TransLib: An Ada 95 Object Oriented Framework for Building Transactional Applications[*]

R. Jiménez Peris[†], M. Patiño Martínez[†], S. Arévalo[‡⋆] and F. J. Ballesteros[‡⋄]

[†] Facultad de Informática
Universidad Politécnica de Madrid
E-28660 Boadilla del Monte, Madrid, Spain
{rjimenez, mpatino}@fi.upm.es
[‡] Escuela de Ciencias Experimentales
Universidad Rey Juan Carlos
E-28933 Mostoles, Madrid, Spain
[⋆] s.arevalo@escet.urjc.es
[⋄] nemo@choices.cs.uiuc.edu

This article has been accepted for publication in the
Special Issue on Developing Fault-Tolerant Systems with Ada 95 of the
Int. Journal on Computer Systems: Science and Engineering. 1999.

Abstract

TransLib is an Ada 95 object oriented framework to program fault-tolerant applications, more concretely, it allows to program transactional distributed applications. Transactions are one of the most widely used fault-tolerance mechanisms. They provide data consistency in the presence of failures and concurrent activities.

One of the novelties of *TransLib* is that it uncouples concurrency control and recovery code from the code of data objects used by transactions. This feature enables to use regular objects in a transactional setting and vice versa what greatly improves reusability. *TransLib* provides commutativity-based locking, and different kinds of recovery. It also allows users to define their own recovery and concurrency control algorithms.

Transactions and exceptions have been integrated in *TransLib*, that is, backward and forward recovery. In this integration, exceptions that cross transaction boundaries cause transaction abortion and transaction abortions are notified as exceptions.

TransLib implements a new transactional model, *Group Transactions*, which integrates two existing fault tolerance techniques: transactions and process groups. In this transactional model transactions can be multitask and/or multi-process.

A set of design patterns documents *TransLib*. Each one describes a component of *TransLib* architecture, such as recovery and concurrency control mechanisms.

In this paper we describe the overall architecture of *TransLib*, as well as the design patterns that document it. Additionally, some aspects of the Ada 95 implementation are also discussed.

Keywords: Transactions, fault-tolerant architectures, backward and forward recovery, object oriented design patterns, distributed systems.

1 Introduction

Transactions are a mechanism to provide data consistency in the presence of concurrent activities and system failures. Transaction systems were first developed for banking applications, but today

[*]This research has been partially funded by the Spanish National Research Council CICYT under grant TIC98-1032-C03-01 and by the Madrid Regional Research Council CAM under grant 07T/0012/1998.

they are widely used in many areas to build fault-tolerant applications [1] such as communications, finance, flight reservations systems, electronic commerce, manufacturing, ...

Transactions have four properties, known as ACID properties, which are useful for constructing fault-tolerant applications:

1. *Atomicity* guarantees that the effect of a transaction is all or nothing. If a transaction does not end successfully it is aborted and its effects are undone.

2. *Consistency* ensures a correct transformation of the state. It is responsibility of the application programmer to guarantee this property.

3. *Isolation* (serializability) ensures that concurrent transactions will not interfere. That is that the effect of concurrent transactions will be equivalent to some serial execution.

4. *Durability* ensures that once a transaction has finished successfully (committed) its effects will remain. In other words, its state changes will survive failures.

Concurrency control mechanisms have been proposed to deal with isolation. Locking is one of the most well known concurrency control mechanisms. Recovery mechanisms preserve atomicity and durability in the presence of failures. Logging is generally used to support these mechanisms.

Transactions involving data located in different nodes in a network are called distributed transactions. Distributed atomic commit protocols are used to ensure the atomicity of a distributed transaction.

Transactions can be nested, that is, a transaction can be started within another transaction. Top level transactions are those not enclosed within other transactions. Nested transactions [2] or subtransactions are useful for two reasons. First, they allow additional concurrency within a transaction (particularly in distributed systems) by running concurrently nested independent subtransactions. Second, the failure of a subtransaction does not force the parent transaction to fail, providing in this manner a kind of firewall against failures. However, the concurrency allowed inside a transaction is very limited, as concurrent subtransactions cannot cooperate due to the isolation property.

In this paper we present *TransLib*, an Ada 95 object oriented framework to build distributed transactional applications. *TransLib* is composed of a set of design patterns that describes its architecture and, at the same time, documents how to use *TransLib*. The two main design patterns of *TransLib* are *TransLock* for concurrency control and *TransRecovery* for recovery.

TransLib provides commutativity-based concurrency control [3] which allows more concurrency than the traditional read/write locking, as well as several predefined recovery mechanisms. One of the contributions of *TransLib* is to allow the redefinition of recovery and concurrency control mechanisms. This feature allows customizing the framework to the application needs. Recovery and concurrency control mechanisms in traditional systems are usually wired, and cannot be changed with the exception of Arjuna [4].

Another contribution of *TransLib* is that there is no distinction between transactional objects and non-transactional ones. This feature allows object reuse in different setups (maybe non-transactional), and it also makes possible to change the concurrency control and recovery policies without modifying transactional objects, that is, they can be varied independently.

TransLib integrates backward and forward recovery, more specifically, transactions and exceptions. In this integration [5], unhandled exceptions propagated outside of a transaction cause the transaction abortion and transaction abortions are notified as exceptions. That is, a transaction that ends exceptionally is aborted. The enclosing scope of an aborted transaction is notified by means of the exception that caused its abortion.

TransLib implements a new transactional model, *Group Transactions*, that integrates the nested transactions and process groups paradigms. In this model, transactions can be multi-task and/or multi-process and a transactional server can be a group of processes. Nested transactions are also supported, as they are a particular case of *Group Transactions*.

As transactions in *Group Transactions* can be composed of several tasks and/or processes, exceptions can be raised concurrently within a transaction, so exception resolution is used. A novelty of the exception resolution provided by *TransLib* is that it is structured hierarchically, applying local resolution among the local exceptions, and then applying distributed resolution among distributed ones.

TransLib restricts the use of some Ada 95 statements. In particular it restricts the use of task abortions and asynchronous transfer of control. Those statements can prevent the execution of *TransLib* code necessary for the correct behaviour of the system. For instance, the task `abort` can prevent the abortion of a transaction.

There is some work related to the implementation of fault-tolerance mechanisms in Ada 95, like atomic actions [6, 7] and recovery blocks [8]. Some of the most important transactional systems are Argus [9], Camelot/Avalon [10], and Arjuna [4, 11]. However, none of them provide the flexibility and adaptability of *TransLib*. There has also been work on transactions with Ada. In particular [12] presents an ad-hoc Ada implementation of transactions for air-traffic control systems. A project with similar goals than ours in an earlier stage can be found in [13].

The paper is structured as follows. Transactional concurrency control and recovery support are presented in Sections 2 and 3 respectively. Section 4 introduces the *Group Transactions* model. Section 5 describes the exception model used in *TransLib*. We have included a subsection discussing the use of Ada 95 advanced features at the end of Sections 2, 3 and 5. Finally, we present a comparison with related work and our conclusions.

2 *TransLock*: A Design Pattern for Transactional Concurrency

Concurrency control mechanisms guarantee the isolation property of transactions, that is, the final effect of executing a set of concurrent transactions is as if they were executed sequentially in some order. This property is also called serializability.

Concurrency control mechanisms are classified as optimistic or pessimistic. Optimistic mechanisms allow transactions to run even if they conflict. At commitment time, if there were conflicting accesses, conflicting transactions are aborted. On the other hand, pessimistic mechanisms delay the execution of conflicting transactions. This yields to another problem, circular waits or deadlocks that must be broken aborting one of the transactions in the wait cycle. As optimistic mechanisms can yield to many abortions, they are not widely used.

