Maintaining Consistency in Distributed Software Engineering Environments

J. Walpole  G.S. Blair  J. Malik  J. R. Nicol

Department of Computing, University of Lancaster, Bailrigg, United Kingdom, LA1 4YR.

ABSTRACT

Considerable research effort in the Software Engineering domain has been focused on the development of more advanced programming environments and Software Engineering Environments (SEEs). It is generally agreed that these systems should be built on a distributed base. However, the distribution of computer systems introduces several problems which make it very difficult to maintain the consistency of data. The concept of atomic transactions has been used extensively, particularly in database systems, to maintain consistency. However, conventional database transaction mechanisms are unsuitable for use in distributed Software Engineering Environments.

In this paper we argue that immutability is a suitable base on which to build distributed Software Engineering Environments. We then discuss the various approaches to maintaining consistency in immutable object systems and compare Reeds model of time domain addressing with our own model of domain relative addressing. Finally, we demonstrate the suitability of domain relative addressing for use in distributed Software Engineering Environments.

1. Introduction

Considerable research effort in the Software Engineering domain has been focused on the development of more advanced programming and Software Engineering Environments (SEEs) [1]. These systems are intended to ease the problems of managing large software projects which involve the coordination of teams of development personnel.

It is generally agreed that such environments should be built on a distributed base. Distributed systems have several potential advantages, e.g. extensibility, availability, and performance [2]. There are however, several difficult problems to be solved before the advantages of distributed systems can be realised fully. These problems are caused by the following important differences between distributed and single machine systems [3,4]:-

(1) There is the possibility of partial failure in distributed systems due to individual node failures, or network failure. This is far more difficult to deal with than total failure.

(2) Distributed systems are based on an unreliable communications sub-system which can cause the loss or delay of messages.

In such an environment reliability becomes an important concern.

To achieve high reliability in a distributed SEE, the problems of failures and concurrency must be solved. In an attempt to solve these problems in database systems, and more recently in general purpose distributed systems, the notion of atomic transactions has been introduced [5,6,7,8,9]. Transactions provide the means to group operations together and execute them atomically. That is, either all the operations complete successfully and their results become visible, or none of their effects become visible and the transaction aborts. Furthermore, transactions are indivisible with respect to other concurrently running transactions. These features simplify the treatment of failures and concurrency and provide the user with a mechanism which, provided transactions are programmed correctly, guarantees the consistency of data in the system [5].

The transaction concept is clearly applicable to distributed SEEs. However, existing transaction mechanisms were not designed for distributed SEEs and therefore their suitability for application to this environment requires some consideration.

This paper investigates the problems of maintaining the consistency of data in a distributed SEE and outlines the special requirements which are placed on transaction mechanisms by distributed SEEs. A new transaction mechanism, designed specifically for use in distributed SEEs, is presented. This forms an integral part of the Cosmos Distributed Programming Environment which is being developed at the University of Lancaster [10].

This paper consists of several sections. Section 2 outlines the special requirements placed on transaction mechanisms by distributed SEEs. Section 3 then introduces an abstract model developed in the Cosmos project. This model includes immutability as one of its main features and is a major influence on the transaction work. The repercussions of immutability for transactions are then discussed in section 4. Reed's model of time domain addressing is described and is shown to be inadequate for supporting SEEs. An alternative approach, called domain relative addressing is then introduced. The mechanisms required to support domain relative addressing are outlined in section 5. Section 6 then discusses synchronisation and recovery in the context of domain relative addressing. An evaluation of domain relative addressing for use in SEEs is presented in section 7. Finally, section 8 contains some concluding remarks.

2. Transactions and Distributed Software Engineering Environments

It is generally recognised that databases should be a central feature of SEE designs. A comparison of conventional database systems and SEE databases reveals several significant differences which have serious repercussions for the design of
SEE transaction mechanisms. Firstly, the granularity of objects in SEE databases is coarser than in conventional databases. Objects in a SEE database often contain large amounts of possibly unstructured data, such as text or source code. Secondly, transactions in a SEE database frequently involve interaction with users in activities such as editing. The combination of these two features makes the duration of transactions in a SEE database typically much longer than in conventional databases. For example, a transaction to build part of a software system may involve edits and compilations of large objects. This process could last for several hours or even days. Support for long term transactions is therefore an important issue in SEEs.

