Inheritance of Synchronization and Recovery Properties in Avalon/C++

David L. Detlefs, Maurice P. Herlihy, and Jeannette M. Wing
Carnegie Mellon University

Many object-oriented programming languages provide inheritance mechanisms that allow programmers to define new data types as extensions of previously existing types. By supporting incremental modification, inheritance mechanisms are generally thought to enhance modularity and reusability. In this article, we describe our experience adapting inheritance mechanisms to a new application domain: reliable distributed systems. We give an overview of Avalon/C++, a programming language under development at Carnegie Mellon University. Avalon/C++ allows programmers to "customize" the synchronization and fault-tolerance properties of new data types by letting them inherit properties such as serializability and crash recovery from a library of basic types. We believe that inheritance can facilitate implementing and reasoning about programs that must cope with the complex behavior associated with concurrency and failures.

Programs in reliable distributed systems are more complex than their sequential counterparts. Avalon/C++ helps programmers cope with the behavior associated with concurrency and failures.

Reliable distributed systems are inherently more complex than their conventional sequential counterparts. In addition to the usual concerns about functional correctness, the programmer must address issues arising from concurrency and fault-tolerance. In the presence of concurrency and failures, the data these systems manage must satisfy application-dependent consistency constraints, which can encompass objects stored at multiple nodes in a distributed system. The data must be highly available, that is, highly likely to be accessible when needed. Data must also be reliable, that is, unlikely to be lost or corrupted by system failures. Examples of applications that require such properties include databases, airline reservations, and electronic banking systems, where incorrect or unavailable data can be extremely expensive.

The following sections describe the transaction model used to organize distributed computations, some relevant features of C++, and an overview of the Avalon/C++ base hierarchy. We then describe in more detail each of the hierarchy's classes and some restrictions on their use that must be obeyed to preserve their semantic intent. An extended example illustrates a directory-type implementation that uses all three of the base classes. Finally, we discuss related work.

Transaction model of computation

A distributed system consists of multiple computers (called nodes) that communicate through a network. Distributed systems are typically subject to several kinds of failures: nodes can crash, perhaps destroying local disk storage, and communications can fail, via lost messages or network partitions. A widely accepted technique for preserving consistency in the presence of failures and concurrency involves organizing computations as sequential processes called transactions. Transactions are *atomic*, that is, *serializable* (transactions appear to execute in a serial order), *transaction consistent* (a transaction either succeeds completely and commits, or aborts and has no effect), and *persis-*

0018-9162/88/1200-0057$01.00 © 1988 IEEE

tent (the effects of a committed transaction survive failures).

A program in Avalon consists of a set of servers, each of which encapsulates a set of objects and exports a set of operations and a set of constructors. A server resides at a single physical node, but each node can be home to multiple servers. An application program can explicitly create a server at a specified node by calling one of its constructors. Rather than sharing data directly, servers communicate by calling one another's operations. An operation call is a remote procedure call with call-by-value transmission of arguments and results. Objects within a server can be stable or volatile; stable objects survive crashes, while volatile objects do not. Avalon/C++ includes a variety of primitives (not discussed here) for creating transactions in sequence or in parallel, and for aborting and committing transactions. Each transaction is the execution of a sequence of operations and is identified with a single process.

Transactions in Avalon/C++ can be nested. A subtransaction's commit depends on its parent's; aborting a parent will roll back a committed child's effects. A transaction's effects become permanent only when it commits at the top level. We use standard tree terminology when discussing nested transactions: A transaction T has a unique parent, a (possibly empty) set of siblings, and sets of ancestors and descendants. A transaction is considered its own ancestor or descendant.

Avalon/C++ provides transaction semantics via atomic objects. Atomic objects ensure the serializability, transaction consistency, and persistence of transactions that use their operations. All objects shared by transactions must be atomic. Avalon/C++ provides a collection of built-in atomic types, and users can define their own atomic types by inheritance from the built-in types.

Sometimes, guaranteeing atomicity at all levels of a system can be too expensive. Instead, implementing atomic objects from nonatomic components is often useful. In Avalon, such components, called recoverable objects, guarantee persistence in the presence of crashes.

Avalon relies on the Camelot system[1] to handle operating-system-level details of transaction management, internode communication, commit protocols, and automatic crash recovery.

C++

C++[2,3] is an object-oriented extension of C[4] designed to combine advantages of C, such as concise syntax, efficient object code, and portability, with important features of object-oriented programming, such as abstract data types, inheritance, and generic functions.

C++ uses the class construct to define abstract data types. (We use the terms "class" and "data type" interchangeably.) A class contains members, which are objects of any C++ type, including functions. For example, one might enforce bounds checking on array accesses by the vector class in Figure 1. The elts and count members are private; operations of the vector class can access them, but external clients of the class cannot. The other members are public; they provide the only means for clients to manipulate the object. Bounds checking is ensured by allowing clients to access the vector elements only through the store and fetch operations. Declaring store to be void indicates that the operation returns no results.

We can also define new classes in C++ by inheritance from an existing class. The old class is the superclass, and the new class the derived class. Each derived class has a single, explicit superclass. (A newer version of C++ supports multiple inheritance, where a class can inherit from more than one superclass.) The public members of the superclass become public members of the derived class. The derived class cannot access the superclass's private members. Thus, even in the derived-class implementation, the inherited superclass object must be manipulated using its public members.

For example, Figure 2 defines a vector2 class that extends the vector class by allowing a lower index bound other than zero. Vector2 keeps track of the index lower bound in the private member lbound. It directly inherits the size operation, but it overloads the fetch and store operations with slightly modified operations that explicitly call the corresponding operations of the superclass. C++ allows us to declare operations of a class to be virtual. A virtual operation of one class can be overloaded by any of its derived classes, but a derived class that does not need a special version of a virtual operation need not provide one. Instead, the operation of the superclass (or its superclass, etc.) is used. C++ guarantees that the most specific operation is invoked at runtime. Many other languages would call virtual operations "generic functions."

Class vector is a public base class of class vector2, so a public member of class vector is a public member of vector2. Omitting the keyword public from the definition would result in a public member of the superclass becoming a private member of the derived class.

