Transaction Support for Ada

Jörg Kienzle[1], Ricardo Jiménez-Peris[2], Alexander Romanovsky[3], M. Patiño Martinez[2]

[1] Software Engineering Laboratory
Swiss Federal Institute of Technology Lausanne
CH - 1015 Lausanne EPFL
Switzerland
Joerg.Kienzle@epfl.ch

[2] Facultad de Informática
Universidad Politécnica de Madrid
E - 28660 Boadilla del Monte, Madrid
Spain
{rjimenez,mpatino}@fi.upm.es

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

Abstract. This paper describes the transaction support framework OPTIMA and its implementation for Ada 95. First, a transaction model that fits concurrent programming languages is presented. Then the design of the framework is given. Applications from many different domains can benefit from using transactions; it is therefore important to provide means to customize the framework depending on the application requirements. This flexibility is achieved by using design patterns. Class hierarchies with classes implementing standard transactional behavior are provided, but a programmer is free to extend the hierarchies by implementing application-specific functionalities. An interface for Ada programmers is presented and its use demonstrated via a simple example.

Keywords. Transactions, Open Multithreaded Transactions, OPTIMA Framework, Design Patterns, Fault-Tolerance, Ada 95.

1 Introduction

Ada [1] 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. It is not surprising that there has also been various research in the fault-tolerant area, such as providing replication for Ada 95 partitions [2].

Ada also provides support for lightweight and heavyweight concurrency (tasks and partitions). But among these active entities, concurrency control and synchronization are reduced to single method, procedure or entry calls (protected objects, rendezvous). These mechanisms are adequate to build small-scale applications where tasks and their synchronization are designed together. However, these mechanisms do no scale well (e.g. several method calls can not be executed as one atomic operation). Complex systems often need more elaborate features that can span multiple operations.

Transactions [3] are such a feature. 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. Transactions have the so-called

ACID properties: *Atomicity*, *Consistency*, *Isolation* and *Durability*. If something unexpected happens during the execution of a transaction that prevents the operation to continue, the transaction is 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.

This paper presents a framework called OPTIMA (OPen Transaction Integration for Multithreaded Applications) that provides transaction support for Ada 95. The next section introduces a new transaction model that fits our needs; section 3 outlines how an Ada programmer should interface with our transaction service; section 4 presents the design of the framework that provides support for open multithreaded transactions; section 5 presents an example program and the last section draws some conclusions and presents future work.

2 Open Multithreaded Transactions

When introducing transactions into a concurrent programming language such as the Ada language, it is important to support concurrency inside a transaction in a natural way. In Ada, a task can terminate or fork another task at any time. This section presents a new transaction model named *Open Multithreaded Transactions*. For a complete description of the model see [4]. It allows tasks to join an ongoing transaction at any time. Tasks can also be forked and terminated inside a transaction. There are only two rules that restrict task behavior:

- It is not allowed for a task that has been created outside a transaction to terminate inside the transaction.
- A task created inside a transaction must also terminate inside this transaction.

Exceptions [5], a standard Ada feature, are also integrated into the model. 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 foreseen and unforeseen errors that prevent an invoked operation from completing successfully. Exceptions are events that must be handled in order to guarantee correct results. If such a situation is not handled, the application data might be left in an inconsistent state. Aborting the transaction and thus restoring the application state to the state it had had before the beginning of the transaction will guarantee correct behavior. For this reason we have decided that unhandled exceptions crossing the transaction boundary result in aborting the open multithreaded transaction [6, 7].

The following rules describe open multithreaded transactions in detail. Tasks working on behalf of a transaction are referred to as participants. External tasks that create or join a transaction are called *joined participants*; tasks created inside a transaction by a participant are called a *spawned participants*. The data that can be modified from inside a transaction is stored in so called *transactional objects*. The transaction support guarantees the ACID properties for this data. Participants of a transaction collaborate by accessing the same transactional objects.

Starting Open Multithreaded Transactions

- Any task can start a transaction. This task will be the first joined participant of the transaction. A newly created transaction is *open*.
- Transactions can be nested. A participant of an open multithreaded transaction can start a new (nested) transaction. Sibling transactions can execute concurrently.

