

# Exception Handling in Open Multithreaded Transactions

Jörg Kienzle

*Software Engineering Laboratory  
Swiss Federal Institute of Technology  
CH - 1015 Lausanne Ecublens  
Switzerland  
email: Joerg.Kienzle@epfl.ch*

**Abstract.** This paper describes a model for providing transaction support for object-oriented concurrent programming languages. In order to achieve seamless integration, the use of the concurrency features provided by the programming language should not be restricted inside a transaction. A transaction model that meets this requirement is presented. Threads inside such a transaction may spawn new threads, but also external threads are allowed to join an ongoing transaction. A blocking commit protocol ensures that no thread leaves the transaction before its outcome has been determined. Exceptions are used to inform all participants in case a transaction aborts.

**Keywords.** Cooperative Concurrency, Competitive Concurrency, Transactions, Open Multithreaded Transactions, Exceptions, Fault Tolerance.

## 1 Introduction

From the very beginning, computer scientists had to deal with concurrency on different levels. Concurrency can be located inside a single processor, such as SIMD processors or super-scalar processors, it can be found in computers with multiprocessor architectures, or it can take its rise from distributed systems, where multiple individual components communicate. Progress in all three fields, especially the recent explosion of distributed systems with the advent of the Internet, shows that the importance of concurrency is constantly increasing.

Modern object-oriented programming languages such as Java [1] or Ada 95 [2] reflect this trend, since they incorporate support for different forms of concurrency. They provide more or less elaborate concurrency features such as active objects and monitors to the application programmer. Distribution of a single program on multiple processing nodes is also often supported. But most of the time, concurrency control and synchronization is reduced to single method or procedure calls. Unfortunately, these mechanisms do not scale well. Complex systems often need more elaborate features that can span multiple operations.

Transactions [3] have been used for many years to provide consistent access to databases. A transaction groups an arbitrary number of simple actions together, making the whole appear indivisible with respect to other concurrent transactions. Using transactions, data updates that involve multiple objects can be executed without worrying about concurrency. If something unexpected happens during the execution of a transaction that prevents the operation to continue, the transaction can be aborted, which will undo all state changes made on behalf of the transaction. The ability of transactions to hide the effects of concurrency and at the same time act as firewalls for failures makes them appropriate building blocks for structuring reliable distributed systems in general.

These observations have lead us to analyze the problems of providing transaction support for concurrent object-oriented programming languages. This paper concentrates on choosing an appropriate transaction model for this purpose. Section 2 gives an overview of concurrency in general, looks at some of the concurrency features found in modern object-oriented programming languages and then presents different sorts of transaction models; section 3 introduces a new transaction model called Open Multithreaded Transactions that suits our needs; section 4 looks at exception handling in Open Multithreaded Transactions and the last section draws some conclusions and presents future work.

## 2 Dealing with Concurrency

According to [4] concurrency comes in two flavors: *competitive* and *cooperative*.

*Competitive concurrency* exists when two or more active components are designed separately, are not aware of each other, but use the same passive components. Programmers (would like to) live in an artificial world in which they do not care about other concurrent activities. They access objects as if they had them at their exclusive disposal. This form of concurrency is used for example in databases.

*Cooperative concurrency* exists when several components cooperate, i.e. do some job together and are aware of this. They can communicate by resource sharing or explicitly, but the important thing is that they have been designed together so that they cooperate to achieve their joint goal and use each other's help and results.

The reasons for encountering concurrency in computing systems are two-fold. In a distributed system, concurrency is caused by the fact that the individual components are active. They evolve independently and sometimes they communicate with each other in order to synchronize or to exchange data. Concurrency is an inherent part of a distributed system and cannot be avoided. But even centralized problems that can be solved sequentially can benefit from concurrency, for example for simulation purposes or to elegantly handle sporadic incoming events, such as events generated by user interfaces or network traffic.

