Combining Tasking and Transactions, Part II: Open Multithreaded Transactions

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

Alexander Romanovsky
Department of Computing Science
University of Newcastle
NE1 7RU, Newcastle upon Tyne
United Kingdom
email: Alexander.Romanovsky@newcastle.ac.uk

Abstract

This position paper is a follow-up paper of [1], presented at the last IRTAW workshop. The paper describes a model for providing transaction support for concurrent programming languages such as Ada 95. In order to achieve smooth integration, the use of the concurrency features provided by the Ada language should not be restricted inside a transaction. A transaction model that meets this requirement is presented. Tasks inside such a transaction may spawn new tasks, but also external tasks are allowed to join an ongoing transaction. A blocking commit protocol ensures that no task leaves the transaction before its outcome has been determined. Exceptions are used to inform all participants in case a transaction aborts. Possible interfaces for the Ada programmer are discussed.

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.

Ada 95 [2] reflects this trend, since it incorporates support for different forms of concurrency. It provides elaborate lightweight concurrency features such as protected types and tasks. Distribution of a single program on multiple processing nodes is supported through the Distributed Systems Annex. But among these active entities, concurrency control and synchronization is reduced to single method, procedure or entry calls. 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.

Ada has a strong reputation for its error-prevention qualities, such as strong typing, modularity, and separate compilation; it has been extensively used for the development of mission-critical and safety-critical software. Support for transactions in Ada 95 would be a powerful tool for a programmer of a fault-tolerant application.

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, threads and processes. The former is supported in Ada through tasks and task types. Communication between tasks can be *asynchronous* (through *protected types*) or *synchronous* (through rendezvous). The latter is supported in Ada 95 by means of the Ada Distributed Systems Annex. An Ada program can be split into multiple partitions, which can be configured to execute on different processing nodes. The communication mechanism provided between partitions is synchronous or asynchronous remote procedure call.

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

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 of data and to provide fault tolerance with respect to hardware failures [3]. A transaction groups an arbitrary number of simple update operations on data objects 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 is 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 and visible to the outside. 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* [5] 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 [6] or in the *SAGAS* model found in [7].

Another possibility to increase concurrency between transaction is to allow certain transactions to view the results of other transactions before they commit / abort (as done in the *Recoverable Communicating Actions* model [8], the *Split Transaction* model and the *Joint Transaction* model [9]. Of course, this creates a certain dependency between these transactions.

Cooperative Concurrency

Conversations

The concept of a *conversation* has been introduced in [10] in 1975. It allows a fixed number of 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*.

Atomic Actions

Later on, conversations have been enhanced with additional forward error recovery and exception resolution [11], 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. An Ada 95 framework consisting of programming guidelines and techniques for programming atomic actions is presented in [12], and a complete discussion is given in [13].

Combining Cooperative and Competitive Concurrency Recently, some transaction models have evolved to allow cooperative concurrency inside a transaction. The C++ extension Arjuna [14] 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 [15].

[16] describes a model called *Multithreaded Transactions*. A multithreaded transaction has precise semantics: Once a task (the *main* task) has started a transaction, it can fork new tasks that work on behalf of it to take advantage of concurrency. Before the main task can commit or abort the transaction, these forked tasks must all run to completion. The same model is also used in *Cooperative Transactional Object Groups* presented in [17, 18].

The atomic action concept has also been extended. The *Coordinated Atomic Action* model [19] 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 a concurrent programming languages such as Ada, it is important to support concurrency inside a transaction in a natural way. In particular, the use of the concurrency constructs provided by the language should not be restricted inside a transaction, if possible.

The multithreaded transaction model [16, 17] 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 task, and then spawn new tasks inside the transaction. These spawned tasks must run to completion before the transaction can be committed. Creation and deletion of tasks can be very time-consuming and programmers try to avoid it whenever possible. Process control and especially real-time systems tend to use a static number of tasks, created once and for all during the initialization of the system. A transaction model for concurrent object-oriented languages should allow existing tasks to join an ongoing transaction. Therefore the multithreaded transaction model [21, 17] cannot be used as it does not allow already existing tasks come together and decide to perform a job on a set of objects in a transactional manner.