Nested transactions [11] are also very important in a SEE. Nested transactions allow the structure and interdependencies of operations within large transactions to be represented. This is particularly useful in SEEs for activities such as system building. Nested transactions can also be used to increase the reliability and performance of large transactions.

Several other SEE database requirements should also influence the design of the transaction mechanism. In particular, the requirements for version control [12,13] and configuration control [14,15] are highly relevant. Transactions, version control and configuration control have been traditionally treated as separate problems. However, they are all associated with controlling change in a SEE database. Integration of transactions, version control and configuration control would be a major step forward forSEE design.

Existing transaction techniques are not well suited to the specific requirements outlined above. For example, long term transactions cause severe problems for existing synchronisation techniques which are based on the notion of serialisability [16,17]. Traditionally, consistency and serialisability have been seen to be inseparable [5]. However, to support long term transactions adequately it may be necessary to consider the possibility of non-serialisable transactions.

Nested transactions also present particular problems for transaction mechanisms. For example, the high multiprogramming level generated with nested transactions can exacerbate the problem of deadlock [9]. Long term transactions add considerably to this problem. It is therefore beneficial to design transaction mechanisms for SEEs which are deadlock-free.

Finally, no existing transaction mechanism is designed to be used with version control and configuration control.

The problems mentioned above are emphasised by Gray [9] -

"Our concept of transaction and the implementation techniques we have are inadequate to the task of many applications. They cannot handle nested transactions, long-lived transactions and they may not fit well into conventional programming systems.

We may be seeing the Peter Principle in operation here: "every good idea is generalised to its level of inapplicability". But I believe that the problems I have outlined here (long-lived and nested transactions) must be solved."

This quote applies particularly well to SEEs.

The arguments presented above are a summary of a study carried out by the authors [18]. The conclusion of this work was that existing approaches to distributed atomic transactions are inadequate for use in distributed SEEs.

This observation provided the stimulus for us to consider the problem of designing and implementing transaction mechanisms specifically for distributed SEEs. This led to the development of a new solution to the problem. The solution is based on an abstract model developed within the Cosmos project. This model is therefore described in the next section.

3. The Cosmos Model

The Cosmos model is an abstract model tailored for a programming environment sub-system. It is designed specifically to provide direct support for programming environment functions and also to operate in a distributed environment [19,20,21]. The model has been used as the basis for a pilot implementation of the Cosmos Distributed Programming Environment* [22]. In describing the model, it is convenient to start with the object model (as introduced by Jones [23]).

The Object Model

In the object model, all resources are represented by a single abstraction called an object. An object is an encapsulation of a persistent data item, operations to manipulate the data, and information pertaining to the data (e.g. access rights). All access to the data must be through the well-defined operational interface.

The object model has proved popular with the designers of both SEEs [24,25,26,27] and more advanced (distributed) operating systems (including, for example, Clouds [28] and Eden [29]). The object model has been applied successfully as a structuring concept in complex systems. This is reflected in its wide range of applicability. In addition, the object model lends itself well to distribution [30,10].

Extensions to the Object Model

The object model provided a useful starting point for the Cosmos design. However, several significant enhancements to the object model have been made in Cosmos. These are discussed in the following sub-sections.

Semantic Information: A central feature of Software Engineering Environments is the existence of a database. The semantic information held by such a database holds the key to providing the intelligence and integration required of SEEs. The object model has been enhanced in two ways to provide a database capable of storing the semantic information needed by Cosmos. Firstly, further semantic information pertaining to objects is stored in the form of a general set of attributes. Secondly, object inter-relationships are recorded in the database.

Immutability: The most notable enhancement to the object model in Cosmos is the immutability of objects [31]. Object immutability means that once an object has been created it cannot be changed. The notion of updating an object is replaced by the notion of transformation. A transformation consists of a sequence of operations which produce a new object. The old object remains unchanged. An important property of transformations is that they are atomic. The atomicity of transformations is achieved by creating a shadow copy of the old object. The operations are then performed on this copy which is only made visible when the transformation commits. In this way, the intermediate states of the pending new object remain invisible until commit time. This makes it possible for transformations to maintain the consistency of individual objects.

