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

David Detlefs, Maurice Herlihy, Jeannette Wing

CMU-CS-87-133

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory under contract F33615-84-K-1520. Additional support for J. Wing was provided in part by the National Science Foundation under grant DMC-8519254.

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

David Detlefs, Maurice Herlihy, Jeannette Wing[1]

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

31 March 1987

Abstract

We exploit the inheritance mechanism of object-oriented languages in a new domain, fault-tolerant distributed systems. We use inheritance in Avalon/C++ to transmit properties, such as serializability and crash resilience, that are of specific interest in distributed applications. We present three base classes, RESILIENT, ATOMIC, and DYNAMIC, arranged in a linear hierarchy, and examples of derived classes whose objects guarantee desirable fault-tolerance properties.

1. Introduction

The *inheritance* mechanism provided by a number of object-oriented programming languages (cited in Section 7) is generally thought to be an effective program structuring technique, increasing the modularity and reusability of programs. In these languages, inheritance allows programmers to define types with behavioral properties similar to previously existing types.

In this paper, we show that inheritance can be used effectively in a new application domain, *fault-tolerant distributed systems*. In this domain, we are concerned not only with the usual desired behavioral properties of types as manifest in the sequential domain, but also with properties that are desired in the presence of concurrency and faults. Thus, we show that inheritance is not only useful for transmitting sequential properties from one type to another (e.g., the LIFO property of a stack), but also for transmitting properties like serializability and crash resilience. In particular, we discuss how Avalon/C++ [Herlihy&Wing 87] uses the C++ [Stroustrup 86] inheritance mechanism to provide support for highly concurrent fault-tolerant applications, via objects that provide their own synchronization and recovery.

---

[1]This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory Under Contract F33615-84-K-1520. Additional support for J. Wing was provided in part by the National Science Foundation under grant DMC-8519254.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Fault-tolerant distributed systems require properties that are typically not of interest in the sequential domain. In the presence of concurrency and failures, the data they manage must satisfy application-dependent *consistency constraints*, which may encompass objects stored at multiple sites in a distributed system. The data must be highly *available*, that is, highly likely to be accessible when needed. Data must be *reliable*, that is, unlikely to be lost or corrupted by system failures. Examples of applications that require such properties include airline reservations and electronic banking systems, where incorrect or unavailable data may be extremely expensive.

This paper is organized as follows: Section 2 describes the transaction model used to organize distributed computations, and some relevant features of C++; Sections 3, 4, and 5 describe the base Avalon/C++ class hierarchy; Section 6 describes some restrictions on the use of these classes that must be obeyed to preserve their semantic intent; and Section 7 presents related work and a discussion.

2. Background

2.1. Model of Computation

We assume that distributed computation is done at a set of *nodes* that communicate over a *network*. Such a system is subject to a number of failures: nodes may fail, perhaps destroying local disk storage, and communications may fail, via lost messages or network partitions.

A widely-accepted technique for preserving consistency in the presence of failures and concurrency is to organize computations as sequential processes called *transactions*. Transactions are *atomic*, that is, serializable, recoverable, and permanent. *Serializability* means that transactions appear to execute in a serial order. *Recoverability* means that a transaction either succeeds completely and *commits*, or *aborts* and has no effect. *Permanence* means that the effects of a committed transaction survive failures. Transactions may be nested.

Avalon/C++ provides transaction semantics via *atomic objects*. Atomic objects ensure the serializability, recoverability, and permanence of the 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 may define their own atomic types by inheriting from the built-in types.

Sometimes it may be too expensive to guarantee atomicity at all levels of a system. Instead it is often useful to implement atomic objects from non-atomic components [Weihl&Liskov 85]. In Avalon, such components are called *resilient* objects; they satisfy certain weak consistency properties in the presence of crashes.

2.2. Overview of C++

C++ is an object-oriented extension of C [Kernighan&Ritchie 78], 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.

New types are defined in C++ using the *class* construct. A class is declared to contain *members*. These may be objects, including functions, of any C++ type. Members may either be *public* or *private*. Private members may be manipulated only within the class that implements a type; code external to the class (*clients* of the class) may access only public members. The public/private distinction is often used to provide an abstraction boundary for a type by hiding its representation in private members, and making its operations public members.

