Implementing Exceptions in Open Multithreaded Transactions Based On Ada 95 Exceptions

Jörg Kienzle
Software Engineering Laboratory
Swiss Federal Institute of Technology Lausanne
CH - 1015 Lausanne EPFL
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 shows how Ada 95 exceptions have been used in a prototype implementation of a transaction support in order to provide more elaborate exception handling. The paper summarizes the open multithreaded transaction model, which is a transaction model suitable for concurrent programming languages, reviews in detail its elaborate exception handling features, and analyzes the exception mechanism provided by the Ada 95 programming language. Different interfaces to the transaction support for the application programmer are presented, and the problems encountered during implementation of the prototype with respect to exception handling are discussed.

Keywords: *Exceptions, Open Multithreaded Transactions, Ada 95.*

1 Introduction

Recently, the concept of *open multithreaded transactions* has been developed [wordspaper, 1, 2]. It defines a transaction model that fits the concurrency model used in concurrent programming languages such as Ada 95 [3].

Transactions are atomic units of system structuring that are intended to move the system from a consistent state to some other consistent state. Applications using transactions are fault-tolerant, since transactions provide automatic backward error recovery in case of crash failures.

Most modern programming languages provide exceptions [4] and sequential exception handling in order to deal with abnormal events. Exceptions can also be used for programming fault-tolerant applications, since they allow a programmer to apply forward error recovery. Exception handling is in general also tightly coupled with the structure of a program.

Integration of exceptions and transactions is highly desirable. The open multithreaded transaction model defines precise rules for combining exception handling and transactions, resulting in units of error confinement that provide forward and backward error recovery.

An extensible object-oriented framework providing support for open multithreaded transactions has been developed [5]. An application programmer can customize and tailor it to fit the specific application needs. A prototype of the framework has been implemented in Ada 95.

This paper describes how the exceptions model of open multithreaded transactions can be implemented based on the exceptions provided by Ada 95. Section 2 summarizes the rules for open multithreaded transactions; Section 3 presents in detail the exception model of open multithreaded transactions; Section 4 reviews the exception model of Ada 95; Section 5 shows the different interfaces that the prototype implementation of te framework provides for Ada programmers; Section 6 discusses the suitability of Ada exceptions for implementing more elaborate exception models and the last section draws some conclusions.

2 Open Multithreaded Transactions

Open multithreaded transactions define a transaction model that fits the concurrency model of Ada 95. It 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 the transaction.
- A task created inside an open multithreaded transaction must also terminate inside the transaction.

Within a transaction, a set of transactional objects can be accessed by the participating tasks, and individual tasks inside a transaction collaborate by accessing the same transactional objects. The transactional objects handle two forms of concurrency, namely *competitive* (inter-transaction) and *cooperative* (intra-transaction).

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

Starting an open multithreaded transaction

- Any task can start a transaction: it will be the first joined participant of the transaction.
- 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 task can join a transaction, 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 task can join a top-level transaction if and only if it does not 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.
- A task spawned by a participant automatically becomes a spawned participant of the innermost transaction in which the spawning task participates. A spawned participant can join a nested transaction, in which case it becomes a joined participant of the nested transaction.

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.

3 Exception handling in open multithreaded transactions

In this section we discuss the exception handling mechanism developed for open multithreaded transactions. 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 exception handling mechanism used in open multithreaded transactions.

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 do not outlive the transaction, they cannot signal any external exception except `Transaction_Abort`, which results in aborting the transaction.

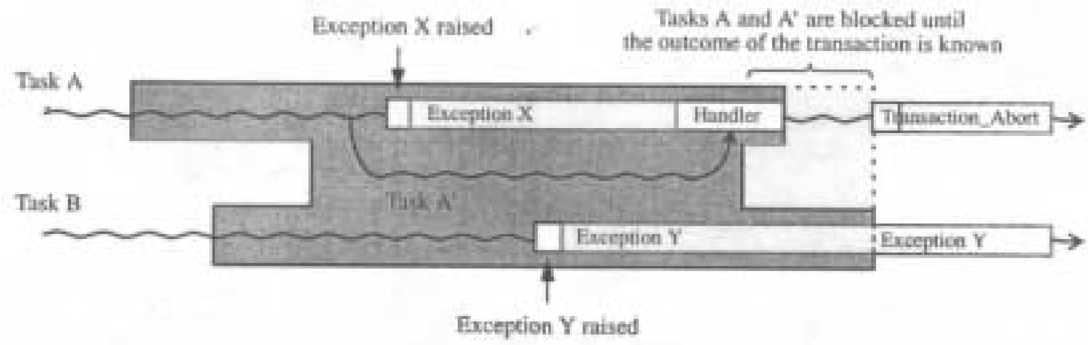

Figure 1: Exceptions in Open Multithreaded Transactions

Because the open multithreaded transaction 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.