A system with immutable objects is considerably different from a conventional, update in place environment. Objects now exist as a series of versions representing their

---

*The term Programming Environment is used because data emphasis has been placed on the implementation phases of the software life-cycle only. Therefore, the pilot implementation is not a full SEE.
evolution. Additionally, in the general case objects may be replicated (for increased availability). To be more precise about this new environment, some terminology is introduced.

A conceptual object refers to a logical or real world entity being modelled by the system (e.g. a user's mailbox). A conceptual object is represented as a history of states (perhaps involving branches). A version represents a node (or state) in the history of a conceptual object. A replica is a representative (or copy) of a version in a conceptual object's history.

4. Consistency in Immutable Systems

In update in place systems, consistency is concerned with relationships between the values of multiple objects. Objects are related by a set of assertions called consistency constraints. The consistency constraints of a system define the relationships which should hold if the system is to remain consistent. In immutable systems, multiple versions of conceptual objects are maintained. Consequently, consistency constraints may hold between some versions of related conceptual objects but not between others. The problem of maintaining consistency in immutable systems is therefore a matter of indicating which objects in the system constitute a particular consistent state. Therefore, transaction mechanisms should be responsible for naming consistent groups of versions of related conceptual objects. The production of new consistent states must also be controlled.

Reed recognised the importance of naming in immutable systems and proposed the technique of time domain addressing as a solution to the transaction problem [32,33].

Time Domain Addressing

In Reed's scheme, multiple versions of objects are maintained. An update operation on an object causes the creation of a new version of that object. In this way objects evolve to contain additional information rather than overwriting the old value with the new one, as is the case in conventional update in place systems.

Each version of an object has two timestamps (called pseudotimes) associated with it. The first of these represents the time of the transaction which created that version, and the second represents the time at which that version was last read.

Synchronisation is achieved by assigning a pseudotime value to each transaction. Consider an object A which has been read at pseudotime p2. If an attempt is made to write to A at pseudotime p1, such that p1>p2, then the write is aborted. This prevents the read from being invalidated (see figure 1).

Figure 1: Read/write conflict in time domain addressing.

In time domain addressing the notion of pseudotime has the effect of temporally ordering all the objects in the system. Transactions are then restricted to only seeing a slice through the system's object space (see figure 2).

Figure 2: Global ordering in time domain addressing.

This global ordering is imposed to ensure that transactions are serialisable. However, serialisability is an unnecessarily strong condition for maintaining consistency in an immutable system. To maintain consistency in immutable systems, it is sufficient to name consistent sets of objects. Some ordering may be imposed on objects, for example to make it meaningful to reason about the most recent version of a conceptual object. However, this extra synchronisation is optional.

There are further problems with time domain addressing. Firstly, the requirement for serialisability in time domain addressing can cause an excessive number of aborts. This problem is exacerbated in long term transactions because pseudotimes are allocated to transactions at their start time. Therefore, the later operations within a transaction have an increased probability of aborting the transaction in the way described above. Secondly, a direct result of the requirement for serialisability is that conceptual objects are forced to have linear version histories; transactions can only transform the most recent version of a conceptual object. This restriction makes it difficult to integrate time domain addressing with the function of version control in a SEE because SEEs require equal access to both old and new versions of an object and the ability to create branches in the version history.

In conclusion, the technique of time domain addressing restricts concurrency unnecessarily by maintaining serialisability. This is a serious restriction, especially for long term transactions. Furthermore, the limitation that version histories must be linear makes time domain addressing unsuitable for use in a SEE.

Rather than ordering the objects in the system, a second approach is to name consistent sets of objects explicitly. This forms the basis of a new approach called domain relative addressing presented in this paper.

Domain Relative Addressing

Domain relative addressing does not impose any global ordering on the system's objects. Instead, consistent states are designated by storing relationships between the appropriate versions of related conceptual objects. In this way, the system's object space is divided into separate consistent domains. Each domain contains one version of each of the conceptual objects in a related set (see figure 3).

