Open Multithreaded Transactions: Keeping Threads and Exceptions under Control

Jörg Kienzle,
Swiss Federal Institute of Technology Lausanne
CH - 1015 Lausanne EPFL
Switzerland
joerg.kienzle@epfl.ch

Alexander Romanovsky,
University of Newcastle upon Tyne
Newcastle upon Tyne
NE1 7RU
United Kingdom
alexander.romanovsky@ncl.ac.uk

Alfred Strohmeier
Swiss Federal Institute of Technology Lausanne
CH - 1015 Lausanne EPFL
Switzerland
alfred.strohmeier@epfl.ch

Abstract

Although transactional models have proved to be very useful for numerous applications, the development of new models to reflect the ever-increasing complexity and diversity of modern applications is a very active area of research. Analysis of the existing models of multithreaded transactions shows that they either give too much freedom to threads and do not control their participation in transactions, or unnecessarily restrict the computational model by assuming that only one thread can enter a transaction. Another important issue, which many models do not address properly, is providing adequate exception handling features. In this paper a new model of multithreaded transactions is proposed. Its detailed description is given, including rules of thread behaviour when transactions start, commit and abort, and rules of exception raising, propagation and handling. This model is supported by enhanced error detection techniques to allow for earlier error detection and for localised recovery. General approaches to implementing transaction support are discussed and a detailed description of an Ada implementation is given. Special attention is paid to outlining typical applications for which this model is suitable and to comparing it with several known approaches (Coordinated Atomic actions, CORBA, and Argus).

1. Transactions

The notion of (atomic) transaction was first introduced in database systems in order to correctly handle concurrent data updates and to provide fault tolerance with respect to hardware failures. A transaction groups a number of simple operations together, making the whole appear indivisible with respect to other concurrent transactions. Under the control of transactions, updates involving multiple objects can be executed as if they happened in a sequential environment. If something happens that prevents normal completion of the transaction, then it can be aborted, i.e. all state changes made within the transaction are undone. Once a transaction has completed successfully, i.e. is committed, the effects become permanent and visible to the outside. The transaction scheme relies on three standard operations: start, abort and commit, which mark the boundaries of a transaction. This approach focuses on preserving and guaranteeing important properties of data, objects, or resources affected by the operations performed inside the transaction. In database systems, the focus is in general on "pure" data, whereas in middleware for distributed systems, a resource is usually data together with related operations, often called a transactional object. Transaction takes care of atomicity, consistency, isolation and durability, called the ACID properties [3]. The atomicity property, understood here as indivisibility of transaction execution with respect to errors (all-or-nothing semantics), is vital for proper system structuring and for providing fault tolerance. Consistency means that the execution of any transaction on its own is a correct transformation of the data and does not violate their integrity. Isolation plays a major role in providing inter-transaction concurrency: when it holds, the designer of a transaction that accesses some piece of data does not have to know about other transactions accessing the same piece concurrently. It is guaranteed that, even when several transactions are executed simultaneously, they do not interfere with each other, and the recovery of any of them is separated from the execution of the others. Durability is understood as the ability of data to survive any assumed hardware faults that happen after the transaction has been successfully completed. The ability of transactions to hide the effects of concurrency and at the same time act as firewalls for errors makes them appropriate building blocks for structuring reliable distributed systems.

The basic transaction model, also called flat transactions, was extended in order to provide more flexible support for concurrency and recovery and to allow for recursive system design. The nested transaction model [12] allows transactions to start subtransactions, thus creating a tree of transactions. A subtransaction can either commit or rollback; however, its commit does not take effect, i.e. is not visible to the outside world, unless the parent transaction commits. The advantage of subtransactions is that they can abort independently without causing the abortion of the "entire" top-level transaction. Since updates of transactional objects by subtransactions are isolated from updates made by other sibling transactions, siblings can be executed concurrently.

Recently the concept of multithreaded transactions (MTT) has been developed to allow several active components, i.e. threads or processes, to take part in the same transaction and to operate together on the same set of data. One thread starts a transaction, then others learn its identity, which they can use to access data that is under the control of the transaction. Finally, one of the threads can abort or commit the transaction. This technique is very general as it leaves thread control to the programmers. Unfortunately this can be dangerous. For example, a thread can decide to leave the transaction and perform some other operations before the outcome of the transaction has been determined, or, a thread can abort the transaction without notifying the other threads. Another problem of MTT is that transactional objects might not be aware of intra-transaction concurrency, and therefore consistent execution of concurrent operations is not guaranteed. Typical examples of systems using the MTT model are the CORBA Object Transaction Service and Arjuna [13].

