The Avalon/C++ Programming Language (Version 0)

**Jeannette M. Wing, Maurice Herlihy, Stewart Clamen,
David Detlefs, Karen Kietzke, Richard Lerner, Su-Tuen Ling**

CMU-CS-88-209 2

Please send direct comments, corrections, and questions to wing@cs.cmu.edu; send bug reports to avalon@cs.cmu.edu. This document should be informally referred to as "The Avalon Report" since it is more than a language manual.

This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD), 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 suppport for J. Wing was provided in part by the National Science Foundation under grant CCR-8620027.

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

The Avalon/C++ Programming Language (Version 0)

Jeannette Wing
Maurice Herlihy
Stewart Clamen
David Detlefs
Karen Kietzke
Richard Lerner
Su-Yuen Ling

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213-3890

21 December 1988

Abstract

Avalon/C++ is a language for implementing reliable distributed programs. People who wish to read or write Avalon/C++ programs should read this document, though not necessarily all of it. It contains a quick overview of the terminology of our intended application domain, a tutorial-by-example introduction to the language, a reference manual for the Avalon extensions to C++, a library of built-in classes, and a list of practical programming guidelines. The appendices include the language's grammar and the UNIX man pages for *acc*, the Avalon/C++ preprocessor.

Please send direct comments, corrections, and questions to wing@cs.cmu.edu: send bug reports to avalon@cs.cmu.edu. This document should be informally referred to as ''The Avalon Report'' since it is more than a language manual.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), 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 J. Wing was provided in part by the National Science Foundation under grant CCR-8620027. 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.

Table of Contents

List of Figures

1. Overview

1.1. Terminology

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 may crash, 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, transaction-consistent, and persistent. *Serializability* means that transactions appear to execute in a serial order. *Transaction-consistency* (''all-or-nothing'') means that a transaction either succeeds completely and *commits*, or *aborts* and has no effect. *Persistence* means that the effects of a committed transaction survive failures.

An Avalon/C++ program 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 may be home to multiple servers. An application program may 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 may be *stable* or *volatile*; stable objects survive crashes, while volatile objects do not. Avalon/C++ includes a variety of primitives for creating transactions in sequence or in parallel, and for aborting and committing transactions. Each transaction is identified with a process, and is the execution of a sequence of operations.

Transactions in Avalon/C++ may be nested. A subtransaction's commit is dependent on that of its parent; aborting a parent will cause a committed child's effects to be rolled back. 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. If transaction B is an ancestor of A, then A is *committed with respect to* B if every transaction that is both an ancestor of A and a proper descendant of B has committed. If B is not an ancestor of A, then A is committed with respect to B if A is committed with respect to the least common ancestor of A and B in the transaction tree.

Avalon/C++ provides transaction semantics via *atomic objects*. All objects accessed by transactions must be atomic to ensure their serializability, transaction-consistency, and persistence. Avalon/C++ provides a collection of built-in atomic types, and users may define their own atomic types by inheriting from the built-in ones.

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, called *recoverable* objects in Avalon; they satisfy certain weak consistency properties in the presence of crashes. Users who define their own atomic types from non-atomic components are responsible for ensuring that their types are indeed atomic.

1.2. Avalon/C++ Specifics

Avalon/C++ is a superset of C++ [14], itself an extension of C [7]. C++ is 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. We assume the reader has some knowledge of C++ and freely use its terminology; see [14] for more information on C++.

Avalon's run-time environment relies on the Camelot system [13, 12] to handle operating-system level details of transaction management, inter-node communication, commit protocols, and automatic crash recovery. We benefited extensively from the Camelot Library [1], which provides a clean interface between the Avalon and Camelot implementors. Some of Avalon's design was influenced by Camelot, in particular those aspects that Camelot implementors worked hard to make efficient; however, the reader is not expected to know Camelot nor use it directly.

Much of Avalon's design has been inspired by Argus [11] and we owe the descriptions of some of Avalon's control structures to the *Argus Reference Manual* [10]. For other papers on Avalon/C++, please see [2, 5, 6, 16].

1.3. A Roadmap to this Document

The rest of this document is divided as follows:

Chapter 2	A tutorial introduction to the language. Detailed walkthroughs of three simple examples.
Chapter 3	A reference manual for the Avalon extensions to C++. Note that it is only about nine pages long.
Chapter 4	A library of Avalon built-in classes and the *catalog server*.
Chapter 5	A list of practical guidelines for novice and expert programmers.
Appendix I	The full grammar for Avalon/C++.
Appendix II	The Unix man pages for running *acc*, the Avalon/C++ preprocessor.

A Note on Specifications

In writing the descriptions of the meanings of operations, in particular a class's member functions, we use the following clauses:

- **modifies**: A list of objects whose values may possibly change as a result of executing the operation.
- **requires**: A pre-condition on any invocation state of the operation. The caller is responsible for ensuring it holds; the implementor may assume it holds at the point of invocation.
- **when**: A condition on the state of the system that must hold before the operation proceeds. This condition is often necessary to give since the state of the system may change between the point of invocation and the actual point of execution of an operation.
- **ensures**: A post-condition on the returning state. The implementor must ensure that it holds; the caller may assume it holds upon return.