By not requiring serialisability, domain relative addressing can also be used to increase concurrency for long term transactions. A simple example of this is presented in example 1.
is especially important for supporting long term transactions.

Another important advantage of domain relative addressing is that it enables the semantics of transactions to be recorded accurately in the database. This is demonstrated in the example above by the fact that the relationships between the object pairs (A1,B1) and (A2,B2) are preserved in the database.

The following section outlines the mechanisms required to support the use of domain relative addressing in a distributed SEE.

5. Mechanisms to Support Domain Relative Addressing

To support domain relative addressing in a SEE, the solution proposed in this paper involves the inclusion of three specific categories of object: *history objects*, *configuration objects* and *transaction objects*. These are discussed in the following sections.

### History and Configuration Objects

The objects in an immutable system can be considered to exist in a two dimensional space. The first dimension corresponds to the temporal ordering of versions of individual conceptual objects. The second dimension encompasses the notion of related groups of versions across several conceptual objects. These constitute consistent domains and are referred to as configurations. This concept is illustrated in figure 5.

The naming mechanism must enable relationships in both dimensions to be distinguished. Hence, in Cosmos, history objects are used to represent the vertical links of figure 5, and configuration objects are used to represent the horizontal links of figure 5. History and configuration objects can be defined more precisely as follows:

* A history object is a directed graph of objects representing changes over time of a particular conceptual object.

* A configuration object is an unordered set of objects which defines a consistent domain.

Together, history and configuration objects constitute a pair of coordinates for naming objects in an immutable system. However, the uncontrolled use of history and configuration objects results in a complex structure in which it is difficult to name objects.

There are two alternative ways of simplifying the task of naming objects using history and configuration objects. Firstly,
configuration objects can be used to distinguish groups of consistent versions. The configuration objects can then be version controlled (see figure 6).

![Figure 6: Versions of configurations.](image)

Hence, a history object would be a directed graph of configuration objects, and a configuration object would be an unordered set of versions of related conceptual objects. This would mean that the evolution of the system from one configuration to the next would be recorded in history objects, and that individual objects would be named relative to a particular configuration.

Alternatively, history objects could be used to store the version histories of individual conceptual objects. Configuration objects could then be used to group together related conceptual objects. However, this would mean that different versions of the same conceptual object would be named by the same configuration object. This does not allow consistent versions of related conceptual objects to be distinguished. This is the main purpose of configuration objects. Therefore, the first approach is clearly preferable.

History and configuration objects can now be redefined more specifically as follows:

* History object: a directed graph of configuration objects.

* Configuration object: an unordered set of versions of related conceptual objects.

Using this scheme, objects are named by first accessing a history object to obtain a configuration object. This designates a particular consistent domain. The desired object is then named relative to this domain.

Once an object has been named in this way, the other objects with which it is consistent are contained in the same consistent domain and are therefore named by the same configuration object. This ensures that all the reads in a transaction are consistent. Similarly, every object which is created (or transformed) by a transaction is grouped to form a new consistent domain. This new domain is named by a single new configuration object. Unchanged objects from the previous consistent domain are inherited into the new domain. If the new objects are only nameable through the configuration object, then a transaction can commit its results simply by linking the new configuration object into the Cosmos naming structure atomically [171]. Thus, the results of a transaction only become visible at the point when the newly created configuration object is linked into the naming structure.

A natural extension to the transaction model is to allow configuration objects to be nested. The resulting hierarchy of configuration objects forms an ideal basis for nested transactions. Each sub-transaction within the overall nested transaction creates its own configuration object when it starts. At sub-transaction commit time, the contents of this configuration object are included in the configuration object of the parent transaction. The parent configuration object is only committed once all its children have committed. Finally the top level transaction is committed by linking its configuration object into the naming structure. Therefore, the results of the complete nested transaction are made visible in a single atomic step.

### Transaction Objects

Transaction objects are used as an entry point for naming the partial results of a transaction. Transaction objects also hold state information about a given transaction and can be stored indefinitely in the database. This is important for supporting long term transactions because it allows transactions to exist across process boundaries.