To handle this situation modern operating systems offer two forms of concurrency support. Processes (or *heavyweight* concurrency) are programs that usually execute in separate address spaces on a computer system. They can execute concurrently, and the processing power of the system is assigned to the processes following different scheduling policies and priorities. Threads and semaphores make concurrency possible inside a single process (*lightweight* concurrency). Here again, the processing power of the system is split up among the threads. Processes and threads may take advantage of multi-processor systems.

To support lightweight concurrency, typical object-oriented concurrent programming languages offer the possibility to declare some sort of active object (such as Java [1] threads or Ada [2] tasks), which, once activated, execute their main method concurrently with the main method of the application. Communication between these objects can be *asynchronous* or *synchronous*. Asynchronous communication is achieved by sending messages or by exchanging data through passive objects. To guarantee consistent updates to such shared data structures some sort of monitors are usually provided by the language (for example Java classes with *synchronized* methods or the more elaborate Ada *protected types*, objects that offer mutual exclusive operations and guarded operations with wait-for queues). Some languages also offer synchronous communication mechanisms between active objects (such as the Ada *rendez-vous* construct), where both the caller and the receiver are blocked during data exchange. Often the language also provides a mechanism to assign different priorities to the active objects running inside a program.

Distribution, or heavyweight concurrency, is also more and more supported by object-oriented programming languages (for example Java RMI or the Ada Distributed Systems Annex). The communication mechanism usually provided between the distributed processes is synchronous or asynchronous remote procedure call or remote method invocation.

Complex systems often need more elaborate concurrency features than the ones mentioned in the two previous paragraphs. Atomic units that encapsulate several operations, making the whole appear indivisible with respect to other atomic units, have been widely used to simplify reasoning about concurrency in large-scale systems. This is even more true when considering adding support for fault tolerance.

Two different forms of atomic units have evolved: *transactions* and their derivatives which emphasize competitive concurrency, and *conversations* and their derivatives which emphasize cooperative concurrency. The authors of [5] name the former *Object and Action* model and the latter *Process and Conversation* model. They claim that the two models are duals of each other, and provide a mapping from one model to the other. Using this mapping, they show that mechanisms used in one model can have interesting counterparts in the other model.

The next subsections will briefly introduce these two models, and then present how they evolved to deal with the “other” aspect of concurrency.

## 2.1 Competitive Concurrency

*Transactions* are the main approach to structuring competitive systems. The notion of transaction has first been introduced in database systems in order to correctly handle concurrent updates and to provide fault tolerance with respect to hardware failures [3]. A transaction groups an arbitrary number of simple actions together, making the whole appear indivisible as far as the application is concerned and with respect to other concurrent transactions. At any time during the execution of the transaction it can *abort*, which means that the state of the system will be restored to the state at the beginning of the transaction (also called *roll back*). Once a transaction has completed successfully (is *committed*), the effects become permanent. The properties of transactions are referred to as the ACID properties: *Atomicity*, *Consistency*, *Isolation* and *Durability*.

The basic transaction model, also called *flat transactions*, has been extended in order to provide more flexible support for concurrency and recovery. *Nested transactions* [6] allow transactions to start *subtransactions*, thus creating a tree of transactions. A subtransaction can either commit or roll back; its commit will not take effect (will not be visible to the outside world), though, unless the parent transaction commits. The advantage of nested transactions is that they can abort independently without causing the abortion of the whole transaction. Only the subtransaction and all its child transactions are rolled back. Since updates of a nested transaction to transactional objects are isolated with respect to other sibling transactions, siblings can be executed concurrently.

To cope with the problems of long-running transaction as they are found in CAD/CAM, VLSI design and software development applications several additional models have been proposed. They all strive to increase concurrency between transactions, mostly by relaxing the serializability criterion such as it is done in the *Cooperative Transaction* model described in [7] or in the *SAGAS* model found in [8].

Another possibility to increase concurrency between transaction is to allow certain transactions to view the results of other transactions before they commit / abort. Of course, this creates a certain dependency between these transactions.