Joining Open Multithreaded Transactions

- A task can join a transaction as long as it is still open, thus becoming a joined participant. To do this it has to learn (at run-time) or to know (statically) the transaction context or the identity of this transaction.
- 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 first join all parent transactions. 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 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.
- Any participant of an open multithreaded transaction can decide to *close* the transaction at any time. Once the transaction is closed, no new joined participants are accepted anymore. If no participant closes the transaction explicitly, it closes once all participants have finished.

Concurrency Control in Open Multithreaded Transactions

- Participant accesses to transactional objects inside a transaction are isolated with respect to other transactions. The only visible information that might be available to the outside world is the transaction identity to be used by tasks willing to join it.
- Accesses of child transactions are isolated with respect to their parent transaction.
- Classic consistency techniques are used to guarantee consistent access to the transactional objects by participants of the same transaction.

Ending Open Multithreaded Transactions

- All transaction participants vote on the outcome of the transaction. After that they do not execute any application activity inside this transaction anymore. Possible votes are *commit* or *abort*.
- The transaction commits if and only if all participants vote 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 that all joined participants of a committing transaction exit synchronously. Only then, the changes made to transactional objects are made visible to the outside world. If the transac-

tion is aborted, the joined participants may exit asynchronously, once changes made to the transactional objects have been undone.

- If a task participating in a transaction disappears without voting on the outcome of the transaction (a deserter task), the transaction is aborted.

Exceptions and Open Multithreaded Transactions

- Each participant has a set of internal exceptions that it can handle inside the transaction and a set of external exceptions which it can signal to the outside, when needed. An additional 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.

3 Ada Interface

The support for open multithreaded transactions in Ada has been implemented in form of a library, without introducing any language changes. This approach has many advantages. It allows us to stay within the standard Ada language, hence making our approach useful for any settings and platforms which have standard Ada compilers and run-times. On the other hand, it requires the application programmer to adhere to certain programming guidelines in order to guarantee correct handling of transactions.

Our transaction support must be called at the beginning (procedure `Begin_Transaction` or `Join_Transaction`) and end of every transaction (procedure `Commit_Transaction` or `Abort_Transaction`). When a transaction is started, the call-

ing task is linked to the transaction using the package `Ada.Task_Attributes`, which offers the possibility to declare data structures for which there is a copy for each task in the system. From that moment on, the transaction support can always determine on behalf of which transaction a task is executing its operations.

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

Fig. 1: Procedural Interface

In order to correctly handle exceptions, a programmer must associate a transaction with an Ada block statement. Using the procedural interface described above, the code of a transaction must look like the code presented in figure 1. Internal exceptions can be handled in the exception section, and if the handling is successful, the transaction can still commit. Any unhandled exception crossing the block boundary will cause the transaction to abort.

To avoid forgetting to vote on the outcome of a transaction and at the same time force the programmer to declare a new block for each transaction, an interface based on controlled types can be used as shown in figure 2.

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

Fig. 2: Interface based on Controlled Types

What is important 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 starts a new transaction. The task can now work on behalf of the transaction. In order to commit the transaction, the `Commit_Transaction` procedure 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 finalized. From within the `Finalize` procedure, `Abort_Transaction` is called.

The state of an application using the transaction support will be stored in a set of data objects. Our transaction service must also be called before and after every operation invocation on such a data object. This can be automated by writing a wrapper object (a *proxy*) offering the same interface as the data object, thus transforming it into a *transactional object*. The implementations of the operations in the transactional object will call the transaction support, and only then invoke the real operations on the data object.

If writing a proxy object for each data object, and adhering to the programming conventions mentioned earlier is too much of a burden for an application programmer, she/he can opt to use Transactional Drago [8], which is an extension to Ada that introduces transactions into the language itself. One of the advantages of having linguistic support is that many programming errors can be detected at compilation time (e.g. correct nesting of transactions). Concurrency control is automatically set in Transactional Drago. Therefore, transactions are programmed just as any other piece of code. The only thing the programmer needs to do is to enclose the transaction code within a transactional block statement.