C++ supports *inheritance*. A class can be *derived* from another class, and becomes a *subclass* of that class. Each derived class has a single, explicit *superclass*. The public members of the superclass become public members of the subclass. The subclass does *not* inherit the superclass's private members; thus, even in the subclass implementation, the inherited superclass object must be manipulated using its public members. The latest version of C++ (Version 1.2) recognizes that finer control over the visibility of inherited members is sometimes necessary, and adds a new member classification called *protected*. Protected members are something of a compromise between public and private members: when a class with protected members is inherited, its protected members become private members of the derived class. The Avalon/C++ class hierarchy uses all three kinds of members. This hierarchy is made up of three classes, RESILIENT, ATOMIC, and DYNAMIC, arranged in a linear inheritance tree (DYNAMIC inherits from ATOMIC, which inherits from RESILIENT). The next three sections describe these classes in detail.

3. RESILIENT Objects

The most basic class in our hierarchy is RESILIENT. RESILIENT provides the means for its derived classes to ensure *operation-consistency:* after a crash, a resilient object will be recovered in a state that reflects only completed operations; the effects of operations in progress at the time of the crash are never observed. Resilient objects are also *permanent:* the recovered state of a resilient object is guaranteed to reflect all operations performed by transactions that committed before the crash. Before presenting the class definition for RESILIENT, we first describe our underlying model of storage.

3.1. Three-Level Model of Storage

Conceptually, there are three kinds of storage for objects: volatile, non-volatile, and stable. We assume that local storage of nodes of a distributed system is structured as a virtual memory system, where *volatile* semiconductor memory serves as a cache for memory pages from *non-volatile* backing store, such as magnetic disk. Resilient objects reside in this local storage of nodes. Since nodes are subject to crashes that destroy all their local storage, if resilient objects are to survive such crashes, they must be written to *stable storage*, a storage medium that survives

crashes with high probability. (Stable storage may be implemented using redundant hardware [Lampson 81] or replication [Daniels 87].) If every resilient object is *logged* to stable storage after modifying operations are performed on it, then an operation-consistent state may be recovered after a crash by "replaying" the log.

Replaying the log is a sufficient method for restoring a system's state (e.g., it is used by the Argus system [Liskov&Scheifler 83]). However, recovering the state of a system entirely from the log is a time-consuming operation. If we note that crashes can be divided into two classes, *node failures* and *media failures*, we can do better. In a media failure, both volatile and non-volatile storage is destroyed, while in a node failure only volatile storage is lost. In practice, node failures are far more common than media failures. To optimize recovery from node failures, a protocol known as *write-ahead logging* [Gray 78] is used. Write-ahead logging uses the following steps to modify an object:

1. The page(s) containing the object are *pinned* in volatile storage; they cannot be returned to non-volatile storage until they are *unpinned*.
2. Modifications are made to the object in volatile memory.
3. The modifications are logged on stable storage.
4. The page(s) are unpinned.

These steps assure that resilient objects always appear in an operation-consistent state in non-volatile storage. In the case of node failure, then, where the contents of non-volatile storage are preserved, the states of resilient objects may be recovered much more quickly. Of course, modifications must still be logged to stable storage against the occurrence of a media failure.

Avalon uses the Camelot system [Spector et al. 86] to handle operating-system level details of transaction management, such as the write-ahead logging protocol described above. Other Camelot facilities we use include inter-node communication, commit protocols, and crash recovery.

3.2. Class Definition

Figure 3-1 shows the class header for RESILIENT:

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

Figure 3-1: The RESILIENT Class

Classes derived from RESILIENT inherit the PIN and UNPIN operations. These can be used to ensure operation-consistency for the derived class. PIN causes the pages containing the object to be pinned, as in the write-ahead logging protocol, while UNPIN logs the modifications to the object and unpins its pages. After a crash, a resilient object will be recovered to a previous state in which no transaction is pinning it. Further, if a transaction makes nested PIN calls, then the changes made within inner PIN/UNPIN pairs do not become permanent until the outermost UNPIN is executed. This allows implementors of classes derived from RESILIENT to guarantee operation-consistency

by enclosing all modifications made by an operation within an outer PIN/UNPIN block.

