Using Metaobject Protocols to Implement Atomic Data Types

R. J. Stroud and Z. Wu

Department of Computing Science, University of Newcastle upon Tyne, UK

Abstract. Many researchers have suggested using atomic data types to implement transactions as a way of achieving atomicity in a modular, encapsulated fashion. However, in practice, implementing user-defined atomic data types is a difficult task. Existing systems fail to make a clean separation between application code and the synchronisation/recovery code needed to guarantee atomicity, and thus make it very hard for programmers to make changes to a system to meet their particular needs. In this paper, we present a metaobject protocol approach to implementing atomic data types. With this approach, application code is implemented at the base level, whilst the synchronisation and recovery code necessary to ensure atomicity is implemented at the meta level in the form of metaobjects. New schemes for ensuring atomicity can be provided for particular applications by introducing new metaobjects into the system without a need to make any changes at the base level. Furthermore, it is possible for an atomic data object to adapt its mechanism for ensuring atomicity dynamically by changing its metaobject at run time.

1 Introduction

Over the past few decades increasing reliance has been placed upon computers to such an extent that many applications are now totally dependent on the correct functioning of their computer systems. A major issue in such applications is preserving the consistency of on-line data in the presence of concurrency and failures. Typically, data consistency is achieved by ensuring that all accesses and updates to shared data objects are made from within atomic activities called transactions. Many researchers have suggested using atomic data types to implement transactions. With this approach, each object that can be accessed concurrently by more than one activity is responsible for ensuring its own atomicity. This use of an object-oriented approach should allow atomicity to be achieved in a modular, encapsulated fashion and enable type-specific object semantics to be exploited to achieve greater concurrency.

In practice, however, implementing user-defined atomic data types is a difficult task. This is because atomic data types define the behaviour of objects in a concurrent and unreliable environment. Thus, in addition to implementing the operations of a normal abstract data type, an atomic data type also needs to implement synchronisation and recovery operations. There are three kinds of approaches to implementing atomic data types: *implicit*, *explicit* and *hybrid*,

according to whether the system or the programmer is responsible for implementing the synchronisation and recovery code necessary to ensure atomicity. With an *implicit approach*, the programmer is only responsible for implementing the basic object operations as if for a sequential reliable environment with no concurrency or failure. The system is responsible for implementing the necessary synchronisation and recovery code using knowledge about the object's semantics provided by the programmer. With an *explicit approach*, the programmer is responsible for implementing both the basic object operations and the synchronisation and recovery code. With a *hybrid approach*, the work of implementing the synchronisation and recovery code is shared by the system and the programmer.

The explicit and hybrid approaches fail to achieve a clean separation between application code and the synchronisation/recovery code necessary to ensure atomicity. Their drawbacks are very obvious. Intertwining synchronisation and recovery code with application code makes the implementation of atomic data types very difficult, requiring considerable sophistication on the part of the programmer. Furthermore, it also makes it impossible to change the synchronisation and recovery scheme for an object without re-implementing the object. This prevents changing the mechanism used to ensure atomicity for an existing atomic data object either to support new application requirements or to improve performance. For example, an object that was initially implemented using an optimistic protocol for its synchronisation scheme might need to be re-implemented using a pessimistic protocol later because the conflicts among applications were much higher than anticipated.

An implicit approach does not require the programmer to implement synchronisation and recovery operations. Thus it seems easy to make a clear separation between application code and synchronisation/recovery code. However, in practice, it is not so straightforward. In order to provide type-specific synchronisation and recovery for an object, it is necessary to monitor or control the object's behaviour at run time. The code provided automatically by the system for this purpose must not become intertwined with the application code provided by the programmer. Otherwise, it will be difficult to change the scheme used to ensure atomicity for an atomic data object because different schemes will need different code for monitoring and controlling the object's dynamic behaviour.

In this paper, we present a metaobject protocol approach to implementing atomic data types. With this approach, the application code and the code for ensuring atomicity (i.e. the synchronisation and recovery code) are totally separated from each other. The application code is implemented at the base level, whilst the code for ensuring atomicity is implemented at the meta level in the form of metaobjects. Thus, the behaviour of the system can be modified incrementally using meta level programming. In this way, a new scheme for ensuring atomicity can be supported by introducing new metaobjects into the system without a need to make any change at the base level.

In the next section, the concept of atomic data types is introduced. Section 3 presents our persistent programming language PC++ which supports an implicit approach to defining atomic data types, and analyses the problems with

the original implementation of PC++ that led us to consider an alternative implementation based on the use of metaobject protocols. In Section 4, the basic idea of the metaobject protocol approach is described, and the reflective object-oriented programming language used for our experiments is introduced. Section 5 introduces our metaobject protocol for implementing atomic data types and Section 6 demonstrates the flexibility of this protocol by illustrating how it can be used to implement two different synchronisation schemes, one optimistic and one pessimistic. Finally, Section 7 summarises the benefits of a metaobject protocol approach for implementing atomic data types and draws some conclusions.

2 Background and Concepts

Maintaining data consistency in a concurrent and unreliable environment is an important issue for many application systems. To support consistency it is useful to make concurrent activities that access data objects *atomic*. Atomic activities are referred to as *transactions*. Although a transaction usually consists of several operations, the properties of transactions ensure that a transaction is performed atomically. That is, if a transaction commits, it should appear to the outside world as if the entire collection of operations comprising the transaction are performed consecutively, with no intervening operations of other transactions, and without getting part way through and stopping. Transactions simplify the problem of maintaining system consistency by decreasing the number of cases that need to be considered [11].

In order to increase the level of concurrency, many researchers have suggested using the semantics of an application for concurrency control. Semantics-based concurrency control protocols can be broadly classified into two groups depending on whether they are based upon the semantics of transactions or upon the semantics of objects [31]. In the database community, work in this area has largely focused on transactions. Lamport [17] proposed using the semantics of transactions to increase concurrency in database systems. The limitations of the traditional transaction model were demonstrated by Gray [10]. Weikum [34] introduced the multilevel transaction model in which the semantics of operations at different levels are used to increase concurrency.