Figure 1 illustrates exception handling in open multithreaded transactions. It depicts an open multithreaded transaction with three participant tasks. Task A starts the transaction, Task B joins it, and at some point Task A spawns Task A'. Task A' performs some work, votes *commit* and terminates. Task A generates an exception while performing its work, but the exception is handled locally. It therefore does not affect the outcome of the transaction; Task A also votes *commit*. Unfortunately Task 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 Task B; in all other joined participants the exception `Transaction_Abort` is raised.

4 Exceptions in Ada 95

Ada 83 was one of the first mainstream programming languages incorporating exceptions. Exceptions in Ada are provided to address the following situations:

- *Error conditions* — like arithmetic overflow, storage exhaustion, array-bound violations, subrange violations, peripheral timeouts, etc. When one of these situations arises, the Ada run-time notifies the application programmer by means of predefined exceptions, e.g. `Constraint_Error`, `Storage_Error`, `Communication_Error`.
- *Abnormal program conditions* — like errors in user input data, need for special algorithms to deal with singularities, or incorrect usage of abstract data types, etc. To address these situations, Ada allows the application programmer to define new exceptions. Later on, when the abnormal condition occurs, the programmer may raise an exception explicitly.

When an exception has occurred, the control is passed to a specified sequence of statements which is called the *exception handler*. Exception handlers are separated, textually, from the place at which the error is raised so that the normal behavior of the program is not obscured. The Ada model of exceptions is based on the termination model [4], and does therefore not allow for an automatic return to the point of the error from within the exception handler.

Any program block or subprogram may contain handlers that can catch exceptions raised during the execution of that block. Unfortunately, Ada does not allow to associate exceptions with subprograms, only with an entire package. This is rather surprising, given that the Ada programming language tries to reduce programming errors by enforcing strict typing rules, by providing safe language constructs, and by performing run-time checks. Java for instance allows to associate exceptions with methods. That way, the compiler can verify that the exceptions declared in a method's interface are handled in the code that calls the method. Otherwise, the exceptions must also be part of the interface of the calling method.

In Ada, unhandled exceptions in a block are propagated to the calling block. This fact, and the fact that exception names can be declared in any declarative region, makes it possible for exceptions to be propagated outside of the scope of the exception name, turning them into so-called *anonymous exceptions*. To still catch these exceptions, the optional handler "others" is provided. It guarantees to catch all exceptions raised in the block, excluding those already mentioned and therefore explicitly handled.

The Package `Ada.Exceptions`

Exceptions in Ada are not objects, and therefore it is not possible to create exception hierarchies or attach data to exceptions by declaring new attributes.

Fortunately, the package `Ada.Exceptions` provides further facilities for manipulating exceptions. It introduces the concept of an exception occurrence, and a number of subprograms to access information regarding the occurrence. Using these subprograms it is for instance possible to obtain a string representation of the exception name, even for anonymous exceptions, or to get information in form of a printable string describing the details of the cause of an exception occurrence.

The package also provides a subprogram named `Raise_Exception` that allows the programmer to attach a specific message in form of a character string to the raising of an exception. During exception handling, this message can be retrieved by calling the `Exception_Message` function. It is also possible to save or copy an exception occurrence, and then raise it again later on.

5 Transaction Support Interface for Ada 95

We want to provide transaction support at the programming language level, without modifying the language in any way. For this reason, approaches such as [6] 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 [7], the transaction framework must be accessed through a well-defined API. 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. The transaction context associated with a participant task can, for instance, be transparently associated with the task by using task attributes.

The following paragraphs present three possible interfaces for the application programmer, a *procedural* interface, an *object-based* interface, and an *object-oriented* interface.

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;`
  Calling this procedure starts a new transaction. If the calling task is already a participant of a transaction, a nested transaction is created. The new transaction ID is stored in a task attribute of the calling task.
- **procedure** `Join_Transaction (T : Task_ID);`
  This procedure allows a task to join the transaction on which the task identified by the `Task_ID` parameter is currently working on. A check is made to verify that the calling task is either a participant of the parent transaction, or is not participating in any transaction at all. Then the calling task is associated with the transaction by storing the corresponding transaction ID in the task attribute.
- **procedure** `Commit_Transaction;`
  This procedure must be called if a task wants to commit the changes that it has made on behalf of the transaction. This procedure blocks until the outcome of the transaction has been determined. If any other participant votes abort, then the transaction is aborted and the exception `Transaction_Abort` is propagated to the calling environment.
- **procedure** `Abort_Transaction;`
  Calling this procedure aborts the current transaction. Aborting a transaction does not block the calling task. In addition, participant tasks that have been suspended while attempting to commit the transaction are released.