The most used pessimistic concurrency control method is locking. Whenever a transaction accesses a datum, it has to request a lock in the appropriate mode. If the requested lock conflicts with other lock held by a different transaction, the requester transaction will be delayed until the end of the holder transaction.

There are several schemes of locking, the most well-known are read/write locking, commutativity based locking and recoverability. Read/write locking was the first proposed scheme, and it is the less concurrent of the three. It provides exclusive writes and concurrent reads. The other two schemes are based on user-defined locks to provide more concurrency. In commutativity-based locking [3] two operations conflict only if they do not commute. Read/write locking is a special case of commutativity-based locking. Recoverability [14] provides yet even more concurrency, as two operations only conflict, if the second depends on the result of the first. However, recoverability increases dramatically the probability of deadlocks. This is why we have chosen commutativity-based locking as predefined concurrency mechanism in *TransLib*.

To illustrate commutativity-based locking, let's study an example: user-defined concurrency control for the set abstract data type (ADT). A set provides operations to insert and remove items, and also an operation to test set membership. Insertions commute between themselves. The same applies for removals. Test membership commutes with insertions and removals, if the test refers to an element different from the one of the update operation. Insertions and removals commute when they have different arguments. The compatibility rules for the set ADT are summarized in Fig. 1.

If we compare the compatibility table just defined with the one of read/write locks shown in

	Insert(n2)	Remove(n2)	IsIn(n2)
Insert (n1)	Yes	n1 $\neq$ n2	n1 $\neq$ n2
Remove(n1)	n1 $\neq$ n2	Yes	n1 $\neq$ n2
IsIn(n1)	n1 $\neq$ n2	n1 $\neq$ n2	Yes

Figure 1: Commutativity table for the set ADT

	Read	Write
Read	Yes	No
Write	No	No

Figure 2: Commutativity table of read/write locks

Fig. 2, it can be observed that more concurrency is achieved with the former locking policy. In particular, using read/write locking, any insertion or removal would be incompatible with any other operation, and membership tests would be just compatible among themselves.

TransLock is a design pattern for transactional concurrency control that provides user defined locks based on operation commutativity. Read/write locking is also provided as a predefined subclass. Participant classes in *TransLock* are the class hierarchies `Lock` and `LockManager`. To add a new kind of lock, a new concrete subclass of `Lock` must be defined, defining a new lock compatibility function (`IsCompatible` in Fig. 3). The `LockManager` class provided by *TransLib* is general in the sense that it can deal with any user defined commutative lock.

2.1 User-Defined Locks: The `Lock` Hierarchy

The `Lock` abstract class provides support for commutative locking. Its methods represent the common information to all user defined commutative locks, as well as a minimum information needed for recovery management. That is, the recovery manager will need to know whether a lock implies the modification of the data object or not in order to take the appropriate recovery actions. On the other hand, the lock manager will need to know if two locks are compatible when they are requested by concurrent transactions.

Each subclass of the abstract `Lock` class implements a locking policy and it must provide constructors (possibly parameterized as in the example of Fig. 1) for each kind of lock, a lock compatibility definition by means of the `IsCompatible` method, as well as the `IsUpdate` function. Instances of concrete `Lock` subclasses are locks and they will be created with the corresponding constructor.

The `ReadWriteLock` class, shown in Figure 3, implements read/write locks, so there are just two constructors: `ReadLock` and `WriteLock`. One for each kind of lock (e.g. a read lock will be created by calling `ReadLock`). The `IsCompatible` method will return true only when both the lock held and the lock requested are read locks. The `IsUpdate` method returns true for a write lock, and false otherwise.

Adding user defined concurrency control to the set ADT presented above is done writing a new concrete `Lock` subclass, `SetLock`. In this case, it is necessary to parameterize locks to be able do decide whether two operations commute. The constructors are:

```
Insert(Item) -> SetLock
Remove(Item) -> SetLock
IsIn(Item) -> SetLock
```

The `IsCompatible` operation will follow the compatibility table shown in Fig. 1. The `IsUpdate` method will return true for the two first kinds of locks, and false for the last one.

Our `Lock` class hierarchy is quite similar to the one of Arjuna [15], especially in what respects the `Lock` class interface. Despite this similarity, there are some important differences between

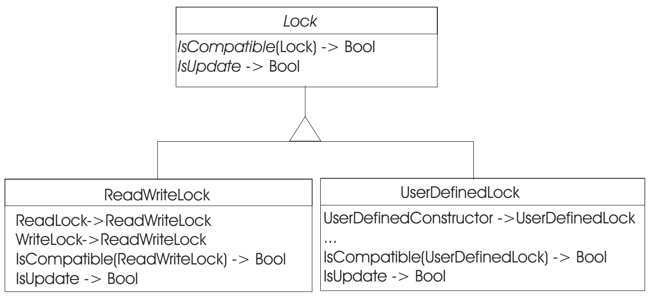
Lock
IsCompatible(Lock) -> Bool
IsUpdate -> Bool
ReadWriteLock
ReadLock->ReadWriteLock
WriteLock->ReadWriteLock
IsCompatible(ReadWriteLock) -> Bool
IsUpdate -> Bool
UserDefinedLock
UserDefinedConstructor ->UserDefinedLock
...
IsCompatible(UserDefinedLock) -> Bool
IsUpdate -> Bool

Figure 3: `Lock` Class Hierarchy

our approach and the one of Arjuna. First of all, in Arjuna lock acquisition is done explicitly in every object method. In our approach locks are acquired automatically by the system and thus object code does not include any concurrency control code. Secondly, Arjuna uses physical logging (logging is introduced in Section 3) what excludes the possibility of concurrent updates, and thus it allows less concurrency than in our approach.

2.2 The class `LockManager`

This class encapsulates the lock manager. The lock manager guarantees transaction isolation (i.e. serializability) by deciding the order in which the operations on data are executed. It currently provides commutativity-based locking and deals with any user-defined lock (instances of `Lock` subclasses). In *TransLib* every data object has an associated `LockManager` that controls its access.

The lock manager deals with both long and short term concurrency control. Locks provide long term concurrency control and are used to provide transaction isolation, that is, they provide logical consistency in presence of concurrent transactions. On the other hand, latches [16] provide short term concurrency control and are used to guarantee data physical consistency (by means of read/write mutual exclusion) in presence of compatible and concurrent write/write or read/write operations. An additional reason for latches in *TransLib* has to do with the fact that transactions can be multithreaded and locks requested from threads of the same transaction do not conflict, so they also need the latches.

The `SetLockAndMutex` method request the lock in the appropriate mode, once the lock is granted, the mutual exclusion is requested and when it is granted control is returned. Due to their different duration, they are freed at different times, and two different methods are needed. The `FreeMutex` method frees mutual exclusion (latch). However, freeing locks is a little more involved. Transaction abortion and top-level transaction commitment free all transaction locks. However, subtransaction commitment implies the propagation of its locks to the parent transaction. The lock manager is notified of transaction ends by means of the `Commit` and `Abort` methods in order to release or propagate locks. Both methods know if the caller is running a top-level transaction or a subtransaction and they take the appropriate actions.

The lock manager blocks transactions that have requested conflictive locks on the object it protects. When a lock is released or propagated, the lock manager unblocks those transactions that can continue their execution.