However, a more object-oriented approach is to use type-specific concurrency control by exploiting the semantics of the data through the definition of abstract data types. A locking protocol based on the notion of commutativity of operations on abstract data types was proposed by Schwarz and Spector [27]. Herlihy, Liskov, and Weihl [14, 33] introduced the concept of atomic data types.

Atomic data types are abstract data types that also have synchronisation and recovery properties. Instances of atomic data types, called atomic objects, are responsible for ensuring their own atomicity. There has been much research into the different issues that arise in implementing atomic data types [4, 6, 27, 32], and a number of systems have been built that support transactions using atomic data types. Examples include Argus [33], Clouds [1], Arjuna [26], TABS [25], and Camelot/Avalon [28]. Although these systems differ in detail, all of them

take either an explicit or a hybrid approach to implementing atomic data types. The drawbacks of these approaches were pointed out in the previous section.

A proposal to add user-defined concurrency control to an object-oriented database language was made within the FIDE project [8]. Although the FIDE proposal allows programmers to specify the concurrent behaviour of an atomic data type declaratively and could therefore be used to support an implicit approach. However, it does not deal adequately with recovery. In particular, transactions are required to commit in the same order in which they were initiated, and the problem of cascading aborts is not addressed.

Atkins [2] described an adaptable server for supporting atomic data types that can choose an appropriate concurrency control strategy on the basis of state information, the history of conflicts encountered, or by using preset transaction priorities.

More recently, Guerraoui [12] has proposed using *o-atomicity* to support the composition of atomic data objects, and has outlined an implementation of *o-atomicity* based on an abstract *AtomicObject* class. The idea is that application-specific atomic data types should be derived from a subclass of *AtomicObject* that uses a particular concurrency control protocol to guarantee *o-atomicity*. Optimistic and pessimistic subclasses of *AtomicObject* are presented. Although there are some similarities between this approach and our approach, Guerraoui requires an *access* operation to be invoked internally every time a transaction invokes an operation on an atomic data object whereas we achieve the same effect using meta level programming. Furthermore, Guerraoui uses a hybrid approach to supporting atomic data types whereas we use an implicit approach.

A great deal of work in the field of concurrent object-oriented programming languages has been concerned with the problem of achieving a clean separation between sychronisation code and the code that implements object operations (the so-called "inheritance anomaly" [21, 23, 24, 9, 22]). Our motivation for using a metaobject protocol to support atomic data types is similar except that we are dealing with the issue of transaction concurrency rather than just process concurrency [33].

3 An Implicit Approach for Atomic Data Types

To simplify the task of constructing atomic data types, we have proposed an implicit approach for defining atomic data types and implemented it in a persistent programming language PC++ [36]. In this section, we present a brief description of PC++ and analyse the problems with its implementation that led us to consider using metaobject protocols to implement atomic data types.

3.1 Defining Atomic Data Types

In PC++ a clear separation is made between the *serial specification* of an atomic data type which describes the permissible behaviour of an instance of the type in the absence of concurrency and failures, and the *concurrent specification* which

describes how an instance should respond to concurrency and failures. Programmers are required to do very little extra work to make an object type atomic. Except for specifying the concurrent semantics of operations declaratively, an atomic data type is defined just like a normal abstract data type implemented in a sequential, reliable environment. In PC++, a very small language has been developed for users to specify the concurrent semantics of object operations declaratively as an *invalidates* relation. PC++ uses this *invalidates* relation to generate type-specific synchronisation and recovery code automatically.

The *invalidates* relation for a data type describes possible conflicts between operations on instances of the type that would limit concurrency. Given two operations on the same type, p and q, we say that p invalidates q if there is some object d whose value is such that the result of executing q on d might not be the same as the result of executing p and then q on d. By result we mean the result returned by the operation, not the effect the operation has on the object state. However, clearly if p invalidates q for some object d, then p has an effect on the state of d that changes the result of invoking q on d. (Note that it is possible for p to modify the state of d without altering the result of invoking q after p on d and in this case, p would not invalidate q.) Since an *invalidates* relation is only concerned with the names and results of operations and not the effect they have on the object state, it is very easy to check whether a given relation is an *invalidates* relation.

When specifying the *invalidates* relation, each operation is represented by its name and result. The result of an operation is simply characterised as either *failed* or *succeeded* (represented by OK), since usually only this distinction makes a significant difference to *invalidates* relations. The first parameter of an operation can also be taken into account if appropriate. The relationship between two parameters can be classified as: $=$ or $\neq$.

For example, suppose there are two operations: *oper1* and *oper2*. If *oper1* invalidates *oper2* only when their first parameter is the same and both of them succeed, then this conflict can be represented as:

(*oper1, OK*) (*oper2, OK*) $=$).

In order to define an atomic class in PC++, the programmer need only place a preprocessor directive *atomic* in front of an ordinary C++ class definition, and add an *invalidates* relation part. For example, consider an *Account* class that has an associated set of operations: *credit* money to an account, *debit* money from an account, and *check* the balance of an account. An appropriate *invalidates* relation can be added to a C++ class definition for *Account* to define a PC++ atomic data type as shown in Fig. 1.

In this example, the *invalidates* relation describes four possible conflicts: a successful *credit* invalidates a successful *check*; a successful *debit* invalidates a successful *check*; a successful *debit* invalidates another successful *debit*; and a successful *credit* invalidates a failed *debit*. The interpretation of the *invalidates* relation by the concurrency control mechanism is handled automatically by PC++. Thus, the application code that implements the *Account* class contains no explicit synchronisation code (although it is necessary to indicate whether an operation succeeds or fails by using a special form of *return* statement).

```
atomic class Account
{
private:
        Money    amount;
public:
                 Account();
        Status   credit(Money);
        Status   debit(Money);
        Money    check();
invalidates relation:
        ((credit, OK) (check, OK))
        ((debit, OK) (check, OK))
        ((debit, OK) (debit, OK))
        ((credit, OK) (debit, failed))
};
```

Fig. 1. The class *Account*

3.2 Constructing Transactions

Programmers can construct transactions that access instances of atomic data types using the *Transaction* class provided by the PC++ system. The computation that constitutes the body of the transaction is controlled by an instance of a *Transaction* object associated with the transaction.

