Developing a Class Hierarchy for Object-Oriented Transaction Processing

Daniel L. McCue

Computing Laboratory
University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU, UK.

Abstract. This paper describes the development of a class hierarchy to support distributed transaction processing. Inheritance and polymorphism, key features of the object oriented programming model, have been used to develop a hierarchy of classes which convey to their subclasses the behaviours of persistence, concurrency-control, recoverability and identity necessary for distributed transaction processing. The development is traced from the requirements of distributed transaction processing to the definition of classes supporting these key properties. The system is interesting in both its development and its results. The development, not based on any rigourous design methodology, illustrates some of the design decisions unique to object-oriented systems. The resulting class hierarchy provides a flexible, object-oriented interface for reliable distributed programming. The paper includes a step-by-step description of the design of the classes and the class hierarchy.

1 Introduction

To perform object-level transaction processing, an application programmer must have a means for accessing and manipulating persistent objects through atomic operations (or operation sequences). This paper explains the development of a class hierarchy which provides such facilities. Starting from interface abstractions, concepts of recovery, persistence, concurrency control, the discussion continues with the development of a complete class hierarchy incorporating facilities for transaction management.

To simplify program access to permanent objects, a programming language facility is provided by which objects can be defined to be persistent (i.e., their state is maintained across program executions). To relieve programmers of the burden of considering the effects of concurrent access to objects or various kinds of system failure during execution, we present a language construct that directly supports atomic actions. Together, these tools allow the programmer to develop robust distributed applications within an object-oriented programming framework.

The remainder of this paper is divided into two parts followed by a brief concluding section. Section 2 describes the evolution of the class hierarchy from the desired properties, at the leaves of the hierarchy, through the necessary supporting classes to the root class, *Checkpointing*. Section 3 describes the class, *AtomicAction*, and its relationship to the property classes. The development of this class library is interesting in itself as an example of an approach to object-oriented design. The concluding section summarises the ideas presented and briefly describes the state of the current implementation of this system.

2 Objects and Atomic Actions

A computational model that has been widely advocated for constructing robust distributed applications uses *atomic actions* (*atomic transactions*) to provide fault-tolerant operations on objects. An object is an instance of some *class*. Each 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 of an object have access to its instance variables and can thus modify the internal state. It is assumed that, in the absence of failures and concurrency, the invocation of any of these operations produces consistent (class specific) state changes to the object.

Operation invocations may be contained within *atomic actions* which have the properties:

- *serialisability*
- *failure atomicity*
- *permanence of effect*

The first property ensures that the concurrent execution of programs which access common objects is free from interference (i.e. a concurrent execution can be shown to be equivalent to some serial order of execution). Some form of concurrency control policy, such as that enforced by two-phase locking, is also required to ensure the serialisability property of actions. The second property, *failure atomicity*, ensures that a computation either terminates normally (*commits*), producing the intended results (and intended state change to the objects involved) or it *aborts* producing no results and no state change to the object(s). Once a computation *commits*, the results produced are not destroyed by subsequent node crashes. This is ensured by the third property - *permanence of effect* - which requires that any state changes produced (i.e. new states of objects modified in the atomic action) are recorded on *stable storage*, a type of storage which can survive node crashes with high probability. A *commit protocol* is required during the termination of an atomic action to ensure that either all the objects updated within the action have their new states recorded on stable storage (committed), or, if the atomic action aborts, no updates get recorded [4].

The object and atomic action model provides a natural framework for designing fault-tolerant systems with persistent objects. In our model, persistent objects are passive and normally reside in object stores which are designed to be stable. Atomic actions are used to ensure consistent state changes to objects, despite system failures.

2.1 Object Properties

Object-oriented programming languages provide an means for programmers to express common operations of a collection of types in the form of an abstract class definition. One way to extend the programming model of an object-oriented programming language like C++ [17] to include abstract concepts such as persistence, is to define *classes* that provide these facilities. New objects, derived from these *property classes* will inherit the behaviour of the parent classes. There are three properties in particular that we would like to provide to applications programmers to support reliable distributed programming: persistence, concurrency control and recovery. In addition, fundamental concepts of object identity and naming must be addressed.