Another activity of the `LockManager` is the initiation of deadlock detection and keeping enough information to detect them. Detection is only started when a transaction blocks due to the request of a conflictive lock. Deadlock detection for nested transactions is more complex than for flat ones. This is due to the commit dependencies between parent and children transactions. In *TransLib* we have extended the algorithm proposed in [17] to adapt it to *Group Transactions*. This extension was needed because the original algorithm is aimed to single-threaded transactions while *Group Transactions* can be multithreaded/multi-process.

In order to provide further flexibility, an abstract `LockManager` class has been defined. The commutativity-based lock manager described above is a concrete subclass of `LockManager`, `Commu-`

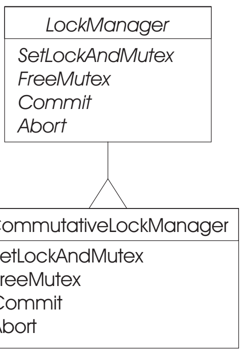

Figure 4: `LockManager` Class Hierarchy

`tativeLockManager`. Thus, it is possible to add other locking mechanisms such as recoverability, encapsulating them into concrete `LockManager` subclasses.

2.3 Ada 95 Features for Transactional Concurrency Control

The class `Lock` has been implemented as an abstract tagged type with two abstract methods, which has been useful to implement polymorphic lists of locks. In particular, the lock manager keeps two of these lists: one with the held locks and another with the conflictive locks requested.

Each concrete `LockManager` contains a protected object, `ProtectedLockManager`, that manages short and long term concurrency control of the associated data object. It also behaves as a monitor that manages concurrent accesses to its internal data structures (e.g. the list of granted locks). A concrete `LockManager` just propagates calls to its protected object.

The interface of the protected object used in the implementation of the lock manager regarding to lock requests, propagations, and releases is the following:

```
protected type ProtectedLockManager is
   entry SetLockAndMutex(lock_to_set : in out LockPtr );
   entry FreeMutex;
   entry T_Commit(tid : in out TransactionIdentifier );
   entry T_Abort(tid : in out TransactionIdentifier );
private
   -- blocked on an incompatible lock
   entry WaitingForLock(lock_to_set : in out LockPtr);
   -- read locks obtained waiting for object operation mutex

   entry WaitingForReading(lock_to_set : in out LockPtr);
   -- write lock obtained waiting for object operation mutex
   entry WaitingForWriting(lock_to_set : in out LockPtr);

   locks_held : LockList;
   ...
end ProtectedLockManager;
```

When a lock cannot be granted the current call to the `SetLockAndMutex` entry is requeued into the `WaitingForLock` entry. However, when a lock is granted, the requester can be delayed until the mutual exclusion is free. When this happens, the call is requeued into `WaitingForReading` or `WaitingForWriting` depending on whether it is waiting for a reading or writing mutex. These three entries are only used by the lock manager to delay requests, so they are kept private. The protected object also encapsulates the list of granted locks, `locks_held`, and the corresponding transaction identifiers (tids). To check whether a lock can be granted the lock manager traverses the list of granted locks and compares each granted lock to the requested lock by means of the `IsCompatible` method of the `Lock` class.

3 *TransRecovery*: Transactional Recovery

Recovery algorithms provide atomicity and durability. A *log* is an append-only linear structure that maintains redundant information to guarantee data recoverability. Transaction initiation and finalization events, together with persistent data updates are logged. Updates can be logged as either the *before image* of the data (the value before the update) or the *after image* of the data (the value after the update), or both. Lost committed updates on persistent data should be redone after a crash, whilst uncommitted updates should be undone after a crash or a transaction abortion.

The most extended method of logging is *physical logging* that logs data images. The main drawback of this method is that the log grows excessively with big objects. Another variant of this method stores image differences, but they can still be too big when the updates involve many fields of the objects.

An alternative method, more space efficient, is *logical logging.* In this method the update operations (operation name and parameters) that have modified the objects are logged. An important advantage of this method is that to redo or undo the operation the state of the object does not need to be physically identical, but just logically equivalent. Logical logging still needs object before images, so it must be used in combination with physical logging.

In order to reduce the number of disk accesses, transactional systems use a cache that keeps a copy of recently used data in volatile memory. When a transaction is going to access a datum, if it is not already in memory, the cache manager brings it from disk to memory. As all data cannot be kept in volatile memory, sometimes it is necessary to free cache slots and as a result objects are propagated to disk.

Whatever replacement strategy the cache manager uses, there are times when cache actions are restricted or forced by the recovery manager. [18] classifies recovery strategies as Undo/No Redo, No Undo/Redo and Undo/Redo depending on the restrictions imposed by the recovery manager to the cache manager. Undo means that dirty objects (i.e. objects with uncommitted updates) can be propagated to disk at any time. Then, if the transaction aborts the uncommitted values written to disk must be undone. Redo means that transaction updates will not be necessarily propagated at commitment time. So, if there is a crash, there will be committed values not reflected in disk, thus the updates must be redone during recovery. The most flexible method is Undo/Redo as the cache manager can both replace a dirty object at any time and delay the propagation of a committed update. But it is also more complex than the other strategies.

Most transactional systems provide a fixed recovery policy because they are oriented towards databases and their main goal is the efficiency under a database access profile (i.e. large collections of homogenous data). One of the exceptions is Arjuna [11, 4] as it provides some flexibility to the programmer to modify concurrency control. However, as Arjuna is based on physical *logging* the admissible concurrency is constrained, as this forbids concurrent update operations.

TransLib is based in a design pattern, *TransRecovery*, that allows using different recovery strategies. An important contribution of *TransLib* is that it uncouples concurrency control from recovery, so they can be modified independently, thus improving flexibility and reuse. Another contribution is that recovery and concurrency control code do not appear in object operations, so they can be extended and reused in (and from) other contexts (e.g. any ADT can be used both in transactional and non-transactional contexts). This contrasts with existing systems where concurrency control and/or recovery code is embedded in the implementation of object operations. The goals of *TransRecovery* are:

- To uncouple cache management policy from the recovery and concurrency controls mechanisms used.
- To uncouple object code from the recovery and concurrency control algorithms chosen.

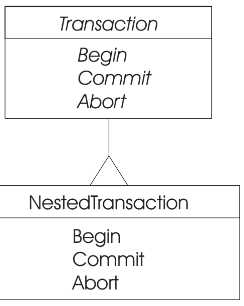

Figure 5: `Transaction` Class Hierarchy

- To allow and provide support for both physical and logical logging.

The participant classes in *TransRecovery* are:

- `Transaction`. It keeps track of all the information necessary to commit or abort a transaction.
- `RecoveryManager`. It encapsulates the recovery strategy.
- `CacheManager`. It encapsulates the cache replacement policy.
- `Log` encapsulates the log object.
- `MemoryObject`. It is the gate to the real object and related volatile information.
- `Operation`. It encapsulates a data object operation call.
- `AtomicCall`. It encapsulates the relationship between recovery and concurrency control.
- `Proxy` objects transform calls into `Operation` instances and use an `AtomicCall` object to make the recovery and concurrency control processing associated with the call.

3.1 Transactions

A transaction object is used to start and end a transaction. A transaction is started by calling the `Begin` method. If the transaction ends successfully, transactional code will commit it by calling the `Commit` method, otherwise it will call the `Abort` method. All these three methods will notify the transaction status (initiated, committed or aborted) to the recovery manager.