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 is 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;
```

Object-Based Interface

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 T 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. If everything goes fine `Commit_Transaction` must be called before exiting the block.

There is no need to provide an abort operation. When T goes out of scope, the `Finalize` procedure is invoked automatically. If the participant has not previously called the commit method, then the transaction will be aborted. The advantages of this are three-fold. Firstly, every participant of a transaction is guaranteed to vote on the outcome of the transaction. If the application programmer forgets to call the commit method of the transaction object, then, following a safe approach, the transaction is aborted. Secondly, unhandled exceptions automatically cause the transaction to abort, because the `Finalize` procedure of the transaction object is invoked when the block in which the object has been declared is left. Finally, deserters, i.e. tasks disappearing without voting on the outcome of a transaction (e.g. due to ATC), can also be detected using the same mechanism, resulting in aborting the transaction.

Object-Oriented Interface

Based on the ideas of [8, 7], 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.

The advantage of the object-oriented interface is that the entire open multithreaded transaction, i.e. the program code for each participant, is grouped together inside an object. This clearly improves readability, understandability and maintainability of the transaction as a whole. Code reuse is also possible, for transactions that want to perform similar work can derive from some other transaction class, override or add new participant methods, and reuse old ones.

6 Discussion

The exception support provided by Ada 95 has proven to be sufficient to implement the exception model of open multithreaded transactions. This section mentions some small problems that have been encountered, and a "wish-list" of features that would have made the implementation simpler and the interface more elegant.

Interface Exceptions

The open multithreaded transaction model distinguishes internal and external exceptions. This difference can not be represented in Ada. In the object-oriented interface, an entire open multithreaded transaction is encapsulated inside an tagged type, and individual participants execute the primitive operations of the type in order to participate. It would have been very convenient to be able to explicitly associate exceptions with such primitive operations in order to specify for each participant the set of external exceptions that might be raised by the transaction.

Exceptions and Finalization

The object-based interface is very elegant, since it forces the application programmer to associate an Ada block, i.e. an exception context, with the transaction. A participant votes *abort* by raising an external exception, i.e. an exception that is not handled inside the Ada block associated with the transaction. The `Finalize` procedure then calls the transaction support to abort the transaction. The model states also that participants that try to commit the transaction must be informed of the abort by means of the predefined external exception `Transaction_Abort`. This is feasible, since the `Commit_Transaction` operation accesses the transaction context, realizes that the transaction has been aborted, and raises the `Transaction_Abort` exception. So far, so good.

The open multithreaded transaction model states that participants that forget to vote on the outcome of the transaction result in aborting the transaction. These participants should also be notified of the abort by means of the predefined exception `Transaction_Abort`. Unfortunately, it is not possible to raise the exception `Transaction_Abort` in this case, since raising exceptions inside of the `Finalize` procedure is considered a bounded error. Even if this was allowed, it is not possible inside the Finalize procedure to determine if there currently is an unhandled exception occurrence, i.e. a participant has raised an external exception, or if the participant has forgot to vote on the outcome of the transaction.

Exceptions and Asynchronous Transfer of Control

If an interface exception has been raised by one of the participants of an open multithreaded transaction, 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.

The only way in Ada 95 for a task to asynchronously signal another task is to use the asynchronous `select` statement. Again, supporting pre-emption requires the application programmer to follow programming guidelines, encapsulating the statements of a participant task inside an asynchronous select statement. The following code shows how this must be done when using the procedural interface:

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

As the example shows, the code gets very complicated, because internal and external exception handling must be dissociated.

It would have been very convenient if Ada provided a mechanism that allows a task to raise an exception in some other task. Of course this may give rise to scoping problems, since the exception name might not be known in both tasks. But anonymous exceptions already exist in Ada, so this might be acceptable.

7 Conclusions

The experience gained when implementing the prototype of our transaction support framework for open multithreaded transactions has shown that the exception handling features provided by Ada 95 are powerful enough to be used as building blocks for constructing a more elaborate exception handling mechanisms involving multiple tasks.

When using the procedural or object-oriented interface, the application programmer must follow programming guidelines in order to intercept exceptions that cross the transaction border as required by the model.

The fact that in Ada 95 exceptions can not be associated with subprograms or primitive operations is rather unfortunate, for it would allow a designer of an open multithreaded transaction to specify the external exceptions for each participant in a precise manner.

A feature allowing a task to raise an exception in some other task would have been useful, and might even be necessary when implementing more collaborative forms of exception handling as can be found for instance in CA actions [9].

8 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).

9 References

[1] J. Kienzle and A. Romanovsky: "Combining Tasking and Transactions, Part II: Open Multithreaded Transactions". In *Proceedings of the 10th International Real-Time Ada Workshop, Castillo de Magalia, Las Navas del Marqués, Avila, Spain, September 2000*, to be published in Ada Letters, ACM Press, 2001.

[2] J. Kienzle: "Exception Handling in Open Multithreaded Transactions". In *ECOOP Workshop on Exception Handling in Object-Oriented Systems, Cannes, France*, June 2000.

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

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

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

[6] 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*, pp. 78 – 89, Lecture Notes in Computer Science **1411**, 1998.

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

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

[9] 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.