notice that not all the transactions have an associated conversation: for instance, if a process in a conversation Conv starts a transaction Trans1 and then a nested transaction Trans2 on the same TDSC, Trans2 has no associated conversation. One can observe that a transaction node has an association arc if and only if it has a creation arc, although the two may not issue from the same conversation node. A transaction nodes that has neither an association arc nor a creation arc is connected in the CG by the parent-child arc from its parent transaction.

Both conversation and transaction nodes have associated descriptors (respectively, *conversation descriptor* and *transaction descriptor*). A conversation descriptor stores the following information: 1) the list of processes involved in the conversation; 2) information for managing commit and rollback (e.g. pointers to copies of checkpointed state, information about which processes are ready to commit, and other temporary variables used by the atomic commit protocol) and, optionally, 3) information about which processes are allowed to enter this conversation (in some models of conversations, the set of processes that may enter a conversation is easily determined at compile time, and the information could be stored into run-time data structures for added protection). A transaction descriptor stores: 1) information about the state of the transaction, which can be "ongoing" or "requested to commit" (after a commit has been requested by an application process, but not yet executed: the interpreter will deny all requests to operate in a transaction whose state is "requested to commit") and 2) information to be used by the multiple atomic commit protocol (for example, information necessary to realise a two-phase commit).

The co-ordination graph is generated, updated or examined by the interpreter when executing the operations involved in the interaction between conversational processes and TDSCs, that is: *enter_conversation*, *close_conversation*, *abort_conversation* (we assume that these three are seen by the application process as procedure calls or system calls), *start_transaction*, *commit_transaction*, *abort_transaction* and all other calls to methods of TDSCs. The high-level specification of these operations, including all the actions performed on the co-ordination graph structure, is given below.

The conversation tree subset of the co-ordination graph contains the information necessary for checking the legality of inter-process communication (when the operations *send*, *receive*, write operations in shared memory, etc., are invoked by a process inside a conversation).

To simplify this description, we now define a few properties and symbols for our entities. Let Proc, Conv and Serv:Trans be a generic process, conversation, and transaction, respectively:

1) *Proc in conversation* means that Proc is a participant in at least one conversation. In terms of the CG, it means that at least one conversation node Conv exists in whose descriptor Proc is registered as a participant;
2) *Conv owns Serv:Trans* is a property on Conv and Serv:Trans: it holds if the transaction Serv:Trans was started inside conversation Conv. In terms of the CG, it means that a creation arc exists directly between nodes Conv and Serv:Trans or between Conv and the nearest ancestor node (according to parent-child arcs) of Serv:Trans for which a creation arc exists;
3) *Serv:Trans committable* means that Serv:Trans can be immediately committed at the moment the *commit_transaction* is invoked, that is, Serv:Trans is not associated with any conversation and no nested transactions are open inside Serv:Trans. In terms of the CG, it means that no creation (or association) arc exists for the node Serv:Trans and the node Serv:Trans is a leaf in its transactional tree;

4) *Proc has right to abort Serv:Trans* is a predicate on Proc and Serv:Trans: it is true if Proc can legally abort Serv:Trans, that is, Proc is a participant in the conversation inside which Serv:Trans was started. This is actually a necessary condition for Proc to know the transaction identifier (capability) Serv:Trans. In terms of the CG, it means that in the CG there is a directed path from a conversation node in whose descriptor Proc is registered as a participant to the node Serv:Trans, starting with a creation arc possibly followed by one or more parent-child arcs;

5) *Proc has right to commit Serv:Trans* is a predicate on Proc and Serv:Trans: it is true if Proc can legally commit Serv:Trans, that is, Proc is a participant in the conversation inside which Serv:Trans was started (that is, the conversation *owning* Serv:Trans) and no nested transactions are open inside Serv:Trans. In terms of the CG, it means that *Proc has right to abort Serv:Trans* **and** Serv:Trans is a leaf in its transaction tree;

6) C_{inner}^{Proc} represents the innermost active conversation in which Proc is a participant. In terms of the CG, it means that C_{inner}^{Proc} is the last node in a path in the conversation tree, which starts from the root and is made up of parent-child arcs connecting conversation nodes where Proc is registered as a participant.

