# Combining Object-oriented Systems and Open Transaction Processing

PAUL TAYLOR*, VINNY CAHILL* AND MICHAEL MOCK†

*Distributed Systems Group, Department of Computer Science, Trinity College, Dublin 2, Ireland
†Research Center for Computer Science (GMD), Schloß Birlinghoven, D-53757 Sankt Augustin, Germany

**Atomic transactions are now a familiar paradigm for distributed programming and have been provided in a number of object-oriented languages. Much effort has also been expended on developing open transaction processing systems which support distributed transactions involving multi-vendor database systems. This paper addresses a number of issues that arise in combining object-oriented distributed programming with open transaction processing. We describe an approach to supporting transactions which can access objects and other resource types, such as files and records, consistently, and which is independent of the use of any particular object-oriented programming language. We discuss both the design of a generic run-time interface which provides language independent support for atomic objects and transactions and, following the X/Open model for open transaction processing, the design of an interface between the transaction manager and a resource manager which is suitable for the requirements of object-oriented systems. We illustrate our approach by describing the transaction sub-system of the Amadeus/RelaX implementation of the Comandos platform which supports a number of popular object-oriented languages and has been integrated with an existing relational database system.**

*Received April, 1994*

## 1. INTRODUCTION

Atomic tranactions are now a familiar paradigm for the construction of reliable distributed applications. Transactions usually provide the well known ACID properties of Atomicity, Consistency, Isolation and Durability (Gray & Reuter, 1992). These properties allow applications to take the distributed state of the system from one consistent state to another consistent state despite failures and concurrency. Nested transactions (Moss, 1981) extend the transaction paradigm by providing the independent failure property for subtransactions and support the modular construction of applications.

Many systems have been developed that successfully combine transaction processing with the object-oriented programming methodology [e.g. Argus (Liskov and Scheifler, 1983), Arjuna (Parrington, 1990) and Avalon/C++ (Detlefs, *et al.*, 1988)]. These systems offer rich transaction functionality including support for nested transactions and provide a powerful linguistic base for developing reliable distributed programs. However these systems are restricted in that they typically only support a single language and do not allow consistent access to data other than objects.

The more traditional domain for transaction processing is in database systems. Generally these systems do not allow for nesting of transactions or support concurrency within a transaction. On the positive side, standards for open transaction processing have been developed with the aim of allowing a transaction to access data from multi-vendor database systems. The work being down by the X/Open Company in developing a reference model for distributed transaction processing (X/Open Company, 1991) is particularly important in this respect. In the X/Open model, Application Programs (APs) access shared resources (e.g. database records) provided by a number of Resource Managers (RMs) under the control of a Transaction Manager (TM). The X/Open XA-interface defines the interface between the TM and a RM and allows different XA-compliant database systems to be involved in a transaction controlled by an XA-compliant TM.

In this paper we discuss a number of issues involved in combining object-oriented systems and open transaction processing so that the guarantees made by the transaction system can hold not only for objects but for other resource types used by an application. We describe the transaction sub-system of the Amadeus/RelaX implementation of the Comandos platform (Cahill *et al.*, 1993), which provides language independent support for atomic objects and transactions, and allows objects and other resource types to be accessed consistently within the same transaction.[1]

In Amadeus/RelaX the *Generic Run-time* library (the GRT) provides the interface to existing object-oriented languages in such a way that a language can be extended to support distribution and persistence as well as atomicity without necessitating changes to its compiler or dictating the way in which this functionality is made visible to the programmer (Cahill *et al.*, 1993).

[1] This work was partially supported by the Commission of the European Community under ESPIRIT contracts 834 and 2071.

The transaction model provided offers the functionality of an object-oriented system including fully distributed nested transactions as well as providing enhanced functionality such as allowing the possibility of relaxing the isolation property of transactions (Nett *et al.*, 1986; Nett and Mock, 1993).

The implementation is structured according to the X/Open model: Amadeus/RelaX acts as a collection of APs and RMs that interact with the RelaX TM (Kröger *et al.*, 1990) which is responsible for distributed transaction management via an interface which extends the X/Open XA-interface. Pure XA-compliant RMs can still be integrated into the system.

In this paper we focus on the design of the major interfaces concerned with transaction management: the (subset of) the GRT interface[2] concerned with support for atomic objects and transactions and the TM-interface. We discuss the separation between generic and language-specific aspects of atomic object management and describe a TM-interface suitable for the requirements of object-oriented systems.

The remainder of this paper is laid out as follows. Section 2 gives an overview of Amadeus/RelaX. The issues involved in supporting atomic objects and transactions in a language independent manner are discussed in Section 3 where the GRT interface is described. In Section 4 we show how other resource types may be used and describe the TM-interface. Section 5 discusses the performance of the main operations related to the management of atomic objects and transactions. Some related work is described in Section 6, and Section 7 presents a summary and some conclusions.

## 2. AMADEUS/RELAX

This section presents an overview of the Amadeus/RelaX system focusing on those aspects that are related to transaction management. Further details may be found in (Mock *et al.*, 1992; Cahill *et al.*, 1993a,b; Taylor, 1993). Section 2.1 describes the computational model; the transaction model is described in Section 2.2 and the internals of Amadeus/RelaX are described briefly in Section 2.3.

### 2.1. Computational model

In line with the Comandos model, Amadeus/RelaX supports persistent, global and atomic objects. A *persistent object* is an object that exists beyond the lifetime of the application that created it. So called persistent objects are only potentially persistent in that only those which are reachable from a designated set of root objects actually persist, others being considered garbage. A *global object* is an object that is remotely accessible. An *atomic object* is an object for which the ACID transaction properties can be guaranteed. Such objects are the units of concurrency control and recovery; an atomic object can be taken from one consistent state to another within a transaction. Applications may use a mixture of local volatile objects and global, persistent or atomic objects.

Distributed processes are supported in the form of *jobs* and *activities*. A job consists of a number of *contexts* (address spaces), one on each node visited by the job, and one or more activities. Activities are lightweight processes that may be distributed over a number of nodes. A job or activity is created to perform a specific invocation on some object. An activity terminates when this initial invocation has completed and a job terminates when all activities belonging to the job terminate.