- *Identity* – For shared or persistent objects, a unique identity must be determined. This identity must be independent of space, time and contents of the object. Local, transient objects may not need this "unique identity" property.

- *Naming* – For human access, mappings must be provided from strings to the unique identities mentioned above. These name mappings might not be unique; a single unique identity may have several names associated with it and a single name might designate different objects in different contexts or at different times.

- *Persistence* –The ability to declare classes of objects to be persistent is useful in its own right and quite orthogonal to the use of atomic actions. Persistent objects and temporary objects should be indistinguishable in the programming model, although persistent objects may require additional support at initialisation and termination (where the persistent objects get their state loaded or stored).

- *Recovery* – The recovery property is largely independent of the persistence property. For backward error recovery schemes (e.g., shadow paging), a persistent object may be "recoverable" in the sense that its old stable state is available until the new stable state replaces it. But this is not a property of all persistence management schemes (e.g., update-in-place). Furthermore, the boundaries of recovery regions do not always coincide with the boundaries of persistence regions, for example, for objects which are modified in the course of execution of nested atomic actions. Finally, recovery may be desirable independent of persistence. In some contexts, it may be useful to have "temporary" objects that can nevertheless establish and manage recovery points. Hence, the property of recovery can and should be separated from persistence.

- *Concurrency control* – Concurrency control is yet a third orthogonal property which a programmer might want to attribute to a class of objects. Shared objects, whether persistent or recoverable or neither, may require concurrency control. For example, in parallel processing systems, concurrency control is routinely applied to temporary, non-recoverable objects (e.g., locking elements of a matrix used in a parallel algorithm to solve PDEs).

Each of these abstract properties may be used independent of each other and of atomic actions. The properties can be freely mixed as all combinations have plausible applications. However, in the context of an atomic action system, the *operations* of these abstract classes must be invoked according to a strict protocol to ensure the three properties of atomic actions: serialisability, atomicity and permanence of effect. For example, a concurrency control class exports operations to acquire and release locks, but the atomic action boundaries define *when* these operations should be called. The following sections describe these properties in more detail.

2.2 Object Identity

Object identity is fundamental to sharing and persistence and hence a consistent view of object identity is critical to the integration of database-style persistence into programming languages. In their excellent paper on the subject of object identity, Khoshafian and Copeland state, "Identity is that property of an object that distinguishes it from all other objects. Most programming and database languages use variable names to distinguish temporary objects, mixing addressability and identity. Most database systems use identifier keys (i.e., attributes which uniquely identify a tuple) to distinguish persistent objects, mixing data value and identity. Both of these approaches compromise identity"[12]. The authors go on to discuss the impact of identity on the semantics of object equality and conclude that, "The most powerful technique for supporting identity is through surrogates [2] [8][11][6]. Surrogates are system-generated, globally unique identifiers, completely independent of any physical location" [12]. This assertion about the usefulness of surrogates applies to the construction of reliable distributed systems: For persistent objects to be retrieved, they must have an identity which is unique in space and time and independent of the state of the object. To maintain the consistency of an object which is shared by two or more atomic actions, the object must have a unique identity, ideally, one which is independent of the state of the object.

A class of surrogates called unique identifiers (UIDs) can be defined. One way to generate identifying numbers which are sure to be unique in a distributed system is to concatenate a local, monotonically increasing timestamp with some kind of unique host-identifier. Thus, each host can produce unique identifiers autonomously without the overhead of consulting any other hosts in the system. This method for construction of identifiers will not defeat the condition of complete independence of physical location laid down by Khoshafian and Copeland as long as the host information in the UID is not relied upon to locate objects.

The adoption of globally unique identifiers for objects results in a two-level addressing scheme for objects. Objects which are active in memory will necessarily be addressed using machine addresses (unless the machine supports direct access via UID such as the Recursiv [9]). Objects which are remote or passive must be addressed by UID since they have no meaningful machine address in the local context.

