Implementing Concurrency Control in Reliable Distributed Object-Oriented Systems

Graham D. Parrington and Santosh K. Shrivastava
Computing Laboratory,The University of Newcastle upon Tyne,
Newcastle upon Tyne, NE1 7RU, U.K.

Abstract

One of the key concepts available in many object-oriented programming languages is that of *type-inheritance*, which permits new types to be derived from, and inherit the capabilities of, old types. This paper describes how to exploit this property in a very simple fashion to implement object-oriented concurrency control. We show how by using type-inheritance, objects may control their own level of concurrency in a type-specific manner. Simple examples demonstrate the applicability of the approach. The implementation technique described here is being used to develop *Arjuna*, a fault-tolerant distributed programming system supporting atomic actions.

Key words and phrases: Object-oriented programming, Concurrency control, Reliability, Atomic actions, Type-inheritance.

1. Introduction

A widely used technique of introducing fault-tolerance - particularly in distributed systems - is based upon the use of *atomic actions* (atomic transactions) for structuring programs [Gray 78]. An atomic action possesses the properties of *serialisability, failure atomicity and permanence of effect.* A wide variety of concurrency control algorithms have been proposed in the literature to ensure the serialisability property of atomic actions. This paper proposes a practical implementation technique for such concurrency control algorithms within the framework of an object-oriented system. The basic idea behind the approach is quite straightforward. It is assumed that the underlying operating system provides rudimentary synchronisation facilities, such as semaphores, for concurrent processes. A type can then be constructed to provide a specific concurrency control technique by exploiting these facilities. User-defined types can *inherit* this underlying concurrency control facility by making use of the type-inheritance mechanism available in object-oriented programming languages. If desired, the inherited concurrency control mechanism can also be *refined* to provide a *type-specific* concurrency control mechanism.

This paper reports on simple experiments that have been performed to show that the approach presented here does provide a very flexible means of implementing concurrency control techniques for object-oriented systems. Other papers [Dixon and Shrivastava 87, Dixon *et al.* 87] have shown how the same type-inheritance technique can be used to add the properties of failure atomicity and permanence of effect to objects with equal ease.

The use of inheritance in this fashion has a number of advantages. Firstly, it is not necessary to design and implement either a new language and run-time system, nor an operating system kernel. Secondly, it enables experimentation with different concurrency control and recovery techniques in a straightforward fashion, since the capabilities are not tied into any particular system. This approach may be contrasted with that taken by other systems in the same area including Clouds [Dasgupta *et al.* 85], Argus [Liskov and Scheifler 83], TABS [Spector *et al.* 85], and ISIS [Birman 86].

This paper is structured as follows. Section two reviews the main ideas on objects, actions and some widely used concurrency control techniques such as two-phase and type-specific locking. Section three introduces the inheritance-based concurrency control implementation technique, after first describing the relevant features of the C++ object-oriented language in which the implementations have been carried out. Sections four and five discuss some specific concurrency control implementations to illustrate the feasibility of the proposed approach. Finally, section six describes how this approach is integrated within *Arjuna* - a distributed system supporting atomic actions.

2. Objects, Actions and Concurrency Control

An object is an instance of some *type* or *class*. Each individual object consists of some variables (its *instance* variables) and a set of operations (its *methods*) that determine the externally visible behaviour of the object. The operations provided by a type have access to the instance variables and can thus modify the state of an object. Furthermore, the type of an object determines what operations may be applied to it.