As an example, consider a class RES_X_Y derived from RESILIENT, containing members X and Y, both of which are pointers to resilient objects.

```
class res_X_Y: public resilient {
  res_X_type *X;                          // X and Y point to resilient objects.
  res_Y_type *Y;
 public:
  void modify ();
}
```

MODIFY is an operation that modifies the objects pointed to by X and Y. The implementation of modify would have the following structure:

```
void res_X_Y::modify() {
  pin();                          // Pin the entire object.

    (*X).pin();                   // Pin X...
    // ...modify X...
    (*X).unpin();                 // ...and unpin X.

    (*Y).pin();                   // Pin Y...
    // ...modify Y...
    (*Y).unpin();                 // ...and unpin Y.

  unpin();                        // Unpin the entire object.
}
```

Without the outermost PIN/UNPIN pair, operation-consistency might be violated, since a crash occurring after modifications had been made to X but before modifications had been made to Y would leave the object in an inconsistent state.

Since PIN and UNPIN should always be called in pairs, Avalon/C++ provides a special control structure, the pinning block, that enforces this constraint. The structure

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

is equivalent to

```
object.pin();
<stmnt>;
object.unpin();
```

with the additional guarantee that the UNPIN will be executed even if <stmnt> causes control to pass outside the block prematurely, e.g., by executing a break or return. If object is omitted in a pinning statement, it defaults to the value this, which refers to the current object in C++.

3.3. Using the Class RESILIENT

Consider defining a resilient array of integers, RES_INT_ARRAY. An object of this type should provide normal array operations such as STORE and FETCH, but should do so in an operation-consistent manner. We could implement RES_INT_ARRAY as a subclass of RESILIENT as follows:

```
class res_int_array: public resilient {
  int elts[100];
 public:
  res_int_array(int);
  int fetch(int);
  void store(int, int);
  void operator=(res_int_array&);      // Array copy.
};

res_int_array::res_int_array(int initial = 0) {
  pinning()
    for (int i = 1; i <= 100; i++) elts[i] = initial;
}

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

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

void res_int_array::operator=(res_int_array& target) {
  pinning(target)
    for (int i = 1; i <= 100; i++) target.elts[i] = elts[i];
}
```

Now, suppose we have a RES_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 to each of the PIN and UNPIN operations. Unfortunately, the log write done by UNPIN is expensive, both in terms of space taken on stable storage, which is a scarce resource, and in terms of time. Clients can choose to avoid this expense by explicitly enclosing the loop in a pinning block. Clients are permitted to call PIN and UNPIN because they are public members of RESILIENT, and are therefore visible to classes derived from RESILIENT. Thus, this loop could be written as:

```
// Pin and log once, rather than 100 times.
pinning (a)
  for (i; i < 100; i++) a.store(i, a.fetch(i) + 1);
```

Here, the PIN and UNPIN calls made by the STORE operation are not expensive because they recognize that their object is already pinned, and return immediately.

In summary, resilient types can be defined as subclasses of RESILIENT. If an operation that modifies a resilient object calls the inherited PIN and UNPIN operations properly, the object will be operation-consistent and permanent. If a client calls an object's operations many times, as in a loop, performance can be enhanced by enclosing those operations in a `pinning` block.

4. ATOMIC Objects

4.1. Class Definition

Figure 4-1 gives the class header for ATOMIC.

```
class atomic: public resilient {
 protected:
  void seize();                          // Gains short-term lock.
  void release();                        // Releases short-term lock.
  void pause();                          // Releases short-term lock for some period
                                         // of time, then regains it.
 public:
  // "Pin" and "unpin" are public, by inheritance from "resilient."
  virtual void commit(tid);              // Called after a transaction that operated on
                                         // the object commits.
  virtual void abort(tid);               // Called after a transaction that operated on
                                         // the object aborts.
}
```

Figure 4-1: The ATOMIC Class

Objects of class ATOMIC are intended to be atomic, that is, serializable, recoverable, and permanent. We will now discuss how the operations of ATOMIC may be used to satisfy these three aspects of atomicity. Permanence is "inherited" from RESILIENT; since ATOMIC is a subclass of RESILIENT, the PIN and UNPIN operations of RESILIENT are public operations of ATOMIC, and may be used in the same way to ensure permanence and operation-consistency.