Emerald [10] and Amber [5] are examples of systems that hide the distinction between local and global references by a combination of compiler-inserted code and run-time support. The possibility of hiding this distinction is only available because these systems employ special programming languages for which customised compilers and run-time support can be constructed. Persistent programming languages like PS-Algol [1] which ignore problems of distribution and heterogeneity can manage to hide this distinction efficiently through a combination of clever compiler and run-time support. In SOS [16] as in the work described in this paper, the distinction between local and global references is made visible to programmers although support tools are provided to make manipulation of the two types of addressing as straightforward as possible.

The class UID provides these unique surrogates. All objects in the system which can be identified by a UID can be grouped together into a separate class, Identified. Thus, "identified" objects are just those objects that have UIDs.

2.3 Object Naming

While system-generated unique identifiers are important as a means of unambiguous identification of objects, they are not especially convenient for programmers to use. Programmers and users would prefer to refer to objects by names such as "/usr/local/telephone.book" or "General Ledger". Naming objects with human-readable string names, typically implemented by a *name server*, can be expressed in the class hierarchy as a new class, derived from UID, called NamedUID. Each instance of the class NamedUID is an object which represents a mapping from a string name to a UID. Because the class NamedUID is derived from UID, it *is* a UID and can be used in any context in which a UID is required (see figure 1).

Figure 1: Abstract Classes for Object Identity and Naming

2.4 Persistence, Recovery, Concurrency Control

The properties desired for programming *persistent* objects using *atomic actions* can be manifest by object classes:

- Persistence can be represented by an abstract class. *Persistent*, which provides operations to manage the activation and update of stable (passive) copies of an object i.e., a stable storage interface;
- Recovery can be represented by another abstract class, *Recoverable* which provides operations to define the start and end of recovery regions [13] and manages the restoration of a consistent object state in the event of a failure;
- Concurrency control can be represented by a *ConcurrencyControlled* (abstract) class which provides operations to enforce some concurrency control protocol e.g., strict two-phase locking, timestamp ordering (See figure 2).

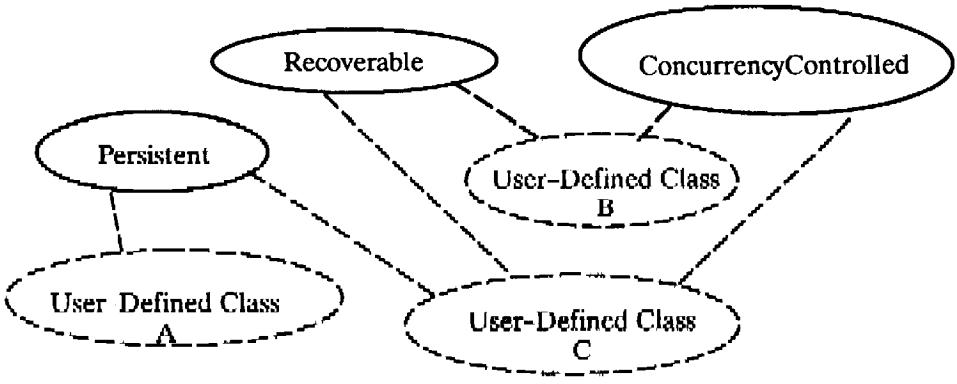

Figure 2: Some possible derivations from object hierarchy including
(A) a simple persistent object class
(B) a temporary recoverable, concurrency-controlled object class
(C) a persistent, recoverable, concurrency-controlled object class.

2.5 Persistence

The existence of a class, *Persistent*, which confers the property of persistence on classes of objects derived from it should simplify the programmer's job of maintaining permanent state. Rather than relying on "files" or "relations" which cannot naturally represent objects, a programmer can create objects that automatically maintain their state across program executions. However, just as files and relations have names that identify the collections that they represent, and elements of files and relations have keys (or seek keys) that identify the individual elements, persistent objects require an identity which can be interpreted across program execution boundaries. The class UID, described above, serves to uniquely identify persistent objects. Since each persistent object is "identified", the class Persistent derives from the class Identified (see figure 3).