The *Transaction* class provides three primitive operations that may be used to declare and control a transaction: *begin_transaction, commit_transaction* and *abort_transaction*. A *begin_transaction* operation starts a transaction which may be terminated by either a *commit_transaction* or an *abort_transaction* operation. A transaction terminated by an *abort_transaction* operation will definitely be aborted, its partial results will be removed. A transaction terminated by a *commit_transaction* operation is not guaranteed to be committed successfully, but programmers will be told whether their attempt to commit the transaction succeeds. The outcome of a *commit_transaction* operation is either a *success*, when the effects of the transaction become permanent, or a *failure*, when the effects of the transaction are removed.

Figure 2 shows a transaction that attempts to transfer some amount of money from one account to another. At first it tries to debit £1000 from account *John*, and if the debit succeeds, it credits £1000 to account *Guang* and commits; otherwise, it aborts.

```
Account      John, Guang;
Transaction  T;

T.begin_transaction();
if (John.debit(1000) == Success)
 {
  Guang.credit(1000);
  T.commit_transaction();
 }
 else
 {
  T.abort_transaction();
 }
```

Fig. 2. A transfer transaction

3.3 PC++ Implementation

From the above description, we can see that it is very simple for programmers to design atomic data types and transactions in PC++. In this subsection, we describe how PC++ provides synchronisation and recovery operations for user-defined atomic data types.

An essential issue for PC++ is how to integrate system-provided synchronisation and recovery code with different kinds of object whilst still allowing type-specific concurrency control. PC++ resolves this issue by using the type inheritance technique of object-oriented programming. The method is quite straightforward. A special type, called *Scheduler*, is provided which implements a specific concurrency control protocol. Atomic data types are translated by the PC++ preprocessor into C++ classes that are derived from this special type and will thereby inherit the underlying concurrency control facility. Furthermore, if the definition of an atomic data type includes type-specific information about the concurrency semantics of its operations, this can be used by the inherited concurrency control mechanism to make synchronisation decisions.

A difficulty particular to an implicit approach is how the system can get the information about the dynamic behaviour of objects necessary to make sychronisation and recovery decisions, because programmers should not be asked to provide it. PC++ overcomes the difficulty via a preprocessor. By preprocessing atomic data type definitions, some code can be added to object operations so that the information necessary for synchronisation and recovery can be collected automatically when object operations are invoked by transactions.

In principle, the type-inheritance technique used by PC++ for providing concurrency control makes it easy to change the synchronisation scheme of an atomic data type. It is only necessary to provide a new type, say *NewScheduler*, that implements the new concurrency control method. Then, user-defined atomic data types that are derived from *NewScheduler* instead of *Scheduler* will auto-

matically use a different mechanism to implement synchronisation and recovery operations.

Unfortunately, the preprocessing approach makes the above flexibility difficult to achieve in practice. Since different concurrency control methods may need to monitor and control the dynamic behaviour of objects in different ways in order to make synchronisation and recovery decisions, implementing a new synchronisation scheme also requires changing the preprocessor so that it generates different code for monitoring and controlling the behaviour of objects. This is not a easy task with a traditional implementation of a preprocessor, especially if the updator is not the original author of the preprocessor. A new approach is required in order to solve this problem.

4 Metaobject Protocols

In order to make PC++ a flexible and extensible system, we re-implemented it using a metaobject protocol approach that opens up the system by using reflection and object-oriented programming techniques. In effect, the metaobject protocol was used as a mechanism for implementing an extensible, flexible, preprocessor. In this section, we introduce the concept of a metaobject protocol and the reflective object-oriented language Open C++ that was used for implementing our metaobject protocols.

4.1 The Basic Idea

Metaobject protocols [16] are interfaces to a system that give users the ability to modify the system's behaviour and implementation incrementally. Reflection and object-oriented programming play a critical role in the metaobject protocol approach. Reflection makes it possible to open up a system's implementation without revealing unnecessary implementation details. Object-oriented programming allows the resulting model of the system's implementation and behaviour to be adjusted locally and incrementally. The crucial difference between an approach based on reflection and an approach based on a preprocessor is *reification*, the process by which a model of the internal workings of the system is presented to the meta level. PC++ required the support of a preprocessor to collect information about the run time behaviour of objects. This information was a reification of the behaviour of the underlying system that was hard-coded into the implementation of the preprocessor. The use of a reflective system allows a more flexible approach to reification.

Reflection [19] is the process by which a system can reason about and act upon itself. In a conventional system, computation is performed on data that represents entities that are external to the computational system. However, a reflective computational system must contain data that represents the structural and computational aspects of the system itself. Moreover, it must be possible to access and manipulate such data from within the system itself, and more importantly, such data must be causally connected to the actual behaviour of the

system. Unlike a conventional system, a reflective system allows users to perform computation on the system itself in the same manner as in the application, thus providing users with the ability to adjust the behaviour of the system to suit their particular needs.

In an object-oriented programming environment, reflection can be implemented using metaobjects. Each object is causally connected with a *metaobject* that represents both the structural and computational aspects of the object. Metaobjects can be manipulated in the same manner as "normal" objects. More importantly, changes made to a metaobject will be automatically reflected to the object. Therefore, the computational behaviour of an object can be adjusted to meet a particular requirement without modifying the implementation of the object but rather by changing its metaobject.

Reflection has been used in many application areas: flexible programming [16], concurrent programming [20], distributed systems [5], soft real-time [15], and fault tolerant applications [7]. More generally, [29] argues that reflection can be used as a general approach to addressing non-functional requirements such as fault tolerance and distribution transparency.

4.2 Open C++

For our experiments, we used Open C++ as a reflective programming environment. Open C++ [5] is a C++ preprocessor that provides the programmer with two levels of abstraction: the base level, dedicated to traditional C++ object-oriented programming, and the meta level which allows certain aspects of the C++ programming model to be redefined. For example, method calls to base level objects can be intercepted at the meta level by metaobjects. A base level object that is controlled by a metaobject in this way is called a *reflective object.* In Open C++, metaobjects are just objects whose class is derived from the class *MetaObj.* We will refer to such classes as *metaobject classes.* Open C++ allows the programmer to associate an application class with a metaobject class and arrange for a particular instances of the application class to be reflective, i.e. controlled by a metaobject of the appropriate metaobject class.