4 The OPTIMA Framework

A framework providing support for open multithreaded transactions must allow tasks working on behalf of transactions to access transactional objects in a consistent manner, guaranteeing the transactional ACID properties.

The design of the framework is a further development of our previous work described in [9]. It makes heavy use of design patterns in order to maximize modularity and flexibility. Using object-oriented programming techniques it can be easily customized and tailored to specific application needs. The design of the framework can be broken into three important components, namely *transaction support*, *concurrency control* and *recovery support*.

4.1 Transaction Support

The transaction support is responsible of keeping track of the lifetime of an open multithreaded transaction. For each transaction, it creates a transaction object that holds the transaction context containing the following data:

- The status of the transaction (open, closed, aborted, committed),
- The current number of participants, their identity and their status (joined participant, or spawned participant),
- A list of subtransactions, and a reference to the parent transaction, if there is one,
- A list of all transactional objects that have been accessed from within the open multithreaded transaction.

When a participant votes on the outcome of an open multithreaded transaction, the transaction support is notified. If a joined participant votes *commit*, and there are still other participants working on behalf of the transaction, the calling task is suspended. Only in case of an abort, or if all participants have voted commit, the transaction support passes the decision on to the recovery support.

4.2 Concurrency Control

The main objective of the concurrency control support is to handle the cooperative and competitive concurrency in open multithreaded transactions. Dealing with *competitive concurrency* comes down to guaranteeing the *isolation* property for each transaction. Transactions running concurrently are not allowed to interfere with each other; participants of a transaction access transactional objects as if they were the only tasks executing in the system. Handling *cooperative concurrency* means ensuring data *consistency* despite concurrent accesses to transactional objects by participants of the same transaction.

These problems can be solved by synchronizing the accesses to transactional objects made by tasks participating in some transaction. Providing consistency among participants of the same transaction requires that operations that update the state of the transactional object execute with mutual exclusion. Competitive concurrency control among concurrent transactions can be *pessimistic (conservative)* or *optimistic (aggressive)*, both having advantages and disadvantages. In any case, the serializability of all transactions must be guaranteed.

The principle underlying pessimistic concurrency control schemes is that, before attempting to perform an operation on any transactional object, a transaction has to get permission to do so. If a transaction invokes an operation that causes a conflict, the transaction is blocked or aborted. Read/write locks are common examples of pessimistic concurrency control.

In optimistic concurrency control schemes [10], transactions are allowed to perform conflicting operations on objects without being blocked, but when they attempt to commit, the transactions are validated to ensure that they preserve serializability. If a transaction is successfully validated, it means that it has not executed operations that conflict with the operations of other concurrent transactions. It can then commit safely.

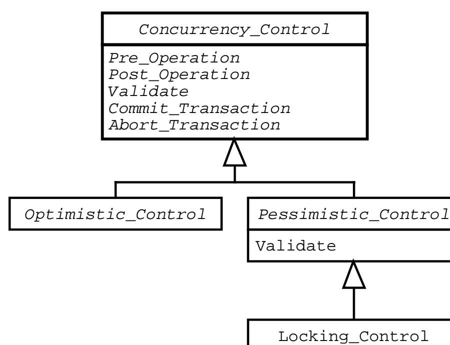

Fig. 3: Concurrency Control

The common interface for concurrency control is shown in the abstract class `Concurrency_Control` in figure 3. The `Pre_` and `Post_Operation` operations must be called before (resp. after) executing any operation on a transactional object.

A call to `Pre_Operation` comprises two phases. First, competitive concurrency must be handled. In optimistic concurrency control schemes based on timestamps for instance, `Pre_Operation` must remember the invocation time of the operation. In a pessimistic scheme based on locking, the calling task must acquire the lock in order to proceed with the operation. If the lock is not compatible with all other locks granted for this transactional object, the calling task is suspended.

The second phase deals with cooperative concurrency. In both concurrency control schemes, `Pre_Operation` will acquire the mutual exclusion lock to access the transactional object. This is needed to guarantee consistent access to the data among participants inside an open multithreaded transaction. `Post_Operation` releases the mutual exclusion lock again, but does not discard the competitive concurrency control information (i.e. discarding the timestamps, or releasing the transaction locks). In general, information about the competitive concurrency control must be kept at least until the outcome of the transaction is known.