### 2.2. Transaction model

The transaction model provides standard distributed and nested transactions and has been extended to provide more flexibility (Nett *et al.*, 1985). The most obvious extensions to support object-oriented programming concern support for concurrency within transactions: both full nesting, with concurrent subtransactions, and concurrent activities inside each individual (sub)transaction are supported. A model based on single writer/multiple reader locks is used for the synchronization of concurrent activities inside of a single transaction. By combining nesting with support for concurrent activities, concurrency at different nesting levels is possible. In addition, nesting of transactions for recovery purposes alone is possible. In this case the synchronization level of a parent and its subtransaction are the same but the subtransaction is able to abort independently from its parent.

In contrast to conventional transaction systems, isolation of transactions is not mandatory although the usual transaction guarantees of atomicity, consistency and durability hold. The system provides a means for the controlled use of uncommitted data in order to increase the concurrency and efficiency of the system. A transaction that uses uncommitted data depends on the transaction that produced the data. Such a transaction cannot commit or abort independently and may, once terminated, be required to wait for the commitment of any transaction on which it depends before committing. The system checks for and keeps track of dependencies between transactions. These are then taken into account during the execution of the commit protocol in order to achieve a transaction consistent system state.

The system can distinguish between the successful termination of a transaction and its commitment so that transactions can terminate in an additional state other than the committed or aborted states, i.e., the completed state. A completed transaction may be committed later in a (potentially distributed) group commit. Note that, in contrast to the database notion of group commit, delaying commitment does not imply preventing access to the results of the transaction.

[2] Of the Comandos Virtual Machine Interface (VMI).

Concurrency control between transactions is implemented using non-strict two-phase read/write locking (Eswaran *et al.*, 1976). The lockpoint indicating the beginning of the shrinking phase of the transaction is not necessarily combined with its commit point. Premature release of locks allows the results of the transaction to be made available before its commitment. Note that this feature does not result in dirty reads since consistency and atomicity are still guaranteed.

### 2.3. System structure

The goal of Amadeus/RelaX has been to allow persistence, distribution and atomicity to be added to languages whose base versions do not support these features, in particular, without necessarily requiring changes to the language's compiler nor imposing entirely new constructs and models on the language.

Typically, a language to be supported will already have its own execution structures implemented by a compiler or preprocessor and a Language-Specific Run-time library (LSRT). These individual LSRTs are supported above the GRT which provides the support for persistence, distribution and atomicity which is required by a range of language implementations but which is independent of any particular language (Cahill *et al.*, 1993a). Whenever language-specific information or actions are required, the GRT makes an *upcall* to code supplied for the particular language; in some cases this code will be specific to the class of object being manipulated. The GRT has been designed specifically to interface to a LSRT rather than to provide an API for application programmers to use directly.

The GRT is supported by the *kernel*. While the GRT is a purely local component, linked into the address space(s) of each application, the kernel is a distributed component providing secondary storage management, location services, security mechanisms, distributed processes, and load balancing. In some sense, the GRT may be seen as the interface between a LSRT and the underlying kernel. The kernel is currently implemented above UNIX[3] as a collection of trusted servers with an associated library which is linked with each application (Cahill *et al.*, 1993b).

#### *2.3.1. Support for atomic objects*

The GRT makes use of a number of generic modules provided by RelaX which support concurrency control based on read/write locking of overlapping fragments of resources, recovery control, interfacing to the RelaX TM, and logging. These generic transaction support components (the so-called RM-library) are resource and language independent.

The key to the resource independence of these modules lies in the abstraction of the actual resource type making it opaque to the RM-library. A resource is identified by an (opaque) identifier and is expected to implement some low-level operations, for example to take or restore a recovery point, which can be called by the RM-library. In the case of Amadeus/RelaX, the resources are objects and the required operations are provided in the GRT in a language independent way (Mock *et al.*, 1992).

Accesses to atomic resources are assumed to be bracketed with calls to the RM-library for concurrency and recovery control. These checks are based on the identifiers of the current transaction and of the resource being accessed, and the mode (read/write) of the attempted access. These parameters are assumed to be passed to the RM-library which then handles the transaction-related aspects of accessing the resource in a language independent way. The detection and reporting of such accesses are discussed in Section 3.

The RM-library cooperates with the RelaX TM to achieve transactional properties, such as establishing consensus about the outcome of a transaction, that require global coordination and thus cannot be handled within the RM-library alone. The interface between the RM-library and the TM is described in Section 4.

## 3. THE GRT INTERFACE

In this section, we address the issues involved in supporting atomic objects and transactions in a language independent object support system. In particular we identify the support that can be provided by the GRT and that which is required from the language layer. Section 3.1 discusses issues related to supporting atomic objects, and Section 3.2 discusses issues related to creating and controlling transactions.

### 3.1. Atomic objects

The support for atomic objects in the GRT must meet the following goals:

- The support must be independent of any language.
- Each language must be free to choose how atomic objects are presented, if at all, to application programmers (e.g. accesses to atomic objects should be possible using the same syntax as for accesses to non-atomic objects).
- The overheads associated with using atomic objects should be minimized.
- The support should not effect the performance of accesses to non-atomic objects.

The use of transactions incurs overheads for commit processing, concurrency control and recovery. However, the transaction properties may not be required by all applications or for all objects manipulated by a given application. Therefore only a subset of objects need be atomic. Atomic and non-atomic objects may be manipulated in a transaction, but the transaction properties are only ensured for atomic objects; nothing is guaranteed about the consistency of non-atomic objects.

[3] UNIX is a trademark of UNIX Systems Laboratories, Inc.

### 3.1.1. Creation of atomic objects

There are a number of possible choices for how a language can present atomic objects to programmers. Atomic types could be supported so that all instances of some specified types are atomic. Alternatively, the decision as to which objects should be atomic and which non-atomic could be made at object creation time, independently of the object's types. Both of these approaches force an early decision to be made about which objects are to be atomic and which not. A more general approach would be to allow the decision to be deferred until later, i.e. to provide a means of conversion between non-atomic and atomic objects and, possibly, vice versa.

Since the GRT is intended to be language independent, a fundamental aspect of the support for atomic objects is that, at the GRT level, atomicity must be orthogonal to type. Thus, the GRT supports *atomic promotion* allowing a non-atomic object to be converted to being an atomic object. This does not prevent a language providing atomic types or other methods of indicating which objects are to be atomic, e.g. by performing promotion at object creation time.

Note that the inverse conversion, *atomic demotion*, is not permitted in order to avoid consistency problems with shared atomic objects. If an atomic object could be demoted, and subsequently modified, outside of a transaction, this could lead to inconsistency if the object was locked by some transaction at the time of its demotion.

