About Conversations for Concurrent OO Languages

A.Romanovsky

Computing Department, University of Newcastle upon Tyne, NE1 7RU, UK
email: alexander.romanovsky@newcastle.ac.uk

Keywords: software fault-tolerance, software diversity, backward error recovery, concurrent object-oriented languages.

The purposes of this paper are as follows: to discuss the problems which arise while attempting to use a conversation scheme in concurrent object oriented languages (COOLs); to propose approaches to solving these problems and to find the most appropriate ways to use conversations in different COOLs (to discuss the most relevant peculiarities of these languages in these terms); to find the ways in which the characteristics of COOLs can facilitate the use of conversation schemes. We realise that we are not able to find answers to all these questions but we believe that it is time to outline these difficult problems and to discuss possible approaches and directions of future research in this field. To our knowledge, these problems have not been addressed before in the context of COOLs, so it seems to be very important to do this to find appropriate ways of introducing design diversity in the concurrent object oriented programs.

1. Introduction

COOLs [A90, CA93] will clearly be the mainstream languages not only for research but also for application use in the coming decade. That is why providing dependability of application becomes a highly important field of research. This paper deals with the problem of software and hardware fault tolerating in concurrent systems. In his fundamental paper [R75] B.Randell proposed the concept of the *conversation* which was meant to provide joint recovery of several processes exchanging information. Each process that takes part in a conversation must save its state when it enters a conversation. While inside a conversation, a process may only communicate with other processes in the same conversation. If any process fails its acceptance test then every process taking part in the conversation rolls back to the saved state, and uses an alternate algorithm. Processes can asynchronously enter a conversation but all must leave it in the same time after the acceptance test in each of them has been satisfied.

This general idea has been embodied in several ways in the later research (for different languages, different kinds of information exchange, etc.). We will discuss the ways to use conversations in COOLs. There are two main approaches to do this. Firstly, a class can be implemented as a set of "alternates" each of which is implemented as a set of active and co-operating objects [RX93]. We consider the second approach in which a part of computation can be implemented with diversity as a set of alternates. In this case a conversation is a set of co-operating alternates from different concurrent computations.

What do we understand by conversation? Conversation concept is the concept of designing the joint activity and joint recovery in a concurrent program. So it has to be a software engineering facility and not only facility to describe recovery. That is why we believe that conversations should be the units of software structurization.

Generally speaking, all approaches to introduce concurrency into COOLs can be split on the two main parts: *active* concurrency and *passive* concurrency. In the former most attention is paid to providing tools to start, complete, abort and synchronise activities, these activities have got the names and they are controlled by these names. In the latter concurrency mostly is stuck to the object (class) in the form of

synchronization constraints which restrict the behaviour of the object; this is the responsibility of an object designer to implement these constraints in such a way that they guarantee a consistency of the object data while concurrent calls of the object methods occur. It is obvious that these two approaches are complementary and that the features for both of them exist in each particular COOL (sometimes in the rudimentary form). For the purposes of this paper we shall restrict the set of COOLs to be dealt with. We shall consider COOLs in which concurrency is represented in one of the following ways: the active concurrency in the form of process concept (Concurrent C++ [G93], CSP/OCCAM C++) or thread concept (Modula-3 [N91], P-Eiffel) and the passive concurrency is the form of synchronization constraints (Ada9X [B93], Eiffel// [C90], ACT++ [KL90], POOL [A87], DRAGOON).

2. Problems

To use conversations in COOLs we should clarify what conversation participants-interlocutors, alternates, rollback and recovery points, information exchange, acceptance test, nested conversations and conversation entrance are.

We would like to start from the following consideration. We assume that the set of cooperative components which form a conversation is described in a way (we will discuss the ways to do this later). First of all we will consider a conversation as an *atomic action* (using the definition of the latter by [LA90] which stresses the absence of the interactions between the atomic action components and the rest of the system). Provided this the recovery should involve only the set of components participating in an action. But because of the implicit nature of information exchange in most COOLs (through shared objects), information can be spread through the objects-servers in uncontrolled for a conversation way. We do not want to force the conversation developers to enumerate all servers (and their servers, recursively) while describing a conversation and, which is absolutely impossible, to describe all clients of these servers which can operate concurrently with the conversation. Our proposal is to include all conversation's servers in a conversation but in such a way that the executing of this action will be *serializable* for all other clients of these servers (this seems to be the most reasonable way to guarantee consistent concurrent access of independently designed programs to the encapsulated data of an object). It is clear that in this way information smuggling in COOLs will be prevented.