The first task of a newly started transaction is to create a transaction object. Transaction objects remain private to the initiator of the transaction until the transaction is committed. Once created a transaction object can be opened and closed. Closing a transaction object causes the partial results of the associated transaction to be saved in a new version of the transaction object, thus creating a save point. Opening a transaction object causes the transaction to be restarted from a specified save point.

When the transaction commits, the committed transaction object becomes a configuration object and is linked into the naming structure. Transaction objects can therefore be considered as configuration objects in a transient state.

### 6. Synchronisation and Recovery in Domain Relative Addressing

Transaction algorithms can generally be considered to consist of two separate parts: synchronisation and recovery [34,35]. Therefore, algorithms can be described as having a synchronisation scheme [36] and a recovery scheme [37,38].

In domain relative addressing, synchronisation and recovery are not of central importance; they have a very different interpretation in domain relative addressing. The exact meaning of synchronisation and recovery in domain relative addressing is discussed in the following sub-sections.

#### Synchronisation in Domain Relative Addressing

The synchronisation problem in domain relative addressing can be reduced to a problem of controlling the branching of version histories. However, for transactions it is the version histories of configuration objects rather than conceptual objects which must be controlled. For synchronisation purposes, configuration objects can be defined as one of the following three classes: free branching, controlled branching, or linear. Each class of object is now considered in turn.

**Free Branching Objects**: For free branching configuration objects, no synchronisation is necessary. Should two transactions concurrently transform the same configuration
the class of the configuration can be changed to controlled. However, concurrent transactions are expected to be rare for this class of configuration object. Free branching is intended to be used for configurations which are not shared. For example, free branching can be used for groups of objects being developed in a user's private workspace. If it becomes necessary to share any of these objects at some later stage then the class of the configuration can be changed to controlled branching. Since no synchronisation is required, free branching is very lightweight. Figure 7 illustrates a possible scenario for concurrent transactions in a SEE. In the case of a free branching configuration, transactions T1, T2 and T3 will always be permitted to proceed concurrently.

Controlled Branching Objects: For controlled branching configuration objects, attempts to transform a single configuration object concurrently result in a warning. To prevent concurrent transformations from occurring, an advisory lock is applied to the configuration object being transformed. This type of lock is maintained for the duration of the transaction. However, it does not prevent other transactions from reading the locked configuration object. Should a transaction attempt to transform the locked configuration object a warning is returned. For example, in the scenario of figure 7, it is possible for either transaction T2 or transaction T3 to receive a warning (depending on the exact timing of the two transactions). It is possible to ignore a warning if required; the competing transaction would then proceed to create a branch in the version history of the configuration. It is important to note that this branch is created intentionally and not unintentionally as is possible in free branching.

Controlled branching configuration objects can be used, for example, in software development where a team of programmers are working together (i.e., members may wish to share access to objects but be informed of potential conflicts).

Linear Objects: For certain conceptual objects, it is desirable for all users to see the same (i.e., most recent) version in the version history. An example of a conceptual object belonging to this category would be a system object containing state information such as user names and their encrypted passwords. Such conceptual objects are referred to as linear objects. To model the semantics of a linear object, the object being transformed is locked for both reading and transformation. This type of lock cannot be ignored by other transactions; other transactions must back off and retry later. To avoid the problem of livelock a binary exponential backoff function is used to calculate retry times. When a transaction commits, a new configuration object is created and the lock on the previous configuration object is made permanent. Old versions with permanent locks can be discarded. This mechanism guarantees that no branching can occur in the object history and also that only one version is ever accessible.

Recovery in Domain Relative Addressing

To discuss recovery in domain relative addressing, it is necessary to distinguish between failures before commit time and failures after commit time. In conventional transaction systems, recovery from failures before commit time involves backward error recovery and recovery from failures after commit time involves forward error recovery. An important feature of domain relative addressing is that no forward error recovery is needed. This is because the commit point of a transaction is signalled by the entry of a configuration object into the naming structure. This is a single atomic operation and is the last operation to occur in a transaction.