Recent versions of C++ also address the occasional need for finer control over the visibility of inherited members by adding a new member classification called *protected*. Protected members are something of a compromise between public and private members; protected members of an inherited class become private members of the derived class.

Avalon/C++ base hierarchy

Conventional sequential languages typically use inheritance to implement an object's functional properties, that is, properties whose meaning can be given by simple pre- and postconditions. In Avalon/C++, however, we use inheritance to implement more complex, nonfunctional properties such as serializability, transaction consistency, and persistence.

The Avalon/C++ base hierarchy consists of three classes, as shown in Figure 3. Each base class provides primitives for implementors of derived classes to ensure the nonfunctional properties of objects of the derived classes. The recoverable class provides primitives for ensuring persistence and thus a means of defining recoverable types. Both atomic and subatomic classes provide primitives for ensuring atomicity and thus two different means of defining atomic types. Putting the recoverable class at the root of the hierarchy makes sense, since atomicity encompasses persistence. Moreover, factoring out recoverable's operations from those of the other two classes lets programmers define nonatomic (but recoverable) objects, such as objects for which synchronization is not a concern (usually because correct synchronization is provided by objects that contain them at a higher level). The difference between the atomic and subatomic classes is that subatomic gives programmers a finer-grained control over synchronization and crash recovery.

Programmers define their own recoverable or atomic types by derivation from the appropriate class. We emphasize that persistence and atomicity, like more conventional functional properties, cannot be inherited automatically. Instead, the base classes provide the means by which the implementor of the derived class can ensure these properties. Users of the derived class can then rely on the guarantee provided by the implementor. For example, an implementor of an atomic_set type would derive from the atomic class, explicitly using the inherited locking primitives to implement operations of atomic_set. Users of the class atomic_set can then treat objects of that type as atomic without additional explicit synchronization. Although inheritance of persistence and atomicity is not automatic, we will give simple guidelines that guarantee persistence of classes derived from recoverable, and atomicity of classes derived from atomic.

The recoverable class

Recoverable—the most basic class in our hierarchy—lets its derived classes ensure persistence. The restored state of a recoverable object is guaranteed to reflect all operations performed by transactions that committed before the crash, and possibly some operations of transactions uncommitted at the time of the crash. Before presenting the class definition for recoverable, we describe our underlying model of storage.

Three-level storage model. Conceptually, there are three kinds of storage for objects: volatile, nonvolatile, and stable. We assume that local storage of a distributed system's nodes is structured as a virtual memory system, where volatile semiconductor memory serves as a cache for memory pages from a nonvolatile backing store, such as a magnetic disk. Recoverable objects reside in this local storage. Since nodes are subject to crashes that destroy all their local storage, to survive such crashes recoverable objects must be written to stable storage—a medium with a high probability of surviving crashes. (Stable storage can be implemented by replicating data.) If we log every recoverable object to stable storage after performing modifying operations on it, we can recover a consistent state after a crash by "replaying" the log.

Replaying the log will restore a system's state (indeed, the Argus system[5] uses this scheme). Nevertheless, recovering the system state entirely from the log is time-consuming. Camelot hastens crash recovery by dividing crashes into two classes: node failures and media failures. A media failure destroys both volatile and nonvolatile storage, while a node failure destroys only volatile storage. In practice, node failures are far more common than media failures. A protocol known as write-ahead logging[6] optimizes recovery from node failures by modifying an object as follows:

(1) The pages containing the object are pinned in volatile storage; they cannot be returned to nonvolatile storage

```
class vector {
  int* elts;                              // Array of elements.
  int count;                              // Size of array.
 public:
  vector(int sz)                          // Create vector.
  ~vector( );                             // Destroy vector.
  int size( );                            // Number of elements.
  virtual void store(int e, int i);       // Store element at index.
  virtual int fetch(int i);               // Fetch element from index.
};

int vector::fetch(int i) {
  if (i < 0 || i >= count) error("vector index out of range");
  return elts[i];
}

// Implementations of other vector operations omitted . . .
```

Figure 1. The vector class.

```
class vector2: public vector {
  int lbound;                             // Low bound.
 public:
  vector2(int sz, int lb);                // Create vector.
  ~vector2( );                            // Destroy vector.
  int low( );                             // Return low bound.
  void store(int e, int i);               // Store element at index.
  int fetch(int i);                       // Fetch from index.
};

int vector2::fetch(int i) {
  return vector::fetch(i – lbound);
}

// Implementations of other vector2 operations omitted . . .
```

Figure 2. The vector2 class.

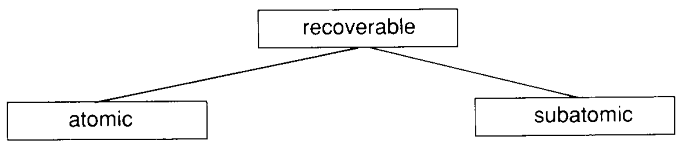

Figure 3. Inheritance hierarchy of the three Avalon/C++ base classes.

```
class recoverable {
  public:
    virtual void pin (int size);       // Pins object in physical memory.
    virtual void unpin (int size);     // Unpins and logs object to stable storage.
}
```

Figure 4. The recoverable class.

```
class rec_X: public recoverable {
    X_type X;
  public:
    void modify( );
};

void rec_X::modify( ) {
    pinning( ) {
       //. . . modify X . . .
    }
}
```

Figure 5. The rec_x class.

```
class rec_int_array: public recoverable {
    int elts[100];
  public:
    rec_int_array(int initial = 0);
    int fetch(int index);
    void store(int index, int value);
    void operator=(rec_int_array& source);      // Array copy.
};

rec_int_array::rec_int_array(int initial) {
    pinning ( )
       for (int i = 0; i < 100; i++) elts[i] = initial;
}

void rec_int_array::store(int index, int value) {
    pinning ( ) elts[index] = value;
}

int rec_int_array::fetch(int index) {
    return elts[index];
}

void rec_int_array::operator=(rec_int_array& source) {
    pinning( )
       for (int i = 0; i < 100; i++) elts[i] = source.fetch(i);
}
```

Figure 6. The rec_int_array class.

until they are unpinned.