Atomic objects must be recoverable, that is, the effects of aborted transactions, including those aborted by crashes, must be undone. Recoverability is a stronger property than resilience. Resilience permits the effects of aborted transactions to be observed, while recoverability does not. To implement recoverability, ATOMIC provides COMMIT and ABORT operations. Whenever a top-level transaction commits (aborts), the Avalon system executes the COMMIT (ABORT) operation on all the atomic objects on which the transaction or its descendents have operated. 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 that is no longer needed. COMMIT and ABORT are C++ *virtual* operations: classes derived from ATOMIC may reimplement these operations, and when COMMIT or ABORT are called by the system, the most specific implementation for the object will be called. (Many other languages would call these *generic* functions.) Thus, ATOMIC allows type-specific commit and abort processing, which is useful and often necessary in implementing user-defined atomic types efficiently, as described in [Greif et al. 86] and [Herlihy&Wing 87].

Many transactions may attempt to execute operations on atomic objects concurrently. They must be synchronized at two levels: at the operation level, so that a transaction may execute an operation without fear of interference from other transactions, and at the transaction level, so that serializability is not violated.

ATOMIC provides the SEIZE, RELEASE, and PAUSE operations to aid derived classes in ensuring operation-level synchronization. Each atomic object contains a *short-term lock*, similar to a monitor lock. Only one transaction may hold the short-term lock at a time. The SEIZE operation obtains the short-term lock, and RELEASE relinquishes it. PAUSE releases the short-term lock, waits, and reacquires it before returning. Thus, these operations allow

transactions mutually exclusive access to atomic objects. Note that these operations are protected members of the ATOMIC class. They are not provided to clients of derived classes, since it would not be useful for clients to call them.

To ensure both operation-level synchronization and transaction-level serializability, operations of atomic types often contain loops of the form:

```
for (;;) {                              // Do forever...
  seize();                              // Enter critical region
  if ( <TEST> ) {                       // OK to proceed?
    <...BODY...>                        // Execute the operation.
    release();                          // Leave critical region.
    return;
  }
  else {
    pause();                            // Try again later.
  }
}
```

<TEST> is a condition that is true when the serializability constraint allows the operation to proceed. It generally involves queries to the Avalon system concerning the state of transactions. **<...BODY...>** represents the substance of the operation. Because this structure is very common, Avalon/C++ provides a special control construct, the `when` statement, to abbreviate it. The following code is equivalent to the code above, with the added guarantee that the short-term lock is released even if a statement of **<...BODY...>** (such as a `return` statement) causes a premature exit.

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

4.2. Using the Class ATOMIC

Consider an example of a user-defined atomic type. We will create the class ATOM_INT_ARRAY, an atomic array of `int`, that ensures synchronization using *two-phase* or *read/write locking* [Eswaran et al. 76]. Below is the interface of a RW_LOCK class that will be used in the representation of ATOM_INT_ARRAY.

```
class rw_lock: public resilient {
 public:
  rw_lock();                            // Create a new lock.
  bool can_read(tid);                   // Returns true => read lock grantable.
  bool can_write(tid);                  // Returns true => write lock grantable.
  void write_lock(tid);                 // Grant write lock.
  void read_lock(tid);                  // Grant read lock.
  void release(tid);                    // Release all locks held by the transaction.
};
```

The representation of ATOM_INT_ARRAY also includes a stack of *versions* of the array, so that we can recover from transaction aborts. This version stack must be kept operation-consistent, so each version is an object of the previously defined class RES_INT_ARRAY. The definition of `version` is shown below.

```
struct version {
  tid who;
  res_int_array* what;
  version(tid t, res_int_array* a){who = t; what = a;};
};
```