The *Recoverable Communicating Actions* model described in [9] for instance allows a transaction (the sender) to communicate with another transaction (the receiver), by sending results of operations, inducing an abort dependency of the receiver on the sender. If the sender aborts, then the receiver must abort as a result of the dependency. Likewise, in order for the receiver transaction to commit, the sender transaction must commit, too.

The *Split Transaction* model [10] allows a user to dynamically split a transaction into two or more smaller transactions and commit / abort them independently. At the time of the split, the operations invoked so far by the transaction must be divided and assigned to the new transactions, making each responsible for some subset of the operations. This allows to make partial results visible to other transactions and hence has the potential to increase concurrency. In the *Joint Transaction* model [10], it is possible for one transaction, instead of committing or aborting, to join another transaction, releasing its objects to the joint transaction. The effects of the joining transaction are committed only if the joint transaction commits.

## 2.2 Cooperative Concurrency

### Conversations

The concept of a *conversation* has been introduced in [11] in 1975. It allows several processes to perform an action together in an atomic way. Processes enter a conversation asynchronously; a recovery point is established in each of them. They freely exchange information within the conversation but cannot communicate with any outside process (violations of this rule are called *information smuggling*). When all processes participating in the conversation have come to the end of the conversation, their acceptance tests are to be checked. If all tests have been satisfied, the processes leave the conversation. Otherwise, they restore their states from the recovery points and may try and execute a different *alternate*. The occurrence of an error in a process inside a conversation requires the rollback of all (and only) the processes in the conversation to the checkpoint established upon entering the conversation. Conversations may be nested freely.

### Atomic Actions

Later on, conversations have been enhanced with additional forward error recovery and exception resolution [12], resulting in so-called *atomic actions* [4]. This means that an exception that has been raised in a process that is part of an atomic action will be propagated to all other participating processes of that action. Since multiple exceptions can be raised concurrently, an exception resolution mechanism must be provided in order to determine the final exception that will be propagated to all participants.

## 2.3 Combining Cooperative and Competitive Concurrency

Recently, some transaction models have evolved to allow cooperative concurrency inside a transaction. The C++ extension Arjuna [13] for instance allows different threads to work on behalf of the same transaction, but without really defining a clear model. One thread starts a transaction, and may communicate its identity to other threads. These can then also perform work on behalf of the same transaction. Finally, one of the threads will abort or commit the transaction. This technique is very general as it leaves complete freedom to the transaction programmer, but from our point of view it is exactly this freedom that can be dangerous. It takes very careful programming to still guarantee the ACID properties of such transactions. Threads can decide to leave the transaction and communicate some of its results to the outside world before the outcome of the transaction has been determined (*information smuggling*). Transactional objects might not be aware of the intra-transaction concurrency and hence won't guarantee consistent execution of concurrent operations. The same sort of transactions are also described in the CORBA transaction service specification [14].

[15] describes a model called *Multithreaded Transactions*. The same model is also used in *Cooperative Transactional Object Groups* presented in [16, 17]. A multithreaded transaction has precise semantics: Once a thread (the *main* thread) has started a transaction, it can fork new threads that work on behalf of it to take advantage of concurrency. Before the main thread can commit or abort the transaction, these forked threads must all run to completion.

The atomic action concept has also been extended. The *Coordinated Atomic Action* model [18] allows the participants of an action, which want to be isolated from the outside world, to also access external objects. Updates to these objects have transactional semantics with respect to other concurrently running coordinated atomic actions.

In both the multithreaded transaction and the coordinated atomic action models, cooperation is supported for participants in the inside of an atomic unit, and competitive concurrency is supported between different atomic units that run in parallel. Coordination is supported for a known set of participants, and all other concurrency is considered to be of competitive nature.

## 3 Open Multithreaded Transactions

As we have seen in the previous section, the classic transaction model has been extended in many ways to satisfy the requirements of different application domains. When introducing transactions into concurrent programming languages, it is important to support concurrency inside a transaction in a natural way. In particular, the use of concurrency constructs provided by the language should not be restricted inside a transaction, if possible.