2. Exception handling

Although transactions are able to tolerate hardware faults, they cannot deal with many other types of faults, e.g. environmental faults, software defects, faults of the underlying software, etc. The most general mechanism for dealing with such abnormalities at the application level is exception handling [2].

Exception handling features allow programmers to *declare* exceptions and provide them with the ability to treat a program block as an *exception context*. *Handlers* are associated with such a context, so that when an exception is raised in this context the execution stops and the handler corresponding to the exception is searched among the handlers. Note that there are some models where an exception can be propagated straight to the outside of the context. In our opinion, a vital feature of any exception handling mechanism is to differentiate between *internal exceptions* to be handled inside the context and *external exceptions* which are propagated to the outside of the context. These two kinds of exceptions are not clearly separated in most programming languages although they are intended for very different purposes. To achieve this separation, the following programming language features are needed:

- Contexts are associated with programming units;
- Programming units have interfaces, where exceptions can be declared;
- Contexts can be nested.

The majority of existing exception handling mechanisms use dynamic exception context nesting. In this case the execution of the context can be completed either successfully or by interface exception propagation. In the latter case the propagated exception is treated as an internal exception raised in the *containing context*. The simplest example of this approach are nested procedure calls. Actually this is the dominating approach very suitable for the client/server and the remote procedure call paradigms and used in the majority of systems.

Exception handling features should be in accordance with the design and programming abstractions. It is clearly incorrect and error-prone to apply sequential exception handling for concurrent systems. Following the previous discussion, the best approach is to merge exception handling and transactions by considering transactions as exception contexts. This approach facilitates system design and makes it possible to tolerate additional kinds of faults. Another important benefit is system state consistency in the presence of exceptions. Existing exception handling mechanisms usually leave the responsibility for consistency to the application programmer, which is error-prone, as practical experience shows.

Many researchers have realised all these benefits. The object-based language Argus [10] is a very interesting example of MTT enriched with powerful exception handling: applications are composed out of guardians, each of which provides an interface consisting of callable procedures, called handlers. A handler can fork concurrent threads, which are joined when the handler is completed. The execution of a handler forms an atomic transaction, and nested handler calls form nested transactions. Argus provides a very powerful extension of sequential exception handling: handlers can have exceptions declared in their interface, which are propagated to a single-threaded caller when they are signalled by a thread inside the transaction. Any thread may decide to signal an exception with or without transaction abort. The Argus model proved to be very influential: several systems have been developed which rely on similar computational models, including Vinari/ML [4] and Transactional Drago [5].

Let us now analyse existing exception handling models in transactional systems. Classical transactional systems usually do not incorporate exception handling and use return error codes instead. There are many problems with this approach. Firstly, using return codes has been always been considered bad practice and a poor substitute for a proper exception handling mechanism. Secondly, even if the host language has exception handling features, they are not integrated with the transaction mechanisms.

The CORBA Object Transaction Service is an example of this situation. CORBA OTS is based on a sophisticated MTT model but it provides only sequential exception handling, i.e. that of the host languages C++ or Java. An exception raised in an MTT can cross the MTT border unnoticed, because an MTT is not an exception context, and each MTT participant deals with its exceptions in isolation. Actually, the border of transaction is not clearly defined as threads taking part in a transaction are not coordinated in any way. Therefore, thread coordination in both normal and abnormal situations is the application programmers' responsibility.

It is symptomatic that the designers of Enterprise JavaBeans have made some efforts in combining exception handling with transactions. Basically the MTT model is similar to the CORBA model but the situation is different with respect to exception handling. If an exception is signalled by a transactional object to the thread participating in the transaction, the exception can affect the execution of the whole transaction: it can prepare the transaction for abort or commit, signal an exception, abort the transaction, etc. However, an MTT is not an exception context and coordination of multiple participants is left to the application programmer.