In Open C++, a metaobject realises its control on a reflective object by trapping the method calls to the latter. For example, suppose a reflective object *account* is bound to a metaobject *meta_account* as shown in Fig. 3. When a method is called at the base level, the call is trapped and handled at the meta level by a virtual method called *Meta_MethodCall.* This meta method makes it possible to redefine the semantics of a method call for a reflective object. Usually, *Meta_MethodCall* invokes the method at the base level, but it may also perform some extra processing before and/or after the invocation. For example, it can try to set a lock on the object before accessing its state and release the lock afterwards. The definition of *Meta_MethodCall* for *meta_account* is supplied by the class of *meta_account*, i.e. the metaobject class associated with the base level class of *account.*

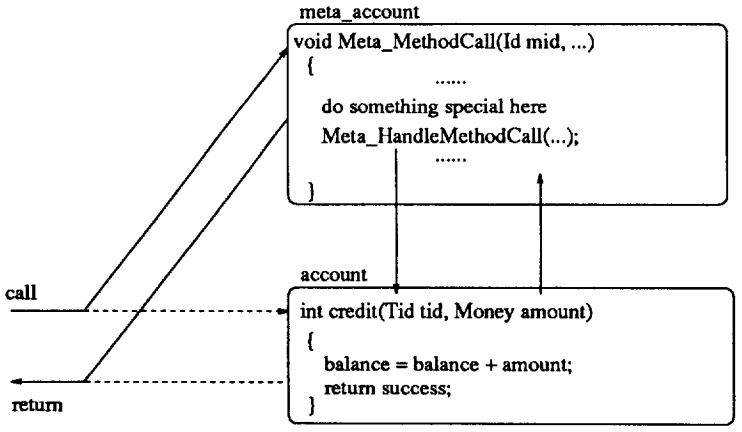

Fig. 3. Invocation trapping in Open C++

5 Using Metaobject Protocols to Implement Atomic Data Types

In this section, we present our metaobject protocol for supporting atomic data types and in the next section we demonstrate the flexibility of our approach with two examples.

5.1 The Computation Model

In our model a system consists of a collection of atomic objects that represent the state of the application, and a collection of transactions that access atomic objects to perform some computations. Atomic objects can only be accessed by operations defined on them and invoked inside a transaction. A transaction usually consists of a series of operations on several atomic objects. The sequence of operations that a transaction invokes on a particular atomic object is referred to as the *transaction component* at that object. An atomic object can deal with invocations from different transactions concurrently. The atomicity of a transaction component at an atomic object is ensured by that atomic object.

There are two kinds of atomicity in a system supporting atomic data objects. The atomicity of transaction components at an atomic object is called *local atomicity* because it deals only with the events involving a particular object. The atomicity of transactions is called *global atomicity* because it deals with events involving all the objects in the system. Generally speaking, the atomicity of a transaction cannot be guaranteed by the atomicity of its components. However, local atomicity is sufficient to ensure global atomicity if certain conditions are satisfied. Weihl [32] has identified three local atomicity properties that result in global atomicity: *static atomicity*, *dynamic atomicity*, and *hybrid atomicity*. If all the atomic objects shared by the transactions in a system provide the same kind

of atomicity, the transactions are guaranteed to be serialisable and recoverable. More recently, Guerraoui has proposed *o-atomicity* as a condition for atomic object composition [12].

A transaction must be started by a *begin_transaction* operation and ended by either a *commit_transaction* or an *abort_transaction* operation. Similarly, a transaction component at an atomic object must be started by a *begin_component* and ended by either a *commit_component* or an *abort_component* operation. However, unlike transaction operations, these operations on components are not directly invoked by application programmers. A *begin_component* operation will be executed when a transaction invokes an operation on an atomic object for the first time. When a transaction commits, the *commit_transaction* operation must invoke a *commit_component* operation on each component of the transaction. Similarly, if a transaction aborts, the *abort_transaction* operation must invoke *abort_component* for each component. The object representing the transaction must keep track of which atomic objects are accessed during the transaction so that it can identify the transaction's components.

5.2 Metaobject Protocol Approach

The execution of a transaction component involves two kinds of operations: object operations defined by programmers which we call *functional operations* because they are primarily concerned with the functionality of objects, and local transaction operations provided by the system which we call *non-functional operations* because they do not add new functionality to objects but are used to ensure the correct execution of object operations in an unreliable, concurrent environment. The *begin_component*, *commit_component* and *abort_component* operations that must be provided by every atomic data type are non-functional; the type-specific operations that implement the functionality of an atomic data type are functional although the synchronisation and recovery code associated with the invocation of these operations is non-functional.

The problems with the original implementation of PC++ resulted from mixing up the code implementing functional operations with the code implementing non-functional operations. In order to make a system flexible, we must make a clear separation between the two kinds of code. This can be achieved by taking the metaobject protocol approach introduced in section 4. The use of meta level programming makes it possible to achieve a transparent separation of functional components from non-functional components [29, 30]. Using a metaobject protocol approach, atomic object operations can be implemented by application programmers at the base level as if for a sequential and reliable environment. All of the work required for providing synchronisation and recovery, such as controlling the invocation of object operations, collecting run time object information, validating, committing and aborting transaction components, can be done separately at the meta level.

In Open C++, it is possible to capture the semantics of an atomic data type within our computational model using the *AtomicMetaObj* metaobject class

shown in Fig. 4. Metaobject classes that realise particular schemes for ensuring atomicity can be derived from this abstract class.

```
class AtomicMetaObj : public MetaObj
{
public:
                AtomicMetaObj();
   virtual void Meta_StartUp();
   virtual void Meta_MethodCall(Id mid, Id cid, ArgPac& args, ArgPac& rslt);
   virtual int  Meta_BeginComponent(Tid tid);
   virtual int  Meta_CommitComponent(Tid tid);
   virtual int  Meta_AbortComponent(Tid tid);
   virtual void Meta_NormalOperation(Tid tid, Id mid, id cid, ArgPac& args,
                                                              ArgPac& rslt);
}
```

Fig. 4. A metaobject class for atomic data types

In order to define an atomic data type, the application programmer writes a class definition for the type as if it were implemented in a sequential, reliable environment, and then specifies using an Open C++ directive that the metaobject class of the application class is an appropriate subclass of *AtomicMetaObj*. In other words, atomic data types are implemented by classes whose metaobject class is derived from *AtomicMetaObj*.