Figure 4 shows a simple example of a co-ordination graph, with the description of a few properties holding in this graph, as shown on the right side of the picture.

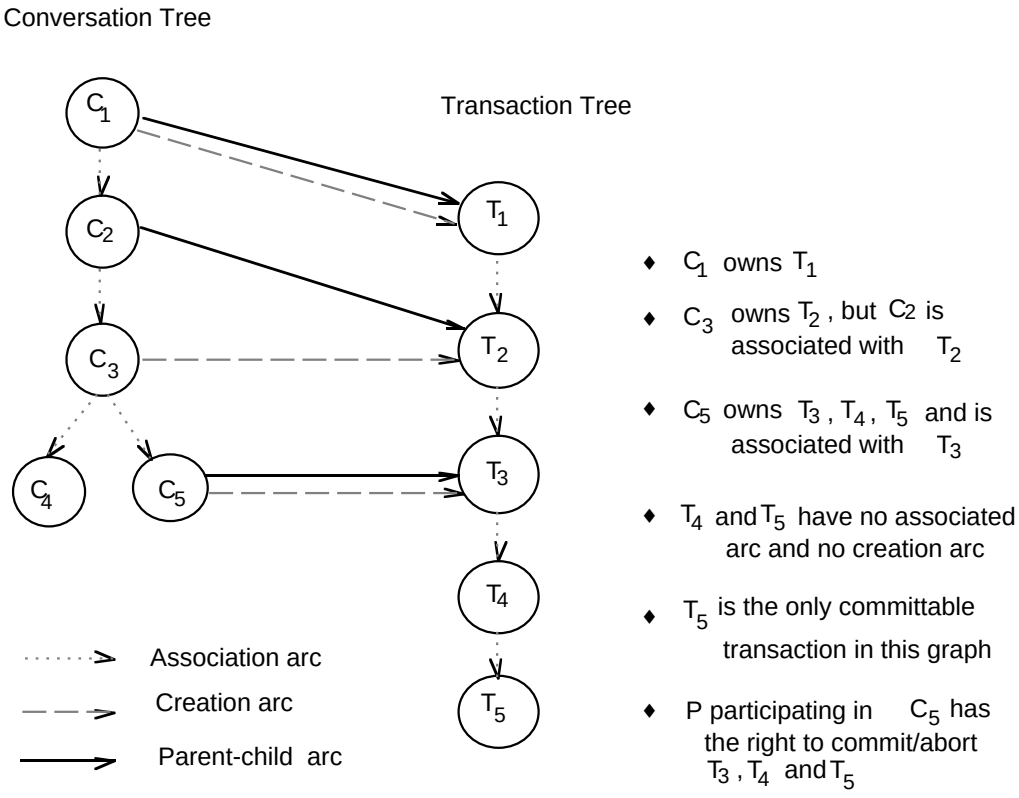

Fig. 4. An example of co-ordination graph

Further notes about the following specification are:

- all operations update the CG in such a way that in every instant it represents the real situation: the conversation and transaction nodes in the CG (with their connection arcs) correspond to the active conversations and transactions in the system;
- for simplicity, we omit the details of those operations required only of the pure conversation-support part of the interpreter (e.g. the checkpointing of processes at the start of conversations), and all redundant checks of the consistency of the CG that could be included in an actual implementation;