We now present the implementation of ATOMIC_INT_ARRAY.

```
class atom_int_array: public atomic {
  rw_lock locks;
  version_stack stack;
 public:
  atom_int_array(int);
  void store(int, int);
  int fetch(int);
  void commit(tid);
  void abort(tid);
};

atom_int_array::atom_int_array(int initial =0) {
  tid self = *(new tid);                    // Own transaction id.
  pinning(){                                // Making update.
    locks.write_lock(self);                 // Grant write lock to self.
    stack.push(version(self, new res_int_array(initial)));
  }
}

void atom_int_array::store(int index, int value) {
  tid self = *(new tid);                    // Caller's tid.
  when (locks.can_write(self))              // Check lock conflict.
    pinning() {                             // Making update.
      locks.write_lock(self);               // Get write lock.
      // Committed wrt last writer?
      if (both(stack.top().who,self)){      // Modify in place...
        version v = stack.pop();            // Copy top element.
        v.elems[index] = value;             // Store.
        stack.push(v);
      } else {                              // Push new version...
        version v(self,new res_int_array);
        v.elems = stack.top().elems;        // Copy top element.
        v.elems[index] = value;             // Store.
        stack.push(v);
      }
    }
}

int atom_int_array::fetch(int index) {
  tid self = *(new tid);
  when (locks.can_read(self))               // Check lock conflict.
    pinning() {                             // Updating lock table.
      locks.read_lock(self);                // Get read lock.
      return stack.top().elems[index];
    }
}

void atom_int_array::commit(tid finished) {
  when (true)                               // Always ok to commit.
    pinning() {                             // Making modification.
      version v = stack.top();              // Get current version.
      while (!stack.empty())
        stack.pop();                        // Discard others.
      stack.push(v);                        // Push back current version.
      locks.release(finished);              // Release locks.
    }
}

void atom_int_array::abort(tid finished) {
  when (true)                               // Always ok to abort.
    pinning() {                             // Making update.
      while (!stack.empty() &&
             both(finished, stack.top().who))
        stack.pop();                        // Discard aborted stack.
      locks.release(finished);              // Release locks.
    }
}
```

The type of read/write locking and version stack recovery used in the above example is a very common method of ensuring atomicity. Avalon/C++ provides a special subclass of ATOMIC, called DYNAMIC, which is optimized for this case, and is described in the next section. While DYNAMIC is a convenient method for implementing many atomic types, for some types greater concurrency can be gained by implementing type-specific transaction synchronization and recovery [Weihl&Liskov 85, Herlihy&Wing 87]. These types need to be derived from class ATOMIC directly.

5. DYNAMIC Objects

DYNAMIC is a subclass of ATOMIC, specialized to provide two-phase read/write locking and automatic recovery. Locking is used to ensure transaction serializability, and the automatic recovery mechanism "rolls back" the effects of aborted transactions on DYNAMIC objects. (The name *dynamic* refers to the particular *local atomicity property* that two-phase locking implements [Weihl 84].)

5.1. Class Definition

Figure 5-1 gives the class header for DYNAMIC.

```
class dynamic: public atomic {
 public:
  void write_lock();                    // Atomically obtains a long-term write lock.
  void read_lock();                     // Atomically obtains a long-term read lock.

  // Note that "commit" and "abort" are inherited from "atomic", and that
  // "pin" and "unpin" are inherited from "resilient" via "atomic."
}
```

Figure 5-1: The DYNAMIC Class

DYNAMIC objects should be thought of as containing *long-term locks*. READ_LOCK gains a *read lock* for its caller. Many transactions may 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 may hold any kind of lock on it. Transactions hold locks until they terminate. READ_LOCK and WRITE_LOCK suspend the calling transaction until the requested lock can be granted, which may involve waiting for other transactions to 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 DYNAMIC 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 may hold any type of lock, so it need not fear interference.

These locking mechanisms can be used to ensure serializability; atomicity also requires that we ensure

recoverability and permanence. DYNAMIC inherits the PIN and UNPIN operations from RESILIENT (via ATOMIC); these can be used to guarantee permanence and operation-consistency. If these are used properly, the Avalon run-time system guarantees recoverability by performing special abort processing that "undoes" the effects of aborted transactions. Thus, atomic types derived from DYNAMIC need not provide commit or abort operations.

5.2. Using the Class DYNAMIC.

We now present another implementation of the ATOM_INT_ARRAY class defined in Section 4. This implementation inherits from DYNAMIC; the contrast in code size and complexity should show the utility of DYNAMIC.

```
class atom_int_array: public dynamic {
  res_int_array elems;
 public:

  atom_int_array(int n, int initial =0) {
    read_lock();
    elems = res_int_array(n, initial);
  }

  void store(int index, int value) {
    write_lock();
    elems[index] = value;
  }

  void fetch(int index) {
    read_lock();
    return elems[index];
  }
}
```

WRITE_LOCK and READ_LOCK are public operations of DYNAMIC, and thus may be called by clients of classes derived from DYNAMIC. Clients might want to call these locking 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, there will be a deadlock -- neither will be able to 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 might want to 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.