When an atomic object is created at run time, Open C++ will automatically create a corresponding metaobject that is an instance of the appropriate subclass of *AtomicMetaObj*. This metaobject will be bound to the application object in such a way that all of the operations that the application invokes on the atomic object are intercepted by the metaobject. The metaobject is responsible for enforcing the particular synchronisation and recovery policies being used to guarantee local atomicity and can control whether the invocation is allowed to proceed at this time or record information about the invocation that will be used for validation later.

The following four operations are provided by the *AtomicMetaObj* class for implementing local atomicity: *Meta_BeginComponent*, *Meta_CommitComponent*, *Meta_AbortComponent*, and *Meta_NormalOperation*.

Open C++ ensures that all the operations invoked on an atomic object are intercepted by the *Meta_MethodCall* operation of the corresponding metaobject. *Meta_MethodCall* will check the category of an intercepted call and then invoke an appropriate operation to deal with the invocation.

The *Meta_MethodCall* operation will invoke the *Meta_BeginComponent* operation when a *begin_component* operation is called on an atomic object. Similarly, the *Meta_CommitComponent* operation is invoked in response to *commit_component* and likewise for *Meta_AbortComponent* and *abort_component*.

When a normal object operation is called on an atomic object from a transaction component, *Meta_MethodCall* will invoke the *Meta_NormalOperation* operation which decides when the invocation is allowed to proceed and records any information necessary for validation at commit time.

In this way, the code for implementing a particular scheme for achieving local atomicity can be clearly separated from the code implementing the functionality of an atomic data type. Therefore, changes to the scheme for achieving atomicity can be made totally transparent to application programmers. Conversely, the code for implementing atomicity is not affected if application programmers make changes to the implementation of the object's operations.

6 Implementing Concurrency Control Methods Using an Atomic Metaobject

In this section, we illustrate the flexibility of our approach and demonstrate the use of the metaobject protocol defined by *AtomicMetaObj* by presenting two different metaobject classes derived from *AtomicMetaObj* that implement an optimistic and a pessimistic concurrency control policy respectively.

6.1 An Optimistic Concurrency Control Metaobject Class

PC++ was developed for a multimedia project called OPERA [18]. Based on our belief that an optimistic concurrency control method is more suitable for real-time multimedia applications than a pessimistic method, PC++ uses an optimistic protocol called the *dual-level validation method* (DLV) [36] as its default concurrency control strategy. In this subsection we describe a metaobject class that uses the DLV method to implement local atomicity for atomic objects. Since we are dealing with local atomicity, we need only consider the effect of a transaction component at a single object.

The Dual-Level Validation Method. We view an atomic object as a two-layered architecture. The higher layer, called the *logical level*, is concerned with the set of abstract operations defined on the object, which are the only means for applications to access the object. The lower layer, called the *physical level*, is concerned with the set of operations provided by the system to manage primitive data objects.

The DLV method is an optimistic protocol. Transactions operate on shadow copies of (components of) objects, relying on commit-time validation to ensure serialisability. The two levels of the DLV method are concerned with the two levels of the object architecture, logical and physical. Logical level validation ensures that a transaction that has used an object and is requesting a commit is serialisable with other transactions. Physical level validation ensures that the logical level object operations are elementary. Logical level validation can be done by making use of the semantics of the object so that greater concurrency can be achieved.

One approach to implementing optimistic concurrency control is to take a *shadow copy* of a whole object at the start of a transaction, or perhaps at the time of the first operation that updates the object. All subsequent invocations are on this copy and are validated against the persistent object when the transaction requests commit [13].

However, our approach is different. Our objects are tree structured with primitive objects as leaves. We assume that objects may be large. Our approach is to take a shadow copy only of the sub-object of the (tree structured) physical object that is required for a given invocation. A copy of a sub-object is taken on the first invocation that updates that part of the object and all subsequent invocations on that sub-object, for read or update, are performed on that shadow.

A transaction, in general, encloses operations on several objects. The sequence of operations that a transaction invokes on a particular object forms the *transaction component* at that object. An execution of a transaction component consists of two, or three phases: a *read phase*, a *validation phase*, and possibly a *write phase*.

During the read phase, an operation invocation is executed immediately. However, if the invocation involves an update, this takes place on a local *shadow copy* of the physical sub-object affected. Each atomic object also maintains a record of which object operations have been invoked by each transaction component, and which physical (sub)objects have been read or written by each transaction component.

The logical validation phase begins when the execution of a transaction component reaches its end. During this phase, an atomic object validates the transaction component to establish whether any of the invocations of the transaction component have been invalidated by the invocations of concurrent transaction components. The information recorded in the previous phase is essential for this validation.

After entering the write phase, a transaction component is validated again by the atomic object to check whether it can be accepted at the physical level. The purpose of this validation is to check whether the values read by the transaction component are still current. If they are, the transaction component is committed by merging its shadow copies into the permanent state. Otherwise the operations of the transaction component are re-executed. Again, the information recorded during the read phase is required for the physical validation and the re-execution.

A more detailed description of the DLV method can be found in [36] and the correctness of the DLV method is proved in [35].