TransLib implements the nested transaction model and thus transactions can be nested by starting new transactions in dynamically nested scopes. A transaction object keeps references to all accessed objects, as well as the recovery and cache managers. It also keeps references to all its subtransactions (local or distributed). It must be noticed that when a transaction is distributed among a set of nodes, there will be a `Transaction` object in each of the nodes. Figure 5 summarizes the `Transaction` class methods.

3.2 The Recovery Manager

There is a recovery manager per node that it is in charge of logging redundant information to guarantee atomicity and durability of local transactions.

Transaction objects will notify it about transaction begin, commitment and abortion by calling the `Begin`, `Commit` (it makes the transactions updates durable) and `Abort` (it undoes the transaction) methods. Each `Operation` object has an associated recovery processing, so it can be later redone and/or undone. So, it is necessary to notify the recovery manager both before calling the operation (e.g. to log the before image of the object) and just after calling it (e.g. to log the after

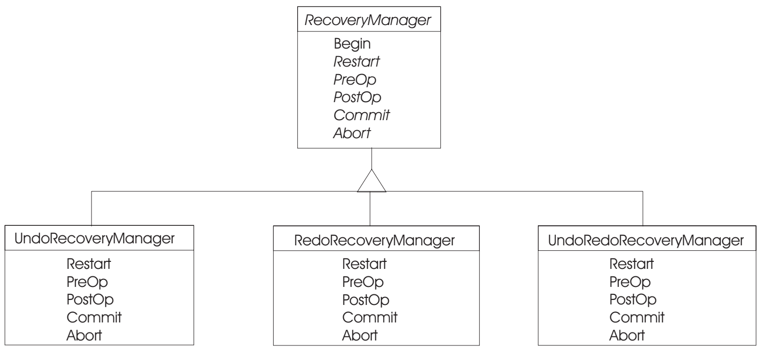

Figure 6: `RecoveryManager` Class Hierarchy

image of the object). For this reason, the `RecoveryManager` class provides the `PreOp` (recovery prologue) and `PostOp` (recovery epilogue) methods. These two operations allow modifying the recovery without affecting the rest of the system. The `Restart` method is in charge of the recovery process of persistent data after a crash. It implies to read and, possibly, modify the log. The recovery manager keeps in its state references to the local log object and cache manager, as well as to all the active transactions.

In order to admit different recovery managers, an abstract `RecoveryManager` class has been defined from where concrete recovery managers inherit as shown in Fig. 6. *TransLib* provides as predefined recovery strategies Undo/Redo, Undo/NoRedo and NoUndo/Redo.

3.3 The Cache Manager

The cache keeps copies of persistent data on volatile memory and it is used to decrease the number of disk accesses. As all persistent data cannot be kept in volatile memory, the cache manager sometimes needs to replace some data items for new ones, propagating the replaced data items to disk.

Additionally, to implement NoRedo schemes it is necessary a method to propagate all the objects modified by a particular transaction during its commitment. To cover these needs the `CacheManager` provides the `RecoveryManager` with the `Flush` and `FlushAll` methods. The `Flush` method propagates all the objects modified by a particular transaction, and then unpins all the objects accessed by the transaction. After the unpinning of an object it can be replaced by other objects. `FlushAll` acts similarly but with all the objects in the cache.

The `MemoryObject` class is used to prevent accessing the cache in each object access. This class encapsulates the information that the cache manager keeps about every object, and thus, the cache manager becomes just a table of `MemoryObjects`. The `NewMemoryObject` method of the `CacheManager` class is used to create new instances of the `MemoryObject` class. Each time an object is going to be loaded or discarded, the corresponding memory object must notify the cache manager so it can apply the replacement policy (before a new object is loaded) and update its information by means of the `ApplyReplacementPolicy` method.

Like the previous managers there is a root abstract class, `CacheManager`, that will be redefined for each concrete cache manager yielding to the hierarchy shown in Fig. 7. Currently, the least recently used replacement policy is provided by *TransLib*.

3.4 Cache objects: the `MemoryObject` class

Instances of this class act as gates to data objects, thus, each instance encapsulates the access to a data object. An instance of this class contains the object disk address and a reference to it when it is in memory. A `MemoryObject` brings the data object to memory (if it is not already present) when a transaction is going to access it. A `MemoryObject` instance also keeps the locks kept on

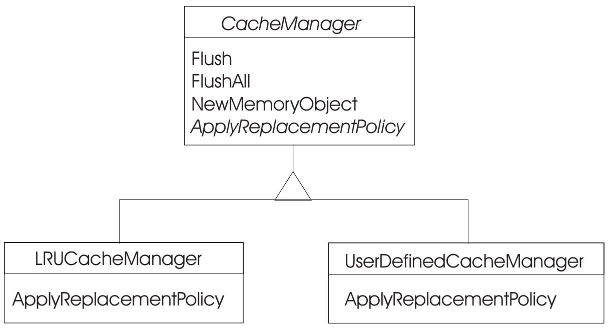

Figure 7: `CacheManager` Class Hierarchy

that object (i.e. a reference to its lock manager) and other information about the object that is not recorded on disk.

For instance, when versioning [9] (a variant of the Redo strategy for nested transactions) is used, the memory object keeps a stack of object before images (one per active subtransaction) to be able to recover the object state in case of a subtransaction abortion. This information is not written to disk because after a crash uncommitted subtransactions will be aborted.

Independently of the replacement policy used by the cache manager, on certain occasions some objects must not be propagated to disk. For instance, during the execution of an operation on an object, the object should not be propagated, otherwise inconsistent data would be propagated to disk. Another situation where objects should not be propagated is when NoUndo/Redo recovery is used. In this kind of recovery, dirty objects (objects reflecting uncommitted updates) should not be propagated to disk until commitment, so they must be retained in memory until this time. This is the motivation of the `Pin` and `UnPin` methods. The recovery manager will use these methods to inform the cache manager that the propagation of a particular object must be delayed. While an object is pinned the cache manager cannot propagate it to disk. When a transaction aborts, the `Discard` method of each modified object is called to discard all the transaction updates and free the associated cache slots. The `Propagate` method forces object propagation to disk. There are also methods to find out whether an object is currently in memory (`IsPresent`), whether it is dirty (`IsDirty`) and whether it is pinned (`IsPinned`).

3.5 Atomic Operations: The `Operation` class

An instance of `Operation` encapsulates a call to an object operation (i.e. an operation made by a transaction on a data object). This class enables logical logging, as operations are objects they can now be logged. The transformation of operation calls into objects allows a uniform treatment of the concurrency control and recovery processing associated with each operation (this is further explained in next section).

The `UnDo` and `Do` methods provide support for logical logging, together with the `Input` and `Output` stream methods[1] provided by Ada 95. The `Do` method executes an operation interacting properly with the `RecoveryManager` and the `LockManager`. The `UnDo` method is used to undo the effect of an operation. This method is needed for logical logging, to undo the effect of an aborted transaction.

In order to reduce the impact of changing the locking scheme on objects, the kind of lock chosen for an operation is encapsulated in the `GetKindOfLock` method. This method returns the kind of lock associated with an operation. Thus, only this method is redefined when a new locking scheme is written. The recovery manager also needs to know whether an operation will update an object or not. This information can be obtained by calling the `IsUpdate` method. Finally, each

[1]They correspond to the `GetState` and `SetState` of the *Memento* [19] design pattern.

operation object keeps the arguments of the call, and possibly part of the object state[2] before the call is made, so the operation can be undone.