In promoting an object to being atomic some language-specific operations may be necessary, for example binding of different class code to the object as discussed below in Section 3.1.2. Thus, an upcall from the GRT to the LSRT, which is called by the GRT when an object is being promoted, is provided. The GRT interface concerned with atomic promotion is thus as given in Figure 1.

### 3.1.2. Access detection and reporting

Atomic objects differ from non-atomic objects in that each access to an atomic object must be detected and reported to the underlying concurrency control and recovery systems. Thus, the GRT provides a prologue operation that must be called immediately before each access to an atomic object. The information passed to this operation includes the identifier of the object being accessed and the mode of access (i.e. read or write).

Since the transaction model supports *non-strict* two-phase locking, there must be a way of indicating when an access to an atomic object has completed (so that locks can be released). Further, to support concurrency within a transaction, single-writer/multiple-readers locks for atomic objects are required. Thus, an epilogue operation is also provided and must be called immediately after every access to an atomic object.

```
grt_make_atomic(Object) -> ack
/* promote object to being atomic */

upcall_make_atomic(Object) -> ack
/* perform language specific operations
   for atomic promotion */
```

**FIGURE 1.** GRT interface for atomic objects.

The major decision here concerns the level at which access detection should be performed: in the kernel, in the GRT or at the language level.

At the kernel level, one possible approach is to use virtual memory faulting techniques provided by modern operating systems [e.g. Mach (Rashid, 1986)]. In such a scheme, any page that contains (part of) an atomic object would be locked as soon as the page is accessed. This technique can be used, not only for access detection, but as the basis for recovery. Using such an approach has several disadvantages. First, it places a dependency on the underlying system. The Amadeus/RelaX platform is currently implemented above UNIX which does not provide such advanced virtual memory facilities as Mach. Second, the granularity of concurrency control and recovery would be a virtual memory page, rather than an individual object. Third, notification of when an access to an atomic object is complete would be difficult to achieve, thus not supporting non-strict two-phase locking and concurrency within a transaction. Finally, recent work suggests that the use of such virtual memory techniques to detect accesses to atomic objects is costly due to the overhead of trap generation and processing (Hosking and Moss, 1993).

An alternative method that could be implemented in the GRT is to provide *wrapper objects* (Baker, 1992). Here, all accesses to an object are directed to a wrapper object which forwards the invocation to the real object and also calls the prologue and epilogue operations appropriately. Wrapper objects do not support atomic promotion well since a test must be performed to determine whether the object is atomic or non-atomic, thus affecting the performance of accessing non-atomic objects. Another problem is that an objects *self* value cannot be passed to other objects. Other problems associated with wrapper objects are discussed in Baker (1992).

In extending a language to provide atomic types the most common and simplest approach to access detection is to require explicit notification of all accesses. This approach is used in the language support for Camelot (Eppinger *et al.*, 1992), Avalon/C++ (Detlefs *et al.*, 1988) and Arjuna (Parrington, 1990). The advantage of this approach is that applications have control over when locks are acquired (and in what mode) rather than relying on system defaults. This approach, though simple, requires extra work from programmers to explicitly declare all accesses to atomic objects. This gives greater scope for programming errors which could undermine the properties of atomic objects (e.g. only acquiring a read lock before modifying an object). There is further scope for errors if programmers also have to deal with epilogue operations.

As an alternative for languages that provide atomic types [e.g. Argus (Liskov and Scheifler, 1983)] the compiler (or preprocessor) can insert the prologue and epilogue operations around accesses to atomic objects. Similar to the explicit approach, the compiler can decide whether a lock is acquired and the correct mode of access depending on certain conditions. The advantage of this approach over the explicit approach is that the compiler can guarantee that all accesses are detected and that the correct mode is used. Of course when extending a language, a disadvantage is that modifying a compiler to insert the extra code may be difficult.

One uncommon aspect of the support for atomic objects in Amadeus/RelaX is the ability to promote an object from being non-atomic to being atomic. An approach to access detection which supports this aspect is to associate two sets of class code with each object: the non-atomic code and the atomic code. A non-atomic object is bound to the non-atomic code which provides normal access to the object with no performance overheads. When the object is promoted to being atomic, the object is bound to the atomic code which provides the same interface to the object except that the atomic code includes the appropriate calls to the prologue and epilogue operations. The atomic code would be produced by the language's compiler or preprocessor and the (re)binding done in a language-specific way in response to the `make_atomic` upcall.

There are two ways of implementing such atomic code. First, the atomic code could be a complete copy of the (normal) non-atomic code with extra code inserted for the prologue and epilogue operations. This has the advantage that the compiler can make better decisions as to when to acquire locks and which mode to use (similar to compiler generated in-line code). The disadvantages are that modification of compilers or preprocessors to produce this code may be difficult and there would be an increase in the volume of code required by an application.

Second, the atomic code could contain only the calls to the prologue and epilogue operations and just forward invocations to the non-atomic code. This is similar to the use of wrapper objects but without the associated problems because there is only one object. Modifying compilers or preprocessors to produce this type of atomic code should be easier because only the class definitions need to be examined. Also, the increase in the volume of code produced would not be significant.

It can be seen that while numerous mechanisms exist for trapping access to atomic objects, the most feasible require support from the language level. Hence, in Amadeus/RelaX access detection for atomic objects must be performed at the language level. This approach allows the language designer the full freedom to choose between the use of explicit (i.e. hand written) and implicit (i.e. compiler/preprocessor generated) access detection mechanisms as appropriate for the particular language. The interface to the GRT consists only of the prologue and epilogue operations which must be called before and after each access respectively (see Figure 2).

```
grt_prologue(Object, AccessMode) -> ack

grt_epilogue(Object, AccessMode) -> ack
```

**FIGURE 2.** GRT interface for access detection.

### 3.2. Transactions

The main issue here is how language independent support for transactions can be provided which allows for correct program continuation after a transaction abort.

Typically, programmers are provided with three primitive operations: BEGIN, COMMIT and ABORT. The BEGIN and COMMIT operations clearly define the boundaries of the transaction and provide points for computational rollback in the case of transaction aborts. When a transaction aborts, computation should continue at the statement immediately following the COMMIT operation.