- in the pseudo-code below, "Proc" designates the process issuing an operation request;
- comments are bracketed with "/*" and "*/";
- the notation *Serv.<operation>* means the invocation of the *operation* exported by the TDSC called Serv;
- the pseudo-code describing the implementation (by the interpreter) of an operation requested by the application process, say `abort_conversation(in conversation_name=Conv*)`, contains an invocation of internal operations of the interpreter including - in this case - "abort conversation Conv*". This is not a circular definition. The operation `abort_conversation` is seen by the application programmer of the client process, and hides a series of operations by the interpreter, including "abort conversation Conv*"; the latter operation (called in the body of one of our pseudo-code subprograms) is the "standard" conversation abort (undo all memory changes, move the program counter to the beginning of the next alternate or, if alternates are exhausted, to an exception handler). Likewise, when the interpreter intercepts a call issued by a client to a TDSC, the interpreter's operations will include forwarding the `start_transaction` call to that TDSC.

```
enter_conversation(in conversation_name=Conv*)
begin
<checkpointing, legality checks and alternate activation>
if (Conv* is not in CG)
        then
                if not (Proc in conversation)
                    then /* update CG: */
                            create a conversation node labelled Conv* and use it as root of a new
                            conversation tree;
                            register Proc among the participants in Conv*;
                            /* operation ends here*/
                    else /* Proc is in a conversation */ update CG:
                            create a conversation node labelled Conv* and register Proc among the
                            participants in Conv*;
                            create a parent-child arc from C_inner^Proc to Conv*;
                            /* operation ends here*/
        else /* Conv* exists in CG */
                if (Conv* is in CG as child of C_inner^Proc)
                    then /* update CG: */
                    register Proc among the participants in Conv*;
                    /* operation ends here*/
                    else <error: illegal request>
end.
```

```
close_conversation()
begin           /* assume C_inner^Proc=Conv*  */
<record Proc in the set of those participants in Conv* that already called close_conversation; if these are not all
the participants yet, wait for next call to close_conversation without returning control to Proc >
find the set S of transactions associated with Conv* and the set Q of transactions owned by Conv*
if (Q includes transactions in "ongoing" state)
        then /* error: for each transaction there should already have been a request to commit or abort it */
             abort Conv*;
             send abort_transaction requests to TDSCs for all transactions in S and Q[9];
```

[9] We are thus creating a "conservative" interpreter, which requires the application programmer always to close all the transactions he/she has opened. An alternative specification would be that of a more friendly interpreter, which helps (or warns) the application programmer to "clean up" at the end of a conversation. Such an interpreter at this point would atomically commit C* plus all transactions in S or in Q. A caution to be used then is to commit the transactions in the appropriate order (bottom-up along their order of

```
            /* update CG: */ delete nodes in S, Q and Conv*
            /* operation ends here */
        else /* multiple-atomic commit of Conv* plus transactions in S */
            if (all transactions in S and the conversation Conv* are ready to commit)
                then
                    for each Serv involved in S:
                    Serv.commit_transaction(all transactions in S on TDSC Serv);
                    commit conversation Conv*;
                    /* update CG: */ delete nodes in S and node Conv*
                else
                    for each Serv involved in S or in Q:
                    Serv.abort_transaction(all transactions in S and Q on TDSC Serv);
                    abort Conv*;
                    /* update CG: */ delete nodes in S, Q and node Conv*
end.


abort_conversation()
begin
        find set R of transactions associated with or owned by Conv*;
        send abort for each of them and abort Conv*;
        /* update CG: */ delete all nodes in R and node Conv*;
end.


start_transaction(in TDSC=Serv, in transaction_type=...,
        in parent_id= .., out transaction_id,
        [in/out TDSC-specific parameters])
begin
result:=Serv.start_transaction(..., .., transaction_id,[TDSC-specific parameters])
if (result ≠ OK) then <return result>
    /* operation ends here */
/* else: start_transaction was successful */
if not (Proc in conversation ) then
        <return result> /* no co-ordination needed: operation ends here */
/* else: Proc is in a conversation */
/* update CG: */
create a transaction node labelled
    {Serv:<transaction_id returned from Serv>} with state="ongoing";
if <this transaction is not nested inside another transaction on the same TDSC, Serv, created by C_inner^Proc>
    then create a creation arc from C_inner^Proc to this transaction node;
        create appropriate association arc for this transaction node
if (parent_id=Null)
    then use this transaction node as root of a new transaction tree
    else create a parent-child arc from Serv:parent_id to Serv:transaction_id
end.


commit_transaction(in TDSC=Server, in transaction_id=Trans)
begin
if not (Proc in conversation)
    then result:=Serv.commit_transaction(Trans);
        <return result>
        /* Trans was not recorded in any CG, no need to update a CG */
    /* else: Proc is in a conversation; Trans is recorded in its CG */
if not (Proc has right to commit Serv:Trans)
    then <error: insufficient rights>
    /* else: Proc has a right to commit Serv:Trans */
if (Serv:Trans committable)
```