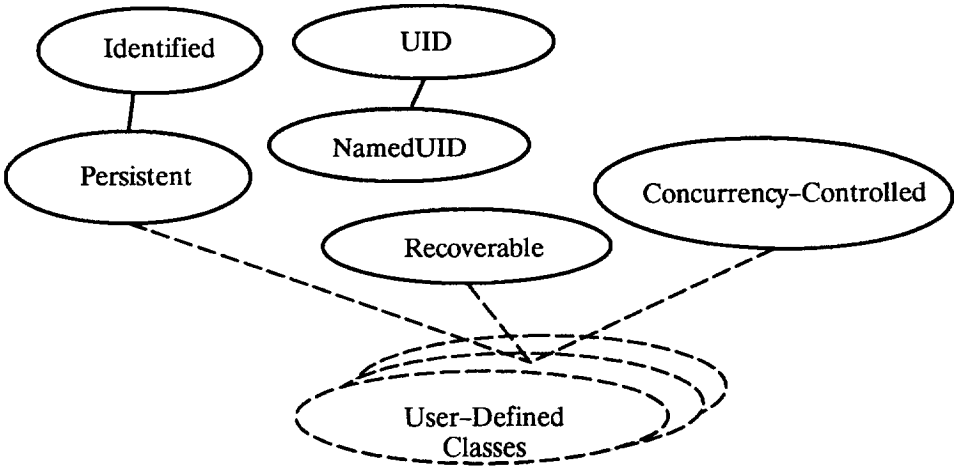

Figure 3: Object hierarchy including the concept of identity

2.6 Saving and Restoring Object State from Stable Storage

Persistence requires another property as well as identity; since the state of the object will be maintained on permanent storage independent of the execution context in which it was created, a mechanism must be provided to convert the state of an active object into a passive state (e.g., a bit-stream). There are several aspects to this conversion.

- Since the passive state of the object may be re-activated in a different execution context, memory pointers and other context dependent data must be converted to a context independent form (e.g., UID).

- Since the passive state of the object may be re-activated on a machine of a different architecture, the state should be saved in an architecture-neutral form analogous to "network byte-order" and possibly including word size and floating-point number representation encoding, depending on the range of architectures involved.

To encapsulate the semantics of an object whose state can be converted back and forth from a passive, context independent representation to an active, "in-memory" representation, a new abstract class, *Checkpointing,* is defined. All objects with the property that their state can be converted between active and passive representations are derived from *Checkpointing.* An additional class is required to represent the passive state of these objects, *ObjectState.* Members of class *Checkpointing* can save their current state in an instance of class *ObjectState* or restore their state from an instance of class *ObjectState.* Since all aspects of an object to be saved are subject to these conversions, *Checkpointing* is a root class for this hierarchy (See figure 4).