With nested transactions, some systems distinguish between creation of top-level transactions and creation of nested transactions while others provide a single BEGIN primitive such that the outermost BEGIN creates a top-level transaction and nested BEGINS create nested transactions. While the former approach has the disadvantage to reducing the modularity of the system, nevertheless, some mechanism must be supplied for creating independent top-level transactions from within a transaction.

In supporting transactions in the Amadeus/RelaX system the following goals were identified:

- Language independence.
- Allowing for correct program continuation.
- Supporting the modular construction of applications while allowing the creation of independent top-level transactions from within a transaction.

#### *3.2.1. Transactions in Amadeus/RelaX*

In general, transaction support can be provided by either embedding the support into a new language or by extending an existing language. An example of the former approach is the Argus language which provides linguistic constructs for the creation of top-level and nested transactions. This approach offers powerful linguistic constructs which, because of their tight integration with the underlying system, can provide a fine degree of control. For example, invalid use of the constructs can be detected at compile time. Managing transaction aborts in this approach is simple because the language constructs can be tailored to the underlying system.

If an existing language is being extended, the language can either be augmented by preprocessing or macro facilities or a library of operations for transaction control can be provided. The former approach offers the same advantages as embedding the support in a new

```
grt_create_transaction(invoc_desc)
        -> COMMITTED | COMPLETED | ABORTED

grt_abort_transaction() -> ack
```

**FIGURE 3.** GRT interface for transaction control.

language except that the support is not as tightly integrated into the system. Providing a set of library routines is the simplest approach but does not offer a fine degree of control for transaction aborts because there is no linguistic link between the start and end of the transaction. In both of these cases the language cannot prevent the transaction's thread of control from running outside the boundaries of the transaction. For example, in the C language support for Camelot (Eppinger *et al.*, 1992), programmers are advised not to execute `goto`, `break` or `continue` statements that transfer control out of (or into) the body of a transaction block.

In Amadeus/RelaX, the method of creating (and ending) transactions must be generic but must ensure correct program continuation when a transaction aborts. Therefore, we decided to use object invocations as the defining boundaries for transactions (see Figure 3). That is, a transaction is created at the start of the invocation and ends when the invocation completes. This provides an intuitive way for programmers to define transaction boundaries. An advantage of this approach is that programmers can write classes without having to know if the code is to be used with transactions. Users of the class can decide by choosing whether to perform invocations on instances of the class within transactions or not. One added benefit of this method is that no explicit COMMIT operation is required.

Providing correct program continuation is simple since all that is required is the premature termination of the invocation. For the application, an abort is translated into an exceptional return from the invocation which is propagated in the appropriate language-specific manner.

In order to provide modular support, there is no distinction between creating a top-level transaction and creating a nested transaction. Top-level transactions are created by default, but inside a transaction, all new transactions are created as nested transactions. Independent top-level transactions may be created by first creating a job and then creating a transaction in the new job. Thus, jobs created within a transaction are not considered part of the transaction.

A transaction may be aborted at any time by the system and applications may explicitly abort the current transaction. Unhandled exceptions also cause the current transaction, if there is one, to abort.

## 4. DIFFERENT RESOURCE TYPES

The key to open transaction processing lies in isolating resource independent transaction functionality (such as distributed commit and abort protocols or the handling of site failures and restarts) from resource dependent

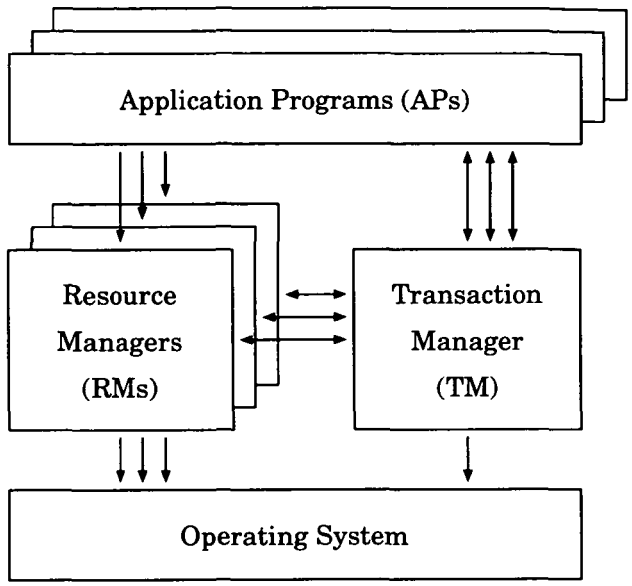


**FIGURE 4.** The RelaX architecture.

transaction functionality (such as saving or undoing the changes to a specific resource made by some transaction). In order to allow different resource types to be supported the interaction between these two aspects of transaction processing — in effect between the TM and individual RMs — must be defined by a standard interface. These basic ideas are reflected in the X/Open model for distributed transaction processing which defines a standard for the interface between a TM and a RM — the so-called XA-interface (X/Open Company, 1991) — as well as in the RelaX architecture depicted in Figure 4.

In the X/Open model RMs were expected to be database management systems providing non-nested, standard ACID transactions. In order to support more general applications and, in particular, to fulfill the needs of object-oriented systems with respect to nesting and concurrency control, Amadeus/RelaX provides extended transaction functionality (see Section 2.2) which requires that an enriched interface between the TM and the RMs be provided. In other words, the XA-interface as currently defined is not capable of supporting the extended transaction functionality provided, since:

- The XA-interface assumes a flat transaction model. Supporting nested transactions requires additional information about the nesting structure to be passed via the TM-RM interface.
- The XA-interface does not specify the behaviour of the TM nor of the RM with respect to concurrency control. Extensions are needed to allow for concurrency control mechanisms to exchange information via the TM-RM interface.
- The XA-interface associated accesses to resources with transactions based on the 'thread of control' which carried out that access. This causes difficulties when trying to combine RMs with different interpretations of what a 'thread of control' is.

```
/* - transaction control - */
join_ta( tid ) -> ack
local_complete(tid, parent_tid, sync_tid )
               -> ack
local_abort( <tid, sync_tid> ) -> ack

/* - transaction global synchronisation - */
local_setlockpoint( tid )

/* - committing transactions - */
local_prepare( cri, <tid> )
               -> PREPARED | FAILED
local_commit( cri ) -> ack
local_abortcr( cri ) -> ack

/* - restart handling - */
prepared_cr( <cri> )
```

FIGURE 5. The RelaX TM-interface.

- The XA-interface only addresses single transactions, group operations are not supported.