---

nesting): one could argue that application programmers relying on such a powerful aid would easily loose control of the effects of multiple transactions.

```
        then
                result:=Serv.commit_transaction(Trans);
                /* update CG: */ delete node Serv:Trans;
                <return result>
        else
                result:=Serv.prepare_to_commit(Trans);
                if (result=OK)
                    then /* update CG: */ set Serv:Trans state to "commit requested";
                    else <return result> [10]
end.


abort_transaction(in TDSC=Serv, in transaction_id=Trans)
begin
if not (Proc in conversation)
        then result:=Serv.abort_transaction(Trans);
                <return result> /* operation ends here */
if (Serv:Trans is in state "commit requested")
        then <error: illegal request> /* operation ends here */
if not (Proc has right to abort Serv:Trans)
        then <error:insufficient rights>
        else result:=Serv.abort_transaction(Trans);
             if (result="abort executed")
                then
                     /* update CG: */ delete node Serv:Trans and all descendants
                     <return result>
end.
```

Each request to a method of a TDSC is treated as follows:

```
 method_name(in TDSC=Serv, in transaction_id=Trans,
           [in/out parameters specific of the method])
begin
if not (Proc in conversation)
        then
                result:=Serv.method_name(Trans,[parameters specific of the method]);
                <return result> /* operation ends here */
        /* else: Proc is in a conversation */
if (Serv:Trans is in state "ongoing")
        then result:=Serv.method_name(Trans,[parameters specific of the method]);
                <return result> /* operation ends here */
        else <error: illegal request>
end.
```

4. Discussion and Conclusions

We have considered the problem of backward recovery in those software systems that include different subsystems designed according to two different computational models: the "object-action" model and the "process-conversation" model. We argue that this is a realistic problem: a computational model is imposed on designers by factors of application peculiarities and design culture that cannot be arbitrarily changed, and large software systems exist where heterogeneous computational models are needed.

[26] showed how techniques developed for each of the two models, the "object-action" model and the "process-conversation" model, could be implemented in the other model as well, so broadening the possibilities of both design styles. One might then investigate, for instance, extensions to the process-conversation model to allow more complex and open patterns of

---

10 We choose not to automatically abort the conversation. The rationale of this choice is explained at the end of Section 3.2 and in footnote 7.

communication. By contrast, we have assumed a net separation in the use of the two computational models. In our scenario, a subsystem based on processes and conversations would offer full interpreter support for a rather constrained pattern of conversations, while atomic transaction servers would incorporate great flexibility (in terms, e.g., of concurrency control policies), mostly obtained in their application-level design. This seems to us a realistic expectation (though not the *only* realistic scenario) about how such facilities could be designed and used in large, heterogeneous systems.

On this basis, we have described how the backward recovery of computations involving both types of subsystems can be co-ordinated. An important aspect of this work is that we do not assume that the interpreter for a large application system can or should be built as a monolithic, homogeneous entity, and we allow for "transactional data server components", which could run on virtually any computing platform, to be used in conjunction with the specially enhanced platform which we propose for client processes.