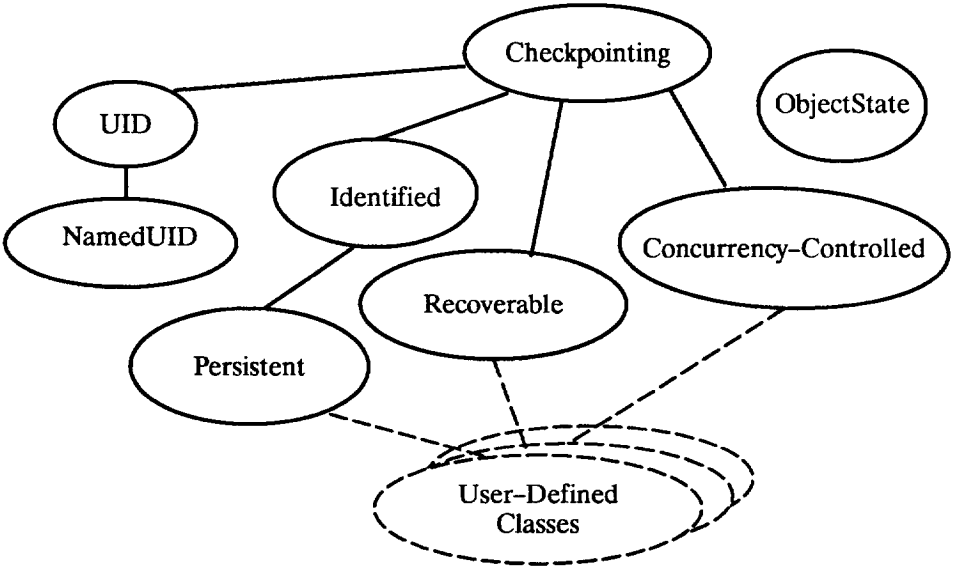

Figure 4: An Object Hierarchy rooted in the class Checkpointing

2.7 Recovery

The definition of the class *Recoverable* is formulated to encompass various recovery techniques (e.g., state-based recovery, operation logging) without modification (i.e., these would simply be different implementations of the interface). Its operation definitions refer only to *establishing* a recovery region, *reverting* to a consistent state defined by the start of the recovery region and *discarding* a recovery region. In the context of atomic actions, these operations would be invoked at action begin, abort and commit respectively. However, in any sort of fault-tolerant software, whether atomic actions are used or not, these operations could be employed to ensure consistent state changes to objects. For example, an exception block construct could be defined which *established* a new recovery region, *reverting* to the prior state in the case of an exception being raised or *discarding* the recovery region when the exception block is exited normally.

2.8 Saving and Restoring Object State for Recovery

For state-based backward error recovery [13], there is a requirement to capture the state of an active object at various points in its execution. The same checkpointing operations required to support the persistence mechanism can be used to save and restore the state of the object for state-based recovery.

A potentially faster means of capturing the state has been proposed: by augmenting the operations of the virtual memory system of the host operating system, it may be possible to "snapshot" the current memory state of an active object [18] [19]. This method, while faster, does make the recovery state dependent on both machine architecture and process context. Migrating an object whose recovery state is so captured could be extremely difficult. There are also potentially large recovery times required when different objects share the same page. However, if migration of active objects is not considered and page placement issues are resolved, the virtual memory paging technique may well be the most efficient way to effect state-based recovery.

2.9 Concurrency Control

The most widely used form of concurrency control in use in transaction processing today is (strict) two-phase locking [4]. Many alternatives or extensions have been proposed (e.g., non-strict two-phase locking, time-stamp ordering, optimistic locking). While these alternative locking strategies may be excellent for specific classes of objects, they all suffer some disadvantage in recovery (e.g., cascaded aborts), distribution (e.g., central timestamp generator) or correctness (e.g., by breaking serializability or failure atomicity) that limits their general applicability. Since conventional transaction processing systems are not flexible enough to vary the concurrency control method by object class or workload, they adopt a single general purpose policy to apply to all classes of objects involved in transactions (e.g., two-phase locking). By encapsulating the semantics of concurrency control in an object class as in figure 4, it is possible to gain back the flexibility to vary the concurrency control policy by object class.

To define a concurrency control class that supports two-phase locking, it is necessary to define a *Lock* class. The *Lock* class encapsulates the lock conflict information and implements a specific locking policy (e.g., simultaneous read, exclusive write). Even within the protocol of two-phase locking, there may a requirement for different lock types for different classes of object. This requirement is met by allowing new *Lock* classes to be derived from the basic *Lock* class. The concurrency controller embodied by the *ConcurrencyControlled* class can continue to manage operation invocation conflicts by invoking polymorphic operations defined by the basic *Lock* class. Since locks may be part of the state of an object which needs to be saved (for recovery, migration or persistence), the *Lock* class is derived from the class *Checkpointing* (see figure 5).