Argus includes exception handling but has a somewhat restrictive computational model: no outside threads can join transactions, and threads can be forked and joined only on the boundary of a transaction.

Introducing a more sophisticated model which has the flexibility of the general MTT model but provides features for disciplined exception handling is important for many applications. Our analysis shows that it is not trivial to devise a consistent scheme, especially in the presence of concurrency.

In the rest of the paper we present a new transaction model called Open Multithreaded Transactions (OMTT), that is more flexible than the Argus model but still allows dealing with threads and exceptions in a disciplined and structured fashion.

3. Open multithreaded transactions

In the computational model of open multithreaded transactions a thread can terminate or fork another thread. The model allows threads to join an ongoing transaction at any time and to be forked and to terminate inside a transaction. There are only two rules that restrict thread behaviour: a thread created outside a transaction cannot terminate inside the transaction, and a thread created inside a transaction must also terminate inside the transaction.

Within a transaction a set of transactional objects can be accessed by the participating threads, and individual threads inside a transaction collaborate by accessing the same transactional objects. The threads have to be synchronised in order to guarantee consistency of the accessed transactional objects. Transactions are units of system structuring. They transition the system from one consistent state to another one. Also, if a transaction aborts, the system state remains unchanged.

Threads working on behalf of a transaction are referred to as participants. External threads that create or join a transaction are called *joined participants*; a thread created inside a transaction by a participant is called a *spawned participant.*

Let us now describe open multithreaded transactions in detail.

Starting an open multithreaded transaction:

• Any thread can start a transaction: it will be the first joined participant of the transaction. The newly created transaction is *open.*

• Transactions can be nested. A participant of an open multithreaded transaction can start a new (nested) transaction. Sibling transactions created by different participants execute concurrently.

Joining an open multithreaded transaction:

• A thread can join a transaction as long as it is open, thus becoming one of its joined participants. To do so it has to learn (at run-time) or to know (statically) the identity of the transaction it wishes to join.

• A thread can join a top-level transaction if and only if it does not participate in any other transaction. To join a nested transaction, a thread must be a participant of the parent transaction. A thread can only participate in one sibling transaction at a time.

• A thread spawned by a participant automatically becomes a spawned participant of the innermost transaction in which the spawning thread participates. A spawned participant can join a nested transaction, in which case it becomes a joined participant of the nested transaction.

• Any participant of a transaction can decide to *close* it at any time. Once the transaction is closed, no new joined participants are accepted anymore. Note that a participant can still spawn a thread. If no participant closes the transaction explicitly, it closes once all participants have finished (see below).

Concurrency control in open multithreaded transactions:

• Accesses to transactional objects by participants inside a transaction are isolated from accesses by other transactions. The only visible information that might be available to the outside world is the transaction identity to be used by threads willing to join.

• Accesses by a child transaction are isolated from accesses by the parent transaction.

• Inside of a given transaction, conventional techniques, like mutual exclusion, are used to guarantee consistency of transactional objects when accessed by several participants.

Ending an open multithreaded transaction:

• All transaction participants *finish* their work inside the transaction by voting on the transaction outcome. Possible votes are *commit* and *abort.*

• The transaction commits if and only if all participants commit. In this case, the changes made to transactional objects on behalf of the transaction are made visible to the outside world. If any of the participants wishes to abort, the transaction aborts. In that case, all changes made to transactional objects on behalf of the transaction are undone.

• Once a spawned participant has given its vote, it *terminates* immediately.

• Joined participants are not allowed to leave the transaction, i.e. they are blocked, until the outcome of the transaction has been determined. This means that all joined participants of a committing transaction exit synchronously. At the same time, but only then, the changes made to transactional objects are made visible to the outside world. Joined participants of a transaction that aborts can exit asynchronously, but changes made to the transactional objects are undone.

• If a participating thread "disappears" from a transaction without voting on its outcome, the transaction is aborted, as this case is treated as an error.

Figure 1 shows two open multithreaded transactions: T1 and T2. Threads A, B and C are joined participants of the containing transaction T1. Inside T1 the thread C forks a new thread D, a spawned participant, which performs some work on behalf of the transaction and then terminates. There is a nested transaction T2 inside T1 with three joined participants: B, C and D.