The following sections describe the extensions to the XA-interface provided by the RelaX TM-interface in order to support object-oriented systems. We show how the Amadeus/RelaX system is seen as a set of RMs from the TM's point of view, and sketch how a standard database system [Informix (Informix Software, 1991)], conforming to the XA-specification, has been integrated into the Amadeus/RelaX system and the resulting restrictions on the use of the database system.

### 4.1. The TM-interface

The following sections describe the RelaX TM-interface and its relationship to the XA-interface in greater detail. It should be noted however that the enriched interface provided by the RelaX TM still allows XA-compliant database systems to be supported as will be described.

The RelaX TM-interface is shown in Figure 5. A RM informs the TM that it is participating in a transaction by calling `join_ta`. The operations `local_complete`, `local_abort`, `local_setlockpoint` are used to inform the RM when a transaction completes, aborts or sets its lockpoint, respectively. Note that announcing the setting of the lockpoint (`local_setlockpoint`) or the completion of a transaction (`local_complete`) relates to the extended transaction functionality discussed in Section 2.2. Setting the lockpoint is a global event for a transaction which is scheduled consistently on all nodes by the TMs. Notifying the RMs about the setting of the lockpoint is an example of including information related to concurrency control in the TM-interface. Completing a transaction without committing is a prerequisite for supporting group commitment. As the completion of a transaction is also a global event detected by the TMs, the RMs must also be informed about this event via the TM-interface.

The RelaX TM-interface also includes additional information about the nesting structure of transactions. In Figure 5 the parameter `tid` identifies the transaction concerned, `parent_tid` identifies its parent transaction and `sync_tid` identifies the next ancestor transaction with the same synchronization level. Note that the RelaX TM-interface supports committing and aborting a group of transactions in a single protocol execution by passing lists of transaction identifiers in the relevant cells.

The operation `local_prepare` requests that the RMs which are participating in the specified transactions stably store the after-images of those transactions and associate them with the commit request identifier `cri`. In response, the RM returns `PREPARED` if the effects of the transactions are stably stored and `FAILED` if not. The TMs determine the outcome of the commit request by executing a distributed commit protocol and inform their local RMs about the outcome through the `local_commit` and `local_abortcr` routines. The RMs then perform the appropriate actions (i.e. replace the committed state with the prepared after-image in case of a commit or discard the prepared after-image in case of an abort) and reply to the TM when completed.

The TM keeps track of the progress of a commit request on a stable outcome log. As the resources are maintained by the RMs, this log only contains outcome records for commit requests, i.e. it has no data entries. Before writing a `prepared` record to its outcome log, the TM must make sure that all effected RMs have prepared the corresponding transactions. It may write the `prepared` record to the outcome log only if all the required RMs return `PREPARED`. In the restart phase after a site crash, the outcome log is scanned for incomplete commit requests. These are resumed by the TM and their final outcome is forwarded to the RMs. If a RM recovers individually, it informs the TM via the operation `prepared_cr` about the commit requests that it has not yet completed.

The X/Open XA-interface (see Figure 6) provides the `xa_abort`, `xa_precom` and `xa_commit` operations to abort, prepare and commit a single trans-action. These are superseded by the operations `local_abort`, `local_prepare` and `local_commit` in the RelaX TM-interface. The X/Open operation `xa_recover` is called in the restart phase and is subsumed by the `prepared_cr` operation. The X/Open `gtrid_reg` operation registers a RM as a participant in a transaction and is the counterpart of the `join_ta` operation.

```
/* - transaction control - */
xa_abort abort a transaction

/* - committing transactions - */
xa_precom prepare a transaction
xa_commit commit a transaction

/* - restart handling - */
xa_recover get the list of prepared transactions

/* - announce RMs and threads - */
gtrid_reg register a RM in a transaction
xa_start associate a thread with a transaction
xa_end disassociate a thread from a transaction
```

FIGURE 6. The XA-interface (extract).

The XA-interface provides operations related to the association of application programs and transactions. The operation `xa_start` associates the 'thread of control' that executed that operation with the transaction identifier which is passed as a parameter. The RM associates every subsequent operation by that 'thread of control' with that transaction identifier until `xa_end` is called from within the same 'thread of control'. The definition of the notion of 'thread of control' is not part of the XA-specification, but is left open to the individual RMs. A 'thread of control' might be, for instance, a UNIX process or a thread within such a process.

In contrast, in Amadeus/RelaX, the association of an application program in execution with a transaction identifier is handled in a library linked to the application program and consequently, does not appear at the RelaX TM-interface. The library maintains state information such as the name of the current transaction, which is encoded in the form of a pathname denoting nested transactions, and therefore allows accesses to resources, whether they are local or remote, to be tagged with that transaction identifier. This approach leads to greater flexibility in the construction of systems with components that have different notions of 'thread of control.'

### 4.2. Informix-integration

To integrate Amadeus/RelaX into the structure prescribed by the RelaX architecture required a mapping between the components related to object management and the components of the architecture, i.e. APs and RMs.

The resulting process structure — in terms of UNIX processes — can be captured by the RelaX architecture but to some extent exceeds the process model that is supported in the definition of the X/Open model. In Amadeus/RelaX, a context, which is actually realised by a UNIX process, acts as a RM for the objects mapped in the context. Activities that access these objects execute within that context. Note that these activities potentially belong to different transactions. Thus, a single UNIX process plays the role of a RM and a number of APs at the same time. In terms of the XA-specification, the RM recognises each activity to be a separate 'thread of control.' In terms of the RelaX TM-interface, this does not cause confusion as long as the guarantee that all accesses to resources are tagged with the correct transaction identifiers holds. This is easy to achieve by having each activity hold the current transaction identifier for that activity.

When integrating the Informix database, the weakness of the X/Open 'thread of control' concept comes to light. In the case of the Informix database, a 'thread of control' is a UNIX process. Thus, a UNIX process can only be associated via `xa_start` with one transaction on the Informix database at a time. In contrast, multiple transactions can be executed concurrently within an Amadeus/RelaX context (which is also a UNIX process) because the 'threads of control' within that RM are represented by activities. Thus, combining different RMs with different interpretations of the notion of 'thread of control' can lead to inconsistencies in the X/Open model. Since the Informix system could not do any better in this point than the X/Open model would allow, we had to disallow concurrent accesses to the Informix database from different transactions running in the same context.