On the other hand we don't want to forbid spawning new tasks inside a transaction. This would exclude the use of the coordinated atomic action [19] model, since it requires the number of participants to be fixed in advance.

The kind of collaboration we are looking for is also different from that in the coordinated atomic action model. Participants of a coordinated atomic action collaborate closely; they rely on each other. This is possible, for they know the identity of the other participants and are assured of their presence. They have been designed together and hence are tightly coupled, communicating explicitly or through shared local resources. The collaboration we want to achieve in transactions is somehow different, less entangling. Communication between tasks will be done exclusively through transactional objects. The number of participant tasks will not be fixed in advance, since at the beginning of a transaction it may sometimes not even be foreseeable how many tasks 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 task and maybe some accounting system that will participate in such a transaction, but the number of bidder tasks is not known in advance. A bidder might also want to join an already ongoing auction.

The rest of this section presents a new transaction model named *Open Multithreaded Transactions* that fits our needs. The model allows tasks to be created, to run to completion, or to join an ongoing transaction at any time. There are only two rules that restrict task behavior:

- A task created outside of an open multithreaded transaction cannot terminate inside this transaction.
- A task created inside an open multithreaded transaction must also terminate inside this transaction.

The individual tasks inside an open multithreaded transaction collaborate loosely. They evolve independently, besides when exchanging data with other task that work on behalf of the transaction through transactional objects. In that case they have to be synchronized with respect to other participating tasks in order to guarantee the consistency of the accessed data.

The following paragraphs describe the rules for open multithreaded transactions. Tasks working on behalf of an open multithreaded transaction are referred to as *participants*. External tasks that join an open multithreaded transaction are called *joined participants*; task created inside an open multithreaded transaction by a participant are called *spawned participants*:

Starting Open Multithreaded Transactions

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

Joining Open Multithreaded Transactions

- A tasks can join an open multithreaded transaction is it is still open, thus becoming *joined participants* of the transaction.
- A task can join a top-level open multithreaded transaction if and only if it does not yet participate in any other transaction. To join a nested transaction, a task must be a participant of the parent transaction. A task can only participate in one sibling transaction at a time.
- Tasks spawned by participants will automatically become *spawned participants* of the transactions in which the spawning task participates. Spawned participants are allowed to join a nested transaction. If they do so, they are considered joined participants of the nested transaction.
- A participant of an open multithreaded transaction can decide to *close* the transaction at any time. Once the transaction is closed, no new task can join the transaction anymore. If no participant closes the transaction explicitly, it closes once all participants have given their vote.

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.
- Accesses of child transactions are isolated with respect to their parent transaction.
- Classic consistency techniques (i.e. protected types) are used to guarantee consistent updating of the state of transactional objects by participants of the same transaction.

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*. In that case, the changes made to transactional objects on behalf of the transaction are made visible to the outside world. If any of the participants votes *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 (they are blocked) until the outcome of the transaction has been determined. This means in particular that all joined 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. If the transaction is aborted, the joined participants may exit asynchronously, once changes made to the transactional objects have been undone.

Figure 1 depicts a non-nested open multithreaded transaction with 6 participants. Task C starts the transaction, task A, B and D join it later on. During the transaction, task C forks a new task, task C'. This spawned participant performs some work on behalf of the transaction and then terminates after having given its vote on the outcome of the transaction. The same is true for task B'. In this example, all tasks vote *commit*. The joined participants A, C, and D are therefore blocked until the last participant, here Task B, has finished its work and given its vote.