(2) The object in volatile memory is modified.
(3) The modifications are logged on stable storage.
(4) The pages are unpinned.

This protocol ensures that programmers can return a recoverable object to a consistent state quickly and efficiently.[1] Of course, they must still log modifications to stable storage to protect against media failure.

Class definition. Figure 4 shows the class header for recoverable. Classes derived from recoverable inherit pin and unpin operations, which can ensure persistence for the derived class. The pin operation causes the pages containing the object to be pinned as required by the write-ahead logging protocol, while unpin logs the modifications to the object and unpins its pages. A recoverable object must be pinned before it is modified and unpinned afterwards. After a crash, a recoverable object will be restored to a previous state in which it was not pinned. Further, if a transaction makes nested pin calls, then the changes made within inner pin/unpin pairs do not become persistent until the outermost unpin is executed. This allows implementors of classes derived from recoverable to guarantee persistence by enclosing all modifications between calls to pin and unpin.

Programmers usually do not explicitly call the pin and unpin operations; instead, Avalon/C++ provides a special control structure, the pinning block, both for syntactic convenience and as a safety measure. The statement

```
pinning (object) <stmnt>;
```

is equivalent to

```
object->pin(sizeof(*object));
<stmnt>;
object->unpin(sizeof(*object));
```

with the additional guarantee that the unpin will execute even if <stmnt> passes control outside the block prematurely, such as by executing a break or return. If a pinning statement within a class definition omits the object name, it defaults to the value "this," which refers to the object whose member is being defined.

Using the recoverable class. Figure 5 shows a class definition for a class rec_X derived from recoverable and containing a member X, and an operation modify that modifies X. Without the pinning block, the modification to X would never be written to stable storage. Persistence could then be violated if a transaction that executed the rec_X operation committed. If a crash occurred, that transaction's effects would not survive.

Since the pin and unpin operations are public members of recoverable, they are public members of classes derived from recoverable. To see why it is useful to make pin and unpin public, consider a recoverable array of integers, rec_int_array. An object of this type should provide normal array operations such as store and fetch, but should do so in a way that ensures persistence. We could implement rec_int_array as a derived class of recoverable as shown in Figure 6.

Now, suppose we have a rec_int_array of 100 integers, and we want to add 1 to each element. We can use a loop where each element is fetched, incremented, and stored back into the array. Given the above implementation of store, we would make 100 calls each to pin and unpin. Unfortunately, unpin's log write is expensive in terms of both stable storage, which is a scarce resource, and time. Clients can avoid this expense by explicitly enclosing the loop in a pinning block (see Figure 7). Here, the pin and unpin calls made by store are much less expensive, because the implementations of pin and unpin recognize when an object is already pinned, and return immediately.

The pin and unpin operations can be overloaded. Consider implementing a recoverable array whose size can be adjusted dynamically. The dynamic array is implemented as a list in which each element includes a 100-element integer array, a size indicating how much of that array is used, and a pointer to the next list element, possibly null (see Figure 8). This implementation has the disadvantage that the pin operation exported by rec_dyn_array pins only the first list element. If the array is repeatedly updated in a loop, as discussed above, then each access to a subsequent list element will generate a new log record.

A simple remedy is to overload the pin and unpin operations to dereference and pin the next list element (see Figure 9). The redefined pin pins the first element by an explicit call to the pin operation provided by recoverable and then recursively pins its successor. This example illustrates how the combination of inheritance and overloading can help customize properties such as failure recovery.

In summary, we can define recoverable types as subclasses of recoverable. If an operation that modifies a recoverable object calls the inherited pin and unpin operations properly, the object will be persistent. If a client calls an object's operations many times, as in a loop, then enclosing those operations in a pinning block can enhance performance.

```
// Pin and log only once instead of 100 times.
pinning(&a)
   for (int i = 0; i < 100; i++) a.store(i, a.fetch(i) + 1);
```

Figure 7. A pinning block for the rec_int_array class.

```
class rec_dyn_array : public recoverable {
   int elts[100];                        // 100 element array
   int size;                             // How much of elts is used
   rec_dyn_array* next;                  // Null => end of list
 public:
   . . .
};
```

Figure 8. Implementation of a recoverable dynamic array.

```
void rec_dyn_array::pin(int size) {
   recoverable::pin(size);
   if (next) next->pin(sizeof(*next));       // Pin next element
}

void rec_dyn_array::unpin(int size) {
   recoverable::unpin(size);
   if (next) next->unpin(sizeof(*next));     // Unpin next element
}
```

Figure 9. Redefinition of pin and unpin.

The atomic class

Atomic, the second base class in our hierarchy, is a subclass of recoverable, specialized to provide two-phase read/write locking and automatic recovery. Locking ensures serializability, and an automatic recovery mechanism for objects derived from the atomic class ensures transaction consistency. (Note that we differentiate between objects derived from the atomic class and atomic objects as defined in the section "Transaction model of computation.")

Class definition. Think of objects derived from the atomic class as containing long-term locks (see Figure 10). Read_lock gains a read lock for its caller. Many transactions can simultaneously hold read locks on an object. Write_lock gains a write lock for its caller; if one transaction holds a write lock on an object, no other transaction can hold either kind of lock. Transactions hold locks until they commit or abort. Read_lock and write_lock suspend the calling transaction until the requested lock can be granted, which can involve waiting for other transactions to

```
class atomic: public recoverable {
 public:
  // "Pin" and "unpin" are inherited from "recoverable."
  virtual void write_lock( );          // Atomically obtains a long-term write lock.
  virtual void read_lock( );           // Atomically obtains a long-term read lock.
}
```

Figure 10. The atomic class.

```
class at_int_array: public atomic {
  rec_int_array elems;

 public:
  at_int_array(int initial = 0) : elems(initial) { };

  void store(int index, int value) {
    write_lock( );
    elems.store(index, value);
  }

  int fetch(int index) {
    read_lock( );
    return elems.fetch(index);
  }
};
```

Figure 11. An at_int_array class that inherits from atomic.

complete and release their locks. If read_lock or write_lock is called while the calling transaction already holds the appropriate lock on an object, it returns immediately.