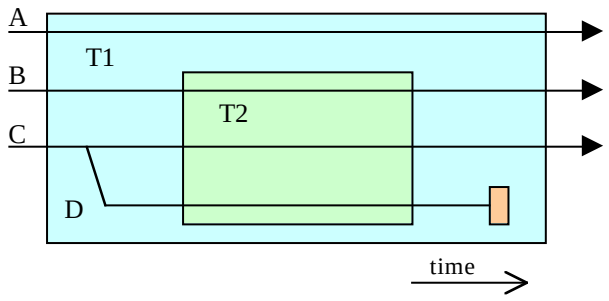

Figure 1. An example of open multithreaded transactions

4. Exception handling in open multithreaded transactions

In this section we discuss the exception handling mechanism developed for OMTT. Two important design decisions are:

• Distinguish between internal and external exceptions, also called interface exceptions;

• Interpreting any external exception propagated from a transaction context as an abort vote passed by this participant.

The following rules govern the OMTT exception handling mechanism.

Classification of exceptions:

• Each participant has a set of internal exceptions that must be handled inside the transaction, and a set of external exceptions which are signalled to the outside of the transaction, when needed. The predefined external exception Transaction_Abort is always included in the set of external exceptions.

Internal exceptions:

• Inside a transaction each participant has a set of handlers, one for each internal exception that can occur during its execution.

• The termination model is adhered to: after an internal exception is raised in a participant, the corresponding handler is called to handle it and to complete the participant's activity within the transaction. The handler can signal an external exception if it is not able to deal with the situation.

• If a participant "forgets" to handle an internal exception, the external exception Transaction_Abort is signalled.

External exceptions:

• External exceptions are signalled explicitly. Each participant can signal any of its external exceptions.

• Each joined participant of a transaction has a containing exception context.

• When an external exception is signalled by a joined participant, it is propagated to its containing context. If several joined participants signal an external exception, each of them propagates its own exception to its own context.

• If any participant of a transaction signals an external exception, the transaction is aborted, and the exception Transaction_Abort is signalled to all joined participants that vote commit.

• Because spawned participants don not outlive the transaction, they cannot signal any external exception except Transaction_Abort, which results in aborting the transaction.

Because the OMTT model provides transaction nesting, the exception handling rules have to be applied "recursively" by the programmer. All external exceptions of a joined participant are internal exceptions of the calling environment.

If an interface exception has been raised, all participants should be informed about the abort of the transaction as soon as possible. There are two distinct approaches: *blocking* and *pre-emptive*.

In the blocking approach, each participant completes the transaction by voting commit or by signalling an interface exception. If a participant votes abort, the other participants are informed of the abort only when they have completed or also signal an interface exception.

In the pre-emptive approach, the transaction does not wait for the participants to complete, but interrupts all participants as soon as one of them has signalled an external exception.

5. Transactional objects

5.1. Two-level concurrency control

In the OMTT model, access to transactional objects must be controlled at two levels. The first concern is to deal with competitive concurrency, i.e. to guarantee the isolation property of all updates made within a transaction with respect to other transactions running concurrently. The second concern is to handle cooperative concurrency within a transaction, i.e. to ensure mutual exclusion of individual operations performed by participants of the same transaction. Generally speaking, competitive concurrency control can use existing optimistic or pessimistic concurrency control techniques [3]. With the optimistic techniques, when transaction abort is used to compensate for a consistency violation, the abort can be either reported to the containing transaction by signalling the Transaction_Abort exception or the same transaction can be re-tried. The latter approach requires additional run-time support for restoring the thread states. In the presence of cooperative concurrency, consistency can be guaranteed simply by using monitors or similar techniques found in modern concurrent languages, for example protected objects in Ada 95, or objects with synchronised methods in Java.

5.2. Enhanced error detection

Early error detection is vitally important for reliable modern applications; it makes error recovery faster and more effective. To achieve this goal, special methodologies must be used during development. The OMTT model strives to make, whenever possible, the recovery local to individual transaction participants. Checks can be performed either by a participating thread or by a transactional object.

Firstly, all participants have to check all operation parameters they exchange with transactional objects. This is in line with the principles of defensive programming.

Secondly, error detection is enhanced by *self-checking transactional objects*. Such objects incorporate pre- and post-conditions and invariants. When any of these is violated during the execution of an operation, an exception is propagated to the calling participant.