The multithreaded transaction model comes closest to what we need. One drawback however is that the only way of having concurrency inside a transaction is to start a transaction in one thread, and then spawn new threads

inside the transaction. Also, these spawned threads must run to completion before the transaction can be committed. The multithreaded transaction model therefore cannot be used to deal with the situation where several already existing threads come together and decide to perform a job on a set of objects in a transactional manner.

Unfortunately, creation and deletion of threads can be a complicated task and often programmers try to avoid it whenever possible. Process control and especially real-time systems tend to use a static number of threads, created once and for all during the initialization of the system.

Due to the dynamic nature of problems it is sometimes not even foreseeable at the beginning of a transaction, how many threads will participate. An example of such a system is an online auction system, where the individual auctions are structured using transactions. There will always be a vendor thread and maybe some accounting system that will participate in such a transaction, but the number of bidder threads is not known in advance. A bidder might also want to join an already ongoing auction. This dynamic behavior also excludes the use of the coordinated atomic action model, that works with a fixed number of participants.

In order to overcome these limitations, a new transaction model named *Open Multithreaded Transactions* has been defined. Lightweight and heavyweight concurrency are treated orthogonally with respect to transactions. The model allows threads to be created, to run to completion, or to join an ongoing transaction at any time.

The following description describes the rules for open multithreaded transactions. Threads working on behalf of an open multithreaded transaction are referred to as *participants*:

### Starting Open Multithreaded Transactions

- Any thread can start an open multithreaded transaction. This thread will be the first participant of the transaction.
- Open multithreaded transactions can be *nested*. A participant thread that starts a new open multithreaded transaction will start a nested transaction. Sibling transactions can execute concurrently.

### Joining Open Multithreaded Transactions

- Zero or more threads can join an open multithreaded transaction, thus becoming participants of the transaction, if and only if they do not yet participate in any other transaction. To join a nested transaction, a thread must first join all parent transactions. A thread can only participate in one sibling transaction at a time.
- Threads spawned by participants will automatically become participants of all the transactions in which the spawning thread participates.

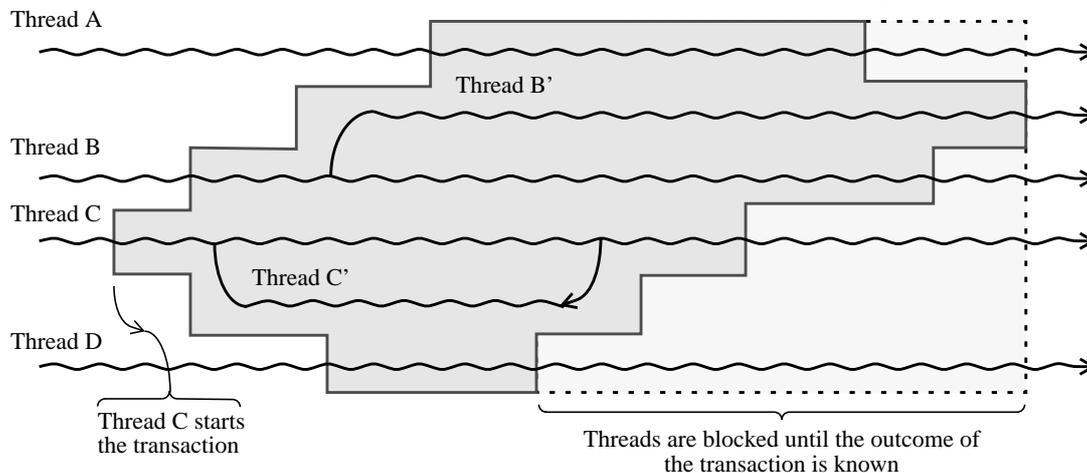
### Concurrency Control in Open Multithreaded Transactions

- Accesses to transactional objects are isolated with respect to other open multithreaded transactions. The only visible information that is available to the outside world is the existence of the transaction.
- Classic consistency techniques (i.e. monitors) are used to guarantee consistent updating of the objects state by participants.