3.6 The `AtomicCall` Class

A key feature of *TransRecovery* is the independence of data objects code from recovery and concurrency control. The `AtomicCall` class encapsulates the concurrency control and recovery processing in its `AtomicDo` method. In the current implementation of *TransLib* (for locking-based concurrency control) `AtomicDo` executes the following actions:

1. *Concurrency control prologue.* Calls `SetLockAndMutex` of the associated `LockManager` passing as argument the `Operation` to be executed.
2. *Recovery prologue.* Calls `PreOp` of the local `RecoveryManager` to execute the recovery prologue (it will depend on the concrete policy it implements).
3. *Object operation call.* Calls the `Do` method of the `Operation` object that calls to the real object.
4. *Recovery epilogue.* Calls `PostOp` of the local `RecoveryManager` to execute the epilogue defined by it (again, it will depend on the concrete recovery policy implemented).
5. *Concurrency control epilogue.* Calls `FreeMutex` of the associated `LockManager` to free the object mutual exclusion.

3.7 The `Proxy` Class

Object operation calls require to do some mechanical activities, such as `Operation` object creation, use of an `AtomicCall` object to make the call, and so on. In order to simplify transactional code, objects will be accessed by means of a *proxy* [19] that will encapsulate these mechanical activities. A *proxy* provides the same interface as the real object, and its methods just create the corresponding `Operation` instance and use an `AtomicCall` object to execute the operation atomically. Currently, the programmer of the object must write the proxy. This is one of the disadvantages of using a library instead of a language. We plan to implement a proxy generator to free the programmer from this mechanical task.

3.8 Interaction between the components of *TransRecovery*

The relationships among the objects involved in an object operation call within a transaction can be seen in Fig. 8. Each `Transaction` object keeps references to the local `RecoveryManager` and `CacheManager`. Additionally, it keeps a list of all the objects (more exactly the associated `MemoryObjects`) accessed by the transaction. The `RecoveryManager` references the `CacheManager` and the `log`, and it also keeps a list of active transactions in that node. A `proxy` object keeps a reference to its `MemoryObject`, so it can propagate calls to the real object. Instances of `AtomicCall` only need to reference the local `RecoveryManager`.

The `CacheManager` is just a table of `MemoryObjects`. Each `MemoryObject` encapsulates the real object and its associated `LockManager`.

The complete interaction corresponding to a transactional object operation call is shown in Fig. 9.

[2]An operation that writes the contents of an object field will keep the previous value of field to be able to undo the operation.

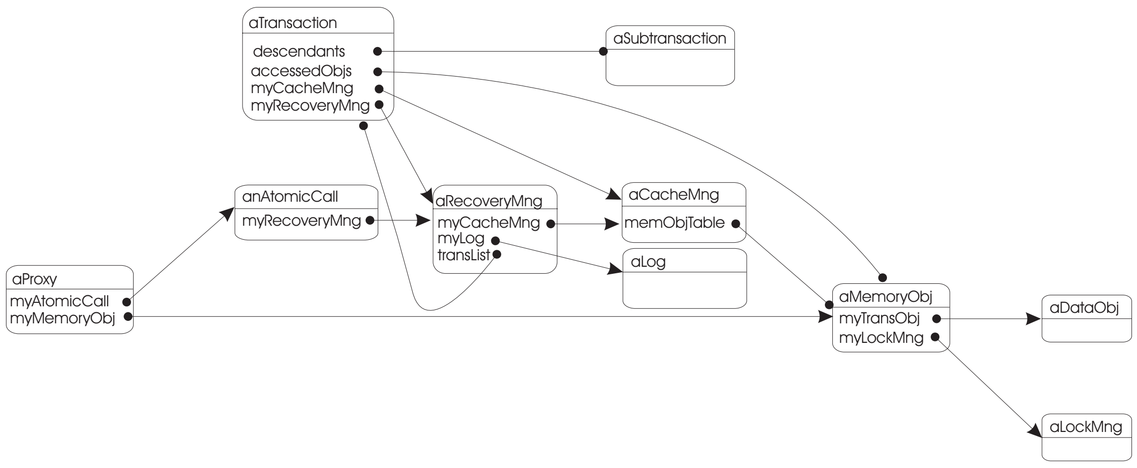

Figure 8: Object diagram of *TransRecovery*

3.9 Transactional Recovery and Ada 95

Streams have been one of the most useful features of Ada 95 for recovery. They have been used for both physical and logical logging. Ada 95 provides default flattening and unflattening operations for any user-defined type what saves a lot of mechanical work to the programmer. The standard stream interface for direct access files has been used to access the log, thus log records are flattened (unflattened) by the `Input` and (`Output`) operation. A most welcome feature was the reuse of these operations for any kind of stream. Log records can contain either object images or operations. With physical logging data objects are logged, whilst in logical logging `Operation` objects are logged.

Mixin-inheritance has made possible to provide generic class extensions and thus improve further reuse along *TransLib*. This feature has been especially useful to automate the stub generation. We have not found any need for multiple inheritance. Package hierarchies has helped to structure *TransLib* functionality. Generic child packages have helped to automate many activities such as the server main loops.

Limited types have helped to keep the consistency of class instances, but we have missed the possibility of redefining the assignment operator as in C++. Controlled types have been useful on many occasions, but sometimes the impossibility of finding out which event has fired the execution of `finalize` has prevented its use. The same conclusion has been drawn in other works like [20].

Protected objects have been widely used to protect objects from concurrent accesses, for instance the cache, the recovery manager and memory objects. In general, concurrency support has made *TransLib* highly portable, thus not depending on specific operating system support.

The systems programming annex [21, 22] has played an important role to simplify *TransLib* interface. It has helped to free the programmer from managing tids on many occasions. This annex allows task state extension and its manipulation. The state of tasks belonging to a transaction has been extended. As a task may start nested transactions, its state has been extended with a stack that contains the active transactions being executed by that task. The operations that modify the state are the ones that start and end transactions. An excerpt of the interface of the package that deals with task state is the following:

```
package TransactionalTaskState is
no_transactions : exception;
type TaskAttribute is tagged limited private;
type TaskAttributePtr is access all TaskAttribute'class;

function CurrentTID return TransactionIdentifier;
```