Thus we introduce the concepts of *implicit* and *explicit* conversation participants and require their different behaviour within a conversation. The explicit participants (components which have been designed with diversity and which are included in a conversation description) have to have several alternates as opposed to the implicit ones which serve them. The implicit participants do not have to be involved in all alternates, they have no design diversity and, generally speaking, they can be shared by several conversations or by clients which do not participate in a conversation. It is inevitable that all participants should have to be designed in special ways: they have to have special methods to save and to restore the object state (recovery point tools). And, besides, the explicit participants have several alternates and acceptance tests. The next important question is about providing atomicity for implicit participants. We believe that the solution is in delaying all of them in an alternate till the acceptance test checking [SD91].

How an implicit participant has to be implemented by the object-designer? Within our approach an object-server is to be considered as a small data base with the parallel access and the common transaction model should be applied. The main problem is its clients are anonymous and it is impossible for it to distinguish their calls. One of key features of a transaction is that it is the named set of operations. So, it is clear that a server that can be involved in many conversations cannot be an usual object. All calls of its methods must be marked with a transaction identifier, and for our purposes a conversation name is the only reasonable way to choose this identifier.

State restoration is the base of conversation schemes. It is obvious that saving the state of entire system (of all its objects) is non practical (though correct) solution. That is why peculiarities of COOLs should be taken into account to diminish the amount of saved and restored information. Though in COOLs with thread and process concepts it is possible to introduce the concept of thread or process state this is only partial solution because these threads or processes go through several objects and change their states as well. It would be natural to consider the saving of the object state as a base tool for COOLs of all above-mentioned kinds. In this case the clear specification of implementing state restoration tool can be formulated for designer of each object (class).

The next problem is that of loose and tight design of conversations. Whether a conversation should have a master participant which knows the names of all other participants; whether each participant should know the names of all other participants; whether the entire conversation should be designed by one programmer or each participant can be designed independently by programmers who know only the conversation name and the specifications of (service provided by) some other participants. This opposition often serves as a basis for comparing and discussing the advantages of different approaches to conversation schemes. We believe that both of these extremes are important, have a right to existence, and there are no reasons to consider that one of them is better. In different applications with different approaches to concurrent program design and recovery, each of them can prove necessary. It is a complex problem and a point has to be found to reach a reasonable balance between the particular concurrent language (and its run time system), software engineering conventions for the implementation of the application and the conversation scheme (and its run time system).

Notice that the computational model of COOLs per se relies essentially on the client/server relation between objects. That is why there is a danger of the well-known problem of the capture effect when a server can be involved in a conversation till the acceptance test is satisfied even if there is a need to use it in just one alternate or if it is used in conversation on a condition [GK89]. We believe that this is the inevitable problem of implementation and that it is impossible to provide a "clever" server operating transparently during run-time. It is no wonder. If the server has to be involved in several conversations simultaneously it has to be designed in this way (e.g. by serializing the operation of these conversations). If the server is to be involved on a condition, then a new nested conversation which includes the server has to be started if necessary. It is possible to implement a server in such a way that it will be considered as an explicit conversation participant (and its name will have to enumerated in the list of the explicit participants) but it is also possible to implement it as a server for any conversation (as is proposed in [CG91]).

3. Conversation Concept

Depending on the COOL and the application, there are several approaches to introducing the concept of a conversation. Generally speaking, it can be:

a). The *conversation of processes*. If we have the concept of process in a COOL, it seems to be quite natural to consider these processes as conversation participants. These conversation participants communicate directly by send/receive messages. In these COOLs, all conversation schemes proposed can be used straight away (with all their advantages and disadvantages). The problems exist if these processes want to share an object; that is why the "capsule" mechanism has been introduced in Concurrent C++ [G93] .

b). The *conversation of threads*. If we have the concept of thread it seems to be quite natural to have process-like extension of conversation concept. In this case an approach similar to the named-linked recovery block when the conversation identifier

is used and *ENTER_CONVERSATION(CONV_NAME)* method is explicitly called in each participant. The problem is that since these conversation participants are supposed to communicate, they have to do this implicitly by object sharing. It is not clear what to do with this shared object (evidently, it should be regarded a conversation participant too, then we have to roll it back as well) and what is global test in this approach. Besides, all problems enumerated by [GK89] will have to be coped with.

c). The *conversation* as a set *of interconnected objects*. This approach is implicitly proposed in [SMR93]. There is no specific proposals of implementation but we believe that it is actually a general discussion of the following approach:

d). The *conversation* as an *object* (i.e. resource used by several participants). This way seems the most object-oriented. If conversation participants communicate by object sharing, then this object can be considered as a conversation "flag" (unit, name) which unifies all of them. This approach allows any kinds of entities (processes, objects, threads) to be conversation participants. They have to compete for this resource (for conversation) and only if they take it they can communicate. In a complex case we can accept even sharing this resource: when two conversations use it concurrently, and in this case we can rely only on the serialisability of these two conversations with respect to the shared resource (conversation). This object has to be recoverable: if the acceptance test is not satisfied, the states of it and of other participants have to restored.