When the transaction support is ready to commit a transaction, the `Validate` operation is called for each accessed transactional object. In optimistic concurrency control schemes, `Validate` verifies that there the serializability property has not been violated. If this has happened, the transaction will abort. For pessimistic concurrency control schemes, `Validate` always succeeds.

Optimistic and pessimistic concurrency control schemes must be able to decide if there are conflicts between operations that would compromise the serializability of transactions. They must also know if an operation modifies the state of the transactional object.

This is what the operation information hierarchy shown in figure 4 is there for. For each operation of a transactional object, an operation information object must be

written providing the operations Is_Update and Is_Compatible. Is_Update is needed for dealing with cooperative concurrency. It returns true if the operation modifies the state of the transactional object. This determines if mutual exclusion is needed among participants of the same transaction.

Is_Compatible is necessary for dealing with competitive concurrency. It must determine whether an operation conflicts with the other operations available for this transactional object with respect to transaction serializability.

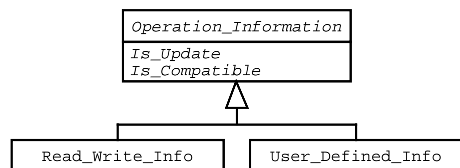

Fig. 4: Operation Information

Without knowledge of the semantics of operations, one can easily see that update operations conflict with each other, and also with read operations. Read operations however do not conflict with other read operations, since they do not modify the state of the transactional object. The operation information hierarchy contains a Read_Write_Info class that provides this behavior. Inter-transaction concurrency can be increased if one knows more about the semantics of the operation itself. For instance two invocations of an insert operation that inserts an element into a set do not conflict with each other. They *commute*. An application programmer can provide his own operation information class if she/he wants to use commutativity-based concurrency control [11].

Once the outcome of a transaction has been determined by the transaction support, the concurrency control is notified by means of the operations Commit_Transaction or Abort_Transaction. When a transaction aborts, the collected information can be discarded (timestamps, locks). When a transaction commits, the information must be passed to the parent transaction (the parent transaction "inherits" the modifications made on behalf of a nested transaction). Only when a top-level transaction commits, the information can be discarded safely.

4.3 Recovery Support

The recovery support provides open multithreaded transactions with *atomicity* and *durability* properties in spite of system failures. To achieve durability the state of transactional objects is stored on a non-volatile storage device. Atomicity means that either all modifications made on behalf of an open multithreaded transaction are reflected in the state of the accessed transactional objects, or else none is.

The recovery support must therefore keep track of all the modifications that the participants have made to transactional objects on behalf of a transaction. In order to recover from a crash failure, this information, also called a transaction *trace*, must be stored on some kind of storage, called a *log*, that will not be affected by the system failure. That way, once the system restarts, the recovery support can consult the log and perform the cleanup actions necessary to restore the system to a consistent state. The information that must be written to the log depends on the chosen recovery strategy, and the necessary cleanup actions depend on the strategy, the status of the transaction and whether the modifications made to the transactional objects have already been propagated from the cache to the non-volatile storage unit or not.

The essential components of the recovery support are the *Persistence_Support*, the *Cache_Manager*, the *Recovery_Manager* and the *Log_Manager*. The following subsections present these components in more detail.

4.3.1 Persistence Support

The persistence support provides three basic functionalities:

- It provides a device independent interface that is used by the cache manager to store the state of transactional objects on some non-volatile storage.
- It provides stable storage (not affected by crash failures) to the log manager.
- It provides a means for identifying transactional objects stored on some storage device which is independent of the actual device used.

The implementation is based on the persistence support presented in [12].

4.3.2 The Cache Manager

The state of transactional objects is kept in main memory in order to improve the performance of the overall system, since accessing memory is in general significantly faster than accessing non-volatile storage. However, in systems that are composed of lots of transactional objects, it is often not possible to keep the state of all objects in memory at a given time, and therefore it is sometimes necessary to replace objects that are already in the cache.