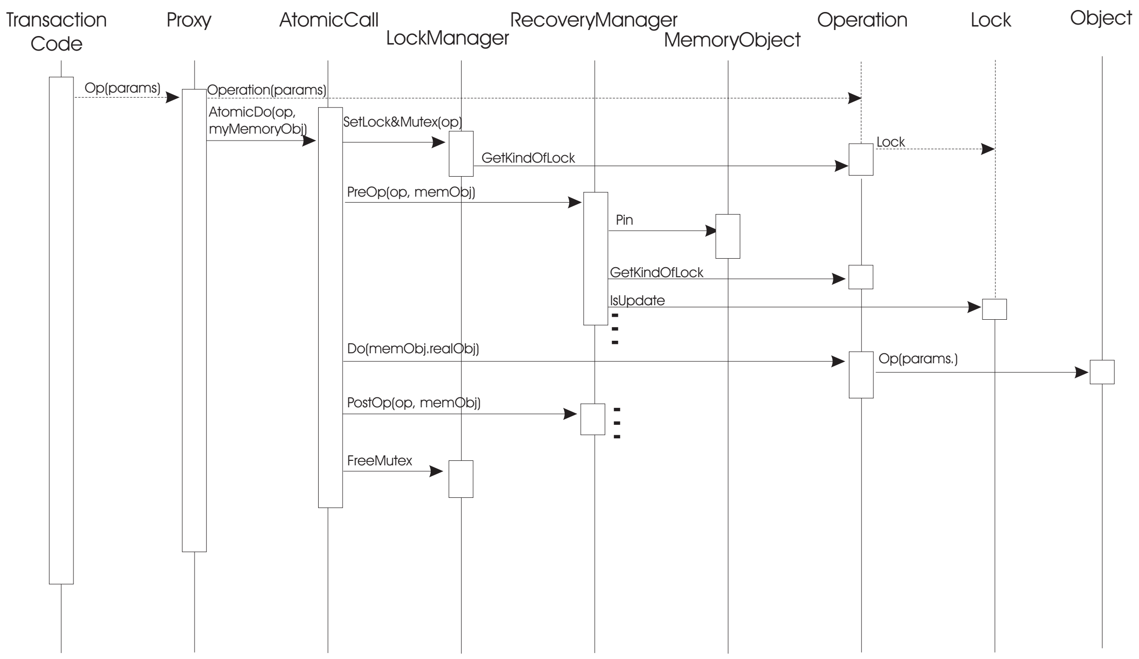

Figure 9: Interaction Diagram of an Atomic Call

```
function AmIRunningTransactions return Boolean;
procedure T_Begin;
procedure T_Begin_Finger(finger_kind : Finger_Kind_Type;
                         tid : TransactionIdentifier );
procedure T_End;
procedure T_Abort(reason : Exception_Occurrence_Access );
...
end TransactionalTaskState;
```

The operation `CurrentTID` is provided to obtain the tid of the transaction the task is currently executing (the one on the top of the stack). This operation allows knowing on behalf of which transaction an object operation is being executed. This operation has been useful for concurrency control management where the lock manager needs to know which transaction is requesting a lock. The lock manager can find out this information by calling the `CurrentTID` procedure. Tasks can check whether they are running transactions by calling the operation `AmIRunningTransactions` (this operation is usually called internally by *TransLib*).

To further simplify the use of `Transaction` objects, operations to start and end transactions have been added to this package. Thus, a new (sub)transaction is started by just calling `T_Begin`. `T_Begin` creates a `Transaction` object and stores it in the task state. A transaction is committed with `T_End` and aborted with `T_Abort`. Both operations remove the transaction from the task state. This means that the programmer has also been freed from the management of `Transaction` objects, and (s)he just uses the transaction start and end operations almost as language primitives. A task can join an ongoing transaction by calling `T_Begin_Finger` (a finger is a transaction thread).

As can be seen `T_Begin` does not return the tid, and termination operations (i.e. `T_End` and `T_Abort`) do not need tids as arguments, nor it is necessary for a task to notify its tid in each transaction operation (that is, each data object access). An additional benefit of this approach is that *TransLib* can ensure that transactional objects are only accessed within a transaction (just by calling to `AmIRunningTransactions`). When *TransLib* finds out that a transaction operation is being executed outside of a transaction it raises the `no_transactions` exception. The tid management remains thus hidden in the code of transaction management operations (provided by *TransLib*) in all the cases except in the `T_Begin_Finger` operation.

The need for the tid in the `T_Begin_Finger` operation it is due to the impossibility of identifying the parent task (i.e. its task identifier) of a particular task in Ada 95. The lack of this feature prevented us from hiding tid management during transaction thread creation. If this feature would have been available, transactions child tasks could be created without providing them the tid of the transaction they belong to, as they could have obtained the parent task identifier internally and use it to access its extended state and to find out the transaction tid.

4 *Group Transactions*

Process groups and the group communication model [23], in particular Causally and Totally Ordered Communication (CATOCS) [24] have also been proposed as an adequate way to build reliable distributed applications [25]. A group provides a common interface to a distributed server composed of a set processes. Clients call the group by multicasting the request to all the group members. The group helps to abstract the client from the fact that there is more than one process involved in the service. Causal and total ordering ensures that all the messages sent to a group are delivered in the same order to all the group members, preserving its causal order. This kind of communication is used in a variety of applications like management of replicated data, observation of a distributed system, financial systems, air-traffic control, multimedia systems, mobile computing environments and teleconferencing. A group composed of replicated distributed processes can tolerate hardware failures. The *Isis* toolkit [26] provides a collection of libraries to manage process groups that communicate using CATOCS primitives.

According to [27, 28, 29] a current important topic of research is the integration of the group communication and transaction models. [27] integrates these models. In this integration all the operations of a transaction are packed in unique service. The client multicasts the call to the group server and all the members of the group service atomically the call. However, this approach only deals with the isolation property of transactions and it does not take into account recoverability. The relationship between multicast and distributed commit protocols is discussed in [29], where they propose a protocol, DT-multicast, that can be used to implement both protocols.

Our opinion is that both models are complementary. We have developed a new transaction model, *Group Transactions*, that integrates both approaches. We have extended a group oriented fault-tolerant distributed language, *Drago* [30], with transactional mechanisms in a new version of the language called *Transactional Drago* [31]. *TransLib* has also been used to provide run-time support for *Transactional Drago*. *TransLib* uses as communication layer, the group communication library *GroupIO* [32]. This library provides reliable multicast. *Group Transactions* provide replicated and cooperative transactional groups. Transactional replicated groups improve the availability of transactional servers, by replicating them and their associated data. Transactional cooperative groups reduce the latency of their services by distributing the work among the group members.

One of the main features of *Group Transactions* is that it allows several flows of execution (concurrency) within a transaction. This *intra-transactional concurrency* allows decreasing the latency of transactions. There are two kinds of intra-transactional concurrency:

1. *Multithreaded transactions.* A transaction can have several threads, that is, several concurrent tasks can act on behalf of a transaction. This allows taking advantage of multiprocessor and multiprogramming capabilities.

2. *Multi-process transactions.* Requests to a cooperative group will be provided by all the members of the group. A group call is executed as a single multi-process (distributed) subtransaction of the client transaction. Due to all the members of a group participate in the same transaction, they will be able to cooperate using intragroup communication or by interacting with other groups.

We have called *fingers* the flows of execution of a transaction, concurrent (tasks) or distributed (processes). Fingers of the same transaction are called *sibling fingers*. A finger can initiate a subtransaction at anytime; that subtransaction will be atomic with respect to its sibling fingers. A transaction can commit only when all its fingers have finished successfully.

Group Transactions are serialized in the traditional way. The final effect of the concurrent execution of two transactions must be the same as if they were executed sequentially in some order; that is, as if all the fingers of the transaction serialized in first place were executed before all the fingers of the transaction serialized in second place.

Client/server interaction in *Group Transactions* is based on multithreaded rendezvous [33]. This interaction mechanism allows to build transactional servers with conversational interfaces [34] based on Ada rendezvous. A design pattern, *MultithreadedRendezvous*, has been proposed to implement it in [35]. Its main advantage is that it simplifies both, the code of the server, and the enforcement of the client-server protocol. When a client calls to a server group, the call is multicast to all group members. With *MultithreadedRendezvous* the interaction with a server is based in rendezvous, however servers do not deal with calls from different transactions in the same flow of execution. When a client calls for the first time to a server, a server thread (in each group member) is created to service all the calls from this client. A server thread accepts calls from a client like an Ada task, and thus clients interact with servers by means of rendezvous. This is a natural extension of Ada rendezvous.

Replication is traditionally used to provide highly available data. In our model, replicated groups not only provide highly available data, but also highly available transactions, as we allow replicating both clients and servers. A transaction can be initiated in a replicated group (i.e. clients can be replicated). The failure of a replica does not abort that transaction. Replicated groups can also act as servers, providing available services in spite of node failures.