DYNAMIC uses specially optimized facilities provided by the Camelot system, and is therefore quite efficient. It is probably appropriate for deriving most atomic types. As noted previously, however, for certain types more concurrency can be allowed by a customized implementation inheriting from ATOMIC instead of DYNAMIC.

6. Restrictions on Containers

Some types are (conceptually) parameterized over the types of objects that they can contain. In order to preserve the intended meaning of the type, some restrictions are necessary on the types that can be used to instantiate these parameterized container types.

6.1. Restrictions for Resilience

Let us consider the class RES_ARRAY, a generalization of the RES_INT_ARRAY class that is parameterized over the element type of the array. It is necessary to ask what kinds of objects we can put in RES_ARRAYs, and still maintain resilience of the array object considered as a whole. First, any type that is 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 in-line. Similarly, a (C++) `array` whose elements are all in-line is in-line. Note that if a RES_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 start to consider pointer types. If we declare `A` to be a RES_ARRAY of pointers to `int`s, is `A` a resilient object? The answer is no, since `A[1]` points to an `int`, which is not a resilient 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, allowing permanence to be violated.

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

The inverse problem occurs when we have a object that is not meant to be resilient, but conceptually contains some resilient object. The Camelot system requires that resilient and non-resilient data be allocated in different sections of memory. If we allow a non-resilient object to contain an in-line resilient object, then we must allocate space for the aggregate object in one of these sections of memory. We cannot put it in the non-resilient section of memory since then the resilient object would not be resilient. We also cannot put the object in the resilient section of memory, for a more subtle reason. If we were to allocate memory there, and there were a node crash, the non-resilient 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 non-resilient objects to contain resilient objects "in-line;" they can point only to resilient objects.

6.2. Restrictions for Serializability

A rule very similar to the rule for resilience applies for serializability. If a container type is intended to ensure serializability of the transactions accessing it, then it should be instantiated either with an in-line type or with a pointer type to another type that ensures serializability. Care must be taken that nested atomic objects do not lead to deadlock.

7. Related Work and Discussion

The use of inheritance to provide resilience and atomicity in Avalon/C++ is not closely tied to the details of the C++ inheritance mechanism. It could be adapted to inheritance mechanisms in languages such as Smalltalk [Goldberg&Robson 83], Flavors [Moon 86], CommonLoops [Bobrow et al. 86], CommonObjects [Snyder 85], and Owl [Schaffert et al. 86]. The *abstract* types of the Emerald language [Black et al. 87] provide capabilities similar to inheritance. It would be especially interesting to investigate what form our extensions might take in a language allowing *multiple* inheritance, where a class may inherit from more than one superclass.

Avalon/C++ is in many ways similar to Argus [Liskov&Scheifler 83], a language designed to provide support for fault-tolerant distributed computing. Like Avalon/C++, Argus allows users to write atomic data types and transactions. The major additional features offered by Avalon/C++ are: (1) user-defined atomic types are defined using type inheritance, whereas in Argus such types are typically defined by including atomic objects in the representation; (2) Avalon provides explicit linguistic support for both hybrid and dynamic atomicity, not just dynamic atomicity; and (3) Avalon uses the Camelot system for its low-level support, thereby gaining the ability to recover efficiently the state of the system's objects from non-volatile storage after node failures. Argus always reconstructs state from the log after crashes.

Dixon and Shrivastava [Dixon&Shrivastava 87] explore the use of C++ for just recoverability. They assume a simpler failure model, where transactions "voluntarily" abort; crashes are not handled. Crash recovery places more stringent demands on our language. Synchronization primitives are not integrated into their class structure at all.

We are currently implementing Avalon/C++. The implementation will take the form of a preprocessor that produces C++ code. We make extensive use of the Camelot system for low-level transaction support; Camelot, in turn, relies on the Mach operating system [Accetta et al. 86] for memory management, inter-node communication, and lightweight processes.

We have shown how inheritance can be applied to a novel domain: fault-tolerant distributed computing. In doing so, we have used inheritance to transmit properties quite different from the normal behavioral properties transmitted in the serial domain. We believe that the inheritance of properties such as synchronization and recovery is an important paradigm for the extension of object-oriented languages to distributed environments, especially in environments with stringent reliability requirements.

References

[Accetta et al. 86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young.
Mach: A New Kernel Foundation for UNIX Development.
In *Proceedings of Summer Usenix*. July, 1986.