In a distributed system, an operation on an object is typically invoked via a *remote procedure call* (RPC) (see [Liskov and Scheifler 83, Spector *et al.* 85, Birman 86, Dixon *et al.* 87] for some typical fault-tolerant, object-based architectures). Programs which operate on objects are executed as *atomic actions* with the properties of (i) *serialisability*, (ii) *failure atomicity*, and (iii) *permanence of effect*. The first property ensures that concurrent execution of such actions is free from interference (that is, concurrent executions can be shown to be equivalent to some serial order of execution [Eswaran *et al.* 76, Best and Randell 81]). The second property ensures that an action can either be terminated normally (*committed*) producing the intended results, or *aborted*, producing no results. This property can be obtained by the appropriate use of *backward error recovery*, which is invoked whenever a failure that cannot be masked occurs. Typical failures causing an action to be

aborted are node crashes and communication failures such as lost messages. It is reasonable to assume that once an action terminates normally, the results produced are not destroyed by subsequent node crashes. This is the property of permanence of effect which ensures that state changes produced are recorded on *stable storage* which can survive node crashes with a high probability of success. A *commit* protocol is required during the termination of an action to ensure that either all of the objects updated within the action have their new states recorded on stable storage (normal or committed termination), or no updates get recorded (aborted termination).

In object-based systems, the encapsulation properties of objects make it seem natural to require that the implementation of each object type be made responsible for enforcing concurrency control and for implementing properties such as backward error recovery. A separate paper [Dixon and Shrivastava 87] has described how a type can implement backward error recovery by exploiting the type-inheritance facilities of an object-oriented programming language. This paper applies similar ideas to the problem of concurrency control.

2.1. Object-Based Concurrency Control

A very simple and widely used concurrency control technique is to regard all operations on objects to be of type *read* or *write*, which must follow the well known synchronisation rule permitting concurrent read operations but exclusive write operations. This rule is imposed by requiring that any action intending to perform an operation that is of a *read* type (*write* type) on an object must first acquire a *read lock* (*write lock*) associated with that object. To guarantee serialisability, all actions must follow a *two-phase* locking policy [Eswaran *et al.* 76] . During the first phase, termed the *growing* phase, an action can acquire locks on objects but must not release any acquired locks. The last phase of the action constitutes the *shrinking* phase, during which time held locks can be released, but no new locks can be acquired. It is also necessary to make the shrinking phase appear instantaneous by releasing all of the held locks simultaneously, to ensure that an action can be aborted unilaterally, without affecting other ongoing actions and thus avoiding the possibility of cascade-rollback.

This policy of shared read but exclusive write represents a *lock conflict rule* for an object: read-read locks do not conflict but read-write and write-write locks do. There are situations where this conflict rule can be regarded as overly restrictive, from the point of view of permissible concurrency. Consider a simple example.

Suppose there is some directory object providing the operations: *addentry*(...) (to add an entry in the directory), *rmentry*(...) (to remove a specified entry from the directory), and *lookup*(...) (to look for a given name in the directory). If *addentry* and *rmentry* operations are taking place on different entries then there is generally no reason why these two operations

cannot be permitted to occur concurrently. This observation leads to the notion of *type-specific locking*, whereby a type definition includes a type-specific lock conflict rule [Schwarz and Spector 84]. Type-specific locking is a promising technique for increasing the permissible concurrency in object-based systems supporting atomic actions.

3. Utilising Type-Inheritance for Concurrency Control

Several modern programming languages support the property of *type-inheritance* - the ability for newly constructed types to acquire the properties and operations of the base types out of which they have been constructed. This section examines how this property can be used to implement object-oriented concurrency control. The approach is to construct an appropriate concurrency control base type which can then be used to derive more specific (user) types as required.

3.1. Type-Inheritance in C++

The language we will use to describe our objects is C++ [Stroustrup 86] largely because of its ease of availability and its incorporation of the features we require. C++ is an object-oriented superset of the language C, and includes facilities for type-inheritance, data abstraction, and operator overloading. The data abstraction and type-inheritance facilities are based on the *class* concept. Instances of a class are objects, with specific operations provided for their manipulation. The type-inheritance mechanism of C++ works as follows: given a base class C_1, another class C_2 - a *derived* class of C_1 - can be defined so that it inherits some or all of the operations of C_1.