In a conventional cache, any object can be chosen. The situation for a cache used in a transaction system is more complicated [13]. Firstly, we distinguish between *Steal* and *No-Steal* policy. In the *Steal* policy, objects modified by a transaction may be propagated to the storage at any time, whereas in the *No-Steal* policy, modified objects are kept in the cache at least until the commitment of the modifying transaction. We also make a distinction on what happens during transaction commit. In the *Force* policy, all objects that a transaction has modified are propagated to their associated storage unit during the commit processing, whereas in the *No-Force* policy no propagation is initiated upon transaction commit.

In practice, caches are very effective because of the *principle of locality*, which is an empirical observation that, most of the time, the information in use is either the same information that was recently in use (temporal locality), or is information “nearby” the information recently used (spacial locality). The behavior of caches can be tailored in order to get a better hit ratio, i.e. by adjusting the size of the cache, or by choosing appropriate fetch and replacement algorithms. It is therefore important for our framework to allow a user to define his own cache policy.

Again this flexibility is achieved by providing an abstract root class `Cache_Manager`, with abstract methods such as `Apply_Replacement_Policy`. An application programmer can choose the appropriate cache policy depending

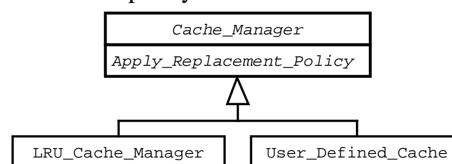

Fig. 5: The Cache Manager Hierarchy

on the application requirements. She/he can also extend the cache manager hierarchy, providing his own implementation as shown in figure 5.

Although introducing a cache is completely transparent for the users of the transaction support, it significantly complicates the reasoning about the consistency of the state of the system. When using a cache, the current state of a transactional object is determined by the state of the object in the cache, or if it is not present in the cache, by the state of the object on the storage. When a transaction aborts, the state changes made on behalf of the transaction are undone in the cache. It might be that these changes have already been propagated to the storage. However, we do not have to undo them, since they will be undone the next time we update the state of the object on the storage. When a transaction commits, we must ensure that at some time in the future, the changes of the transaction will be propagated to the associated storage unit.

Using a cache has a significant impact on the actions that must be taken when recovering from a crash failure. On a system crash, the content of the cache is lost, and therefore, in general, the state of the objects on the storage can be inconsistent for the following reasons:

- The storage does not contain updates of committed transactions.
- The storage contains updates of uncommitted transactions.

When recovering from a system crash, these situations must be remedied. The former problem can be solved by *redoing* the changes made by the corresponding transactions, the latter by *undoing* the changes made by the corresponding transactions. Depending on the cache policy, undo, redo or both are necessary [14].

Undo/Redo
The *Undo/Redo* recovery protocol requires both undo and redo actions and allows great flexibility in the management of the cache by permitting *Steal* and *No-Force* object replacement policies. It maximizes efficiency during normal operation at the expense of less efficient recovery processing.

Undo/No-Redo
The *Undo/No-Redo* recovery protocol requires undo but never redo actions by ensuring that all the updates of committed transactions are reflected in the storage. It therefore relies on *Steal* and *Force* object replacement policies. The commitment of a transaction is delayed until all its updates are recorded on non-volatile storage.

No-Undo/Redo
The *No-Undo/Redo* recovery protocol, also known as *logging with deferred updates*, never requires undo actions, but relies on redo actions. Updates of active transactions are not propagated to the storage, but recorded in the system log, either in the form of an after-image of the state or as a list of invoked operations, also called an *intention list*.

4.3.3 The Log Manager

The system log is a sequential storage area located on stable storage. It is important that the log is stored on stable storage, since it must always remain readable even in the presence of failures in order to guarantee the correct functioning of the transaction sys-

tem. The purpose of the log is to store the information necessary to reconstruct a consistent state of the system in case a transaction aborts or a system crash occurs. The required information can be split into 3 categories:

- Undo Information
- Redo Information
- Transaction Status Information

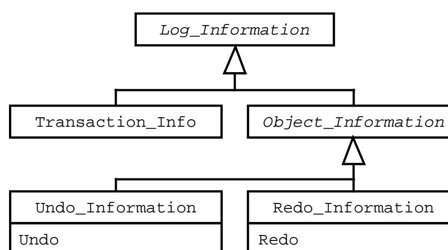

Fig. 6: Log Information

This information is organized in a hierarchy as shown in figure 6.