A Metaobject Class for the DLV Method. From the above description of the DLV method, we can see that all the work done at the logical validation phase and the write phase is non-functional. At the read phase, the execution of object operations is functional, but recording information about transaction components is non-functional. We implement the DLV method using a metaobject class called *DlvMetaObj* that is a subclass of *AtomicMetaObj*, as shown in Fig. 5.

```
class DlvMetaObj: public AtomicMetaObj
{
public:
        DlvMetaObj();
    void    Meta_StartUp();
    void    Meta_MethodCall(Id mid, Id cid, ArgPac& args, ArgPac& rslt);
    int     Meta_BeginComponent(Tid tid);
    int     Meta_ValidateComponent(Tid tid, Timestamp tmstp);
    int     Meta_CommitComponent(Tid tid);
    int     Meta_AbortComponent(Tid tid);
    void    Meta_NormalOperation(Tid tid, Id mid, Id cid, ArgPac& args,
                                                          ArgPac& rslt);
protected:
    long    latest_committed;
    struct {
        Tid         tid;
        Timestamp   timestamp;
        Timestamp   last_committed;
        Operation   operations[MaxOpers];
        RWSet       read_set[MaxObjs];
        RWSet       write_set[MaxObjs];
    } history[MaxTrans];
};
```

Fig. 5. The metaobject class for the DLV method

The *latest_committed* variable of a DLV metaobject is used to record the timestamp of the latest committed transaction component at its atomic object. The *history* array is used to record information about each transaction component at that object. The identifier and the timestamp of a transaction component are recorded in *tid* and *timestamp* respectively. The timestamp of the last committed transaction before a transaction component starts is recorded in *last_committed*, whilst the operations performed by each transaction component are recorded in *operations*. *read_set* and *write_set* are used to record the variables that are read and written by a transaction component respectively.

Whenever a *begin_component* operation is called on an atomic object, the *Meta_MethodCall* operation will invoke the *Meta_BeginComponent* operation. The latter will register the new transaction component with the atomic object, and assign an element of the *history* array as the data buffer to record information related to the transaction component. The behaviour of *begin_component* is shown in Fig. 6.

The *Meta_ValidateComponent* will be invoked by *Meta_MethodCall* when a *commit_component* call is intercepted. It will perform the logical validation by using the information recorded in the *history* array. *Meta_CommitComponent* will be invoked to commit the transaction component if it passes its logical

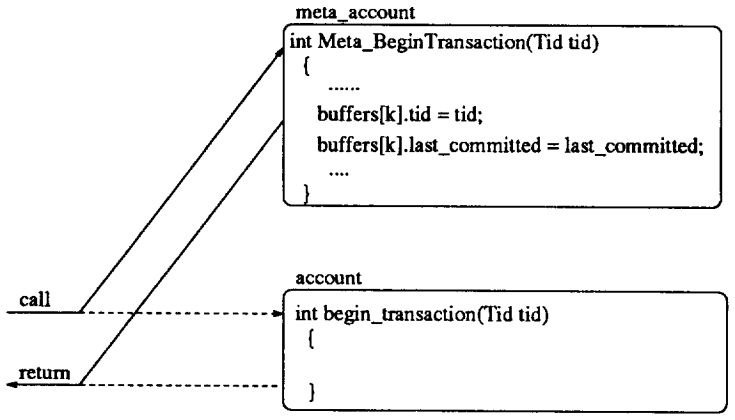

Fig. 6. The behaviour of *begin_component*

validation; otherwise *Meta_AbortComponent* will be invoked to abort it.

When a normal object operation is called on an atomic object from a transaction component, *Meta_MethodCall* will invoke *Meta_NormalOperation*. The *Meta_NormalOperation* operation will first record the method identifier *mId*, the category identifier *cId* and the arguments of the method *args* into the data buffer assigned for the transaction component. It will then invoke the original object operation on the atomic object. After the invocation, it will record the result of the invocation in the data buffer before returning the result to *Meta_MethodCall* which will in turn pass the result to the original caller. The behaviour of a normal object operation is shown in Fig. 7.

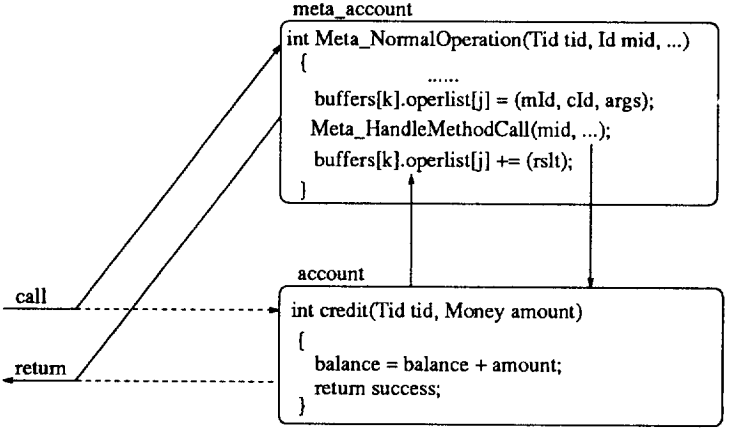

Fig. 7. The behaviour of normal operations

6.2 A Pessimistic Concurrency Control Metaobject Class

In order to demonstrate the flexibility of the metaobject protocol defined by *AtomicMetaObj*, we will now illustrate how it can be used to support a pessimistic concurrency control metaobject class.

The pessimistic concurrency control method we are going to support is the widely used 2-phase-locking (2PL) method. In this method, object operations are classified as either *read* operations if they do not make any change to an object; or *write* operations if they may update an object. This classification will be specified by the application programmer as part of the definition of an atomic data type. Before invoking an object operation, a transaction component must acquire an appropriate lock for that operation. A transaction can acquire a lock on an object only if no concurrent transaction component holds a conflicting lock on the object.

The metaobject class implementing the 2PL method is described in Fig. 8. The *current_locks* field records the current locks set in an object together with their modes and owners. The *shadows* field stores a list of snapshots of the object state.

```
class TplMetaObj: public AtomicMetaObj
{
public:
        TplMetaObj();
    void    Meta_StartUp();
    void    Meta_MethodCall(Id mid, Id cid, ArgPac& args, ArgPac& rslt);
    int     Meta_BeginComponent(Tid tid);
    int     Meta_ValidateComponent(Tid tid, Timestamp tmstp);
    int     Meta_CommitComponent(Tid tid);
    int     Meta_AbortComponent(Tid tid);
    void    Meta_NormalOperation(Tid tid, Id mid, id cid, ArgPac& args,
                                                    ArgPac& rslt);
protected:
    struct {
        Lock    lockid;
        Mode    mode;
        Tid     tid;
    } current_locks[MaxLocks];
    struct {
        long length;
        void* addrs;
    } shadows[MaxTrans];
}
```

Fig. 8. *The metaobject class for the 2PL method*

Meta_BeginComponent will be invoked when a *begin_component* call is intercepted. It simply makes a snapshot of the object state. *Meta_CommitComponent* will be invoked when a *commit_component* call is intercepted. It needs only to release the locks held by a transaction component. *Meta_AbortComponent* will be invoked when an *abort_component* call is intercepted. It first restores the object to a consistent state by replacing the object state with the snapshot taken for that transaction component, and then releases the lock set by the transaction.