Classes are defined in the manner shown in Figure 1(a) which is a skeleton declaration of a class called *baseclass*. The variable and function declarations which occur before the *public* label are private members of the class; the only operations which may access private variables or invoke private operations are the member operations of the class itself (in this example, *baseclass*, ~*baseclass*, *op1*, *op2* and *op3*). The variable and operation declarations following the public label constitute the public interface to objects of the class (here, *op2* and *op3* in Figure 1(a); the operations *baseclass* and ~*baseclass* are special, see below). An example of a class derived from the *baseclass* class is shown in Figure 1(b). This new class, called *derived*, inherits the public operations *op2* and *op3* from *baseclass*. In this example these inherited operations are also made public operations of the derived class by the use of the keyword *public* in the class header.

Each class may have a *constructor* which is a public operation with the same name as the class (*baseclass*() and *derived*()), and which will be invoked each time an instance of the class is created. There is also a complementary operation (~*baseclass*() and ~*derived*()),

```
class baseclass
{
        int val1;
        int val2;
        op1 ();
public:
        baseclass ();
        ~baseclass ();

        op2 ();
        op3 ();
};

        (a)
```

```
class derived : public baseclass
{
        int val3;

public:
                derived ();
                ~derived ();

                dop4 ();
                dop5 ();
};

        (b)
```

Figure 1: C++ syntax

called a *destructor*, which is invoked automatically when the object is deleted. The constructor allows an object to perform type-specific initialisation, and the destructor enables an object to tidy up before it is deleted. Both operations are special in that they will be automatically invoked when objects are created or deleted and even though a part of the public interface to the object, they cannot be directly invoked by a user of the object.

3.2. The Basic Concurrency Control Scheme

The basic concurrency control scheme relies on the inheritance features of the language C++. Concurrency control is achieved by first defining an appropriate class (called *LockCC*) which provides the basic concurrency control facilities that are required, and then using that class as a basis from which to derive other specific user-defined classes. Given such a class, Figure 2 shows how a user-defined type called *File* is defined so that it inherits the capabilities of *LockCC*. Use of the operations provided by *LockCC* must be explicitly coded as part of the code for the operations of the newly derived class (for example, the *open* operation in Figure 2 should contain appropriate calls on the operations of *LockCC*). This point will be explained further in a later section.

3.3 Representing Locks

In order to make the system as flexible as possible, locks are deliberately not made special immutable system types (in contrast to systems such as Clouds and Argus). Rather, in this system locks are simply another type that can be refined as required in the same fashion as any other type. This approach has several advantages. Firstly, locks can be

```
class File : public LockCC
{
        ...                     // private file data
public:
        File ();
        ~File ();

        open (...);             // standard file operations
        close (...);
        ...
}
```

Figure 2: The class *File*

created and manipulated in the same fashion as any other object in the system. Secondly, we do not require any new language features or modifications to the run-time environment to support them. Thirdly, the approach is very flexible, permitting different concurrency control policies to be adopted with surprising ease.

In Figure 3 we show a skeleton declaration of the *Lock* class. Instances of this class are

```
class Lock: public Object
{
        modetype lockmode;
        Uid owner;
        ...
public:
        Lock (modetype) ;
        ~Lock ();

        modetype getmode () { return lockmode; }
        Uid getowner () { return owner; }
        ...
        virtual boolean operator!= (Lock*);
}
```

Figure 3: The class *Lock*

created as needed by the user and then passed as arguments to the *setlock* operation provided by *LockCC*.

Since instances of the *Lock* class are simply objects, they obey the usual object-oriented rules regarding encapsulation. Thus the *Lock* class provides operations to retrieve certain parts of its internal state rather than allow public access to the actual instance variables. Note that *Lock* is itself a derived class (of *Object*), and thus inherits the capabilities and operations of that class. The reasons for this will be covered in section six.