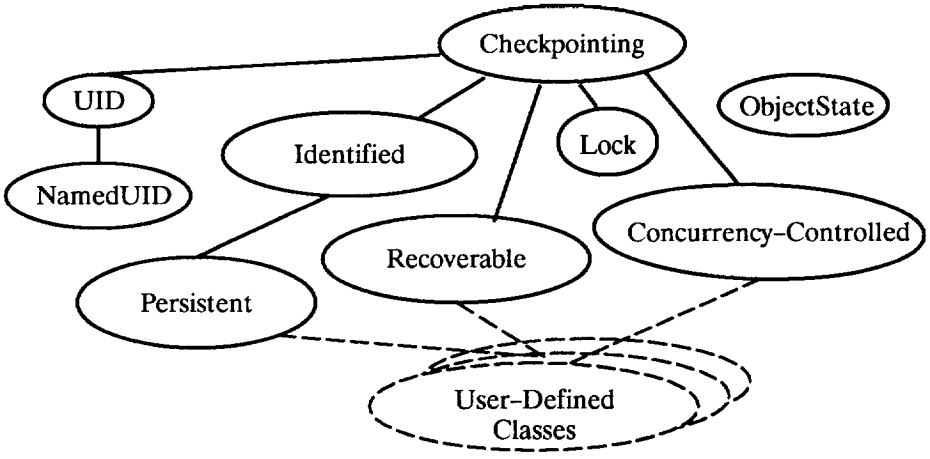

Figure 5: Principal Classes in the object hierarchy

3 Atomic Actions

In the spirit of the object programming model, atomic actions can be defined as objects exporting the operations, *Begin, Commit* and *Abort*. Given the hierarchy shown in figure 5, where should such a class be derived? How can the operations of persistence, recovery and concurrency control be invoked in response to changes in the state of an atomic action?

In conventional atomic action systems, a certain amount of state associated with each atomic action must be recorded on stable storage. This data, sometimes called the *intentions list*, is usually written to a *transaction log* [4] which contains a (roughly) time-ordered sequence of transaction records. In the model presented here, the class *AtomicAction* can simply be derived directly from the class *Persistent* (see figure 6), the intentions list being captured as the permanent state of the atomic action object.

The atomic action data itself is not recoverable so there is no need to derive from the class *Recoverable*. Since the information associated with each atomic action is encapsulated in a separate object which is accessed in a controlled way, there is no need to apply concurrency control to these objects (unlike a transaction log). In effect, the concurrency control problem has been shifted from the log to the object store where concurrent accesses to objects are serialised by the object storage system.

3.1 Integrating Atomic Actions with other classes

The properties exported by such classes as *Persistent*, *Recoverable*, etc are useful in their own right, but in an atomic action processing environment, there is a definite protocol imposed on the invocation of the operations of these classes. When an action commits, for example, locks should be released (an operation of the *ConcurrencyControlled* class), permanent state should be recorded on stable storage (an operation of the *Persistent* class) and recovery state should be discarded (an operation of the *Recoverable* class). In fact, all the operations of these classes should be invoked *only* in respect of state changes to the enclosing atomic action (i.e., an individual object should not release locks mid-transaction). How can we arrange for the correct operations to be invoked at the correct times?

One possibility for co-ordinating the operations of the atomic action with the operations of the "property" classes, would be to require programmers to explicitly insert those invocations with the atomic action invocations. For example, after invoking AtomicAction.Commit, an application could invoke operations on each object involved in the atomic action e.g., ConcurrencyControlled.ReleaseAllLocks. If application programs were actually written in some higher-level language which was translated down into C++, this option might be acceptable - all those explicit calls could be inserted by the translator. However, in the absence of such a preprocessor, we would like a way to trigger the correct operations "automatically" for instances of classes derived from these "property" classes.

A common characteristic, or attribute, of these property classes is that they need to take action in respect of state changes in the controlling atomic action. The object paradigm provides a mechanism for capturing common attributes of classes: inheritance. To ensure that each of the classes, *Persistent*, *Recoverable*, *ConcurrencyControlled*, is involved in these state changes, the class hierarchy is modified to derive these classes from a new common root, *ActionRelated* (See figure 6). The class *ActionRelated* captures the semantic of "classes which are dependent on atomic action state changes".