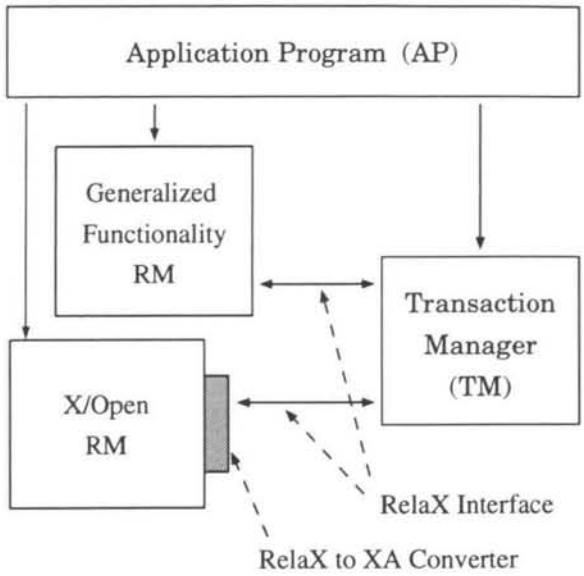


**FIGURE 7.** Integration of RMs.

As depicted in Figure 7, all RMs, i.e. RMs supporting extended transaction functionality and the full RelaX TM-interface, and standard RMs supporting only the XA-interface, look alike from the TM's point of view. In order to enable a standard RM to participate in transaction processing, a converter is linked to that RM in order to map the RelaX TM-interface calls issued by the TM to XA-interface calls for this RM and vice versa. This converter performs the following functions:

- Since a standard RM only knows about top-level transactions, we decided that, for simplicity, only top-level transactions are allowed to access resources maintained by the standard RM [although we realise that other more sophisticated approaches are possible (Gray and Reuter, 1992)]. Therefore, information concerning the nesting structure has to be stripped off.
- The converter transforms group operations related to the abort or commit of a set of transactions into a number of calls for individual transactions.
- The calls in the RelaX TM-interface to deal with aspects related to the extended transaction functionality are suppressed in the converter since standard RMs do not support this functionality.
- Finally, the converter provides name conversion to map procedure names used in the RelaX TM-interface to those of the XA-interface and vice versa.

## 5. PERFORMANCE

This section presents performance figures for the main Amadeus/RelaX operations related to the management

**TABLE 1.** Compilation of C** classes (all times in s)

| | *Number of classes* | | | | | *Number of methods* | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | *1* | *10* | *100* | *200* | *1000* | *1* | *10* | *100* | *200* | *1000* |
| Non-atomic | 21.5 | 22.3 | 32.8 | 50.2 | 184.6 | 20.9 | 21.8 | 25.0 | 28.2 | 61.7 |
| Atomic | 22.4 | 22.1 | 37.2 | 53.1 | 198.5 | 21.6 | 22.3 | 25.0 | 27.6 | 64.2 |

of atomic objects and transactions. The language used in obtaining these results is an extension to the C++ (Stroustrup, 1991) language called C** (Distributed Systems Group, 1992). C** directly supports promotion of objects to be atomic objects. Atomic objects are supported by producing an atomic class for each class and binding the atomic class code to the target object during atomic promotion. Invocations on atomic objects are intercepted by the atomic class code which brackets the real invocation with calls to the GRT prologue and epilogue operations.

Most of the performance figures presented in this section were obtained from an implementation of Amadeus/RelaX running on DEC station 5000s under Ultrix 4.3 connected by a 10 Mbit/s Ethernet. The figures were obtained by three different methods:

1. By using a distributed measurement system called Jewel, also developed at GMD (Lange *et al.*, 1992). Jewel provides highly accurate timings with a granularity of $10^{-7}$ s. Note that the figures from Jewel were obtained in a different environment and with an earlier version of Amadeus/RelaX.
2. By executing an operation a large number of times and taking the average of the total elapsed time. Here, the total time was measured using the UNIX `gettimeofday` system call which does not provide as fine a degree of granularity as Jewel.
3. By running a program using the UNIX `time` command and taking the elapsed time output by this command.

### 5.1. Compilation of atomic classes

The support of atomic objects in C** imposes a compilation overhead because the compiler must produce atomic classes. Table 1 compares the times to compile a C** source file without support for atomic objects to the time to compile the same file with support for atomic objects. The compilation times (which exclude the time for linking) were obtained by using the `time` command. The first part of the table shows the effect of the number of classes on compilation time. Here, all classes were identical, consisting of a single integer data member and a single method. The second part shows the effect of the number of methods in a class. Here, there was just one class and all methods were identical (i.e. taking no parameters and with empty bodies). The overhead in both cases is small: an average of 6% for the first part and 1.5% for the second part.

**TABLE 2.** *Object creation and atomic promotion (all times in μs)*

| | |
|---|---|
| Non-atomic object creation | 523 |
| Atomic object creation | 999 |
| GRT mutual exclusion locks | 117 |

**TABLE 3.** Accessing a non-atomic and an atomic object (all times in μs)

| | *Non-atomic object* | *Atomic object* | *Atomic object (no locking)* |
|---|---|---|---|
| Read | 16.38 | 3260.0 | 67.89 |
| Write | 16.29 | 3220.0 | 66.15 |

### 5.2. Management of atomic objects

The times for creating a non-atomic object and creating an atomic object (i.e. creation followed immediately by atomic promotion) are shown in Table 2. The `gettimeofday` system call was used to obtain these results.

The overhead of creating an atomic object is 476 μs. About one third of this time is taken by the time required to acquire and release GRT mutual exclusion locks.

The times to access a non-atomic and an atomic object are shown in Table 3. Both objects were instances of a class consisting of a single integer data member and two methods with empty bodies. The `gettimeofday` system call was used to obtain these results.

For the atomic object, the times for read and write accesses are similar because the additional recovery operations that are necessary for modifying operations are only required for the first modification. These figures clearly show that there is a large overhead associated with accessing atomic objects. The last column shows the times for just detecting an access and reporting the access to the GRT from C** (i.e. with empty prologue and epilogue operations). The largest proportion of the overhead comes from the prologue and epilogue operations.

More detailed times for the prologue and epilogue operations, obtained from Jewel with an older version of Amadeus/RelaX, are shown in Table 4. The prologue operation consumes the most time so the times for each of its constituent parts are also shown (external synchronization deals with synchronization between transactions and internal synchronization deals with synchronization between activities belonging to the same transaction).