The mode of a lock object is immutable. Thus having declared a lock object to be a read lock, the object is always a read lock. If a write lock is required a new lock object with the appropriate mode must be created. This restriction relates to the way in which locks are expected to be used. It seems unlikely that having created a lock the programmer would want to change its mode from say read to write (this process of *lock conversion* can be handled in a different fashion in this system as will be explained in a later section).

The boolean operator *!=* defined for the *Lock* class is declared to be *virtual* so that it may be redefined in any class that is derived from *Lock*. This operator is used by one of the internal (private) operations of *LockCC* to ascertain whether two locks conflict; its use will become clearer in the following section.

3.4 The Concurrency Control Class LockCC

The previous section outlined the basic *Lock* class that is supplied as a parameter to the operations of the concurrency control class *LockCC*. In this section the *LockCC* class itself is described, as is the manner in which it manipulates the locks that are passed to it.

Recall that, as mentioned in section 3.2, a user-defined class requiring concurrency control is derived from the class *LockCC* which manages the concurrency control information. So, if several instances of such a user-defined class are created, each instance will possess the capability of maintaining its own concurrency control information on a purely local basis.

The *LockCC* class provides two basic public operations; *setlock*, whose task is to set a lock upon the user-defined object; and *releaselock* whose task is to unlock the object. The private information maintained by the *LockCC* class includes a list of *Lock* objects that are currently being held, which is used to determine whether any new lock can be set. Given this information each object can determine whether a new lock request can be granted based purely on its own local information without reference to the other objects in the system. In addition, since multiple processes may be attempting to set locks upon an object concurrently this private information is updated inside a critical section protected by the semaphore *mutex*.

Given the above basic description an outline of how the *setlock* operation works is illustrated in Figure 5*. As shown here, *setlock* attempts to determine whether a conflict exists by calling the private operation *lockconflict*. If this returns the result TRUE then a conflict exists between the requested lock and (at least) one of the other locks currently set on the object, in which case the mutual exclusion semaphore is freed and the call blocks.

---

* We show a simple implementation of *setlock* where the calling process simply keeps on trying (after a brief pause) until the lock is granted. Clearly optimisations are possible, but are not discussed here.

```
class LockCC: public Object
{
        Lock_List locks_held;
        Semaphore *mutex;           // and other private concurrency
                                    // control information
        ...
        boolean lockconflict (Lock*);

public:
        LockCC ();
        ~LockCC ();
        lockstatus setlock (Lock*);
        lockstatus releaselock (Uid*);

}
```

Figure 4: The concurrency control class *LockCC*

```
lockstatus LockCC::setlock (Lock *reqlock)
{
        boolean conflict = TRUE;
        do
        {
                P(mutex);
                if (conflict = lockconflict(reqlock))
                {
                        V(mutex);
                        sleep();            // wait for a while
                }
        } while (conflict);
        locks_held.insert(reqlock);    // add to list of locks
        ...
        V(mutex);
        return (GRANTED);
}
```

Figure 5: The *setlock* algorithm

When the call resumes, the conflict check is again performed to see if the requested lock can now be set.

Releaselock is not intended to be called directly by the programmer, rather it is called automatically when the atomic action using the object terminates (hence the unusual parameter type). This ensures that the two-phase policy is always followed. It occurs in the public interface because in *Arjuna* atomic actions are themselves objects (instances of the class *Action*, see section 6), not part of the underlying system, and thus have no special

privileges to access objects, instead they can only use the same public interface that any other object in the system could use.

The operation of checking whether two locks conflict is more sophisticated than simply allowing *lockconflict* to compare the modes of the lock objects, since in the case of type-specific locking extra information may be used to allow greater concurrency. Instead the *Lock* objects themselves are required to ascertain whether a conflict exists. This check is performed by utilising the *!=* operation provided by the *Lock* type. This operation is defined such that if *L1* and *L2* are two instances of the *Lock* class, then execution of the comparison operation *L1 != L2* returns the value TRUE if the two locks conflict, and returns FALSE otherwise.