### Ending an Open Multithreaded Transactions

- All participants must vote on the outcome of the transaction. Possible votes are *commit* or *abort*.
- The open multithreaded transaction commits if and only if all participants voted *commit*. If any of the participants votes *abort*, the transaction aborts.
- Participants are not allowed to leave the transaction (they are blocked) until the outcome of the transaction has been determined. This means in particular that all participants of an open multithreaded transaction that commits exit synchronously. Only then, the changes made to transactional objects are made visible to the outside world.

The following figure depicts a non-nested open multithreaded transaction with 6 participants:



**Figure 1:** Open Multithreaded Transactions

Thread C starts the transaction, thread A, B and D join it later on. During the transaction, thread C forks a new thread, thread C'. This thread performs some work on behalf of the transaction and then terminates after having given his vote on the outcome of the transaction. Thread B also forks a thread, thread B', but this thread will continue to run after the transaction. In this example, all threads vote *commit*, and are therefore blocked until the last participant, here Thread B', has finished its work and given its vote.

An obvious problem that has not been discussed yet are *deserters*, i.e. threads participating in an open multithreaded transaction that suddenly disappear without voting on the outcome of the transaction. This can happen if a thread is explicitly killed<sup>1</sup>, or when the process of a participant thread dies incidentally.

Different solutions are possible:

- Participant threads that terminate without voting on the outcome of the transaction cause the transaction to abort. The advantage in this scenario is that if the thread terminated due to some unknown failure, the application state remains consistent, since it is restored to the state that was valid before the beginning of the transaction.
- The disappearance of participant threads causes the transaction to abort, except for participant threads that have been spawned by some other participant thread. This rule also makes sense, since such participant threads can be considered auxiliary threads of the parent participant thread, and it is the parent that will vote on the outcome of the transaction.

## 4 Exceptions in Open Multithreaded Transactions

We have faced several problems while extending the open multithreaded transaction model with exception handling. Coordinated atomic actions provide coordinated exception handling. A set of internal and external exceptions is associated here with each action. Each participant has a set of handlers for some or all of the internal exceptions. When an exception is raised in any participant, the appropriate handlers are executed in all participants. This allows the participants to cooperate not only during normal execution of the action, but also when handling abnormal situations. A coordinated atomic action can either terminate normally, or by signalling an external exception.

In the cooperative transactional object groups model [17], exceptions raised concurrently in threads participating in a multithreaded transaction are first resolved locally, and then a distributed resolution algorithm is initiated. If the resolved exception can not be handled and crosses the transaction boundary, then the transaction is aborted.

From our point of view, the fact that unhandled exceptions crossing the transaction boundary result in aborting the transaction is a good idea. Transactions are atomic units of system structuring that move the system from a consistent state to some other consistent state if the transaction commits. Otherwise the state remains unchanged. The exception mechanism is typically used to signal unforeseen events such as situations in which a desired operation could not be performed as requested. Exceptions are events that must be handled in order to guarantee correct results. If such a situation is not handled, the application data might be left in an inconsistent state. Aborting the transaction and thus restoring the application state to the state that was valid before the beginning of the transaction will guarantee correct behavior. If this happens, it is of course important that all participants are notified of the abortion of the transaction.

On the other hand it is not clear if exception resolution and coordinated exception handling among participants is necessary, or even feasible due to multiple reasons. Firstly, concurrency in open multithreaded transactions is dynamic, that is participants can join transactions at any time. Due to this asynchronous entering, a participant can not really rely on the presence of some other participant, especially for exception handling. The thread that starts the transaction might cause an exception to be raised even before any other participant has a chance to join the transaction. Secondly, an exception declared in one participant might have no meaning in some other participant, since they might have been designed separately, cooperating only through shared data objects. Thirdly, concurrent (and potentially distributed) exception resolution can be very time consuming and should therefore be avoided if it is not absolutely necessary.