A replica is composed of a scheduler and a set of server threads (one for each client transaction) that execute the same code. To guarantee replica determinism, it has been necessary to restrict the Ada 95 constructs that can be used in the code of server threads. The server thread code must be deterministic (and thus sequential) what excludes the use of asynchronous transfer of control, delay statements, and any other construct that could compromise determinism. Additionally, to guarantee the determinism of the replica as a whole it is necessary to use a deterministic scheduler. The scheduler ensures that only a transaction is executing at a given time and that context changes take place deterministically. Only when the active transaction blocks, the control is transferred again to the scheduler.

5 *TransLib* Exception Model

TransLib exception model [5] is novel in that it integrates backward recovery and forward recovery. In our approach forward recovery is used to cope with anticipated errors, that is, errors predicted by the programmer for which (s)he has written an exception handler to obtain a consistent state. If the application state cannot be transformed into a consistent one or the error has not been foreseen, then backward recovery is automatically used to restore the state to a previous consistent one by aborting the enclosing transaction. In addition, we use exceptions to notify transaction abortions to the enclosing scope. In this way, transactions act as firewalls confining damage produced by unhandled errors, and exceptions are used as notification mechanism.

The Ada exception model is based on the termination model. In this model, when an exception is raised, the current scope is abandoned. If there is an exception handler for the raised exception that is associated with the scope where the exception has been raised, control is transferred to it and execution follows just after the terminated scope. If the exception is not handled, it is *propagated* to the dynamic enclosing scope. If the exception is not handled in that scope, it is propagated again. An unhandled exception in the main program causes program termination with a run-time error, and an unhandled exception in a task provokes the task termination, and the exception is lost.

TransLib deals with multithreaded and/or multi-process (distributed) transactions that can raise exceptions concurrently. In this case exception resolution is needed. Exception resolution [36, 37] is used to choose an exception that represents all the exceptions that have been concurrently raised. *TransLib* provides a default resolution scheme that it is applied when multiple fingers of a transaction raise exceptions. This default resolution scheme propagates the predefined exception *several_exceptions*. It must be noticed that in Ada no exception resolution is applied, and exceptions that cause task termination are just lost. Programmers can define their own resolution scheme in order to provide more information about the source of the transaction abortion than the predefined exception *several_exceptions* does.

The task that starts a transaction is called root finger. The server threads created in a group as a consequence of a call from a transaction are termed primary fingers. Fingers created by a primary finger are called secondary fingers. Based on this distinction *TransLib* provides two levels of exception resolution: local and distributed. Local resolution is used to solve concurrent exceptions among a primary finger and its secondary fingers (see Fig. 10). Thus, this resolution will resolve more related exceptions (those local to a server thread). Distributed resolution is used to solve concurrent exceptions among the root and its primary fingers (see Fig. 10). Local resolution is always applied before distributed resolution. Thus, we create a two-level *hierarchical exception resolution* based on the finger hierarchy. This approach is more coherent than a global exception resolution at transactional level, which would have to solve totally unrelated concurrent exceptions.

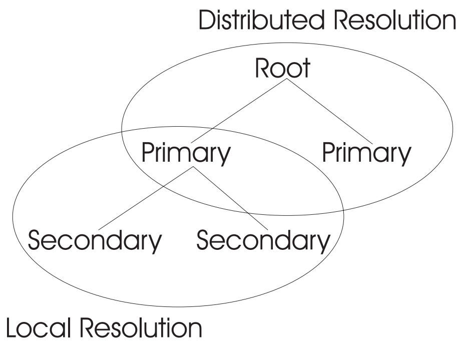

Figure 10: Hierarchical Exception Resolution

Replicated groups behave as a single member group, thus all the replicas will raise the same exception, so exception resolution does not apply.

5.1 Exceptions in Ada 95 and *TransLib*

Ada exceptions have been essential to provide forward recovery, but they have also been very useful to implement the backward recovery provided by transactions. The package `Ada.Exceptions` has been extremely helpful to manage exceptions as data, especially exception occurrences and operations to reraise them. Exception occurrences have allowed capturing exception information what has been useful in two different scenarios. First of all, they have been used to reraise an exception automatically in the code of the `Abort` primitive, and thus freeing the programmer from this work. And second, they have also been used to flatten an exception raised in a group member during remote interaction, and propagate it to the rest of the group members and the client. Blocks have also provided an essential support to create the necessary exception scopes (used as transaction scopes) for local subtransactions.

6 Related Work

We have already mentioned that there are some languages like Argus [9] and Avalon [10] that implement the nested transaction model. However, library support is necessary to write transactional applications with standard languages.

One of the main features of Avalon is that it provides hybrid atomicity as concurrency control mechanism. This mechanism provides more concurrency than commutativity-based locking, the concurrency control used in *TransLib*. However, their approach presents two disadvantages: first of all, although hybrid atomicity provides more concurrency, it also increases the probability of deadlocks (in fact, it is possible to create a deadlock with a single object), and secondly, the object code must take care of both concurrency control and recovery. In *TransLib* we have chosen commutativity-based because of its simplicity, it is a declarative approach, and because it has allowed us to remove the concurrency control code from object code.

Arjuna [4, 11] is an object oriented library implementing the nested transaction model. *TransLib* is similar to Arjuna in that it provides an Ada 95 object oriented library for building transactional applications. Although it is possible in Arjuna to customize concurrency control, this customization is very limited due to the fact Arjuna only allows physical logging what eliminates the possibility of concurrent updates. *TransLib* is novel in the flexibility it provides to change recovery and concurrency control mechanisms, and its integration of exceptions and transactions. Furthermore, it implements a more general transactional model *Group Transactions* being the nested transaction model a particular case of it.

Argus [9] is the only system to our knowledge that also provides exceptions and transactions. However, they do not really integrate both mechanisms what originates several problems. First of all, a transaction can end exceptionally (and possibly leaving an inconsistent state) and commit. Secondly, it is also possible for a transaction to end successfully and abort (i.e. without raising any exception notifying the abortion), what could be interpreted by the caller as a transaction commitment when the transaction has aborted and thus undone. For these reasons, we find our model more coherent as it identifies successful end (i.e. when no exception is raised) with commitment and exceptional end with abortion.

There has been several works on the implementation of software fault-tolerance mechanisms in Ada 95. [6, 7] discuss how to implement atomic actions in Ada 95. Atomic actions differ from transactions in that they do not support persistent data, and both proposals only address non-distributed implementations. *TransLib* is far more general as it implements distributed transactions and it can also be used to implement atomic actions.

[8] suggests an Ada 95 implementation of distributed recovery blocks. The main difference with the previously mentioned papers is that they do not deal with concurrent participants, but on the other hand they deal with distribution. They also do not deal with persistent data.

7 Conclusions

We have presented *TransLib* an Ada 95 object oriented framework for building distributed transactional systems. *TransLib* is documented as a set of object oriented design patterns. The main *TransLib* features can be summarized as:

- Transactional concurrency control and recovery can be redefined by the user. This flexibility allows customizing *TransLib* to the application needs.
- Code of objects used by transactions does not include synchronization neither recovery code, that is, it is independent from the concurrency control and recovery mechanisms used. Thus, regular objects can be reused in transactional applications, and vice versa.
- A novel integration of transactions and exceptions. Transactions that finish successfully are committed. Transactions that end exceptionally are aborted, that is, exceptions that cross transactional boundaries cause transaction abortions. Additionally, transaction abortions are notified as exceptions.
- *TransLib* implements the *Group Transactions* model that integrates the transaction and group communication paradigms.