Using this approach leads to the implementation of *lockconflict* as shown in Figure 6.

```
boolean LockCC::lockconflict (Lock *reqlock)
{
        Lock__Iterator next (locks__held);
        Lock* heldlock;

        while ((heldlock = next()) != Null)
        {
                if (*heldlock != reqlock)
                        return TRUE;
        }
        return FALSE;
}
```

Figure 6: The *lockconflict* algorithm

This implementation makes use of an iterator *next* which when called returns the next lock from the list of currently held locks. The implementation of the conflict check between locks is obviously type-specific, however, it is assumed that the conflict operator of the basic *Lock* type supports the traditional multiple reader, single writer policy, thus giving the implementation shown in Figure 7.

3.5. Extensions for Type-Specific Locking

The basic locking scheme of the previous section may be extended to take advantage of more knowledge about the semantics of individual types. Not only can derived classes inherit operations from a base class unchanged, they may also modify those operations to make them more suitable for their individual needs. This property can be utilised to allow new types of lock to be derived from the basic *Lock* type. Using this approach, the new lock type can provide its own version of the conflict operator *!=*. By doing this the user-defined type can determine what level of concurrency it will support since the programmer of the

```
boolean Lock::operator!= (Lock *otherlock)
{
    if (otherlock→getowner() != owner)   // no conflict if locks from same action
        switch (lockmode)
        {
            case READ:                   // holding read
                if (otherlock→getmode() != READ)
                    return TRUE;
                break;
            case WRITE:                  // holding write
                return TRUE;
        }
    return FALSE;
}
```

Figure 7: The basic conflict check of *Lock*

individual operations decides the type of lock that needs to be passed to the concurrency controller of the object. Thus type-specific locking is handled in this scheme in an extremely simple manner. All that is required is the derivation of a new type of lock from the basic *Lock* type and an appropriate redefinition of the conflict operation. Obviously this conflict operator can take advantage of any extra information available about the new lock type to provide additional concurrency. The examples in the next section should help to make this clear.

4. Examples of Object-Oriented Concurrency Control

In this section some simple examples are considered to illustrate the scheme in action. In the first example a simple class that implements a file type is described. It makes available the usual file operations to the client and uses the basic capabilities it inherits (from *LockCC*) to provide simple file locking.

The second example illustrates how the basic facilities can be overridden to increase concurrency using a type-specific approach. The technique adopted of deriving a new type of lock allows this to be undertaken with surprising ease.

4.1. A File Class

Consider a *File* class that allows the usual operations of *read, write, open, close*, etc. Such a class was illustrated in Figure 2. With this organisation Figure 8 illustrates how the *open* operation might be implemented. All that is required is the creation a new instance of the *Lock* class (which is automatically initialised to contain the correct information by its

```
File::open(mode)
{
        setlock(new Lock (mode));

        // now actually open file and do other housekeeping
        ...
}
```

Figure 8: The implementation of *open* for the class *File*

constructor), and to pass that instance to *setlock*. The standard implementation of the lock conflict operator is used to determine whether the requested lock can be applied and the *open* operation only proceeds when the lock has been granted.

4.2. A Directory Class

This second example illustrates a directory class that uses a type-specific locking scheme for concurrency control. A new class - a type-specific lock (*TypeSpecLock* (Figure 9))

```
class TypeSpecLock : public Lock
{
        InstanceId Id;                  // some identifier that
                                        // identifies this lock
public:
        TypeSpecLock (mode, Id);        // TypeSpecLock constructor
        ~TypeSpecLock ();

        InstanceId getId () { return Id; }  // A means of accessing the Id

        virtual boolean operator!= (Lock *);
}
```

Figure 9: The class *TypeSpecLock*

is created, derived from the basic *Lock* class and it therefore inherits all of the attributes of the *Lock* class. However, this class has one important addition - a new field by which it can be identified and an operation by which that field can be interrogated.