These reasons led us to define the following rules for exception handling in open multithreaded transactions:

### Exception Handling in Open Multithreaded Transactions

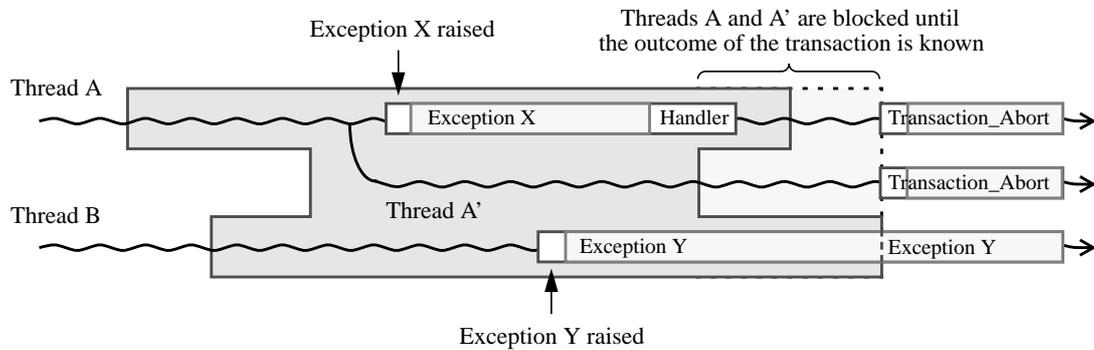
- Each participant has a possibly distinct set of associated internal exceptions that it can handle. The set of interface exceptions of an open multithreaded transaction is also distinct for each participant. It is comprised of the exceptions that can be raised inside the participant and are not handled there augmented by the pre-defined interface exception `Transaction_Abort`.
- Exceptions raised in a participant of an open multithreaded transaction are not propagated to other participants. They can be handled locally by this thread.
- Unhandled exceptions of a participant that cross the transaction boundary will cause the transaction to abort. The exception will be propagated to the calling environment of the participant. Concurrent exceptions crossing the transaction boundary are allowed. Participant threads that did not raise an exception will be informed of

---

1. Most concurrent programming languages offer such features, e.g. the Java `stop` method or the `abort` statement of Ada.

the abort of the transaction as soon as possible<sup>1</sup>. They will propagate the exception `Transaction_Abort` to the calling environment.

Figure 2 depicts an open multithreaded transaction with three participant threads. Thread A starts the transaction, Thread B joins it, and at some point Thread A spawns Thread A'. Thread A' performs some work, and votes *commit*; it is blocked until the outcome of the transaction is known. Thread A generates an exception while performing its work, but the exception is handled locally. It therefore does not affect the outcome of the transaction; Thread A also votes *commit*. Unfortunately Thread B has generated an exception, exception Y, that it could not handle locally. It crosses the transaction boundary, and therefore causes the transaction to abort. The exception Y is propagated to the calling environment of Thread B; in all other participants the exception `Transaction_Abort` is generated,



**Figure 2:** Exceptions in Open Multithreaded Transactions

## 5 Conclusions and Future Work

This paper has highlighted the needs of providing transaction support in concurrent programming languages. A new transaction model, *Open Multithreaded Transactions*, has been defined. It supports concurrency in a natural way, for it does not restrict the use of concurrency constructs provided by typical concurrent languages inside the transaction. Threads inside a transaction can spawn new threads, but also external threads can join an ongoing transaction. A blocking commit protocol ensures that no thread leaves the transaction before its outcome has been determined. Unhandled exceptions that cross the transaction boundary cause the transaction to be aborted. Exceptions are also used to inform all participants in case a transaction aborts.

We are currently investigating the different ways of integrating a transaction support based on the open multithreaded transaction model into object-oriented concurrent programming languages. An implementation of the transaction support for the Ada 95 programming language based on the design of TransLib [17] is in development.