Obviously the synchronization and recovery operations are expensive. This may be due to the current implementation of these operations (consisting of

**TABLE 4.** Prologue and epilogue operations (all times in ms)

| | *Read* | *Write* |
|---|---|---|
| Prologue operation | 3.902 | 4.417 |
| external synchronization | 1.367 | 1.508 |
| internal synchronization | 1.219 | 1.427 |
| recovery operations | 1.316 | 1.482 |
| Epilogue operation | 0.517 | 0.702 |

**TABLE 5.** TS recovery operations (all times in $\mu$s)

| | *No. of references in object* | | | | |
|---|---|---|---|---|---|
| | *0* | *2* | *4* | *8* | *16* |
| `Save` | 58.6 | 64.1 | 62.1 | 64.1 | 69.6 |
| `Delete` | 11.7 | 15.6 | 15.6 | 15.6 | 15.6 |
| `Restore` | 27.3 | 27.3 | 35.1 | 35.1 | 46.6 |
| `StableSave` | 341.0 | 375.8 | 1189.4 | 1269.5 | 2329.7 |

approximately 30 000 lines of C++ code) which provides facilities for coping with concurrent accesses to potentially overlapping fragments of objects. However, the times for reporting accesses from C** are reasonable compared to, for example, the times to process a virtual memory fault.

Table 5 shows the times for recovery operations for atomic objects with different numbers of references. The `Save` operation makes an in-memory copy of an atomic object before the first modification of the object within a transaction; the `Delete` operation removes the in-memory copy after the transaction successfully commits; the `Restore` operation overwrites the real atomic object with an in-memory copy after the transaction aborts; and the `StableSave` operation prepares the object for writing to the log after a transaction commits. Note that these times do not include writing the objects to the log.

The times for `Save`, `Delete` and `Restore` are as expected given the simplicity of these operations, though the times for the `Restore` operation rise more sharply than expected. The times for the `StableSave` operation are high because preparing an object requires that all of its pointers to other objects must be swizzled. These figures clearly show that swizzling references incurs a large overhead.

### 5.3. Transaction management

The figures for the transaction management operations shown in Table 6 were obtained from Jewel. In this table, local transactions involve a single context on one node and distributed transactions involve two nodes with one context per node.

The figures for transaction creation are the times from when the application calls the GRT until the target object is invoked. Here, the values for top-level and nested transactions are similar. These times could be improved by the grouping of messages that are sent to the TM.

**TABLE 6.** Transaction management operations (all times in ms)

| | | *No. of objects involved* | | | | | |
|---|---|---|---|---|---|---|---|
| | | *commit* | | | *abort* | | |
| | *Create* | *1* | *10* | *100* | *1* | *10* | *100* |
| Local top-level | 10.9 | 108.4 | 294.2 | 529.1 | 78.3 | 192.0 | 526.9 |
| Local nested | 9.9 | 26.1 | 71.3 | 82.1 | 21.1 | 73.4 | 107.0 |
| Distributed top-level | — | 263.4 | 439.5 | 682.8 | 165.4 | 289.7 | 609.5 |
| Distributed nested | — | 37.2 | 82.6 | 90.8 | 43.5 | 73.4 | 107.0 |

The difference between top-level and nested transaction can be seen in the times for committing and aborting transactions. The figures shown are the times from when the GRT is requested to commit or abort the transaction until control is returned to the application. As expected, top-level transactions are more expensive that nested transactions because the effects of top-level transactions must be durable which requires the use of the log. Commitment of transactions here uses group commitment of ten transactions. That is, the first nine transactions end in the completed state and the 10th transaction commits with the result that all 10 transactions are committed at the same time.

This table shows two surprising results. First, transaction aborts have similar times to transaction commits (sometimes even greater). This is unexpected since there is no two-phase commit protocol executed for aborting transactions. Second, as the number of objects involved (i.e. modified) in a transaction increases, the time for committing the transaction increases substantially and, in addition, the ratio between the times for local and distributed commit processing decreases even more substantially. Both results seem to indicate that the costs of local processing predominate over the communication costs in the execution of the distributed commit protocol.

Finally, the times for setting a transaction's lockpoint are shown in Table 7. Setting a transaction's lockpoint is a global event which is reliably broadcast to all participating RMs. In this table, there is one participating RM (i.e. an Amadeus/RelaX context) on each node. The times were obtained using the `gettimeofday` system call. Obviously this is an expensive operation due to the broadcast. One possible optimization would be to remove the broadcast if the transaction was only involved in one context.

## 6. RELATED WORK

In this section we briefly describe some systems which have influenced our work. These systems fall into two categories: those that support transaction processing in object-oriented systems and those that support open transaction processing.

### 6.1. Argus

Argus (Liskov and Scheifler, 1983) is an integrated object-oriented programming language and transaction system that supports the construction of robust

**TABLE 7.** Setting a transaction's lockpoint (all times in ms)

| *No. of nodes* | |
|---|---|
| 1 | 234.4 |
| 2 | 429.7 |
| 3 | 468.7 |

distributed applications. The transaction model provides fully nested transactions and allows independent top-level transactions to be created. Argus separates atomicity from durability by providing atomic objects and stable data. Atomic objects are instances of either built-in or user-defined atomic types. User-defined atomic types are built using linguistic constructs and provide a means of increasing the amount of concurrency allowed. Argus is a closed system, both in terms of only supporting a single language and not allowing consistent access to other resource types.

### 6.2. Arjuna

The Arjuna (Parrington, 1990) system supports the construction of fault-tolerant, persistent and distributed applications in a C++ programming environment. A class hierarchy is used as the basis for all properties of objects (thus the C++ language has not been modified). Type specific concurrency control and recovery are implemented by redefining operations of certain base classes. A class is also used for creating transactions. The transaction model supports nested transactions. Arjuna is also a single language system and does not support access to other resource types.

### 6.3. Camelot

Camelot (Eppinger *et al.*, 1992) is a distributed transaction facility layered on top of the Mach operating system and based on the client/server model. Both clients and servers can create transactions and access data provided by one or more servers. The transaction model is flexible, supporting a number of commit protocols and logging strategies. A C language interface is provided to access the Camelot facilities. More sophisticated linguistic facilities are provided in the Avalon (Detlefs *et al.*, 1988) language which is an extension of the C++ language. Avalon is heavily influenced by Argus and, like Arjuna, uses the C++ inheritance mechanism to provide certain properties for objects. Though multiple languages can use the facilities provided by Camelot, access to resources provided by external database systems is not permitted.