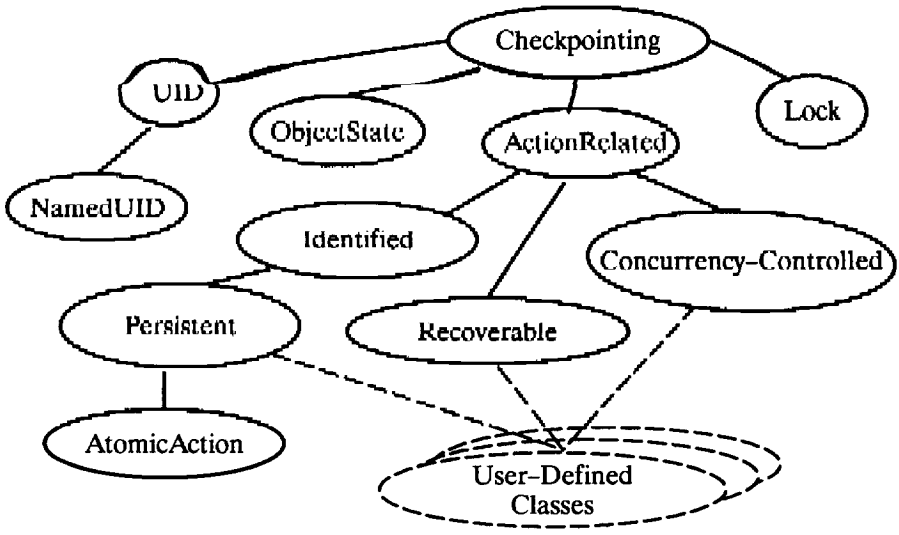

Figure 6: Object hierarchy including *ActionRelated* mechanism

For each object of a class derived from *ActionRelated* there exists a list of operations to perform for each state transition of the action (e.g., for prepare, commit, abort). The number and behaviour of elements on this list is a function of the classes from which the object is derived. For example, if a class X is multiply derived from both *Persistent* and *Recoverable*, the objects of class X will each (logically) contain a list of two elements – one specifying which *Persistent* operations need to be performed for any action state change, the other element specifying which *Recoverable* operations need to be performed for any action state change. For a different class Y derived only from *Persistent*, the list will be of length one, specifying only the effects on persistence of each action state change. These lists are created when the class constructors execute and maintained by the *ActionRelated* class. But how do these lists come to be associated with actions? How does an action state change trigger evaluation of the elements of these action lists?

To ensure that some action is taken on behalf of each object of the *ActionRelated* classes, a list of such objects must be associated with each atomic action. Whenever an operation of a class is invoked, it must "join" the current action (if any). That is, each object must insure (by explicit invocation of a join operation) that it is associated with (i.e., added to the list of objects affected by) the current atomic action. Because this explicit "join" invocation is inserted by the class developer, the user of the class is free to declare objects and actions as required, assured that the appropriate persistence, recovery and concurrency-control actions are invoked as the actions move from state to state. Thus, the user of a class never needs to invoke any operations of the classes, *Persistent*, *Recoverable*, or *ConcurrencyControlled* explicitly. These operations are invoked according to the strict protocol of the action to effect the desired properties of failure atomicity, permanence of effect and serialisability.

The class hierarchy shown in figure 6 has all the necessary functionality to support atomic action processing for persistent objects. An application developer can derive new object classes with precisely the properties desired (i.e., persistence, recovery, concurrency control). By instantiating objects of the class *AtomicAction*, the applications developer can create atomic actions to manipulate objects of these classes. The addition of distribution, replication and migration properties would extend the breadth of the hierarchy but would not require any structural changes.

4 Conclusion

A class hierarchy has been presented as a framework for atomic action processing on persistent objects. The development of the class hierarchy was explained starting with the requirements – necessary properties of objects for atomic action processing – and proceeding up the hierarchy to incorporate necessary supporting class structure. The resulting class hierarchy presents a simple, flexible interface to programmers, allowing the precise requirements of objects to be expressed through the use of inheritance.

The classes described in the foregoing sections have been implemented in C++. In the non-distributed environment, these classes already provide a useful transactional extension to the language. With the addition of run-time support for distribution, these classes form a complete infrastructure for the development of reliable distributed, object-oriented applications.

Further research is presently underway to extend the hierarchy to include replication and migration support. It is unclear whether these extensions should take the same form as the existing "property" classes or use external support in the form of pre-processing – in effect, extending the (C++) language and run-time system.