An interesting consideration about the two faces of duality is one of granularity. To co-ordinate an atomic transaction changing the states of data items which are all encapsulated in the same TDSC, a "co-ordinating agent" only needs authority on the set of data in that one TDSC. So, designers can implement an atomic transaction capability as a responsibility of entities, possibly at the application level, with visibility and authority limited to individual TDSCs. However, an atomic transaction involving two such TDSCs could only be managed by an entity with authority on both: either a third application-level entity designed as a "transaction co-ordinator", or some facility in the run-time support. Two levels of granularity are thus apparent; no such distinction usually appears in the study of recovery in the process-conversation model, where if two processes can communicate, then the co-ordination of their recovery must be guaranteed. While we have assumed no facility for transactions on multiple TDSCs, the co-ordination mechanism we have specified automatically allows the building of multi-TDSC transactions: all transactions owned by the same conversation are committed together. It is also interesting to notice that our co-ordination mechanism is an extension of the application-transparent co-ordination schemes, studied for the process-conversation model. Three styles of co-ordination can thus coexist in a system like this: i) conversations, where for each multi-process interaction the application programmer must pre-plan the participation of each process, while non-interference and recovery policies are pre-packaged in the interpreter; ii) atomic transactions, where the interaction of processes via data protected by TDCSs can be decided at run-time, is monitored by the TDSC, and non-interference and recovery policies are decided by the programmer of the TDSC; and iii) application-transparent co-ordination between conversations and transactions, where the programmer of a TDSC does not need to pre-plan participation in a conversation, and the interpreter keeps track of which TDSCs are involved in a conversation and co-ordinates their recovery.

We believe that the proposed organisation can successfully hide some of the inherent complexity of planning error recovery in a large system with heterogeneous sub-systems. Some of the complexity is avoided by taking care of difficult tasks (co-ordination) in the interpreter, and some by constraining the application designer to use stereotyped design templates. Risks in this style of system construction exist, e.g. in the possibility of deadlock between clients attempting multiple, conflicting transactions. These interaction problems need to be studied in each case, and of course their existence must limit the complexity of any attempted design. On the other hand, if an application requires such complex interactions, our scheme seems able to constrain them to patterns amenable to analysis.

We have specified in some detail how support for this co-ordination model could be implemented. This specification seems to us to be reasonably general, in that it makes few assumptions on the varieties of conversations and of transactions existing in the system.

To verify the practical feasibility of our scheme, we studied its implementation in the form of extensions to a programming language [28]. The natural candidate for such an extension would be a full-fledged object-oriented language: one could build new libraries of classes with the required properties and the user could use them with very little concern for their internal implementation. However, we found that the field of concurrency in object-oriented languages is still moving too fast for us to choose a convincing candidate. We then studied the possibility

of using the Ada language. We were able to specify an implementation of co-ordinated recovery employing concurrent recovery blocks as the chosen form of conversation, and with atomic-TDSCs implemented as Ada packages. We chose Ada mainly to use its concurrency facilities, and because compatibility with a popular language is an important feature for any proposed design scheme. However, Ada and its run-time support are complex and monolithic, and previous researchers have found it difficult to extend them to support backward recovery schemes. So, we regard our success in defining an implementation strategy for our scheme (albeit in a constrained variant) in Ada as a reasonable indication of feasibility of the scheme in different contexts. As part of this work, we also specified a viable implementation of conversations (more specifically, concurrent recovery blocks) in Ada to a level of detail that could be used in practice (an improved version of this latter work is described in [23]). We had to assume the use of a pre-processor, rather than simple procedural extensions, to allow a natural-looking programmer interface for our added facilities (since we did not think it realistic to postulate changes in the language compiler and its run-time support). A syntactic problem was the impossibility of using the names of packages as arguments to procedure calls; a more basic one is the lack of provisions for building multi-program applications to be combined with some degree of openness. Our use of a "dialect" and a pre-processor may be questioned in terms of the philosophy of Ada, but Ada subsets and dialects, with tools for their support, are already a practical necessity elsewhere (for instance, "safe" or time-predictable subsets). We did not deal with the whole issue of building fault-tolerant applications in Ada. This would involve the problem of which lower-level fault tolerance mechanisms can be included in an Ada (distributed) run-time support and how these should be made visible to the application programmer. In view of these problems with the Ada language, we think Ada9X [5] is a promising candidate for an actual implementation, although considerations of complexity would lead us to prefer a simpler, object-oriented language.