Classes derived from atomic should divide their operations into writers and readers, that is, operations that do and do not modify the objects of the class. To ensure serializability, reader operations should call read_lock on entry, and writer operations should call write_lock. Note that no short-term mutual exclusion lock on the object is necessary: If any transaction holds a read lock on an object, then no transaction holds a write lock, so all are free to read the object without fear of its being modified as they read it. Conversely, if one transaction holds a write lock on an object, no other transaction can hold either type of lock, so it need not fear interference.

Atomic objects must also be transaction consistent, that is, the effects of aborted transactions, including those aborted by crashes, must be undone. The Avalon runtime system guarantees transaction consistency by performing special abort processing. Thus, implementors of atomic types derived from the atomic class need not provide explicit commit or abort operations.

Finally, persistence is inherited from recoverable. Since atomic is a subclass of recoverable, the pin and unpin operations of recoverable are public operations of atomic and can ensure persistence.

Using the atomic class. Figure 11 shows an implementation of an at_int_array class that inherits from atomic. Since write_lock and read_lock are public operations of atomic, clients of classes derived from atomic can call them. Clients could call these operations explicitly to decrease the likelihood of deadlock. Two transactions, T1 and T2, might each want to obtain write locks on objects A and B; if T1 gets A, and T2 gets B, deadlock will occur—neither can make any progress. Deadlock can be avoided if all transactions obtain locks on the objects they require in some system-wide canonical order. Therefore, clients could structure their code so that each transaction obtains all the locks it requires before executing any operations. They would do this with explicit calls to read_lock and write_lock.

The atomic class uses specially optimized facilities provided by the Camelot system. It is probably appropriate for deriving most atomic types.

The subatomic class

The third, and perhaps most interesting, base class in our hierarchy is subatomic. Like atomic, subatomic allows objects of its derived classes to ensure atomicity. While atomic provides a quick and convenient way to define new atomic objects, subatomic provides more complex primitives to give programmers more detailed control over their objects' synchronization and recovery mechanisms. Programmers can use this control to exploit type-specific properties of objects, permitting higher levels of concurrency and more-efficient recovery.

Transaction identifiers. The Avalon trans_id class creates and tests transaction identifiers. (see Figure 12). Note that Avalon/C++ defines bool to be an enumeration type with TRUE set to 1 and FALSE set to 0.

A new trans_id is created by a call to the constructor:

```
trans_id tid = trans_id();
```

Rather than simply returning the calling transaction's identifier, the trans_id constructor creates and commits a dummy subtransaction, returning the subtransaction's trans_id to the parent. We chose this alternative semantics because it is often convenient for a transaction to generate multiple trans_ids (for example, one for each of its operations) ordered in the order of their creation.

We can test the system's knowledge about the transaction serialization ordering by the overloaded operators < and >. For example, if the expression t1 < t2 evaluates to true, then if t2 commits, t1 will also commit and be serialized before t2. Note that < induces a partial order on trans_ids; if t1 and t2 are active concurrently, both t1 < t2 and t2 < t1 will evaluate to false.

It is sometimes convenient to test whether one transaction is a descendant of another in the transaction tree. If descendant(t1, t2) evaluates to true, then t1 is a descendant of t2. (A friend in C++ is a nonmember operation that is allowed access to the private part of a class.)

```
class trans_id: public recoverable {
    . . .                                          // Hidden representation.
 public:
    trans_id( );                                   // Constructor.
    bool operator==(trans_id& t);                  // Equality.
    bool operator<(trans_id& t);                   // Serialized before?
    bool operator>(trans_id& t);                   // Serialized after?
    // Is 1st a child of 2nd?
    friend bool descendant(trans_id& t1, trans_id& t2);
};
```

Figure 12. The trans_id class.

Class definition. A subatomic object must synchronize concurrent accesses at two levels: short-term synchronization ensures that concurrently invoked operations are executed in mutual exclusion, and long-term synchronization ensures that the effects of transactions are serializable. Short-term synchronization helps to guarantee operation consistency of objects derived from subatomic. Operation consistency means that an operation completes entirely or not at all. Since a transaction is a sequence of operations, operation consistency is a weaker property than transaction consistency; it permits the effects of aborted transactions to be observed, while transaction consistency does not.

Subatomic provides the seize, release, and pause operations for operation-level synchronization (see Figure 13). Each subatomic object contains a short-term lock, similar to a monitor lock or semaphore. Only one transaction can hold the short-term lock at a time. The seize operation obtains the lock, and release relinquishes it. Pause releases the lock, waits for some duration, and reacquires it before returning. Thus, these operations allow transactions mutually exclusive access to subatomic objects. Note that these operations are protected members of the subatomic class. They are not provided to clients of derived classes, since it would not be useful for clients to call them.

```
class subatomic: public recoverable {
 protected:
  void seize( );                          // Gains short-term lock.
  void release( );                        // Releases short-term lock.
  void pause( );                   // Temporarily releases short term lock.
 public:
  // "Pin" and "unpin" are public, by inheritance from "recoverable."

  virtual void commit(trans_id& t);       // Called after transaction commit.
  virtual void abort(trans_id& t);        // Called after transaction abort.
}
```

Figure 13. The subatomic class.

Like pin and unpin, Avalon/C++ programmers typically do not call these operations directly. Instead, Avalon/C++ provides a special control construct, the when statement, to enhance safety and syntactic convenience:

```
when (<TEST>) {
  <. . . BODY . . .>
}
```

The when statement is a kind of conditional critical region. The calling process calls seize to acquire the object's short-term lock, repeatedly calls pause until the condition becomes true, and then executes the body. It calls release when control leaves the body, either normally or by statements such as break or return. In addition, the when statement can ensure operation consistency: Avalon guarantees that no partial effects are observed if a failure occurs while executing a when.

To implement transaction consistency, subatomic provides commit and abort operations. Whenever a top-level transaction commits (or aborts), the Avalon runtime system calls the commit (or abort) operation of all objects derived from subatomic accessed by that transaction or its descendants. Abort operations are also called when nested transactions "voluntarily" abort. Abort operations usually undo the effects of aborted transactions, while commit operations discard recovery information no longer needed. Since commit and abort are C++ virtual operations, classes derived from subatomic are allowed (and, in this case, expected) to reimplement these operations. When the system calls commit or abort, the most specific implementation for the object will be called. Thus, subatomic allows type-specific commit and abort processing, which is useful and often necessary when implementing user-defined atomic types efficiently. Note that users need not invoke commit and abort explicitly; the system automatically invokes them when appropriate.