There are two situations in which the log must be updated:

- A transaction is committed or aborted.
- An operation that modifies the state of a transactional object is invoked.

Undo and redo information can be stored in the log in two ways. In the first technique, called *physical logging* [3], copies of the state of a transactional object are stored in the log. These copies are called *before-images* or *after-images*, depending on if the snapshot of the state of the object has been taken before or after invoking the operation. Unfortunately, physical logging only works if read/write concurrency control is used. If semantic-based concurrency control such as commutative locking schemes are used, undo and redo information must be saved using *logical logging*. In this technique, the operation invocations and their parameters are written to the log. In order to support undo, every operation op of a transactional object must provide an *inverse operation* op^{-1}, i.e. an operation that undoes the effects of calling op.

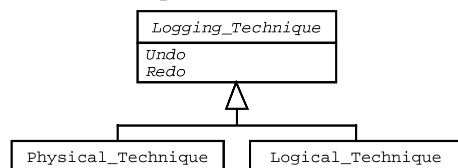

Fig. 7: Logging Techniques

These two logging techniques are captured in the class hierarchy presented in figure 7.

After a system crash, the entire log needs to be scanned in order to perform all the required redo and undo actions. What algorithm must be used to recover from a crash, depends on the chosen recovery strategy and is detailed in [13]. The same paper also shows how to prevent the log from growing too long by using a technique called *checkpointing*.

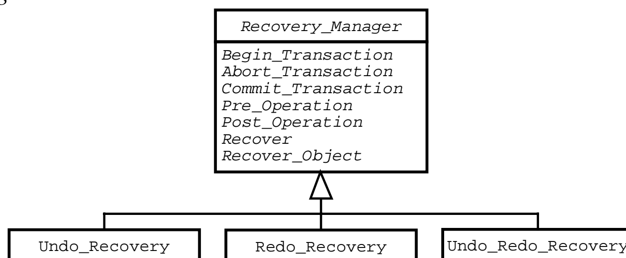

Fig. 8: The Recovery Manager Hierarchy

4.3.4 The Recovery Manager

The *Recovery_Manager* implements the recovery strategy, and therefore controls the *Cache_Manager* and the *Log_Manager*. Again, a class hierarchy of recovery managers is provided to the application programmer. She/he can select the most appropriate recovery strategy for his application by instantiating one of the concrete recovery managers shown in figure 8.

5 Auction System

This section describes how open multithreaded transactions can be used to structure an electronic auction system. The general idea of such a system is the following: First, a user must register with the system by providing a username and password, and deposit some money, most of the time in form of a credit card number. From then on, the user can login and consult the current auctions. He can decide to bid for items he wants to acquire, or sell items by starting new auctions.

Such an auction system is an example of a dynamic system with cooperative and competitive concurrency. The concurrency originates from the multiple connected users, that each may participate in or initiate multiple auctions simultaneously. Inside an auction, the users cooperate by bidding for the item on sale. On the outside, the auctions compete for external resources such as the user accounts. The system is dynamic, since a user must be able to join an ongoing auction at any time. An additional non-functional requirement of such a system is that it must be fault-tolerant, e.g. money transfer from one account to the other should not be executed partially, even in the presence of crash failures.

All these requirements can be met if an individual auction is encapsulated inside an open multithreaded transaction. The seller starts the auction, the individual bidders join it if they want to place a bid.

Implementation in Ada

Figure 9 shows how the main program must initialize the transaction support. The package `Object_Based_Transaction_Interface` defines the object-based interface to the transaction support as mentioned in section 3. The other context clauses define the transactional objects that are used in this example, namely auctions and accounts.