There is a considerable body of research about designing classes of objects together with pre- and post-conditions and invariants, as well as about generating run-time checks for verifying them. The best known example is Bertrand Meyer's "design by contract" methodology supported by features of Eiffel [11]. This is how transactional objects should be designed when used in open multithreaded transactions. Such a design guarantees early error detection and localises error handling within a single transaction participant.

5.3. Exception handling and transactional objects

The previously described error detection techniques make error containment stronger and increase the chances that an internal exception can be handled locally by a participant. Of course, there will still be situations when they fail. In that case, the transaction is aborted and all the changes made to transactional objects on behalf of the transaction are undone, and an external exception is propagated to the calling context. If additional error recovery is needed, it must be performed at the higher level context.

Just as the OMTT model does not support tight collaboration among participants by providing means for direct communication, there is also no automatic cooperative exception handling. Loose collaboration among participants of an OMTT transaction can take place through transactional objects. Exception handling follows the same pattern. Two approaches are possible here. With the first one, a transactional object may be left in an erroneous state after propagating an exception to a participant that uses it. Subsequent operation invocations by other participants are likely to raise an exception as well. The second approach allows the programmer to apply a form of loose cooperative exception handling: the participants may decide to perform corrective or compensation activities on some transactional objects as a part of their local handling to recover from the error which might have spread within the transaction.

6. Typical uses of OMTTs

Our analysis shows that the OMTT model is useful for complex dynamic systems in which more than one thread can participate in a transaction, but the number of transaction participants is not known statically. It is also useful for systems in which threads participating in a transaction are created dynamically. Another important kind of use are systems that achieve different types of fault tolerance by using both transactional techniques and exception handling.

An important aspect of these systems is autonomy of the threads that eventually participate in a transaction: although they perform joint work inside a transaction and have a common goal, they are not tightly synchronised and can perform their jobs even when not all of them are in the transaction. In order to preserve their autonomy, participants cooperate via transactional objects only and handle their internal exceptions separately.

Without the cooperation facilities present in the OMTT model, such systems must be designed and structured out of individual transactions rather than of units of cooperation. Such an architecture might be much more complex. Also, because all these transactions

compete for the same transactional objects, performance might be poor.

Typical examples of such systems are auctions, systems with many readers / writers, systems with workers sharing jobs, etc. We are designing and implementing an online auction system. An auction is organised around a vendor, and is designed as an OMTT. Besides the vendor, there is an accounting system that participates in the auction transaction, but the number of bidders is not known in advance and may vary over time, because a bidder might want to join an already ongoing auction.

Another promising application area for OMTT is developing complex systems of systems. Individual subsystems are essentially autonomous and it is important to allow them, whenever possible, to deal separately with their exceptions even when they participate in a joint activity. Besides, by the very nature of these systems, they have to deal with faults from a wide range of types, and have to accommodate flexible participation because they evolve.

7. Implementation approaches

There are several ways of implementing the OMTT model. Different run-time supports are needed for single-processor, multi-processor and distributed settings because of different fault assumptions and different communication models. A complete distributed decentralised implementation has to rely on ordering and group communication protocols. If the communication protocol is based on the Internet, then the OMTT implementation can be used to develop complex Internet applications.

In order to minimise implementation effort, the best approach is to rely as much as possible on existing transaction middleware and thread control offered by a concurrent programming language. The implementation effort then consists in integrating these two capabilities.

A very promising approach for distributed systems is to rely on existing transactional services, for example that of CORBA, and to add thread coordination with regard to transactions.

The main idea is to introduce a coordinator object for each open multithreaded transaction, that keeps references to all transaction participants. In turn, all transaction participants must keep a reference to the coordinator object. The coordinator object also keeps references to all coordinator objects of nested transactions.

Each open multithreaded transaction is mapped to a CORBA transaction. The coordinator object executes the CORBA transaction begin, abort and commit operations.

Clearly, the thread handling facilities of the host language can not be used as such. Thread spawning and joining has to be implemented in such a way that the coordinator keeps track of all thread creations and terminations, and of all exception raising and signalling.

OMTT-specific concurrency control policies must be developed for transactional objects to correctly address intra- and inter-transaction concurrency.