Meta_NormalOperation is invoked when the invocation of an object operation from a transaction component is intercepted. It first tries to acquire an appropriate lock for the transaction component. If successful, it invokes the original object operation, and then returns the result to the caller. This procedure is shown in Fig. 9.

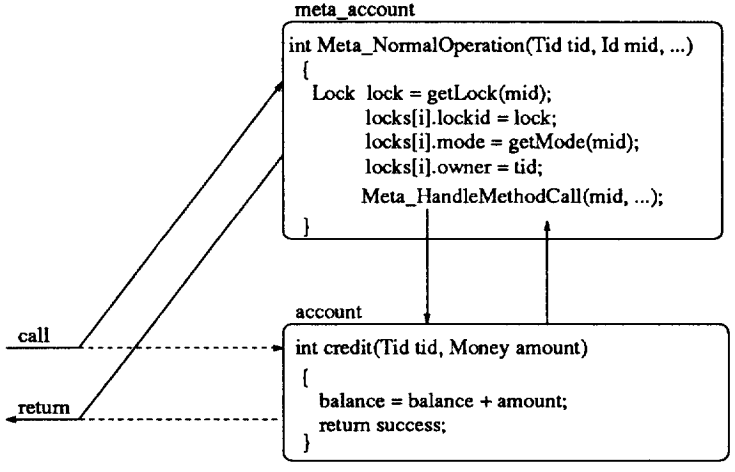

Fig. 9. The behaviour of a normal object operation in 2PL method

These examples have demonstrated that the metaobject protocol defined by *AtomicMetaObj* is flexible enough to support either an optimistic or a pessimistic concurrency control method. Furthermore, because of the separation between the functional and non-functional aspects of the implementation of atomic data types, it is possible for an object to change its synchronisation scheme simply by binding to a different metaobject. This can be done dynamically at run time if the reflective programming environment allows an object to change its metaobject at run time. (The current version of Open C++ does not support this facility.)

7 Summary and Conclusions

Many researchers have suggested using atomic data types to maintain data consistency in a concurrent system. However, existing approaches to implementing atomic data types mix up the code for implementing the functional aspects of

an atomic object with the code for implementing the non-functional aspects of an atomic object (e.g. the synchronisation and recovery code used to guarantee atomicity), and this makes it very hard to change the scheme used to implement local atomicity. The metaobject protocol approach presented in this paper solves this problem by using meta level programming to separate the synchronisation and recovery aspects of an atomic data type from its functionality.

With a metaobject protocol approach, non-functional aspects of atomic data types such as synchronisation and recovery are implemented in metaobject classes at the meta level, whilst functional aspects such as object operations are implemented at the base level. This clear separation between the two kinds of code allows either of them to be freely changed without affecting the other. Programmers can change the particular synchronisation and recovery scheme used by an atomic object without modifying the implementation of the object but simply by changing its metaobject. Furthermore, the use of a metaobject protocol to implement atomic data types allows new synchronisation and recovery schemes to be introduced into the system incrementally.

There are four tangible benefits of taking the metaobject protocol approach to implementing atomic data types. Firstly, the separation of functional and non-functional code makes it possible for the realisation of non-functional aspects of atomicity (e.g. the synchronisation and recovery scheme) to be transparent rather than intrusive as far as the application programmer is concerned, thus solving the problems associated with the explicit and hybrid approaches to implementing atomic data types. Secondly, allowing programmers to build new metaobjects enables them to adjust the design and implementation of the system to suit their particular needs easily. Programmers can introduce new synchronisation and recovery schemes for achieving local atomicity simply by defining new metaobject classes to meet special requirements from their applications. Thus the metaobject protocol approach solves the problems associated with the preprocessor approach. Thirdly, permitting each object to have its own metaobject makes it possible for an application to apply different synchronisation and recovery strategies for different objects according to their characteristics. Fourthly, relying on the metaobjects to deal with a wide range of user requirements allows the basic implementation of the system to be simpler and thus easier to analyse with respect to its correctness. Other authors who have experimented with metaobject protocols have reported similar benefits [16, 3, 5, 7, 29].

Although our experiments with using metaobject protocols to implement atomic data types are still at an early stage, and although limitations of the current version of Open C++ prevent us from exploiting important reflective features such as dynamic binding between objects and metaobjects, we have demonstrated the feasibility of our general approach by outlining the implementation of two different schemes for achieving atomicity using metaobject classes derived from an abstract metaobject class defining a protocol for implementing atomic data types. Issues still to be resolved properly before our new implementation can be used as conveniently as the existing PC++ system include the mechanism by which type-specific information about the concurrent semantics

of an atomic data type is translated into meta data to be used by the atomic metaobject class to resolve conflicts and ensure serialisability. Since the form that this specification takes may depend on the concurrency control method used, we believe that each metaobject class must be responsible for parsing its own meta data. This is much easier to achieve using an interpreted language such as CLOS than a compiled language such as C++ and it may be necessary to retain some kind of preprocessing mechanism to convert a declarative specification expressed using some form of syntactic sugar by the application programmer into meta data or code to be exploited by the metaobject class. Another topic for future research is the issue of performance although we believe that any inefficiencies inherent in our approach arise from limitations in the implementation of the underlying reflective programming system and that as reflective implementation technologies become better understood, reflection will just come to be viewed as another form of indirection mechanism that is no more expensive to use than inheritance or delegation but can be used to solve a different class of problems.

