

Conversations with Fixed and Potential Participants

Alexander Romanovsky and Paul Ezhilchelvan

Department of Computing Science, University of Newcastle upon Tyne

Newcastle upon Tyne, NE1 7RU, UK

1. Introduction

The conversation concept [3] was introduced in order to structure cooperative concurrent systems and to provide their fault tolerance. Several processes take part in a conversation, which they enter logically at the same time (i.e. concurrently), and establish recovery points. While in a conversation, participants send/receive messages to/from other participants only (violation of this rule is called information smuggling [2]). Error recovery is achieved by using a set of diversely designed alternates. When participants leave conversation synchronously after ensuring the acceptance test, they discard the recovery points. If the test fails, all participants restore the recovery points and try the next alternate. The execution of conversations is atomic, indivisible and invisible from outside. Conversations may be nested freely, in which case only the participants of the containing conversation can take part in the nested ones.

Several problems related to the applicability of conversations are discussed in [1]. The authors considered some solutions for resolving these problems but rejected all of them as unworkable. The first problem is related to using servers in conversations. The client/server paradigm assumes that servers may not know the list of all processes they will be serving and that it should be possible for the servers to make a non-deterministic choice of which to serve. This is why the canonical conversation scheme requires all servers to take part in a conversation even if they are not used due to the choice made dynamically inside the conversation. For example, if a conversation participant P1 can choose (inside the conversation) to use either server S1 or server S2 depending on the intermediate conversation results, the traditional approach requires both servers to enter the conversation and to be kept inside until the conversation is over. We agree with the authors of [1] that the nature of servers is different from that of (active) conversation participants; that is why the traditional conversation scheme cannot solve this problem. Our solution, which extends the original conversation scheme, emphasises this difference and relies on treating them in different ways. The second problem is related to using shared objects. Unrestricted access to these objects can cause information to be smuggled out of the conversation. Unlike the authors of [1], we believe that one cannot expect any solution to work that does not serialise the execution of several conversations which use the same shared data because this would allow information smuggling between conversations. It is clear that if units of cooperation whose execution is atomic with respect to each are introduced, other these units, rather than individual processes should share the objects. An approach which relies

on using a guardian (server) process that wraps the shared data effectively reduces this problem to the first one.

2. New Conversation Scheme

The scheme makes it possible for any process participating in a conversation to dynamically invite into the conversation another process which it is going to use. We start by extending the run time system by two special operations. In the new conversation scheme, each conversation declaration includes a conversation name, a list of *fixed participants* and a list of *potential participants*. Fixed participants enter the conversation logically at the same time (one can implement different entry protocols which use the fact that each fixed participant knows the list of all fixed participants). Later on they decide, depending on some internal reasons, which potential participants should join the conversation and at which step of the execution this should be done. After this the required potential participants join the conversation.

The potential participants are involved in the conversations dynamically. They enter conversations only after and if a fixed participant has issued a request for this. This agrees well with the way servers usually work: they wait for clients' requests and choose one of them non-deterministically. Each potential participant knows the complete list of conversations it may take part in, so a set of different execution scenarios (in different conversations) is designed for it. Within this approach, the list of conversation participants is not fixed and different executions can involve different subsets of potential participants.

This approach requires run-time support to provide two operations. Procedure `enter_any(C1, ...)` is executed by a potential participant when it is ready to enter any conversation from the list `C1, ...`; this process has the code which describes what it will be doing in each of these conversations and waits until a conversation invites it to join; the conversation is chosen non-deterministically if several of them request this process (e.g. Ada **select** statement can be used to implement this). Operation `need_server(S1, C1)` is executed by a fixed participant inside conversation `C1` when it requires potential participant `S1` to join `C1`; the list of potential participants in the conversation declaration is used for compiler-time checking of all operations `need_server`.

The run-time support controls the execution of potential participants to involve them in conversations and to guarantee consistency of all conversation participants and properties of the conversation itself. State restoration features should be available for potential participants, so that the support can use them when required. A potential participant can be in just one conversation at a time. The run-time support works in the following way: when it is requested that a potential participant should join the conversation, this process is locked and its recovery point is set; when the conversation acceptance test is ensured, the potential participant is unlocked and its recovery point is discarded; when it is found that the acceptance test of the conversation in which the potential participant is involved cannot

be ensured, the participant's state is restored and this participant is unlocked (which makes it possible for this process to enter another conversation: the next alternate may ask other potential participants to join the conversation).

The rules for using potential participants in nested conversations can be drawn from the general rules of forming conversations [3]. Potential participants of nested conversations should be in the list of potential participants of the containing conversation. After the containing conversation involves a potential participant this participant can be used in any nested conversation. If a nested conversation involves a potential participant, then it should be kept inside the containing conversation until it is completed; this means that if a nested conversation tries several alternates with different potential participants, all of them should be kept inside the containing conversation. The modification of the above way in which the run-time support works in the case of nested conversation follows these rules.

The new conversation scheme adds flexibility and improves system performance. It relaxes the rules of conversations in such a way that, at the application level, all properties of the conversations are preserved, so that application programmers have conventional conversations with traditional properties at their disposal. At the level of conversation abstraction the original and the new scheme are the same: all participants enter logically at the same time and no information smuggling is allowed.