The conversation concept has to be the one which expresses the joint activity (joint cooperative work) of several participants. And the most reasonable way for several concurrent activities to do something "together" in COOLs seems to be manipulating an object (objects) jointly. Generally speaking, there are several ways to do this:

1) we can use one shared resource as a "flag" of the conversation;

2) we can use all shared resources (objects) shared by the participants;

3) we can invent a fictitious resource of *CONVERSATION* class which joins in some way all participants and all their servers involved in the conversation.

The first two cases are actually too restrictive because in the second alternate there can be no need to use the same resource(s). But it is likely that including all shared resources from all alternates is inevitable. So, we can introduce class *CONVERSATION* and any object which will be used in a conversation can be created by multiple-inheriting from the class of the initial object and from this class.

4. Implementation

We believe that most of the problems of conversation implementation can be solved in COOLs without updating languages and run time systems. It appears that the orientation of conversation schemes in process oriented systems onto direct updating of languages, on the one hand, and the lack of language facilities (like inheritance, polymorphism), on the other hand, blocking the use of conversation schemes. There are several ways to benefit from using OO features.

First, each conversation participants can inherit methods from ready-made classes which make its structure correct. Another way can be to use a delegation to forward the messages sent to an object to the delegatee - conversation participant with changed behaviour. A third way is to provide the object developer by a class library of different ways to involve an object in a conversation, which determine the object recovery-specific behaviour and the corresponding restoration tools. A fourth way could be

reusing and inheriting concurrency control rules from the initial class to have a behaviour for an object of the derived class which can be involved in a conversation.

Acknowledgements This research has been done during the author's postdoctoral fellowship, sponsored by the Royal Society, UK. The work described has benefited greatly from discussions with colleagues at University of Newcastle-upon-Tyne, UK: B.Randell, J.Xu, R.Stroud, Z.Wu and with V.Vassiliev from Driver-Inter Ltd, St.Petersburg, Russia.

References

[A90] G.Agha. “Concurrent object oriented programming”, *CACM*, v. 33, 9, 1990, pp. 125-141.

[A87] P.America. "POOL-T: A parallel object-oriented language". In M.Tokoro, A.Yonezawa, *Object-Oriented Concurrent Programming*. MIT Press, 1987, pp. 199-220.

[B93] J. Barnes. "Introducing Ada 9X". Ada 9X Project Report. February 1993. Intermetrics, Inc.

[C90] D. Caromel. "Programming Abstractions for Concurrent Programming," Proc. TOOLS Pacific'90 Conference. November, Sydney, Australia. 1990.

[CA93] CACM, “Concurrent Object-Oriented Programming”, Special Issue, *CACM*, v.36, no.9, 1993.

[CG93] A.Clematis, V.Gianuzzi. "Structuring Conversation in Operation/ Procedure Oriented Programming Languages". *Computer Languages*, v.18, 3, 1993, pp. 153-168.

[G93] N.H.Gehani. “Capsules: A Shared Memory Access Mechanism for Concurrent C/C++", *IEEE Trans. Paral. and Distr. Sys.*: v.4, 7, 1993, pp. 795-812.

[GK89] S.T.Gregory, J.C.Knight. "On Provision of Backward Error Recovery in Production Programming Languages", Proc. FTCS-19, 1989, pp.506-511.

[KL90] D. Kafura, K. H. Lee. "ACT++: Building a Concurrent C++ with Actors," *JOOP*, v.3, 1, 1990, pp.25-37.

[LA90] P. A. Lee, T. Anderson. "Fault Tolerance, Principles and Practice, Dependable Computing and Fault-Tolerant Systems". Wien - New York, Springer-Verlag, 1990.

[N91] G. Nelson. "System Programming with Modula-3," Prentice Hall. 1991.

[RX93] B. Randell, J. Xu. "Object-Oriented Software Fault Tolerance: Framework, Reuse and Design Diversity", PDCS2 ESPRIT basic research project, Year Report, 1993.

[R75] B. Randell. "System Structure for Software Fault-Tolerance", *IEEE Trans. Soft. Eng.* SE-1, 2, 1975, pp.220-232.

[SD91] L.Strigini, F.Di Giandomenico. "Flexible Scheme for Application-Level Fault Tolerance", Proc. IEEE 10th Int. Conf. Reliable Distributed Systems, 1991.

[SMR93] S.K.Shrivastava, L.V.Mancini, B.Randell. "The Duality of Fault-Tolerant System Structures". *Software: P&E.*, v.23, 7, July, 1993, pp.773-798.