Before starting a transaction, the programmer must call the `System_Init` procedure. This procedure can take additional parameters that allow the application programmer to customize the transaction support according to his needs. In particular, a recovery manager, a cache manager, and a

```
with Object_Based_Transaction_Interface;
use Object_Based_Transaction_Interface;
with Transactional_Auctions;
use Transactional_Auctions;
with Transactional_Accounts;
use Transactional_Accounts;
with File_Storage_Params;
use File_Storage_Params;
procedure Main is
   -- Task and other declarations
begin
   System_Init;
   -- Run auctions
   System_Shutdown;
exception
   when others =>
      System_Shutdown;
end;
```

Fig. 9: System Initialization

storage unit to be used to store the log can be specified. The default implementation chooses a LRU cache manager, a Redo/NoUndo recovery manager, and a mirrored file for storing the log.

The package `Transactional_Auct-ions` defines the auction objects, with the operations `Create_Auction`, `Restore_ Auction`, `Get_Current_Bid`, `Accept_Bid`, `Finished` and `Bid_Accepted`. The package `Transactional_Accounts` defines the account objects, with the usual operations `Deposit`, `Withdraw` and `Get_Balance`. Figure 10 shows how the seller tasks and bidder tasks make use of these objects when performing an auction.

The seller task begins its work by declaring a reference to a transactional auction, and then asks the user to input the duration of the auction. Next, a new open multi-threaded transaction named "Auction" is started by declaring a `Transaction` object; the scope of the transaction is linked to the scope of this object. Inside the transaction, a new auction object is created by calling the `Create_Auction` operation. The `Storage_Params_To_String` function of the `File_Storage_Params` package is called in order to tell the system to store the state of the auction object in the file "Auction.file". Then, the seller task suspends itself for the duration previously specified by the user.

```
task body Seller_Task is
   Auction : Transactional_Auction_Ref;
   Auction_Time : Duration := Ask_User;
   Auction_Transaction : Transaction
      (new String' ("Auction"));
   Current_Bid : Natural;
begin
   Auction := Create_Auction (String_To_
      Storage_Params ("Auction.file"),
      -- + initialization parameters,
      -- e.g. description, minimum bid...
   delay Auction_Time;
   Current_Bid :=
      Get_Current_Bid (Auction.all);
   if Current_Bid >= Minimum_Bid then
      Accept_Bid (Auction.all);
      declare
         My_Account : Transactional_
           Account_Ref := Restore_
           Account (String_To_Storage_
           Params ("Seller_Acc.file"));
      begin
         Deposit
           (My_Account.all, Current_Bid);
      end;
   end if;
   Commit_Transaction
      (Auction_Transaction);
end Seller_Task;
```

```
task body Bidder_Task is
   Auction_Transaction : Transaction
      (new String' ("Auction"));
   Auction : Transactional_Auction_Ref
      := Get_Auction;
   Current_Bid : Natural;
   My_Bid : Natural;
begin
   while not Finished (Auction.all) loop
      Current_Bid :=
         Get_Current_Bid (Auction.all);
      My_Bid := Ask_User (Current_Bid);
      Bid (Auction.all, My_Bid);
   end loop;
   if Bid_Accepted (Auction.all) then
      declare
         My_Account : Transactional_
         Account_Ref := Restore_Account
         (Storage_Params_To_String
         ("Bidder_Acc.file"));
      begin
         Withdraw
            (My_Account.all, My_Bid);
      end;
   end if;
   Commit_Transaction
      (Auction_Transaction);
end Bidder_Task;
```

Fig. 10: Seller and Bidder Tasks

The bidder tasks join the ongoing transaction by also declaring a transaction object. This time, the object is initialized with a call to `Join_Transaction`. The bidder task then obtains the current bid from the auction object, and asks the user for his new bid. Next, the `Bid` operation of the auction object is invoked. Other bidder tasks will see the

new bid, for they are all participants of the same transaction, and, in turn, are allowed to place new bids. When the time-out expires, the seller task accepts the current bid, deposits the money on his account and commits the transaction. The bidders leave the loop and the bidder that won the auction withdraws the money from its account. The open multithreaded transaction ends once the last bidder calls `Commit_Transaction`.

If any unforeseen exception is raised inside the transaction, the transaction is aborted and the predefined external exception `Transaction_Abort` is raised in all other participants. The abort results in undoing the deposit and withdrawal, and in deleting the created auction object. This also holds in case of a crash failure.