Given this class, a directory type might be implemented as shown in Figure 10. Then in a similar fashion to the *open* operation for the *File* class, the *addentry* operation of the *Directory* class might be coded as illustrated in Figure 11 (assuming that the operations *addentry* and *rmentry* require write locks, while *lookup* requires a read lock). As was noted earlier both the *addentry* and the *rmentry* operations may proceed concurrently providing that they both manipulate different directory entries. So, to increase concurrency the implementations of *addentry* and *rmentry* construct and pass instances of the new lock type

```
class Directory : public LockCC
{
    ...                          // private directory information
public:
    Directory ();
    ~Directory ();

    addentry (char *Name ...);
    rmentry (char *Name ...);
    lookup (...);
    ...
}
```

Figure 10: The class *Directory*

```
Directory::addentry (char *Name ...)
{
    // first set an appropriate lock...
    setlock(new TypeSpecLock(mode, (InstanceId)Name));

    // then actually manipulate the directory
    ...
}
```

Figure 11: The implementation of *addentry* for the class *Directory*

(*TypeSpecLock*) to the concurrency controller via *setlock*. The implementation of the conflict operator for *TypeSpecLock* is shown in Figure 12.

```
boolean TypeSpecLock::operator!= (Lock *otherlock)
{
    if (otherlock→getowner() != owner)
        switch(lockmode)
        {
            case READ:
                if (otherlock→getmode() == WRITE)
                    return TRUE;   // Read conflicts with Write
            case WRITE:
                if((otherlock→getmode() == READ) ||
                  (Id == ((TypeSpecLock*)otherlock→getId() ))
                    return TRUE;   // RW or WW conflict on same entry
        }
    return FALSE;
}
```

Figure 12: Lock conflict check for the class *TypeSpecLock*

5. Alternative Approaches To Concurrency Control

Most concurrency controllers assume that lock requests emitted by the same action do not conflict, thus an action holding a read lock will be permitted to acquire a write lock providing no other action holds a conflicting lock. This process is often termed *lock conversion*, since its effect is to convert a weaker mode lock into a stronger mode lock. Lock conversion can be handled in the system described in this paper simply by arranging that *Lock* objects with the same owner do not conflict (as determined by the conflict operator *!=*). This simple test was included in the conflict operation *!=* in both Figures 7 and 12.

An alternative approach, adopted in ISIS, requires that conversion is only possible if the original lock had been a *promotable read lock*. Such locks are easy to implement with the approach adopted in this paper. A new type of lock - the *PLock* (derived from *Lock*) - is created and provided with an appropriate conflict operator (Figure 13) which checks all

```
boolean PLock::operator!= (Lock *otherlock);
{
    switch (lockmode)
    {
        case READ:
            if (otherlock→getmode() == WRITE)
                return TRUE;
            break;
        case PREAD:
            if (otherlock→getmode() == READ)
                break;
            if (owner != otherlock→getowner())
                return TRUE;
        case WRITE:
            if (owner != otherlock→getowner())
                return TRUE;
    }
    return FALSE;
}
```

Figure 13: The *PLock* conflict algorithm

locks for conflict regardless of ownership.

A radically different approach to locking, which enforces a *pessimistic concurrency control* policy, is the *optimistic concurrency control* policy [Kung and Robinson 81] where actions are allowed to execute without any synchronisation. At termination (commit) the action is *validated* by analyzing read/write conflicts with other ongoing actions. The validation succeeds if the committing action preserves the serialisability property, otherwise the action is aborted. Just as it is possible to define a conflict rule for type-specific

locking, so is it possible to define a type-specific validation rule [Herlihy 86]. By providing another base class (*OptCC*), objects can utilise this type of concurrency control by being derived from it rather than from *LockCC* (for further details see [Parrington 88]).

6. Integration with Atomic Actions

This section describes how the concurrency control implementation technique described earlier is integrated into a reliable distributed programming system called *Arjuna* currently being developed at the University of Newcastle upon Tyne.