- It provides commutativity-based locking which provides more concurrency than read/write locking.
- *TransLib* implements Redo, Undo and Redo/Undo recovery mechanisms.

Acknowledgments

We are sincerely grateful to the anonymous reviewers for their useful comments, which helped to improve the quality of this article greatly.

References

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

[2] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* MIT Press, Cambridge, MA, 1985.

[3] B. Weihl. Commutativity Based Concurrency Control for Abstract Data Types. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988.

[4] S. K. Shrivastava. Lessons Learned from Building and Using the Arjuna Distributed Programming System. In K.P. Birman, F. Mattern, and A. Schiper, editors, *Theory and Practice in Distributed Systems*, volume LNCS 938, pages 17–32. Springer, 1995.

[5] M. Patiño Martínez, R. Jiménez Peris, and S. Arévalo. Exceptions, Groups and Transactions: The Transactional Drago Approach. In *Submitted to the IEEE Trans. on Software Engineering*, 1999.

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

[7] A. Romanovsky, S.E. Mitchell, and A.J. Wellings. On Programming Atomic Actions in Ada 95. In K. Hardy and J. Briggs, editors, *Proc. of Int. Conf. on Reliable Software Technologies*, volume LNCS 1251, pages 254–265, London, United Kingdom, June 1997. Springer.

[8] Y. Kermarrec, L. Nana, and L. Pautet. Providing fault-tolerant services to distributed Ada 95 applications. In *Proc. of ACM TriAda Conf.*, pages 39–47, Philadelphia, PA, December 1996. ACM Press.

[9] B. Liskov. Distributed Programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[10] J. L. Eppinger, L. B. Mummert, and A. Z. Spector, editors. *Camelot and Avalon: A Distributed Transaction Facility.* Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[11] G. D. Parrington, S. K. Shrivastava, S. M. Wheater, and M. C. Little. The Design and Implementation of Arjuna. *USENIX Computing Systems Journal*, 8(3):255–308, 1995.

[12] C. J. Thompson and V. Celier. DVM: An Object-Oriented Framework for Building Large Distributed Ada Systems. In *Proc. of ACM TriAda'95 Conf.*, Anaheim, CA, November 1995. ACM Press.

[13] J. Kienzle and A. Strohmeier. Shared Recoverable Objects. In M. González-Harbour and J.A. de la Puente, editor, *Proc. of Int. Conf. on Reliable Software Technologies*, volume LNCS 1622, pages 397–411, London, United Kingdom, June 1999. Springer.

[14] B. R. Badrinath and K. Ramamritham. Semantics-Based Concurrency Control: Beyond Commutativity. In *Proc. of 3rd IEEE Int. Conf. on Data Engineering*, pages 304–311, Los Angeles, CA, February 1987. IEEE Computer Society Press.

[15] G. D. Parrington and S. K. Shrivastava. Implementing Concurrency Control in Reliable Distributed Object-Oriented Systems. In *Proc. of 2nd European Conf. on Object-Oriented Programming, ECOOP88*, volume LNCS 322, pages 233–249, Oslo, Norway, August 1988. Springer.

[16] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[17] M. Rukoz. Hierarchical Deadlock Detection for Nested Transactions. *Distributed Computing*, 5(4):123–129, 1991.

[18] V. Kumar and M. Hsu, editors. *Recovery Mechanisms in Database Systems.* Prentice Hall, NJ, 1998.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison Wesley, Reading, MA, 1995.

[20] P. Rogers and A.J. Wellings. An Incremental Recovery Cache Supporting Software Fault Tolerance. In M. González-Harbour and J.A. de la Puente, editor, *Proc. of Int. Conf. on Reliable Software Technologies*, volume LNCS 1622, pages 385–396, Santander, Spain, June 1999. Springer.

[21] *Ada 95 Reference Manual, ISO/8652-1995.* Intermetrics, 1995.

[22] A. Burns and A. Wellings. *Concurrency in Ada.* Cambridge University Press, 1995.

[23] Group Communication. Special Section. *Communications of the ACM*, 39(4):50–97, April 1996.

[24] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. In S. Mullender, editor, *Distributed Systems*, pages 97–145. Addison Wesley, Reading, MA, 1993.

[25] K.P. Birman. *Building Secure and Reliable Network Applications.* Prentice Hall, NJ, 1996.

[26] K. P. Birman and R. Van Renesse. *Reliable Distributed Computing with Isis Toolkit.* IEEE Computer Society Press, Los Alamitos, CA, 1993.

[27] A. Schiper and M. Raynal. From Group Communication to Transactions in Distributed Systems. *Communications of the ACM*, 39(4):84–87, April 1996.

[28] G. D. Parrington, S. K. Shrivastava, S. M. Wheater, and M. C. Little. The Design and Implementation of Arjuna. Technical Report 65, BROADCAST Project, October 1994.

[29] R. Guerraoui and A. Schiper. Transaction model vs Virtual Synchrony model: bridging the gap. In K.P. Birman, F. Mattern, and A. Schiper, editors, *Theory and Practice in Distributed Systems*, volume LNCS 938, pages 121–132. Springer, 1994.

[30] J. Miranda, Á. Álvarez, S. Arévalo, and F. Guerra. Drago: An Ada Extension to Program Fault-tolerant Distributed Applications. In A. Strohmeier, editor, *Proc. of Int. Conf. on Reliable Software Technologies*, volume LNCS 1088, pages 235–246, Montreaux, Switzerland, June 1996. Springer.

[31] M. Patiño Martínez, R. Jiménez Peris, and S. Arévalo. Integrating Groups and Transactions: A Fault-Tolerant Extension of Ada. In L. Asplund, editor, *Proc. of Int. Conf. on Reliable Software Technologies*, volume LNCS 1411, pages 78–89, Uppsala, Sweden, June 1998. Springer.

[32] F. Guerra, J. Miranda, Á. Álvarez, and S. Arévalo. An Ada Library to Program Fault-Tolerant Distributed Applications. In K. Hardy and J. Briggs, editors, *Proc. of Int. Conf. on Reliable Software Technologies*, volume LNCS 1251, pages 230–243, London, United Kingdom, June 1997. Springer.

[33] M. Patiño Martínez, R. Jiménez Peris, and S. Arévalo. Synchronizing Group Transactions with Rendezvous in a Distributed Ada Environment. In *Proc. of ACM Symp. on Applied Computing*, pages 2–9, Atlanta, Georgia, February 1998. ACM Press.

[34] B. Walter. Nested Transactions with Multiple Commit Points: An Approach to the Structuring of Advanced Database Applications. In *Proc. of 10th Int. Conf. on Very Large Data Bases*, pages 161–171, Singapore, August 1984. Morgan Kaufmann Publishers.

[35] R. Jiménez Peris, M. Patiño Martínez, and S. Arévalo. Multithreaded Rendezvous: A Design Pattern for Distributed Rendezvous. In *Proc. of ACM Symp. on Applied Computing*, pages 571–579, San Antonio, Texas, March 1999. ACM Press.

[36] A. Romanovsky. Extending conventional languages by distributed/concurrent exception resolution. *Journal of Systems Architecture*, 46(1):79–95, Sept. 1999.

[37] R. H. Campbell and B. Randell. Error Recovery in Asynchronous Systems. *IEEE Transactions on Software Engineering*, 12(8):811–826, August 1986.