6 Conclusions and Future Work

This paper has presented the design of the OPTIMA framework supporting *Open Multithreaded Transactions*, a transaction model that does not restrict the concurrency features found in Ada, but still keeps the tasks participating in a transaction under control, enforcing the ACID properties. The framework makes heavy use of design patterns in order to maximize modularity and flexibility. Using object-oriented programming techniques it can be easily customized and tailored to specific application needs. A more detailed description of the OPTIMA framework can be found in [15].

Interfaces for the Ada programmer have been laid out. The framework has been implemented in form of a library based on standard Ada only. This makes our approach useful for any settings and platforms which have standard Ada compilers.

The paper also describes parts of an implementation of an auction system based on open multithreaded transactions. This case study has shown the benefits of using open multithreaded transactions for system structuring and fault-tolerance in concurrent systems.

The open multithreaded transaction model does support distribution, but the current framework only addresses single nodes. In the future, we intend to add distribution to our framework. This has an impact on the transaction support component, which must be extended to provide distributed transaction control (i.e. two phase commit protocol), and on the cache manager, which must provide distributed access to transactional objects. Another promising direction of the research is to try and integrate the transaction support with the CORBA Object Transaction Service [16]. Using our transaction support an application programmer can easily implement a transactional CORBA resource. Our intentions are to provide a bridge that will intercept calls to the `prepare`, `rollback` and `commit` methods of CORBA resources and forward them to our transaction support.

7 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). Ricardo Jimenez-Peris and Marta Patino-Martinez have been partially supported by the Spanish Research Council (CICYT) grant #TIC98-1032-C03-01 and by the Madrid Research Council (CAM) grant #CAM-07T/0012/1998.

References

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

[2] Wolf, T.; Strohmeier, A.: "Fault Tolerance by Transparent Replication for Distributed Ada 95". In Harbour, M. G.; de la Puente, J. A. (Eds.), *Ada-Europe'99*, pp. 411 – 424, Lecture Notes in Computer Science **1622**, 1999.

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

[4] Kienzle, J.; Romanovsky, A.: "Combining Tasking and Transactions, Part II: Open Multithreaded Transactions". *10th International Real-Time Ada Workshop, Castillo de Magalia, Spain*, to be published in Ada Letters, ACM Press, 2001.

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

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

[7] Patiño-Martinez, M.; Jiménez-Peris, R.; Arevalo, S.: "Exception Handling in Transactional Object Groups". In *Advances in Exception Handling Techniques*, Lecture Notes in Computer Science **2022**, Springer Verlag, 2001.

[8] Patiño-Martinez, M.; Jiménez-Peris, R.; Arevalo, S.: "Integrating Groups and Transactions: A Fault-Tolerant Extension of Ada". *Reliable Software Technologies - Ada-Europe'98*, pp. 78 – 89, Lecture Notes in Computer Science **1411**, 1998.

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

[10] Kung, H. T.; Robinson, J. T.: "On Optimistic Methods for Concurrency Control". *ACM Transactions on Database Systems **6**(2)*, pp. 213 – 226, June 1981.

[11] García-Molina, H.: "Using Semantic Knowledge for Transaction Processing in a Distributed Database". *ACM Transactions on Database Systems **8**(2)*, pp. 186 – 213, June 1983.

[12] Kienzle, J.; Romanovsky, A.: "On Persistent and Reliable Streaming in Ada". In Keller, H. B.; Plöderer, E. (Eds.), *Reliable Software Technologies - Ada-Europe'2000*, pp. 82 – 95, Lecture Notes in Computer Science **1845**, 2000.

[13] Haerder, T.; Reuter, A.: "Principles of Transaction Oriented Database Recovery". *ACM Computing Surveys **15**(4)*, pp. 287 – 317, December 1983.

[14] Bernstein, P. A.; Goodman, N.: "Concurrency Control in Distributed Database Systems". *ACM Computing Surveys **13**(2)*, pp. 185 – 221, June 1981.

[15] Kienzle, J.: *Open Multithreaded Transactions: A Transaction Model for Concurrent Object-Oriented Programming*. Ph.D. Thesis, Swiss Federal Institute of Technology Lausanne, Switzerland, April 2001, to be published.

[16] Object Management Group, Inc.: *Object Transaction Service,* May 2000.