Finally, since subatomic is a subclass of recoverable, it inherits persistence from recoverable (as did atomic).

Using the subatomic class. Consider the implementation of an atomic FIFO queue. The easiest way to define such a queue is to inherit from the atomic class. A limitation of this approach is that enqueue and dequeue operations would both be classified as writers, permitting little concurrency. Instead,

```
struct enq_rec {
    int item;                                    // Item enqueued.
    trans_id enqr;                               // Who enqueued it.
    enq_rec(int i, trans_id& en) { item = i; enqr = en; }
};

struct deq_rec {
    int item;                                    // Item dequeued.
    trans_id enqr;                               // Who enqueued it.
    trans_id deqr;                               // Who dequeued it.
    deq_rec(int i, trans_id& en, trans_id& de);
     { item = i; enqr = en; deqr = de; }
};
```

Figure 14. Enqueue and dequeue records.

```
class atomic_queue : public subatomic {
    deq_stack deqd;                 // Stack of deq records.
    enq_heap enqd;                  // Heap of enq records.
  public:
    atomic_queue( ) { };            // Create empty queue.
    void enq(int item);             // Enqueue an item.
    int deq( );                     // Dequeue an item.
    void commit(trans_id& t);
    void abort(trans_id& t);
    ~atomic_queue( );               // Destroy queue.
};
```

Figure 15. Implementation of an atomic FIFO queue.

```
void atomic_queue::enq(int item) {
    trans_id tid = trans_id( );
    when (deqd.is_empty( ) || (deqd.top( )->enqr < tid))
        enqd.insert(item, tid);
}

int atomic_queue::deq( ) {
    trans_id tid = trans_id( );
    when ((deqd.is_empty( ) || deqd.top( )->deqr < tid)
            && enqd.min_exists( ) && (enqd.get_min( )->enqr < tid)) {
        enq_rec* min_er = enqd.delete_min( );
        deq_rec dr(*min_er, tid);
        deqd.push(dr);
        return min_er->item;
    }
}
```

Figure 16. Implementations of enq and deq.

we will implement a highly concurrent atomic FIFO queue by inheritance from subatomic. Our implementation supports more concurrency than commutativity-based concurrency control schemes such as two-phase locking. For example, it permits concurrent enq operations, even though enqs do not commute. The implementation also supports more concurrency than any locking-based protocol because it takes advantage of state information. For example, it permits concurrent enq and deq operations while the queue is nonempty.

The representation. Information about enq invocations is recorded in the struct in Figure 14. The item component is the enqueued item. The enqr component is a trans_id generated by the enqueuing transaction, and the last component defines a constructor operation for initializing the struct. Information about deq invocations is recorded similarly (see Figure 14).

The queue is represented in Figure 15. The deqd component is a stack of deq_recs used to undo aborted deq operations. The enqd component is a partially ordered heap of enq_recs, ordered by their enqr fields. A partially ordered heap provides operations to enqueue an enq_rec, to test whether a unique oldest enq_rec exists, to dequeue it if it does, and to discard all enq_rec's committed with respect to a particular transaction identifier.

Our implementation satisfies the following representation invariant:

- Assuming all enqueued items are distinct, an item is either enqueued or dequeued, not both. If an enq_rec containing [x, enq_tid] is in the enqd component, then there is no deq_rec containing [x, enq_tid, deq_tid] in the deqd component, and vice-versa.
- The stack order of two items mirrors both their enqueuing order and their dequeuing order. If d1 is below d2 in the deqd stack, then d1.enqr < d2.enqr and d1.deqr < d2.deqr.
- Any dequeued item must have been enqueued previously. For all deq_recs d, d.enqr < d.deqr.

The operations. Enq and deq operations can proceed under the following conditions. A transaction P can dequeue an item if (1) the most recent dequeuing transaction is committed with respect to P, and (2) a unique oldest element exists in the queue whose enqueuing transaction is committed with respect to P. The first condition ensures that P will not dequeue the wrong item if the earlier dequeuer aborts, and the second condition ensures that there is something for P to dequeue. Similarly, P can enqueue an item if the last item dequeued was enqueued by a transaction Q committed with respect to P. This condition ensures that P will not be serialized before Q, violating the FIFO ordering.

Given these conditions, enq and deq are implemented as shown in Figure 16. Enq checks whether the item most recently dequeued was enqueued by a transaction committed with respect to the caller. If so, the current trans_id and the new item are inserted in enqd. Otherwise, the transaction releases the short-term lock and tries again later. Deq tests whether the most recent dequeuing transaction has committed with respect to the caller and whether enqd has a unique oldest item. If the transaction that enqueued this item has committed with respect to the caller, it removes the item from enqd and records it in deqd. Otherwise, the caller releases the short-term lock, suspends execution, and tries again later.

Commit and abort are implemented as shown in Figure 17. When a top-level transaction commits, it discards deq_recs no longer needed for recovery. (The representation invariant ensures that all deq_recs below the top are also superfluous and can be discarded.) Abort has more work to do. It undoes every operation executed by a transaction that is a descendant of the aborting transaction. It interprets deqd as an undo log, popping records for aborted operations and inserting the items back in enqd. Abort then flushes all items enqueued by the aborted transaction and its descendants.

```
void atomic_queue::commit(trans_id& committer) {
  when (TRUE)
    if (!deqd.is_empty( ) && descendant(deqd.top( )->deqr, committer)) {
      deqd.clear( );
    }
}

void atomic_queue::abort(trans_id& aborter) {
  when (TRUE) {
    while (!deqd.is_empty( ) && descendant(deqd.top( )->deqr, aborter)) {
      deq_rec* d = deqd.pop( );
      enqd.insert(d->item, d->enqr);
    }
    enqd.discard(aborter);
  }
}
```

Figure 17. Implementations of commit and abort.