In this approach based on a middleware, the main effort is directed towards thread coordination implementation. Another approach starts with a concurrent programming language providing extensive thread control, and adds transaction support. This is the approach taken in our prototype implementation presented in the next section.

8. Prototype implementation

We are implementing OMTT support for Ada 95 [7]. One goal of this development is the smooth integration of the native concurrency features of Ada and the newly developed OMTT features. Generally speaking, there are two types of services: the ones related to thread control, tasking in Ada terminology, and the other ones related to transactional objects. The latter is implemented as an extension of the TransLib framework presented in [5, 6]. The approach is based on design patterns in order to maximise modularity and flexibility. The OMTT support can be easily customised for specific application needs by using object-oriented programming techniques, because:

- it supports optimistic and pessimistic concurrency control;
- it provides different locking schemes, like read / write locking and commutativity based locking;
- it is able to employ different kinds of storage devices [8, 9];
- it offers a number of recovery strategies, i.e. Undo/Redo, NoUndo/Redo and Undo/NoRedo;
- it can use physical logging or logical logging;
- it supports different caching techniques, like last recently used or least frequently used.

In order to become a transactional object, each data object has an associated memory object, a storage unit, a lock manager and a proxy object. The proxy object provides the same interface as the object itself. Instead of accessing a data object directly, participants invoke the operations on the proxy object, which performs the required recovery, logging and concurrency control operations. If locking is used, it starts by obtaining a lock for the operation from the lock manager. That is the place where cooperative and competitive concurrency control shows through. In order to execute an operation on a transactional object, two locks must be acquired: the transactional lock and the mutual exclusion lock. The transactional lock guarantees isolation with respect to other concurrently executing transactions. The mutual exclusion lock guarantees consistent updating of the data by participant threads. Once both locks are acquired, the proxy object informs the recovery manager of the operation call, and then passes the operation on to the memory object. The memory object calls the cache manager, that loads the data object's state from the

associated storage unit, if necessary. Finally, the operation is invoked on the data object. After the call has been executed, the recovery manager and the lock manager are notified. Only the mutual exclusion lock is released, though. The transactional lock is only released once the outcome of the transaction has been determined.

Our goal was to provide the OMTT support at the programming language level without modifying the language. We therefore developed a set of service procedures, an API, and programming guidelines to be followed when using OMTT. Clearly, the API necessarily depends on the features of the programming language. Fortunately, Ada is very powerful and many implementation details can be hidden from the API, while still helping the programmer by enforcing correct use. However, when it comes to OMTT exception handling, we have to impose somewhat complicated conventions, because the native Ada exception handling mechanism is basically sequential and block-oriented.

Let us now explain the reason for the simplicity of the API in Ada. Once a thread is part of a transaction, it can invoke operations on transactional objects. The recovery and lock manager must know on behalf of which transaction the operation is executed. Most implementations will therefore need to pass a transaction identifier (ID) as a parameter to every operation of a transactional object. Ada offers a native mechanism that avoids this explicit parameter passing. Using task attributes, it is possible to declare a data type where a transaction ID can be stored. There is a copy of this data type associated with each thread in the system, and when the thread joins a transaction, it is used for storing the corresponding transaction ID. When an operation of a transactional object is called, the support can check the ID. As a result, there is no difference for an application programmer in calling a transactional object or a normal object.

There are several possibilities for defining an API for the OMTT model. Let us start with the procedural interface. Several procedures are needed:

- The procedure Start_Transaction starts a new transaction. If the calling thread is already a participant of a transaction, a nested transaction is created.
- The procedure Join_Transaction allows a thread to join a transaction. It checks that the calling thread is either a participant of the parent transaction, or is not participating in any transaction.
- The procedure Close_Transaction closes the transaction by forbidding further joining.
- The procedure Commit_Transaction commits the changes made on behalf of the transaction; it blocks until the outcome of the transaction has been determined.
- The procedure Abort_Transaction aborts the current transaction; it does not block the caller.