Not surprisingly, the type-inheritance mechanism is also employed for making user-defined types recoverable [Dixon and Shrivastava 87]. A base class *Object* provides the basic capabilities that allows a type to be recoverable. Thus a user-defined type inherits properties of recoverability from the class *Object* and concurrency control capability from the class *LockCC*. The overall class hierarchy of Arjuna is shown below as Figure 14 (see [Dixon *et al.* 87] for more details).

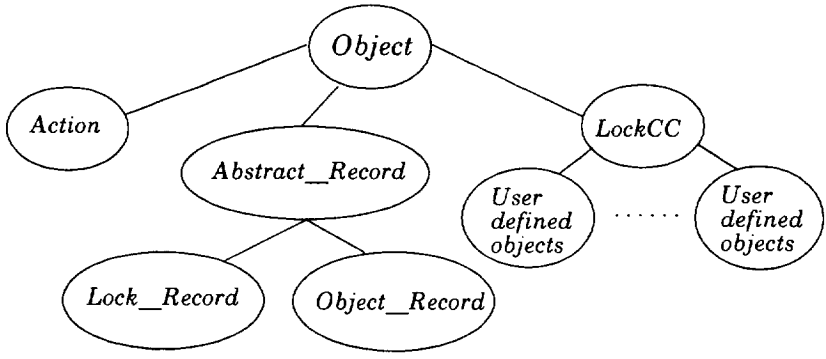

Figure 14: The basic *Arjuna* class hierarchy

Arjuna is novel with respect to other such systems in taking the approach that every major entity in the system is an object. Thus, just as locks are objects, the management of atomic actions is handled by instances of another type of object (called *Action*).

Atomic actions are available through the use of the class *Action*, which provides the operations normally associated with atomic actions, such as *Begin_Action*, *Commit_Action*, etc. *Action* manages information provided to it by *Object* and *LockCC* to ensure that the atomic action abstraction is maintained (for example, locks are released at action commit). See [Dixon 88] for more precise details.

User-defined objects which have been derived from *LockCC* are treated as *persistent*, and are normally stored in local (to a node) object stores. An object without any locks held on it (which implies that no action is currently accessing it) is treated as *passive* and its state

is stored in the object store. When an action is granted a lock on an object, that object is made *active* (if it is not already so) by copying its state from the object store and associating a server process capable of receiving operation invocation requests with the object. If the client action aborts the object state held in the server process is discarded. If the action commits, the state is placed back in the object store using the capabilities provided by the class *Object* (see [Dixon *et al.* 87]).

Actions in *Arjuna* may be nested in the normal way which requires that the lock-based concurrency controller implemented by *LockCC* only finally releases locks when the top-level action commits. When a nested atomic action commits locks are propagated to the parent action [Moss 81]. Further details of this and the implementation of other concurrency controllers can be found in [Parrington 88].

7. Conclusions

The use of type-inheritance has enabled the design and implementation of a concurrency control scheme that is highly adaptable and flexible without resorting to designing a new language or system. Using this approach programmers have control over what level of concurrency a type supports. Of course, this flexibility is not without its potential penalties, since careless programming could lead to chaos as objects are manipulated without being supervised by a concurrency controller. The *Arjuna* system employs type-inheritance for incorporating the serialisability, recoverability and permanence of effect properties of atomic actions.

Of the other comparable robust object-based systems described in the literature only Avalon [Herlihy and Wing 86] is exploring using type-inheritance as opposed to constructing an entirely new language and/or system. However, the approach adopted in Avalon is different from that presented here in that control over concurrency is based on the concept of hybrid atomicity [Weihl 84] and providing user-defined operations for the commit and abort of actions. Nevertheless, its aims are similar. Their work enforces our belief that type-inheritance provides a very powerful concept for incorporating fault-tolerance in systems.