Restrictions on containers

Some types are (conceptually) parameterized over the types of objects they can contain. To preserve a type's intended meaning, some restrictions are necessary on the types that can instantiate these parameterized container types.

Restrictions for recoverable. Consider the class rec_array, a generalization of the rec_int_array class parameterized over the element type of the array. We must ask what kinds of objects we can put in rec_arrays and still maintain persistence of the array object considered as a whole. First, any type stored in-line is permissible. An in-line type is any type that contains no pointers. The fundamental types of C++ (char, int, or float) are in-line. A struct whose members are all in-line is itself in-line. Similarly, a C++ array whose elements are all in-line is also in-line. Note that if a rec_array has an in-line element type, then logging the array to stable storage will log all the elements as well.

Problems arise when we consider pointer types. If we declare A to be a rec_array of pointers to ints, is A persistent? The answer is no, since A[1] points to an int, which is not a recoverable object. We could change the value of this int during a transaction, thus conceptually modifying the state of the array, but no record of this modification would ever reach stable storage, violating persistence.

Here, then, is a rule for ensuring that a type is persistent: If objects of a type can contain other objects, and if the containing type is intended to be persistent, then the contained objects must be an in-line type or a pointer to a recoverable object. This rule ensures that the latest version of a recoverable object will be written to stable storage every time an operation that modifies it completes.

The inverse problem occurs with an object not meant to be persistent, but which conceptually contains some recoverable object. The Camelot system requires allocation of recoverable and nonrecoverable data in different sections of memory. If we allow a nonrecoverable object to contain an in-line recoverable object, we must allocate space for the aggregate object in one of these sections of memory. We cannot put it in the nonrecoverable section, since the recoverable object would become nonrecoverable. We also cannot put the object in the recoverable section for a more subtle reason. If we allocated memory there and a node crash occurred, the nonrecoverable part of the object would become meaningless after recovery; the storage allocator would think it had been allocated, although no variables reference it. This type of garbage would build up over time. Therefore, as a rule we forbid nonrecoverable objects to contain recoverable objects in-line; they can only point to recoverable objects.

Restrictions for serializability. Similar restrictions apply to serializability. If a container type is intended to ensure serializability of the transactions accessing it, it should be instantiated either with an in-line type or with a pointer to another type that ensures serializability. Care must be taken that nested atomic objects do not lead to deadlock.

An extended example

All three base classes can combine to implement an atomic directory type. A directory stores pairs of values, where one value (the key) is used to retrieve the other (the item). The insert operation

```
bool directory::insert(key k, item i)
```

inserts a new binding in the directory, returning FALSE if the key is already present and TRUE otherwise. The remove operation

```
bool directory::remove(key k)
```

removes the item bound to the given key,

```
// Lock modes
enum mode {
   INSERT_T_LOCK,              // Successful insert.
   INSERT_F_LOCK,              // Unsuccessful insert.
   REMOVE_T_LOCK,              // Successful remove.
   REMOVE_F_LOCK,              // Unsuccessful remove.
   ALTER_T_LOCK,               // Successful alter.
   ALTER_F_LOCK,               // Unsuccessful alter.
   LOOKUP_LOCK                 // Lookup.
};

// All synchronization is done through the lock manager.
struct lock_info {mode m; key k;};

class lock_mgr: public recoverable {
   // Private representation not shown ...
  public:
   lock_mgr( );
   bool conflict(key k, mode m, trans_id& t);   // Any conflicts?
   void acquire(key k, mode m, trans_id& t);    // Grant lock.
   bool is_locked(key k);                       // Is key locked?
   lock_info* release(trans_id& t);             // Release and return lock.
};

// Cells are atomic in order to get automatic commit and abort processing.
struct cell: public atomic {
   item value;
   cell(item i) { pinning( ) value = i; }
   item operator=(item rhs);                    // Assign an item to the cell.
   operator item( );                            // Coercion from cell to item.
};

// Discard binding when unlocked and present = 0.
struct binding: public recoverable{
   int present;                                 // inserts - removes.
   cell* target;                                // Current item.
   binding(item i) {
      pinning( ) { present = 1; target = new cell(i); }
   }
   ~binding( ) { delete target; }
};

// Maps keys to bindings.
class map: public recoverable {
   // Private representation not shown ...
  public:
   map( );
   void insert(key k, binding* b);
   void remove(key k);
   binding* lookup(key k);
};
```

Figure 18. Auxiliary definitions for directory example.

returning TRUE if the key is in the directory, and FALSE otherwise. The alter operation,

```
bool directory::alter(key k, item i)
```

alters the item bound to the given key, returning FALSE if the key is absent. Finally, the lookup operation,

```
item directory::lookup(key k)
```

returns the item bound to the given key. For brevity, we assume the key is bound.

The directory example further illustrates how we can use the Avalon/C++ synchronization primitives for type-specific synchronization. Here, all synchronization is done on a per-key basis, so transactions that operate on disjoint sets of keys never interfere. Internally, concurrent operations synchronize by strict two-phase locks. The lock conflict table appears in Table 1. An interesting aspect of this scheme is that lock conflicts take into account not only the names and arguments to operations, but also the operations' results. For example, because an unsuccessful insert (denoted by insert/F) does not modify the key's binding, it need not conflict with a concurrent lookup operation on the same key. On the other hand, a successful insert (insert/T) does modify the key's binding, hence it must conflict with lookup.

The example also illustrates the utility of user-defined commit and abort processing. Transaction recovery is straightforward for objects inheriting from atomic. When a transaction is aborted, the object's earlier value is restored by a bit-wise copy; if it commits, the recovery data is discarded. Camelot does this bit-wise recovery directly, and we use it in the example for operations, such as alter, that overwrite existing bindings. Bit-wise recovery, however, is inadequate for more complex operations such as insert and remove that create or destroy bindings. Instead, commit and abort processing for these operations relies on the commit and abort operations inherited from subatomic and overloaded by the directory implementation.

Class definition. As shown in Figure 18, the enumeration type mode defines lock modes for each operation. The insert, remove, and alter operations have different lock modes depending on whether they return successfully. For example, an insert that returns TRUE must acquire a lock of mode INSERT_T_LOCK, while one that returns FALSE must acquire INSERT_F_LOCK. A lock manager is a recoverable object that keeps track of locks. It provides operations to acquire a lock, to release a lock, and to test whether another transaction holds a conflicting lock.