Exception Handling in Open Multithreaded Transactions

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

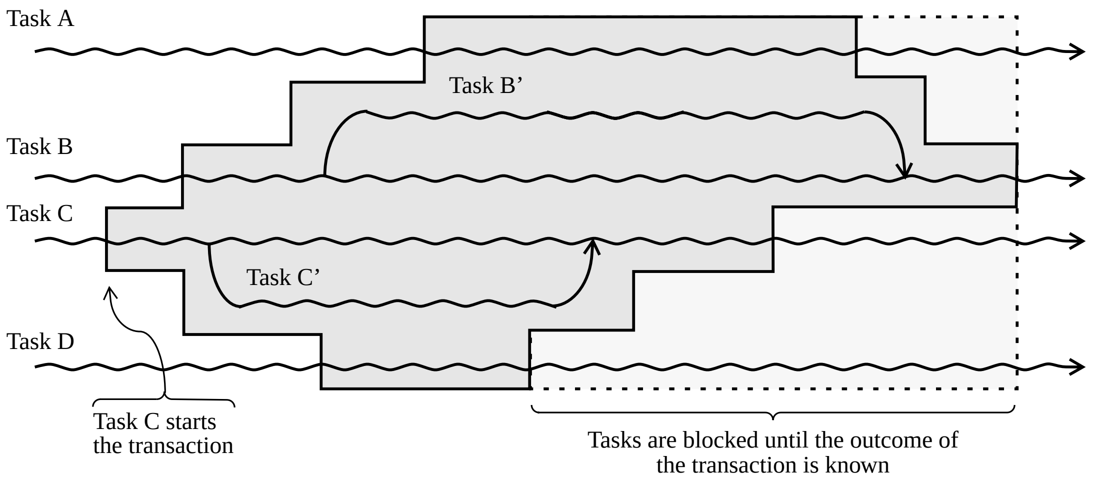
Task A
Task B'
Task B
Task C
Task C'
Task D
Task C starts the transaction
Tasks are blocked until the outcome of the transaction is known

Figure 1: An Open Multithreaded Transaction

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. For this reason, and following the ideas of [17], we have decided that unhandled exceptions crossing the transaction boundary result in aborting the open multithreaded transaction. If this happens all participants must be notified of the abortion of the transaction.

As mentioned above, tasks inside an open multithreaded transaction cooperate loosely. Each participant task has its own local exception context, and must handle its exceptions separately. Unlike [17, 19] we have decided against some form of coordinated exception resolution for multiple reasons. Firstly, the number of participants of an open multithreaded transaction is not determined 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 task might have no meaning or even be undefined in some other participant. Thirdly, we would not like to impose any unnecessary synchronization because participants do not cooperate tightly; we would like to allow them to act as independent as possible. Finally, concurrent and potentially distributed exception resolution can be very time-consuming and difficult to program.

The following rules summarize exception handling in open multithreaded transactions:

- Each participant has a set of internal exceptions that it can handle inside a transaction and a set of external exceptions that it can signal outside. An additional external exception `Transaction_Abort` is always included in the set of external exceptions.
- Internal exceptions raised in a participant of an open multithreaded transaction are not propagated to other participants.
- The termination model of exceptions [20] is adhered to: after an internal exception has been raised in a participant a corresponding handler is called to handle the exception and to complete the participant activity within the transaction. The handler can decide to signal an external exception to the outside in case it is not able to handle the exception properly. The default handler for internal exceptions is to raise the external exception `Transaction_Abort`.
- Each joined participant of a transaction has a containing exception context associated with the containing transaction.
- When an external exception is signaled by a joined participant, it is propagated to its containing context and the transaction aborts. When a spawned participant signals an external exception, the exception is lost since the task terminates immediately after voting, but it still results in transaction abort.
- If joined participants raise external exceptions concurrently, the transaction is aborted and the external exceptions are propagated to the corresponding contexts of the participants that raised them.
- When the transaction aborts, all joined participants that have not raised an external exception are notified of abort by means of the `Transaction_Abort` exception as soon as possible[1].