This work has a rather speculative nature: it concerns a complex toolset for supporting fault tolerance in large applications, and therefore in projects that are quite outside the reach of academic research. We only realised a toy example of use (using a restrictive model of conversations) (see [28] or [29]). An actual proof of feasibility and usefulness can only come from a complete implementation and use in reasonably realistic projects. This is a shortcoming common to many such proposals, notably in the field of fault tolerance. We would expect some help in reducing the gap between conception and trial use from the ongoing research [10, 22, 24] on the use of object-oriented methods to facilitate the construction of toolsets for fault-tolerant programming.

ACKNOWLEDGEMENTS

This research was supported in part by the European Commission through the ESPRIT Basic Research Action 6362 "Predictably Dependable Computing Systems" (PDCS2) and the "OLOS" HCM Network (Contract No. CHRX-CT94-0577). Part of the work was done during Dr. Romanovsky's postdoctoral fellowship with the University of Newcastle upon Tyne, UK, sponsored by the Royal Society.

The work described has benefited greatly from discussions with colleagues in these projects.

References

[1] ANCONA, M., CLEMATIS, A., DODERO, G., FERNANDEZ, E.B., and GIANUZZI, V.: 'A system architecture for fault tolerance in concurrent software', *IEEE Computer* , 1990, **23**, (10), pp. 23-31.

[2] ANDERSON, T., and KNIGHT, J.C.: 'A Framework for Software Fault-Tolerance in Real-Time Systems', *IEEE Trans. Soft. Eng.*, 1983, **SE-9**, (3) pp. 355-364.

[3] ANSI/MIL-Std-1815a: 'Reference manual for the Ada programming language', Technical Report American National Standards Institute, 1983.

[4] BARIGAZZI, G., and. STRIGINI, L: 'Application-transparent setting of recovery points', Proc. 13-th IEEE Int. Symposium on Fault-Tolerant Computing, Milano, Italy, 1983.

[5] BARNES, J.: 'Introducing Ada9X. Ada9X Project Report', Technical Report Intermetrics, Inc., 1993.

[6] BIRMAN, K.P., JOSEPH, T.A., RAEUCHLE, T., and ABBADI, A.E.: 'Implementing Fault-Tolerant Distributed Objects', Proc. 4-th IEEE Symposium on Reliability in Distributed Software and Database Systems, 1984.

[7] CLEMATIS, A., and GIANUZZI, V.: 'Structuring conversation in operation/procedure oriented programming languages', *Comput. Lang.*, 1993, **18**, (3) pp. 153-168.

[8] DI GIANDOMENICO, F., and STRIGINI, L.: 'Implementations and extensions of the conversation concept', Proc. 5-th Int. GI/ITG/GMA Conference on Fault-Tolerant Computing Systems - Tests, Diagnosis, Fault Treatment, Nürnberg, Germany, September 1991.

[9] ELMAGARMID, A.K.: 'Database Transaction Models For Advanced Applications' (Morgan Kaufmann Publishers, Inc., 1992).

[10] FABRE, J.C., and RANDELL, B.: 'An object-oriented view of fragmented data processing for fault and intrusion tolerance in distributed systems', Proc. ESORICS 92, Y. Deswarte, G. Eizenberg, J. J. Quisquater, Eds. Lecture Notes in Comp. Sci., vol. 648, Springer-Verlag, New York, 1992.

[11] GOPAL, G., and GRIFFITH, N.D.: 'Software fault tolerance in telecommunication systems', Proc. 4-th ACM SIGOPS European Workshop, Bologna, Italy, 1990.

[12] GRAY, J., and REUTER, A.: 'Transaction Processing: Concepts and Techniques' (Morgan Kaufmann Publishers, Inc., 1994).