## 6 References

- [1] Gosling, J.; Joy, B.; Steele, G. L.: *The Java Language Specification*. The Java Series, Addison Wesley, Reading, MA, USA, 1996.
- [2] ISO: *International Standard ISO/IEC 8652:1995(E): Ada Reference Manual*, Lecture Notes in Computer Science **1246**, Springer Verlag, 1997; ISO, 1995.
- [3] Gray, J.; Reuter, A.: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, California, 1993.
- [4] Lee, P. A.; Anderson, T.: "Fault Tolerance - Principles and Practice". In *Dependable Computing and Fault-Tolerant Systems*, volume 3, Springer Verlag, 2nd ed., 1990.
- [5] Shrivastava, S. K.; Mancini, L. V.; Randell, B.: "The Duality of Fault-tolerant System Structures". *Software-Practice and Experience* **23**(7), pp. 773 – 798, July 1993.
- [6] Moss, J. E. B.: *Nested Transactions, An Approach to Reliable Computing*. Ph.D. Thesis, MIT, Cambridge, April 1981.
- [7] Korth, H.; Kim, W.; Bancilhon, F.: "On Long-Duration CAD Transactions". *Information Sciences* **46**(1-2), pp. 73 – 107, October - November 1988.

1. Depending on the concurrency features of the language and on performance considerations it might be preferable to do this at once, on the next invocation of an operation of a transactional object, or at the very end of the transaction, once the participant votes on its outcome.

- [8] Garcia-Molina, H.; Salem, K.: "SAGAS". In Dayal, U.; Traiger, I. (Eds.), *Proceedings of the SIGMod 1987 Annual Conference*, pp. 249 – 259, San Francisco, Ca, May 1987, ACM, ACM Press.
- [9] Vinter, S.; Ramamritham, K.; Stemple, D.: "Recoverable Actions in Gutenberg". In *The 6th International Conference on Distributed Computing Systems*, pp. 242 – 249, Los Angeles, Ca., USA, May 1986, IEEE Computer Society Press.
- [10] Pu, C.; Kaiser, G. E.; Hutchinson, N. C.: "Split-Transactions for Open-Ended Activities". In Bancilhon, F.; DeWitt, D. J. (Eds.), *Fourteenth International Conference on Very Large Data Bases*, pp. 26 – 37, Los Angeles, California, 1988, Morgan Kaufmann.
- [11] Randell, B.: "System structure for software fault tolerance". *IEEE Transactions on Software Engineering* *1*(2), pp. 220 – 232, 1975.
- [12] Campbell, R. H.; Randell, B.: "Error Recovery in Asynchronous Systems". *IEEE Transactions on Software Engineering (SE)* *SE-12*(8), August 1986.
- [13] Parrington, G. D.; Shrivastava, S. K.; Wheeler, S. M.; Little, M. C.: "The Design and Implementation of Arjuna". In USENIX (Ed.), *Computing Systems, Summer, 1995.*, volume 8, pp. 255 – 308, Berkeley, CA, USA, Summer 1995, USENIX.
- [14] Object Management Group, Inc.: *Object Transaction Service*, August 1994.
- [15] Haines, N.; Kindred, D.; Morrisett, J. G.; Nettles, S. M.; Wing, J. M.: "Composing First-Class Transactions". *ACM Transactions on Programming Languages and Systems* *16*(6), pp. 1719 – 1736, Nov 1994.
- [16] Patino-Martinez, M.; Jimenez-Peris, R.; Arevalo, S.: "Integrating Groups and Transactions: A Fault-Tolerant Extension of Ada". In *Reliable Software Technologies - Ada-Europe'98*, volume 1411 of *Lecture Notes in Computer Science*, pp. 78 – 89, 1998.
- [17] Jimenez-Peris, R.; Patino-Martinez, M.; Arevalo, S.: "TransLib: An Ada 95 Object-Oriented Framework for Building Transactional Applications". *Computer Systems: Science & Engineering Journal* *15*(1), 2000.
- [18] Xu, J.; Randell, B.; Romanovsky, A.; Rubira, C. M. F.; Stroud, R. J.; Wu, Z.: "Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery". In *FTCS-25: 25th International Symposium on Fault Tolerant Computing*, pp. 499 – 509, Pasadena, California, 1995.