There is obviously "hidden" information smuggling from a conversation when a potential participant joins it. We allow it but keep it under control of the run-time support which manages it by acting properly in all situations which we discuss below. This makes it possible to solve the problems above. Our analysis shows that the world outside the conversation is not affected: the potential participants enter the conversation after the "invitation", leave it either in the same state they were in before entering (if they are not used after rolling back) or in new states which ensured the conversation acceptance test. There is no uncontrolled information smuggling: when the run-time support rolls the conversation back, it guarantees that the entire system is in a consistent state and that all erroneous information is kept inside the conversation. Having said this, we realise that some part of the erroneous information can spread into the run-time support internal data. There are two cases in which controlled spreading of the erroneous information may occur: issuing a wrong request for a potential participant to join and incorrect application-level operations on a potential participant (such that its state can become erroneous). In the first case, either the wrong potential participant joins the conversation: this will be detected by the acceptance test or during the execution, so that the conversation (including the potential participant) is rolled back and this potential participant is freed before the next alternate starts; or there is no such potential participant as is requested (a similar situation happens when it is not ready or does not want to enter the conversation): this is detected by a time-out and the conversation is rolled back because the alternate has a bug. The second case

will be detected by the acceptance test and the conversation rolled back, which includes restoring the states of the potential participant, unlocking and freeing it.

Our rules for involving potential participants into nested conversations guarantee that no information smuggling is allowed and that the execution of conversations (and of nested conversations) appears to be atomic, invisible and indivisible for the surrounding programming environment.

The computational model of potential participants is as it should be for servers: they are ready to be involved in several conversations, wait for an invitation from any of them and execute all application requests within one conversation. Fixed participants, which are more "active" by nature, decide dynamically which potential participants should be in the conversation. Potential participants may wait for an invitation from any conversation. Thus, fixed and potential participants can play the same roles as components in conventional client-server systems.

3. Discussion

The approach proposed solves both problems discussed in [1]. Servers are not kept inside just in case, they are in only if they are used. This approach is flexible because it is easy to add new scenarios for potential participants' behaviour when new conversations are included into the system.

The entry deserter problem (when a process blocks all participants in a conversation by not joining it) [2] can be flexibly solved for potential participants at the application level by using time-outs. If a time-out has expired for a potential participant to join the existing conversation, the fixed participant tries to recover within this conversation. It can either continue the alternate execution, cancel the request for the deserter to join and invite another potential participant if possible, or, which is more general, it can initiate the alternate abort, free all potential participants which have been invited and have joined the conversation, and start the next alternate (which may not use the deserter). This approach allows programmers to complete the conversation successfully in spite of the fact that, generally speaking, the containing conversation has problems, and to continue normal execution (this can help if the reason for the desertion is that the process is busy, slow, or that there are unpredictable delays in the system).

This approach can cause deadlocks when several conversations use the same potential participants. Any approach to deadlock detection and recovery can be used here. A practical solution is to use time-outs as explained above. Note that deadlocks are one of the main reasons for potential participant desertion. Another practical approach to minimise the chance for deadlocks is as follows: fixed participants dynamically form a complete list of required potential participants and issue requests `need_server` for all of them at once.

We believe that the two problems discussed in [1] by no means constitute any inadequacy in the conversation concept. Conversations are intended for cooperative concurrent systems. These problems (solved by our approach) are caused by unsuccessful attempts (in [1]) to apply conversations to competitive concurrent systems (competition here is presented in the form of shared variables and servers). The most general solution is given by Coordinated Atomic (CA) action concept [5] which unifies features of conversations and atomic transactions. The former deals with cooperative participants, the latter guarantee the consistency of the external objects used by the former. It is important to understand that this is not a conversation scheme because active and passive (otherwise referred to as external or transactional objects) participants are treated differently. This scheme assumes a full-pledged transactional support and, generally speaking, does not allow fine-grained application-specific work with external objects explicitly controlled by active participants. The scheme in [4] is in many respect similar to CA actions and can solve the problems above. The authors assume here that there are conversational and transactional supports and show how their operations should be coordinated if conversation participants use transactional objects. In contrast to schemes in [4, 5], our approach is oriented towards using server-processes or shared objects, guarded by these processes, in systems structured using conversations only. Our scheme operates on a different level of abstractions and based on a different understanding of the concept of conversation participants.

There are several conversation schemes which do not assume that the list of participants is known in advance (e.g. in [2]) and in which processes may choose to enter the conversation depending on their will but irrespective of the way the conversation is executed, so they offer a different sort of flexibility and cannot solve problems in [1].

References.

- [1] S.T. Gregory, J.C. Knight, On the provision of backward error recovery in production programming languages, in Proc. FTCS-19, 1989, pp. 506-511.
- [2] K.H. Kim, Approaches to Mechanization of the Conversation Scheme Based on Monitors. IEEE TSE-8 (1982) 189-197.
- [3] B. Randell, System Structure for Software Fault-Tolerance. IEEE TSE-1 (1975) 220-232.
- [4] L. Strigini, F. Di Giandomenico, A. Romanovsky, Coordinated backward recovery between Client Processes and Data Servers. IEE Proc. Softw. Eng. 144, 2 (1997) 134-146.
- [5] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, Z. Wu, Fault tolerance in concurrent object-oriented software through coordinated error recovery, in: Proc FTCS-25, 1995, pp. 499-509.