This API is quite flexible, but it does not protect programmers from making mistakes. For example, it is possible to start or join a transaction, but forget to vote on its outcome, or to let an unhandled exception cross the transaction boundary unnoticed. Programmers must follow guidelines to avoid these kinds of problems. For example, if all transactions in an application are programmed using the following template, the adherence to the OMTT model is guaranteed:

```
begin
    begin
       Start_Transaction;
       -- perform work
       Commit_Transaction;
    exception
      when ...
          -- handle internal exceptions
          Commit_Transaction;
          -- or raise an external exception
      when others =>
          raise Transaction_Abort;
    end;
exception
  when others =>
      Abort_Transaction;
      raise;
end;
```

Another API uses another Ada feature called controlled types for transaction control. Ada controlled types allow programmers to provide their own procedures which are called when an object of the type is created, accessed or goes out of scope. The API offers a type Transaction that can be used in the following way:

```
declare
   T : Transaction;
begin
   -- perform work
   Commit_Transaction  (T);
exception
   when  ...
      -- handle internal exceptions
      Commit_Transaction  (T);
end;
```

In this approach the Ada block is both the transaction and the exception context. When the block starts the procedure Initialize of the Transaction object T is called implicitly and starts a new transaction. Commit_Transaction must be called before exiting the block, or else the implicitly called Finalize procedure will invoke Abort_Transaction.

An object-oriented API is under development as well. It is based on an abstract tagged type Transaction_Type:

```
package Open_Multithreaded_Transactions  is
   type Transaction_Type  is
              abstract tagged limited private;
private
   procedure Start_Or_Join_Transaction
              (T: in out Transaction_Type);
```

```
   procedure Close_Transaction
               (T: in out Transaction_Type);
   procedure Abort_Transaction
               (T: in out Transaction_Type);
   procedure Commit_Transaction
               (T: in out Transaction_Type);
end Open_Multithreaded_Transactions;
```

A concrete transaction derives from this abstract type and adds application-specific operations, one for each participant. When a thread wants to participate in the transaction, it calls the corresponding operation.

8. Discussion

In our opinion, the OMTT model offers the right balance between unrestricted thread behaviour like in CORBA-type MTT and the rather restricted model of Argus [10]. Exception handling is an immanent part of the model: to the contrary of CORBA-type MTTs, exception raising, handling, propagation, etc., are tightly coupled with transactions, because transactions are exception contexts. OMTT always keep all threads accessing transactional objects and all exceptions which can happen inside a transaction under control. Open multithreaded transactions are units of system design, and exception handling is therefore designed at the same time as the transactional structure.

Coordinated Atomic (CA) actions [14] is a well known structuring technique which combines features of atomic actions [1] and transactions. Multiple active participants can enter a CA action to perform a joint activity inside: they can use both local objects, for cooperation, and transactional objects, all updates of which are isolated from the outside world. CA action execution looks like a transaction for the outside world. Exception handling is the main feature of action recovery: all participants are involved in coordinated exception handling if any of them raises an exception. If recovery fails, an external exception is propagated to the containing context.

Part of the motivation for CA actions are similar to those for OMTT and many problems addressed by OMTT can be solved by CA actions. But in our opinion they are different: CA actions effectively add transactional objects to atomic actions, whereas OMTTs add thread coordination to MTT. These are the main distinguishing characteristics of the OMTT model:

• Participants of a CA action collaborate closely: they rely on each other, they are designed together and hence are tightly coupled, and they synchronise their execution explicitly through local resources. OMTTs are intended for designing systems with a different, less entangling type of collaboration, in which participants have their own reasons for taking part in the transaction and are quite autonomous and independent. They are designed rather separately with only limited knowledge about the other OMTT participants.

• The OMTT model is more flexible: threads can be created and terminated inside transactions.

• The OMTT model pays special attention to using enhanced error detection and to providing special mechanisms for supporting it. As a consequence, error recovery is easier and more effective, and chances are increased for successful local error recovery.

• In OMTTs, an attempt is always made to handle any internal exception locally by a participant. We have decided against any form of coordinated exception handling for several reasons. Firstly, the number of OMTT participants is not known in advance, and hence any form of error handling that depends on the presence of participants other than the one that raised the exception can be error-prone. Secondly, exceptions defined in one participant might have no meaning in other participants. Thirdly, we do not want to impose any unnecessary additional synchronisation; we would like to allow them to act as independently as possible. Fourthly, dealing with multiple exceptions which might be raised concurrently (as done in CA actions) complicates the implementation and adds run-time overhead; local handling is usually less expensive and more effective. Finally, using self-checking transactional objects increases the chance that errors are not propagated between participants: participants act independently inside a transaction and their "bad" influence on other participants is limited.