An obvious problem that has not been discussed yet are *deserters*, i.e. tasks participating in an open multithreaded transaction that suddenly disappear without voting on the

1. Depending 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.

outcome of the transaction. This can happen if a task is explicitly killed[1], or when the process of a participant task dies incidentally (e.g. due to a system crash). We view such deserter tasks as a fault, and a reason for transaction abort.

A framework providing support for open multithreaded transactions has been designed and implemented in form of a library [21]. It makes heavy use of design patterns in order to maximize modularity and flexibility. The following section examines different interfaces to the framework.

4 Transaction Support Interface

We want to provide transaction support at the programming language level, without modifying the language in any way. For this reason, approaches such as [17] are not possible, since they augment the language and hence must modify the compiler or provide a preprocessor to recognize the additional keywords.

Similar to [13], we want to provide a framework or programming guidelines that must be used to access our transaction support. The elegance of the interface depends on the features of the programming language. Fortunately Ada offers some special features that allow us to help the application programmer by enforcing certain rules and hiding most parts of the transaction management when accessing transactional objects.

Transaction Identifier Management

Once a task 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 systems therefore need to pass a transaction identifier as a parameter to every operation of a transactional object.

The Systems Programming Annex of the Ada Standard offers the possibility to declare data structures for which there is a copy for each task in the system by means of the generic package `Ada.Task_Attributes`. Using this package, the transaction identifier can be linked to each participant task once it joins the transaction. When calling an operation of a transactional object, the runtime support can retrieve the transaction identifier using the function `Task_Attributes.Value`. This means that for the application programmer there is no difference in calling a transactional object or calling a normal object.

Handling Transactions

Procedural Interface

The most commonly used interface to transactions is the procedural interface. In the open multithreaded transactions model we need four procedures. Again we can get rid of the transaction identifier parameter using task attributes.

- **procedure** `Begin_Transaction;`
- **procedure** `Join_Transaction`
  `(T : Task_ID);`
- **procedure** `Commit_Transaction;`
- **procedure** `Abort_Transaction;`

This procedural interface is flexible, but has some drawbacks. It is possible to start or join a transaction, but forget to vote on its outcome, which results in blocking all other participants that behave correctly. But what's even more annoying is that using the procedural interface we can not guarantee that an unhandled exception crossing the transaction boundary will abort the transaction as required. In order to guarantee this, the programmer must use a construct such as:

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

Using Controlled Types

To avoid forgetting to vote on the outcome of a transaction, one could imagine offering a controlled type `Transaction`:

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

What is interesting here is that the Ada block construct is at the same time the transaction and the exception context. Declaring the transaction object calls the `Initialize` procedure of the transaction type, which on its part calls the transaction support and start a new transaction. The transaction identifier is associated with the calling task. The task can now work on behalf of the transaction, and if everything goes fine `Commit_Transaction` must be called before exiting the block. If a programmer forgets to commit the transaction, or if an unhandled exception crosses the block boundary, the transaction object is final-

1. Using for instance the Ada `abort` statement.

ized. The `Finalize` procedure can then call `Abort_Transaction`. Note that in this case it is not possible to raise the exception `Transaction_Abort`, since raising exceptions inside of the `Finalize` procedure is considered a bounded error.

Transactions As Objects

Based on the ideas of [22, 13], we can also develop an object-oriented approach to open multithreaded transactions:

The package `Open_Multithreaded_Transaction` declares an abstract tagged type `Transaction_Type`. A concrete transaction must derive from this type and add code for each participant by adding primitive operations. A task that wants to work on behalf of the transaction will simply call the corresponding primitive operation.

```
package Open_Multithreaded_Transaction is
    type Transaction_Type is abstract
       tagged limited private;
    -- add code for each participant
    -- using primitive operations
private
    procedure Start_Or_Join_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_Transaction;
```