Each key in the directory is associated with a binding, which is a recoverable struct with two fields. The target field is a pointer to an atomic struct (a cell) that holds the item itself. The present field serves as a count of the number of committed inserts minus the number of committed removes. Thus, present is 1 if the key is bound in the directory's committed state; otherwise, present is 0 and the key appears unbound. Present is initially

1 because new bindings are created only when insertions are performed. A binding can be discarded when it is unlocked (which is true if and only if there are no active insertions or removals) and present is 0.

The association between keys and bindings is maintained by a recoverable map object, also shown in Figure 18. A map provides operations to insert new bindings, to remove existing bindings, to find the binding associated with a given key, and to test whether a particular key is bound. Finally, the directory itself inherits from subatomic (see Figure 19). As its private members it has a lock manager (locks), a map (data), and a collection of auxiliary procedures used to test synchronization conditions.

The simplest operation, lookup, generates a new transaction identifier and enters its critical section when the lock manager reports that no other transaction holds a conflicting lock for that key (see Figure 20). Inside the critical section, the transaction locks the key in lookup mode, finds the key's binding, and returns the associated item.

The insert operation is more complex (see Figure 21). Since the operation's lock depends on the insert's success, it must first check to see whether the key has a binding. The auxiliary insert_check procedure performs this test, checking the status of the binding and the state of the lock manager, and returning a value of an enumeration type. A value of PRESENT indicates that the key is bound and the caller can acquire an INSERT_F_LOCK on that key. A value of ABSENT indicates that the key is unbound and the caller can acquire an INSERT_T_LOCK on that key. A value of BUSY indicates that lock conflicts prevent the binding's status from being determined. The insert operation itself uses the result to determine how to proceed in its critical region. The whenswitch statement is a generalization of the when statement that replaces the Boolean expression with an expression of an enumeration type. If the key is absent, the caller acquires the appropriate lock and then creates (and initializes) a new binding for the key. If the key is present, the caller simply acquires a lock and returns. If the lock returns "busy," the caller suspends and retries later.

The implementations of remove, remove_check, alter, and alter_check are similar.

The commit operation enters its critical section, iterates through the locks held by the committing transaction, and discards any unlocked binding where present is zero (see Figure 22). Handling aborts, in particular for insert and remove operations, is a little more complex. For each successful remove lock, the abort operation locates the associated binding and increments the present field; for each successful insert lock, it decrements present. Finally, it discards superfluous bindings. Note that a key's item is stored in a cell that inherits from atomic, so the effects of aborted alter operations are automatically undone when the cell is recovered (see Figure 23).

Table 1. Lock conflicts for directory.

	insert/T, remove/T	alter/T	insert/F	remove/F, alter/F, lookup
insert/T, remove/T	Conflict	Conflict	Conflict	Conflict
alter/T	Conflict	Conflict		Conflict
insert/F	Conflict			
remove/F, alter/F, lookup	Conflict	Conflict		

```
enum status { PRESENT, ABSENT, BUSY };

class directory: public subatomic {
   lock_mgr locks;
   map data;
   status insert_check(trans_id& t, key k);       // Internal proc.
   status remove_check(trans_id& t, key k);       // Internal proc.
   status alter_check trans_id& t, key k);        // Internal proc.
  public:
   directory( );
   bool insert(key k, item i);
   bool alter(key k, item i);
   bool remove(key k);
   item lookup(key k);
   void commit(trans_id& t);
   void abort(trans_id& t);
};
```

Figure 19. Atomic directory definition.

```
item directory::lookup(key k) {
   trans_id tid = trans_id( );
   when (!locks.conflict(k,
                 LOOKUP_LOCK, tid)) {
      locks.acquire(k,LOOKUP_LOCK,
                                   tid);
      binding* b = data.lookup(k);
      return *(b->target);
   }
}
```

Figure 20. Lookup operation.

The use of inheritance to provide recoverability and atomicity in Avalon/C++ is not closely tied to the details of the C++ inheritance mechanism. It could

```
// Lock modes depend on whether a key is bound.
lock_status directory::insert_check(trans_id& tid, key k) {
  binding* b = data.lookup(k);
  if ((b && b->present == 1) && !locks.conflict(k, INSERT_F_LOCK, tid))
    return PRESENT;
  if ((!b || b->present == 0) && !locks.conflict(k, INSERT_T_LOCK, tid))
    return ABSENT;
  return BUSY;
}

bool directory::insert(key k, item i) {
  trans_id tid = trans_id( );
  whenswitch (insert_check(tid, k)) {
    case ABSENT:
      locks.acquire(k, INSERT_T_LOCK, tid);
      binding* b = new binding(i);
      data.insert(k, b);
      return TRUE;
    case PRESENT:
      locks.acquire(k, INSERT_F_LOCK, tid);
      return FALSE;
  }
}
```

Figure 21. Insert operation.

```
void directory::commit(trans_id& tid) {
  lock_info* info;
  when (TRUE)            // Always ok to commit.
    while (info = locks.release(tid)) {
      key k = info->k;
      binding* b = data.lookup(k);
      if ((b->present==0) && !locks.is_locked(k))
        data.remove(k);
    }
}
```

Figure 22. Commit operation.

```
void directory::abort(trans_id& tid) {
  lock_info* info;
  when (TRUE)
    while (info = locks.release(tid)) {
      key k = info->k;
      binding* b = data.lookup(k);
      switch (info->m) {
        case REMOVE_T_LOCK:
          pinning(b) b->present++;
          break;
        case INSERT_T_LOCK:
          pinning(b) b->present--;
      }
      if ((b->present==0) && !locks.
                        is_locked(k))
        data.remove(k);
    }
}
```

Figure 23. Abort operation.

be adapted to inheritance mechanisms in languages such as Smalltalk,[7] Flavors,[8] CommonLoops,[9] CommonObjects,[10] and Owl.[11] Our extensions would undoubtedly take a slightly different form in a language allowing multiple inheritance.[3]

Avalon/C++ is based on a transaction model of computation. It should be possible to exploit subatomic's provision of user-defined commit and abort operations to support nontransaction-based approaches to crash recovery, which typically use optimistic recovery schemes[12,13] based on rollbacks and replays.