• The OMTT model offers a unified way of participant coordination via transactional objects which suits well the type of systems they are intended for.

9. Conclusions

As we have seen, the open multithreaded transaction model provides features for controlling and structuring not only accesses to objects, as usual in transaction systems, but also threads taking part in transactions. The model allows several threads to enter the same transaction in order to perform a joint activity. It provides a flexible way of manipulating threads executing inside a transaction by allowing them to be forked and terminated, but it restricts their behaviour when necessary in order to guarantee correctness of transaction nesting and structured exception handling. The OMTT model incorporates disciplined exception handling, well adapted to nested multithreaded transactions. It allows individual threads to handle an abnormal situation locally, and promotes a defensive approach for developing transactional objects, so that errors are detected early and dealt with inside the transaction. If local handling fails, the transaction support reverses the system to its “initial” state.

In the future, we will develop case studies in order to experiment with the OMTT model. Another direction of research is to extend our prototype to distributed systems by using group communication protocols or middleware. Finally, it would be important to elaborate a formal description of the OMTT model and its properties.

Acknowledgements. Alexander Romanovsky has been supported by the EC IST RTD Project on Dependable Systems of Systems. Jörg Kienzle has been partially supported by the Swiss National Science Foundation project FN 2000-057187.99/1.

10. References

[1] R.H. Campbell, B. Randell. Error Recovery in Asynchronous Systems. IEEE TSE, SE-12 (8), 1986.

[2] F. Cristian. Exception Handling and Tolerance of Software Faults. In Software Fault Tolerance. M.R. Lyu (Ed). Wiley. pp. 81-108, 1994.

[3] J. Gray, A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[4] N. Haines, D. Kindred, J.G. Morrisett, A.M. Nettles, J.M. Wing. Composing First-Class Transactions. ACM TOPLAS, 16, 6, 1994.

[5] R. Jimenez-Peris, M. Patino-Martinez, S. Arevalo. TransLib: An Ada 95 Object Oriented Framework for Building Transactional Applications. Computer Systems: Science & Engineering J., 15, 1. 2000.

[6] J. Kienzle, R. Jiménez-Peris, A. Romanovsky, M. Patino-Martinez. Transaction Support for Ada. Submitted to the International Conference on Reliable Software Technologies - Ada-Europe'2001, Leuven, Belgium, 2001. Available as Technical Report (EPFL-DI No 2000/348).

[7] J. Kienzle, A. Romanovsky. Combining Tasking and Transactions: Open Multithreaded Transactions. Presented at the 10th Int. Real-Time Ada Workshop, Avila, Spain, 2000 (to be published in AdaLetters).

[8] J. Kienzle, A. Romanovsky. A Framework Based on Design Patterns for Providing Persistence in Object-Oriented Programming Languages. Technical Report CS-TR-688, Dept. of Computing Science, University of Newcastle upon Tyne, April 2000.

[9] J. Kienzle, A. Romanovsky. On Persistent and Reliable Streaming in Ada. In International Conference on Reliable Software Technologies - Ada-Europe'2000, Potsdam, Germany, June 26-30, 2000, LNCS Volume 1845, pp. 82-95. Available as Technical Report (EPFL-DI No 99/323).

[10] B. Liskov. Distributed Programming in Argus. CACM, 31, 3, 1988.

[11] B. Meyer. *Object-oriented software construction*. Prentice Hall,1997.

[12] J.E.B. Moss. *Nested Transactions, An Approach to Reliable Computing*. Ph.D. Thesis, MIT, 1981.

[13] G.D. Parrington, S.K. Shrivastava, S.M. Wheater, M.C. Little. The Design and Implementation of Arjuna. USENIX Computing Systems J., 8, 3, pp. 255 – 308,1995.

[14] J. Xu, B. Randell, A. Romanovsky, C.M.F. Rubira, R.J. Stroud, Z. Wu. Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. In the 25th International Symposium on Fault-Tolerant Computing, USA, pp. 499-509, 1995.