[13] GREGORY, S.T., and KNIGHT, J.C.: 'On the provision of backward error recovery in production programming languages', Proc. 19-th IEEE Int. Symposium on Fault-Tolerant Computing, Chicago, Illinois, 1989.

[14] GUDES, E., and FERNANDEZ, E.B.: 'Combined application datafault recovery', Proc. IEEE Int. Conf. on Computer Systems and Software Engineering, Tel Aviv, Israel, 1990.

[15] KIM, K.H.: 'Approaches to mechanization of the conversation scheme based on monitors', *IEEE Trans. Soft. Eng.*, 1982, **8**, (3), pp. 189-197.

[16] KIM, K.H., and YOU, J.H.: 'A highly decentralized implementation model for the Programmer-Transparent Coordination (PTC) scheme for cooperative recovery', Proc. 20-th IEEE Int. Symposium on Fault-Tolerant Computing, Newcastle-upon-Tyne, U.K., 1990.

[17] LEE, P.A., and ANDERSON, T.: 'Fault tolerance: principles and practice' (Springer-Verlag, Wien - New York, 1990).

[18] LYNCH, N.A., MERRIT, M., WEHIL, W.E., and FEKETE, A.: 'Atomic transactions' (Morgan Kaufmann, 1993).

[19] POWELL, D., BONN, G., SEATON, D., VERISSIMO, P., and WAESELYNCK, F.: "The Delta-4 Approach to Dependability in Open Distributed Computing Systems', Proc. 18-th IEEE Int. Symposium on Fault-Tolerant Computing, Tokyo, Japan, 1988.

[20] RANDELL, B.: 'System Structure for Software Fault Tolerance', *IEEE Trans. Soft. Eng.*, 1975, **1**, (2), pp. 220-232.

[21] RANDELL, B., LEE, P.A., and TRELEAVEN, P.C.: 'Reliability Issues in Computing System Design', *Computing Surveys*, 1978, **10**, (2), pp. 123-175.

[22] RANDELL, B., and XU, J.: 'Object-oriented software fault tolerance: framework, reuse and design diversity', in Predictably Dependable Computing Systems, ESPRIT Basic Research Project 6362-PDCS 2, First Year Report, 1993, pp. 165-184.

[23] ROMANOVSKY, A., and STRIGINI, L.: 'Backward error recovery via conversations in Ada', *Software Engineering Journal*, 1995, **10**, (6), pp. 219-232.

[24] RUBIRA-CALSAVARA, C.M.F., and STROUD, R.J.: 'Forward and backward error recovery in C++', *Object Oriented Systems*, 1994, **1**, (1) (1994), pp. 61-85.

[25] SALTZER, J.H., and SCHROEDER, M.D.: 'The protection of information in computer systems', *Proc. IEEE*, 1975, **63**, (9), pp. 1278-1308 .

[26] SHRIVASTAVA, S.K., MANCINI, L.V., and RANDELL, B.: 'The duality of fault-tolerant system structures', *Soft.: Pract. and Exper.*, 1993, **23**, (7), pp. 773-798.

[27] STRIGINI, L., and DI GIANDOMENICO, F.: 'Flexible schemes for application-level fault tolerance', Proc. 10-th IEEE Symposium on Reliable Distributed Systems, Pisa, Italy, 1991.

[28] STRIGINI, L., ROMANOVSKY, A., and DI GIANDOMENICO, F.: 'Recovery in heterogeneous systems', Technical Report 133, PDCS-2 ESPRIT Basic Research project, 1994, http://www.newcastle.research.ec.org/pdcs/trs/abstracts/133.html.

[29] STRIGINI, L., and DI GIANDOMENICO, F., and ROMANOVSKY, A.: 'Coordinated Backward Recovery between Client Processes and Data Servers', to appear in *IEE Proceedings on Software Engineeering*, 1997, **1** (2).

[30] STROM, R.E., and YEMINI, S.: 'Optimistic recovery in distributed systems', *ACM Trans. Comp. Syst.*, 1985, **3**, (3), pp. 204-226.