[Black et al. 87] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter.
Distribution and Abstract Types in Emerald.
IEEE Transactions on Software Engineering SE-13(1), January, 1987.

[Bobrow et al. 86] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel.
CommonLoops: Merging Lisp and Object-Oriented Programming.
In *Proc. ACM Conference on Object-Oriented Systems, Languages, and Applications*, pages 17-29. Association for Computing Machinery, New York, New York, 1986.

[Daniels 87] D. S. Daniels.
Distributed Logging for Transaction Processing.
In *Proceedings of the 1987 ACM Sigmod International Conference on Management of Data*. Association for Computing Machinery, San Francisco, CA, May, 1987.

[Dixon&Shrivastava 87]
G. Dixon and S. K. Shrivastava.
Exploiting Type Inheritance Facilities to Implement Recoverability in Object Based Systems.
Technical Report, Computing Laboratory, University of Newcastle upon Tyne, 1987.

[Eswaran et al. 76]
K. P. Eswaran, James N. Gray, Raymond A. Lorie, and Irving L. Traiger.
The Notions of Consistency and Predicate Locks in a Database System.
Communications of the ACM 19(11):624-633, November, 1976.

[Goldberg&Robson 83]
A. Goldberg and D. Robson.
SmallTalk-80: The Language and its Implementation.
Addison-Wesley, Reading, MA, 1983.

[Gray 78] J. Gray.
Notes on Database Operationg Systems.
In *Lecture Notes in Computer Science*. Volume 60: *Operating Systems: an Advanced Course*. Springer-Verlag, Berlin, 1978.

[Greif et al. 86] I. Greif, R. Selinger, and W. E. Weihl.
Atomic Data Abstractions in a Distributed Collaborative Editing System.
In *Proceedings of the Thirteenth Annual Symposium on Principles of Programming Languages*, pages 160-172. ACM, January, 1986.

[Herlihy&Wing 87]
M. P. Herlihy and J. M. Wing.
Avalon: Language Support for Reliable Distributed Systems.
In *The Proceedings of the 17th International Symposium on Fault-Tolerant Computing.* Pittsburgh, PA, July, 1987.

[Kernighan&Ritchie 78]
B. W. Kernighan, and D. M. Ritchie.
The C Programming Language.
Prentice-Hall, Englewook Cliffs, NJ, 1978.

[Lampson 81] B. Lampson.
Atomic transactions.
Lecture Notes in Computer Science 105. Distributed Systems: Architecture and Implementation.
Springer-Verlag, Berlin, 1981, pages 246-265.

[Liskov&Scheifler 83]
B. Liskov and R. Scheifler.
Guardians and Actions: Linguistic Support for Robust, Distributed Programs.
ACM Transactions on Programming Language and Systems 5(3):382-404, July, 1983.

[Moon 86] D. A. Moon.
Object-Oriented Programming with Flavors.
In *Proc. ACM Conference on Object-Oriented Systems, Languages, and Applications*, pages 1-8.
Association for Computing Machinery, New York, New York, 1986.

[Schaffert et al. 86]
C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt.
An Introduction to Trellis/Owl.
In *Proc. ACM Conference on Object-Oriented Systems, Languages, and Applications*, pages 9-16.
Association for Computing Machinery, 1986.

[Snyder 85] A. Snyder.
Object-Oriented Programming for Common Lisp.
Technical Report ATC-85-1, Software Technology Laboraty, Hewlett-Packard Laboratories, 1985.

[Spector et al. 86] A. Z. Spector, J. J. Bloch, D. S. Daniels, R. P. Draves, D. Duchamp, J. L. Eppinger, S. G. Menees, D. S. Thompson.
The Camelot Project.
Database Engineering 9(4), December, 1986.
Also available as Technical Report CMU-CS-86-166, Carnegie-Mellon University, November 1986.

[Stroustrup 86] B. Stroustrup.
The C++ Programming Language.
Addison-Wesley, Reading, Massachusetts, 1986.

[Weihl 84] W. E. Weihl.
Specification and Implementation of Atomic Data Types.
PhD thesis, MIT, 1984.

[Weihl&Liskov 85]
W. E. Weihl and B. Liskov.
Implementation of Resilient, Atomic Data Types.
ACM Transactions on Programming Language and Systems 7(2):244-269, April, 1985.