A complex issue which needs further thought is how to modify the existing hierarchy to incorporate different commit protocols. While the specific locking protocol is nicely encapsulated in the implementation of the *ConcurrenyControlled* class, the two-phase commit protocol is visible in the *interface* to the class *ActionRelated*. This implies that a different atomic action object employing a different commit protocol, e.g., three-phase commit, would require a different implementation of the classes derived from *ActionRelated*. In this sense, the *ActionRelated* class is really a *TwoPhaseCommitActionRelated* class in the current system. Yet a persistent object ought to be able to participate (even simultaneously) in atomic actions employing different commit protocols. This is not possible in the current design.

Acknowledgments

The work reported here has been supported in part by grants from the UK Science and Engineering Council and ESPRIT project No. 2267 (Integrated Systems Architecture). The author gratefully acknowledges the continuing support of Professor Shrivastava and the Arjuna team at the University of Newcastle upon Tyne.

References

[1] M. P. Atkinson, K. J. Chisholm, W. P. Cockshott, "PS-Algol: An Algol with a Persistent Heap", ACM Sigplan Notices, Vol. 17, No. 7, pp. 24-31.

[2] J.R. Abrial, "Data Semantics", in Data Base Management, J. W. Klimbie and K. L. Kofferman, (eds.), North-Holland Publishing Co., New York, 1974.

[3] Advanced Networked Systems Architecture (ANSA) Reference Manual, Volume A, Release 1.00, Part VI, Computational Projection, March 1989.

[4] P.A. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[5] J. S. Chase, F. Amador, E. Lazowska, H. Levy, R. Littlefield, "The Amber System: Parallel Programming on a Network of Multiprocessors", in Proceedings of the 12th ACM Symposium on Operating System Principles, Litchfield Park AZ, December 1989, pp. 147-158.

[6] E. F. Codd, "Extending the Database Relational Model to Capture More Meaning", ACM Transactions on Database Systems, Vol 4, No. 4, December 1979.

[7] G. N. Dixon, G. D. Parrington, S. K. Shrivastava and S. M. Wheater, "The Treatment of Persistent Objects in Arjuna", in Proceedings of ECOOP '89, University of Nottingham, pp. 169-204, July 1989.

[8] P. A. V. Hall, J. Owlett and S. J. P Todd, "Relations and Entities", in Modeling in Data Base Management Systems, G. M. Nijssen, ed., North-Holland Publishing Co., New York, 1976.

[9] D. M. Harland, REKURSIV - Object Oriented Computer Architecture, Ellis Horwood, 1988.

[10] Jul, E., H. Levy, N. Hutchinson, A. Black, "Fine-grained Mobility in the Emerald System", ACM Transactions on Computer Systems, Vol 6, No 1, February 1988, pp. 109-133.

[11] W. Kent, Data and Reality, North-Holland Publishing Co. New York 1978.

[12] S. N. Khoshafian and G. P. Copeland, "Object Identity", in Proceedings of OOPSLA '86, September 1986, pp. 406-416.

[13] P. A. Lee and T. Anderson, "Fault Tolerance: Principles and Practice", Second, Revised Edition, Springer-Verlag, 1990.

[14] J. E. B. Moss, "Nested Transactions: An Approach to Reliable Distributed Computing", Technical Report MIT/LCS/TR-260, Massachusetts Institute of Technology, Laboratory for Computer Science, April, 1981.

[15] G. D. Parrington and S. K. Shrivastava, "Implementing Concurrency Control in Reliable Distributed Object-Oriented Systems", in Proceedings of ECOOP '88. Norway, August 1988.

[16] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, and C. Vaolt, "SOS: An Object Oriented Operating System – Assessment and Perspectives", Computing Systems, Vol. 2, No. 7, 1989.

[17] B. Stroustrup, "The C++ Programming Language", Addison Wesley, 1986.

[18] S. Thatte, "Persistent Memory: Merging AI-knowledge and Databases", in Readings in Object-Oriented Database Systems, S. B. Zdonik and D. Maier, (eds.), Morgan Kaufmann, San Mateo CA, 1990, pp. 242–250.

[19] I. L. Traiger, "Virtual Memory Management for Database Systems", ACM Operating Systems Review, Vol. 16, October 1982, pp. 24–48.