Acknowledgements

This work has been supported by the University Research Committee of University of Newcastle upon Tyne, and by funding from the ESPRIT Basic Research Action on 'Predictable Dependable Computing Systems' (PDCS-2), grant no. 6232. Thanks must go to members of the OPERA project in Cambridge, especially to Ken Moody and Jean Bacon, and to our colleague at Newcastle, Brian Randell, for many discussions on all aspects of the work. The comments we received from the anonymous referees have also helped us to improve the presentation of this work.

References

1. J. E. Allchin and M. S. McKendry: Synchronization and recovery of actions. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing*, pages 31–44, August 1983.
2. M. S. Atkins and M. Y. Coady. Adaptable concurrency control for atomic data types. *ACM Transactions on Computer Systems*, 10(3):190–225, August 1992.
3. G. Attardi, C. Bonini, and M. R. Boscotrecase. Metalevel programming in CLOS. In *Proceedings of 3rd European Conference on Object-Oriented Programming*, pages 243–256, 1989.
4. B. R. Badrinath and K. Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, March 1992.
5. S. Chiba and T. Masuda: Designing an extensible distributed language with meta-level architecture. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 482–501, 1993.
6. P. K. Chrysanthis, S. Raghuram, and K. Ramamritham. Extracting concurrency from objects: A methodology. In *Proceedings of the 17th SIGMOD International Conference on Management of Data*, pages 108–117, 1991.

7. J. Fabre, V. Nicomette, T. Perennou, R. J. Stroud, and Z. Wu. Implementing fault tolerant applications using reflective object-oriented programming. In *Proceedings of the 25th IEEE Symposium on Fault Tolerant Computing Systems*, 1995.
8. N. De Francesco, G. Vaglini, L. V. Mancini, and A. Pereira Paz.: Specification of concurrency control in persistent programming languages. In *Proceedings of the Fifth International Workshop on Persistent Object Systems*, pages 126–143, 1992.
9. S. Frølund. Inheritance of synchronization constraints in concurrent object-oriented programming languages. In *Proceedings of 6th European Conference on Object-Oriented Programming*, pages 185–196, 1992.
10. J. N. Gray: The transaction concept: Virtues and limitations. In *Proceedings of the 7th International Conference on Very Large Data Bases*, pages 144–154, Sept. 1981.
11. J Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, 1993.
12. R. Guerraoui. Atomic object composition. In *Proceedings of 8th European Conference on Object-Oriented Programming*, pages 118–138, 1994.
13. M. Herlihy: Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Transactions on Database Systems*, 15(1):96–124, March 1990.
14. M. Herlihy and W. E. Weihl: Hybrid concurrency control for abstract data types. In *Proceedings of the 7th ACM-SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 201–210, March 1988.
15. Y. Honda and M. Tokoro. Soft real-time programming through reflection. Technical Report SCSL-TR-92-016, Sony Computer Science Laboratory, 1992.
16. G. Kiczales, J. des Rivieres, and D. G. Bobrow: *The Art of the Metaobject Protocol.* MIT Press, 1991.
17. L. Lamport: Towards a theory of correctness for multi-user database systems. Technical report, Massachusetts Computer Assoc., 1976.
18. S. L. Lo: *A Modular and Extensible Network Storage Architecture.* PhD thesis, Cambridge University Computer Laboratory, 1994. Technical Report No. 326.
19. P. Maes: Concepts and experiments in computational reflection. In *OOPSLA '87 Proceedings*, pages 147–155, October 1987.
20. S. Matsuoka, T. Watanabe, and A. Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In *Proceedings of the European Conference on Object-Oriented Programming '91*, pages 213–250, July 1991.
21. S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.
22. J. Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In *Proceedings of 7th European Conference on Object-Oriented Programming*, pages 220–246, 1993.
23. C. Neusius. Synchronizing actions. In *Proceedings of 5th European Conference on Object-Oriented Programming*, pages 118–132, 1991.
24. S. C. Reghizzi, G. G. de. Paratesi, and S. Genolini. Definition of reusable concurrent software components. In *Proceedings of 5th European Conference on Object-Oriented Programming*, pages 148–166, 1991.
25. A. Z. Spector, J. Butcher, D. S. Daniels, D. J. Duchamp, J. L. Eppinger, C. E. Fineman, A. Heddaya, and P. M. Schwarz: Support for distributed transactions in

the TABS prototype. *IEEE Transactions on Software Engineering*, SE-11(6):520–530, June 1985.

26. S. K. Shrivastava, G. N. Dixon, and G. D. Parrington: An overview of the Arjuna distributed programming system. *IEEE Software*, 8(1):66–73, January 1991.
27. P. M. Schwarz and A. Z. Spector: Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, 2(3):223–250, August 1984.
28. A. Z. Spector, D. Thompson, R. F. Pausch, J. F. Eppinger, D. Duchamp, R. Draves, D. S Daniels, and J. J. Bloch: Camelot: A distributed transaction facility for Mach and the internet—an interim report. Technical Report CMU-CS-87-129, Carnegie Mellon University, 1987.
29. R. J. Stroud: Transparency and reflection in distributed systems. *ACM Operating Systems Review*, 27(2):99–103, April 1993.
30. R. J. Stroud and Z. Wu: Using metaobjects to adapt a persistent object system to meet application needs. In *Proceedings of 6th SIGOPS European Workshop on Matching Operating Systems to Application Needs*, 1994.
31. A. H. Skarra and S. B. Zdonik: Concurrency control and object-oriented databases. In W. Kim and F. H. Lochovsky, editors, *Object-oriented concepts, Databases and Applications*, pages 395–421. ACM Press, 1989.
32. W. E. Weihl: Local atomicity properties: Modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–282, April 1989.
33. W. E. Weihl and B. Liskov: Implementation of resilient, atomic data types. *ACM Transactions on Programming Languages and Systems*, 7(2):244–269, April 1985.
34. G. Weikum: Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems*, 16(1):132–180, March 1991.
35. Z. Wu: *A New Approach to Implementing Atomic Data Types*. PhD thesis, Cambridge University Computer Laboratory, 1994. Technical Report No. 338.
36. Z. Wu, R. J. Stroud, K. Moody, and J. Bacon. The design and implementation of a distributed transaction system based on atomic data types. *Distributed System Engineering*, 1995(2), July 1995.