Other projects investigating an object-oriented, transaction-based approach to managing persistent data include Exodus[14] and Arjuna.[15] Avalon/C++ in many ways resembles Argus,[5] a language designed to provide support for fault-tolerant distributed computing. Although Avalon/C++ and Argus provide much of the same functionality, such as support for transactions and atomic data types, programs in the two languages have a different flavor.

User-defined atomic objects in Avalon/C++ are implemented by inheritance from the special built-in classes, while such objects in Argus are typically implemented by including atomic objects in the new object's representation. Avalon/C++ and Argus also use different models of serializability. In Argus, concurrency control is based on a generalization of strict two-phase locking. In Avalon/C++, the ability to query the transaction serialization ordering at runtime (via the trans_id type) permits more concurrency than two-phase locking, while remaining compatible with two-phase locking. Finally, Avalon/C++ and Argus use different recovery techniques. Avalon relies on the Camelot system for basic transaction management, using the write-ahead log protocol for efficient recovery from node failures. Argus recovers directly from the log.

We are currently implementing Avalon/C++ on IBM RTs, DEC MicroVAXs, and Sun-3 workstations using version 1.1 of C++. The implementation comprises a preprocessor that transforms Avalon code to C++ code. We use the Camelot system extensively for low-level transaction support; Camelot, in turn, relies on the Mach operating system[16] for memory management, internode communication, and lightweight processes. We are currently able to compile and run all the code presented in this article.

We believe that inheritance provides an effective way to customize and extend the kind of complex nonfunctional properties, such as serializability, transaction consistency, and persistence, needed to support programs for reliable distributed applications. For each of these properties, there is a core of functionality—such as the basic mechanics of locking, pinning, and logging—that is best provided by the underlying language implementation. Nevertheless, support for user-defined data types sometimes requires extending or modifying that functionality, as illustrated by the example in which a recoverable object needs to pin a component object indirectly referenced through a pointer. The combination of inheritance and overloading provides a simple and flexible way to achieve incremental modification of these complex properties that lie outside the domain of conventional programming languages. □

Acknowledgments

This research was sponsored by the Defense Advanced Research Projects Agency, ARPA Order No. 4976 (Amendment 20), under contract F33615-87-C-1499 monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Wright-Patterson AFB. Additional support for Jeannette Wing was provided in part by the National Science Foundation under grant CCR-8620027.

References

1. A.Z. Spector, R. Pausch, and G. Bruell, "Camelot: A Flexible, Distributed Transaction Processing System," *Proc. Compcon 88*, San Francisco, Calif., Feb. 1988, pp. 432-439.

2. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Mass., 1986.

3. B. Stroustrup, "What is Object-Oriented Programming?," *IEEE Software*, Vol. 5, No. 3, May 1988, pp. 10-20.

4. B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice Hall, Englewood Cliffs, N.J., 1978.

5. B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM Trans. Programming Language and Systems*, Vol. 5, No. 3, July 1983, pp. 382-404.

6. J. Gray, "Notes on Database Operating Systems," in *Operating Systems: An Advanced Course*, Lecture Notes in Computer Science series No. 60, Springer-Verlag, Berlin, 1978, pp. 393-481.

7. A. Goldberg and D. Robson, *SmallTalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Mass., 1983.

8. D.A. Moon, "Object-Oriented Programming with Flavors," *Proc. ACM Conf. Object-Oriented Systems, Languages, and Applications*, 1986, New York, pp. 1-8.

9. D.G. Bobrow et al., "CommonLoops: Merging Lisp and Object-Oriented Programming," *Proc. ACM Conf. Object-Oriented Systems, Languages, and Applications*, 1986, New York, pp. 17-29

10. A. Snyder, *Object-Oriented Programming for Common Lisp*, tech. report ATC-85-1, Software Technology Laboratory, Hewlett-Packard Laboratories, 1985.

11. C. Schaffert et al., "An Introduction to Trellis/Owl," *Proc. ACM Conf. Object-Oriented Systems, Languages, and Applications*, 1986, New York, pp. 9-16.

12. D.R. Jefferson, "Virtual Time," *ACM Trans. Programming Languages and Systems*, Vol. 7, No. 3, July 1985, pp. 404-425.

13. R.E. Strom and S. Yemini, "Optimistic Recovery in Database Systems," *ACM Trans. Computing Systems*, Vol. 3, No. 3, Aug. 1985, pp. 204-226.

14. J.E. Richardson and M.J. Carey, "Programming Constructs for Database Implementation in Exodus," *Proc. SIGMod 87*, May 1987, pp. 208-219.

15. S.K. Shrivastava et al., *A Technical Overview of Arjuna: A System for Reliable Distributed Computing*, tech. report, Computing Laboratory, University of Newcastle upon Tyne, 1988.

16. M. Accetta et al., "Mach: A New Kernel Foundation for Unix Development," *Proc. Summer Usenix*, July 1986, pp. 93-112.

David L. Detlefs is in his fourth year of graduate study in the Computer Science Department at Carnegie Mellon University. His research interests include reliable distributed systems and object-oriented programming languages.

Detlefs received the BS degree in computer science from the Massachusetts Institute of Technology in 1982.

Maurice P. Herlihy is an assistant professor in the Computer Science Department at Carnegie Mellon University. His research interests include algorithms for synchronization and fault-tolerance in distributed and concurrent systems, as well as formal and informal aspects of programming language support.

Herlihy received the BA degree in mathematics from Harvard University, and the MS and PhD degrees from the Massachusetts Institute of Technology.

Jeannette M. Wing is an assistant professor in the Computer Science Department at Carnegie Mellon University. Her research interests include formal specifications, programming languages, concurrent and fault-tolerant distributed systems, and visual languages.

Wing received the BS and MS degrees in 1979 and the PhD degree in 1983, all from the Massachusetts Institute of Technology. She is a member of the IEEE and ACM.

Readers may contact the authors at Carnegie Mellon University, Computer Science Department, Pittsburgh, PA 15213.