Recovery from failures before commit time can be divided into two separate problem areas. The first is concerned with removing any objects which have been created as partial results of a transaction. The second problem is concerned with removing any locks which may have been applied by a transaction.

Since the objects created as partial results of a transaction are only made visible after commit time, it is not strictly necessary to remove them after failures. This part of the recovery problem for domain relative addressing has therefore been reduced to a garbage collection problem.

The second recovery problem varies according to the class of configuration object being transformed. For free branching configuration objects, no recovery is necessary because no locks are applied. For both controlled branching and linear configuration objects recovery involves releasing any locks which were applied by the transaction.

In the case of controlled branching objects, any advisory locks which remain after a failure can be ignored, and therefore do not result in a loss of availability for the configuration. However, for linear objects, locks cannot be ignored and therefore the system must ensure that they are released during recovery. Algorithms have been developed to handle recovery of locks [17]. However, more detailed information on the algorithms is beyond the scope of this paper.

7. Evaluation of Domain Relative Addressing

In this section, the new technique of domain relative addressing is evaluated for use in a SEE. The criteria introduced in section 2 are revisited.

* Supporting Nested Transactions

Nested transactions can be accommodated by allowing configuration objects to be nested. As described above, consistency is maintained by linking the top level configuration object into the naming structure at commit time.

* Supporting Long Term Transactions

Long term transactions are supported naturally by the use of transaction objects and save points as discussed.

---

* The algorithm used for advisory locks is based on Gifford's weighted voting scheme [39]. However, a variation of the basic algorithm has been produced for our immutable environment [40]. This algorithm was chosen for its flexibility and for its support for replication.
earlier. Representing transactions in this way allows transactions to exist across process boundaries and to survive failures. Free branching and controlled branching configurations allow an unlimited level of concurrency between transactions, making them ideal for the support of long term transactions. This is primarily due to the fact that neither of these synchronisation schemes require serialisability to be maintained. However, the synchronisation imposed on transactions by linear configurations does have the effect of maintaining serialisability and therefore incurs at least the same overheads as conventional synchronisation schemes. This makes linear configurations unsuitable for supporting long term transactions.

* Supporting Version and Configuration Control

The concept of history and configuration objects provides an ideal basis for the functions of version and configuration control. Configuration objects are used to name consistent groups of objects. Immutability ensures that once created the relationships within a configuration are not invalidated. Furthermore, the transaction mechanism enables new consistent configurations to be produced atomically. Similarly, history objects maintain a complete version history of a given conceptual object through time. Domain relative addressing is also flexible enough to support branching in version histories and also to give warnings of potential conflicts.

Current research is investigating the design of an object management system for Cosmos. This will be built on top of the mechanisms described in this paper and will present a more user oriented interface to versions and configurations. The object management system will also allow the structure and semantics of configurations to be represented.

8. Conclusion

This paper has presented a new approach to atomic transactions, designed specifically for use in distributed SEEs. The approach is based on the concept of immutable objects. As a result of immutability, the consistency problem is replaced by a naming problem. A solution to this naming problem has been presented in the form of domain relative addressing. Domain relative addressing does not require serialisability to maintain consistency and therefore it can support a higher degree of concurrency.

The paper has also addressed the repercussions for synchronisation and recovery in domain relative addressing. Synchronisation has been shown to be optional; consistency can be maintained in the case of free branching configurations without synchronisation. Further synchronisation for controlled branching and linear configurations has been shown to be a problem of controlling the shape of version histories. In the case of free and controlled branching configurations, serialisability is not a requirement. This makes these classes of configuration suitable for supporting long term transactions.

Recovery in domain relative addressing has been shown to be a garbage collection problem. It is possible to maintain consistency without performing recovery. Therefore, recovery is also optional for domain relative addressing. However, recovery will be required to minimise space overheads.

Finally, a very important feature of domain relative addressing is that it provides a unified basis for the functions of version control, configuration control and transactions in a distributed SEE. These functions are treated traditionally as separate problem areas. However, they are all concerned with aspects of the same underlying problem: controlling change in the SEE database. Therefore, the work reported in this paper represents one step towards unifying the various functions to move towards a truly integrated Software Engineering Environment.

REFERENCES