Acknowledgments

Discussions with Graeme Dixon were helpful in formulating these ideas, as were comments by Pete Lee on an earlier draft of this paper. This work was supported by an SERC/Alvey grant in Software Engineering.

References

Best and Randell 81

Best, E., and B. Randell, "A Formal Model of Atomicity in Asynchronous Systems", *Acta Informatica*, 16, pp. 93-124, 1981.

Birman 86

Birman, K.P., "Replication and Fault Tolerance in the ISIS System", Proceedings of 10th Symposium on the Principles of Operating Systems, *ACM Operating Systems Review*, Vol. 19, No. 4, pp. 79-86, 1985.

Dasgupta *et al.* 85

Dasgupta, P., R.J. LeBlanc Jr., and E. Spafford, "The Clouds Project: Designing and Implementing a Fault Tolerant, Distributed Operating System," Technical Report GIT-ICS-85/29, Georgia Institute of Technology, 1985.

Dixon 88

Dixon, G.N., "Managing Objects for Persistence and Recoverability," Ph.D Thesis, Computing Laboratory, University of Newcastle upon Tyne, in preparation.

Dixon and Shrivastava 87

Dixon, G.N., and S.K. Shrivastava, "Exploiting Type-Inheritance Facilities to Implement Recoverability in Object Based Systems", *Proceedings of 6th Symposium on Reliability in Distributed Software and Database Systems*, Williamsburg, pp. 107-114, March 1986.

Dixon *et al.* 87

Dixon, G.N., S.K. Shrivastava, and G.D. Parrington, "Managing Persistent Objects in Arjuna: A System for Reliable Distributed Computing," *Proceedings of the Workshop on Persistent Object Systems*, Persistent Programming Research Report 44, Department of Computational Science, University of St. Andrews, August 1987.

Eswaran *et al.* 76

Eswaran, K.P., *et al.*, "On the Notions of Consistency and Predicate Locks in a Database System", *Communications of the ACM*, Vol. 19, No. 11, pp. 624-633, 1976.

Gray 78

Gray, J.N.,, "Notes on Data Base Operating Systems", in Operating Systems: An Advanced Course, eds. R. Bayer, R.M. Graham, and G. Seegmueller, pp. 393-481, Springer, 1978.

Herlihy 86

Herlihy, M.P., "Optimistic Concurrency Control for Abstract Data Types", *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, pp. 206-216, Calgary, Alberta, August 1986.

Herlihy and Wing 86

Herlihy, M.P., and J.M. Wing, "Avalon: Language Support for Reliable Distributed Systems," *Digest of Papers FTCS-17: Seventeenth Annual International Symposium on Fault-Tolerant Computing*, pp. 89-94, Pittsburgh, July 1987.

Kung and Robinson 81
Kung, H.T., and J.T. Robinson, "On Optimistic Methods for Concurrency Control", *ACM Transactions on Database Systems*, Vol. 6, no. 2, pp. 213-226, June 1981.

Liskov and Scheifler 83
Liskov, B., and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust Distributed Programs", *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 3, pp. 381-404, 1983.

Parrington 88
Parrington, G.D., "Management of Concurrency in a Reliable Object-Oriented Computing System," Ph.D Thesis, Computing Laboratory, University of Newcastle upon Tyne, in preparation.

Schwarz and Spector 84
Schwarz, P.M., and A.Z. Spector, "Synchronizing Shared Abstract Types", *ACM Transactions on Computer Systems*, Vol. 2, No. 3, pp. 223-250, August 1984.

Spector *et al.* 85
Spector, A.Z., *et al.*, "Support for Distributed Transactions in the TABS Prototype", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 6, pp. 520-530, 1985.

Stroustrup 86
B. Stroustrup, *The C + + Programming Language*, Addison Wesley, 1986.

Weihl 84
Weihl, W., "Specification and Implementation of Atomic Data Types," Ph.D Thesis, MIT/LCS/TR-314, MIT Laboratory for Computer Science, Cambridge, Mass., March 1984.