A primitive operation must follow the following programming conventions:

```
procedure Participant_Code
    (T : in out Transaction_Type) is
begin
    Start_Or_Join_Transaction (T);
    -- perform work on behalf of the
    -- transaction
    Commit_Transaction (T);
exception
    when ... =>
    -- handle internal exceptions
    when others =>
       Abort_Transaction (T);
end Participant_Code;
```

The call to the private procedure `Start_Or_Join_ Transaction` starts a new transaction, or join the ongoing transaction if it has already been started by some other participant. Apart from this call, the structure resembles the one used in the procedural interface. This time the procedure construct provides the transaction and exception context.

It is not possible to provide default implementations for participant operations, since we don't know in advance how many there will be. A possible solution might be to provide only one primitive operation `Execute_Participant`, that takes as a parameter an access to subprogram value which will point to the actual participant code. This way the programmer can not forget the call to `Start_Or_Join_ Transaction` and the call to `Abort_Transaction` in case of unhandled exceptions. On the other hand, using access to subprogram types is not very elegant and complicates parameter passing.

Joining Tasks Created Inside a Transaction

The open multithreaded transaction model allows tasks to be created inside a transaction. These tasks must be registered as spawned participants of the transaction. Unfortunately we could not find a way to do this automatically in Ada. One might think that using a controlled task attribute would allow us to call `Join_Transaction` from within the `Initialize` procedure. Unfortunately this is not possible, since task attributes are not required to exist once a task is created, but only when they are associated a value different than the default value. This implies that even if a task attribute is a controlled type, the `Initialize` procedure will not automatically be called upon every task creation.

An additional problem is that it is unfortunately not possible using standard Ada to obtain the creator, parent or master task of a newly created task. We therefore can not implement a procedure that automatically joins the transaction of the creator task. The only solution is to pass the task ID to the new task using a rendez-vous, through a protected object or by means of a discriminant. The new task then has to call `Join_Transaction` just as any other external task that wants to join the transaction.

Detecting Deserters

Deserters are participant tasks that terminate abnormally inside a transaction without previously voting on its outcome, for instance because they have been aborted by some other task, or because they encountered an exception that they could not handle. It is important to detect these deserters, or else all other participants will have to wait forever. Ada allows us to detect them by using a controlled task attribute. Once the task runs to completion, the `Finalize` procedure will be called. Using the transaction identifier stored in the task attribute, the transaction support is notified and can take the appropriate measures.

5 Conclusions

This paper has presented a possible way of providing transaction support in a concurrent programming languages such as Ada 95. 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 the concurrency constructs provided by Ada inside the transaction. Tasks inside a transaction can spawn new tasks, but also external tasks can join an ongoing transaction. A blocking commit protocol ensures that no task 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 notify all participants in case a transaction aborts.

In the second part of the paper we have concentrated on providing an elegant interface to the transaction support library for Ada programmers. Advanced features of Ada such as the package `Ada.Task_Attributes` have proven to be very useful in order to ease the task of the application programmer. Transaction ID management can be hidden completely when accessing transactional objects. Unfortunately it was not possible to provide automatic joining of newly created tasks due to two reasons: it is neither possible in standard Ada to react to each task creation in the system nor to find the creator task of a newly created task. Due to the restriction that forbids raising an exception during finalization of an object, controlled types can not be used to raise an exception when exiting a block or procedure construct.

6 Acknowledgements

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

7 References

[1] J. Kienzle: “Combining Tasking and Transactions”. In *Proceedings of the 9th International Real-Time Ada Workshop, Wakulla Springs Lodge, Tallahassee FL, USA, March 1999*, pp. 49 – 53, Ada Letters **XIX(2), June 1999**, ACM Press, 1999.

[2] ISO: *International Standard ISO/IEC 8652:1995(E): Ada Reference Manual*, Lecture Notes in Computer Science **1246**, Springer Verlag, 1997; ISO, 1995.