### 6.4. Encina

Encina (Transarc Corporation, 1991) is a commercial system supporting open transaction processing. It is layered on top of the Open Software Foundation's Distributed Computing Environment (DCE) and has extended the RPC mechanism to be aware of transaction semantics. The transaction model provides nested transactions and allows concurrency within transactions. Access to resources provided by other database systems is supported with the XA-interface. An interface to the facilities of Encina for the C language is provided. However, Encina does not provide any integrated support for object-oriented languages.

### 6.5. Tuxedo

Tuxedo (UNIX System Laboratories, 1991) is another commercial system supporting open transaction processing which has been widely accepted. The transaction model only allows flat transactions and does not allow concurrency within transactions. Access to other database systems is supported with the XA-interface. A C library is available to provide access to Tuxedo's transaction facilities. As with Encina, Tuxedo has not been combined with object-oriented languages.

## 7. CONCLUSIONS

This paper described the integration of object-oriented systems with open transaction processing and described the transaction support provided in Amadeus/RelaX — a platform for the construction of distributed, persistent and reliable applications.

In particular we described the design of the interface to a Generic Run-time Library providing language independent support for atomic objects and transactions while allowing different languages to adapt the support provided to their own requirements.

We also described the interface between a TM and a RM, based on the X/Open XA-interface, which supports the integration of different RMs while providing support for the use of nested transactions and other enhanced transaction functionality.

The current implementation of the Amadeus/RelaX system supports both C** and an extension to the Eiffel language called Eiffel** (McHugh and Cahill, 1993). We have also been able to run C** applications that access atomic objects provided by Amadeus/RelaX and database records provided by the Informix database system in the same transaction.

## ACKNOWLEDGEMENTS

The authors gratefully acknowledge the input of all those who have contributed to the Amadeus/RelaX system. In particular, the contributions related to the topics of the paper are acknowledged: Rolf Heinl, Reinhold Kröger and Michael Wack at GMD; Seán Baker, Neville Harris, Gradimir Starovic and Brendan Tangney at TCD.

## REFERENCES

Baker, S. (1992) *System Issues in Persistent Programming and OODBMS Integration.* PhD Thesis, Department of Computer Science, Trinity College Dublin.

Cahill, V., Baker, S., Starovic, G. and Horn, C. (1993a) The Amadeus GRT: Generic runtime support for distributed persistent programming. *OOPSLA 1993 Conf. Proc.*, **28**, 144–161.

Cahill, V., Balter, R., Harris, N. and Rousset de Pina, X. (eds) (1993b) *The Comandos Distributed Application Platform.* Springer-Verlag, Berlin.

Transarc Corporation (1991) *Encina — Enterprise Computing in a New Age.*

Detlefs, D., Herlihy, M. and Wing, J. (1988) Inheritance of

synchronisation and recovery properties in Avalon/C++. *Computer*, **21**(12), 57–69.

Distributed Systems Group (1992) *C** *Programmer's guide — Amadeus 2.0*. Technical Report TCD-CS-92-02, Department of Computer Science, Trinity College Dublin.

Eswaran, K., Gray, J., Lorie, R. and Traiger, I. (1976) The notions of consistency and predicate locks in a database system. *Commun ACM*, **19**, 624–633.

Eppinger, J., Mummert, L. and Spector, A. (1992) *Camelot and Avalon: A Distributed Transaction Facility*. Morgen Kaufmann, San Mateo, CA.

Gray, J. and Reuter, A. (1992) *Transaction Processing Systems: Concepts and Techniques*. Morgan Kaufman, San Mateo, CA.

Hosking, A. and Moss. J. (1993) Protection traps and alternatives for memory management fo an object-oriented language. In *Proc. 14th Symp. on Operating Systems Principles*, pp. 106–119, Ashville, NC.

Informix Software, Inc. (1991) *Informix-TP/XA: Transaction Manager Interface, User Manual.*

Kröger, R., Mock, M., Schumann, R. and Lange, F. (1990) RelaX — An extensible architecture supporting reliable distributed applications. In *Proc. 9th Symp. on Reliable Distributed Systems*, pp. 156–164, Huntsville, AL.

Lange, F., Kröger, R. and Gergeleit, M. (1992) JEWEL: design and implementation of a distributed measurement system. *IEEE Trans. Parallel Distributed Syst.*

Liskov, B. and Scheifler, R. (1983) Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Programming Languages Syst.*, **5**, 381–404.

McHugh, C. and Cahill, V. (1993) Eiffel**: an implementation of Eiffel on Amadeus, a persistent, distributed object-oriented applications support environment. In Magnusson, B., Meyer, B. and Perrot, J.-F. (eds) *TOOLS 10*, pp. 47–62, Versailles, France.

Mock, M., Kröger, R. and Cahill, V. (1992) Implementing atomic objects with the RelaX transaction facility. *Comput. Syst.*, **5**, 259–304.

Moss, J. (1981) *Nested Transactions: An Approach to Reliable Distributed Programming*. PhD thesis, MIT.

Nett, E., Grosspietsch, K., Jungblut, A. *et al* (1985) *PROFEMO — design and Implementation of a Fault Tolerant Distributed System Architecture*. GMD-Studie 100, GMD.

Nett, E., Kaiser, J. and Kröger, R. (1986) Providing recoverability in a transaction oriented distributed system. In *6th Int. Conf. on Distributed Computing Systems*, Cambridge, MA.

Nett, E. and Mock, M. (1993) Generic action support for distributed cooperative applications In *Workshop on High Performance Transaction Processing (HPTS)*, Pacific Grove. Also available as GMD-Arbeitspapier 785.

Parrington, G. (1990) Reliable distributed programming in C++: the Arjuna approach. In *USENIX C++ Conf.*, pp. 37–50, San Francisco, CA.

Rashid, R. (1986) Threads of a new system. *Unix Rev.*, **4**(8), 37–49.

Stroustrup, B. (1991) *The C++ Programming Language*, 2nd edn. Addison-Wesley, Reading, MA.

Taylor, P. (1993) *Transactions for Amadeus*. Master's thesis, Department of Computer Science, Trinity College Dublin.

UNIX System Laboratories (1991) *The Tuxedo System: Product Overview*.

X/Open Company Limited (1991) *Distributed Transaction Processing Reference Model: The XA Specification*. Berkshire.