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

[4] P. A. Lee and T. Anderson: “Fault Tolerance - Principles and Practice”. In *Dependable Computing and Fault-Tolerant Systems*, volume 3, Springer Verlag, 2nd ed., 1990.

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

[6] H. Korth, W. Kim, and F. Bancilhon: “On Long-Duration CAD Transactions”. *Information Sciences* ***46****(1-2)*, pp. 73 – 107, October - November 1988.

[7] H. Garcia-Molina and K. Salem: “SAGAS”. In U. Dayal and I. Traiger (Eds.), *Proceedings of the SIGMod 1987 Annual Conference*, pp. 249 – 259, San Francisco, Ca, May 1987, ACM, ACM Press.

[8] S. Vinter, K. Ramamritham, and D. Stemple: “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.

[9] C. Pu, G. E. Kaiser, and N. C. Hutchinson: “Split-Transactions for Open-Ended Activities”. In F. Bancilhon and D. J. DeWitt (Eds.), *Fourteenth International Conference on Very Large Data Bases*, pp. 26 – 37, Los Angeles, California, 1988, Morgan Kaufmann.

[10] B. Randell: “System structure for software fault tolerance”. *IEEE Transactions on Software Engineering* ***1****(2)*, pp. 220 – 232, 1975.

[11] R. H. Campbell and B. Randell: “Error Recovery in Asynchronous Systems”. *IEEE Transactions on Software Engineering (SE)* ***SE-12****(8)*, August 1986.

[12] A. Romanovsky, S. E. Mitchell, and A. J. Wellings: “On Programming Atomic Actions in Ada 95”. In *Reliable Software Technologies - Ada-Europe ’97*, volume 1251 of *Lecture Notes in Computer Science*, pp. 254 – 265, Springer Verlag, June 1997.

[13] A. Romanovsky: “A Study of Atomic Action Schemes Intended for Standard Ada”. *Journal of Systems and Software* ***43****(1)*, pp. 29 – 44, October 1998.

[14] G. D. Parrington, S. K. Shrivastava, S. M. Wheater, and M. C. Little: “The Design and Implementation of Arjuna”. In USENIX (Ed.), *Computing Systems, Summer, 1995.*, volume 8, pp. 255 – 308, Berkeley, CA, USA, Summer 1995, USENIX.

[15] Object Management Group, Inc.: *Object Transaction Service*, August 1994.

[16] N. Haines, D. Kindred, J. G. Morrisett, S. M. Nettles, and J. M. Wing: “Composing First-Class Transactions”. *ACM Transactions on Programming Languages and Systems* ***16****(6)*, pp. 1719 – 1736, Nov 1994.

[17] M. Patiño-Martinez, R. Jiménez-Peris, and S. Arevalo: “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.

[18] R. Jiménez-Peris, M. Patiño-Martinez, and S. Arevalo: “TransLib: An Ada 95 Object-Oriented Framework for Building Transactional Applications”. *Computer Systems: Science & Engineering Journal* ***15****(1)*, 2000.

[19] J. Xu, B. Randell, A. Romanovsky, C. M. F. Rubira, R. J. Stroud, and Z. Wu: “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.

[20] J. B. Goodenough: “Exception Handling: Issues and a Proposed Notation”. *Communications of the ACM* ***18****(12)*, pp. 683 – 696, December 1975.

[21] J. Kienzle, R. Jiménez-Peris, A. Romanovsky, and M. Patiño-Martinez: “Transaction Support for Ada”. *Submitted to International Conference on Reliable Software Technologies - Ada-Europe’2001, Leuven, Belgium, May 14-18, 2001*, Lecture Notes in Computer Science, 2001.

[22] A. Wellings and A. Burns: “Implementing Atomic Actions in Ada 95”. *IEEE Transactions on Software Engineering* ***23****(2)*, pp. 107 – 123, February 1997.