In C++, a pointer to the object for which a member function is invoked is a hidden argument to the function. As C++ does, we refer to this implicit argument as `this` in our specifications.

The absence of a **requires** (**when**) clause is the same as the predicate being TRUE. The absence of a **modifies** clause indicates that no changes are made to the values of any object. This specification style and notational conventions are borrowed from Larch [4].

2. A Tutorial Introduction

An Avalon/C++ *system* consists of a set of *programs*, each of which is an *application* or a *server*. Applications invoke operations on servers, which may, in turn, invoke operations on other servers.

An Avalon server is very much like a C++ class. Just like a class, a server encapsulates some data, and defines the operations that can be used to manipulate that data. A client invokes an operation on a server object using the same syntax it would use to invoke an operation on a class object. There are two main differences between classes and servers. First, a server supports concurrency: more than one client may invoke operations on a server at the same time. These concurrent operations execute as concurrent *threads* (or *lightweight processes*) within the server. The server must be implemented so that this concurrency makes sense. Second, a server's data (if the server is implemented correctly) is *persistent*, i.e., it will survive crashes in a consistent state.

This chapter describes at length three examples, illustrating all the basic features of Avalon/C++. The first example shows how to create, commit, and abort transactions; to invoke operations on servers; and to define and use a simple atomic type derived from the built-in Avalon class **atomic**. The second and third examples illustrate the use of two other built-in classes, **trans_id** and **subatomic**, to show another way Avalon users can define atomic types, and to show what makes Avalon especially different from other (fault-tolerant) distributed programming languages. We hope the reader will see that programming in Avalon/C++ is not much different from ordinary C++ programming.

2.1. Array of Atomic Integers

In this section, we walk through the use and implementation of a simple Avalon server, called "Jill," and client, called "Jack," (so named for historical reasons). The Jill server encapsulates an array of *atomic integers*. From the client's viewpoint, each of these integers is atomic; they are recovered after a crash to the state observed by the last committed transaction, and they ensure the serializability of the transactions that access them. Since each of the elements of the array is atomic, the array as a whole is also atomic. The elements of the Jill array are initially given the value -1 to represent an uninitialized state, after which the Jill server permits only non-negative values to be written in the array.

An atomic array of integers might be useful as a representation for a conference room reservation system. The elements of the array could represent blocks of time, and writing a value into an element could represent reserving the conference room at that time for the person represented by that value. Or, the array could be used to represent a set of bank accounts, indexed by account numbers. Applications that wished to transfer money from one account to another could do so within a transaction, so that no partial transfers would ever happen. These examples are only meant to be suggestive; in both cases, other representations might be more convenient and/or efficient. Still, they show that even a very simple server such as Jill is not too far removed from real-world applications.

2.1.1. Using Jack and Jill

Before we show any Avalon code, let us first see how a user might interact with Jack and Jill. We begin by assuming that the Jill server has been started. To start up Jack on a Unix system (after making sure that the directory containing the **av_jack** executable is on your search path), type:

```
    % av_jack
```

The Jack application starts a transaction and responds with:

```
    Type ? for a list of commands.
    Jack[1]
```

Jack[1] is the prompt. The "1" indicates the current transaction nesting level. If we type "?", we get the following list of commands:

```
Commands are:
    r   Read array element.
    w   Write array element.
    b   Begin nested transaction.
    c   Commit innermost transaction.
    a   Abort innermost transaction.
    A   Abort top level transaction.
    q   Abort top level transaction and quit program.

Jack[1]
```

Let's say we want to read what is stored at location 7 of the array:

```
Jack[1] r
Location to read: 7
Location 7 is uninitialized.
Jack[1]
```

As we can see, we have not yet given location 7 a value. Let's do so:

```
Jack[1] w
Location to write: 7
Value to write: 7
Write succeeded.
Jack[1] r
Location to read: 7
Value at location 7 is 7.
Jack[1]
```

Now we can begin a subtransaction, using the ''b'' command. In this transaction, we first read the value in location 7, and then give it a new value:

```
Jack[1] b
Jack[2] r
Location to read: 7
Value at location 7 is 7.
Jack[2] w
Location to write: 7
Value to write: 27
Write succeeded.
Jack[2] r
Location to read: 7
Value at location 7 is 27.
Jack[2]
```

Note that the prompt has changed to indicate the transaction nesting level. Let's continue with another nested transaction:

```
Jack[2] b
Jack[3] r
Location to read: 7
Value at location 7 is 27.
Jack[3] w
Location to write: 7
Value to write: 37
Write succeeded.
Jack[3] r
Location to read: 7
Value at location 7 is 37.
Jack[3]
```

If we commit this subtransaction, then we return to its parent, with its effects visible:

```
Jack[3] c
Transaction committed.
Jack[2] r
Location to read: 7
Value at location 7 is 37.
Jack[2]
```

Now, however, if we abort the second-level transaction, we return to the top-level transaction, but none of the effects of the aborted transaction (or its children) are visible.

```
Jack[2] a
Transaction aborted as per request.
Jack[1] r
Location to read: 7
Value at location 7 is 7.
Jack[1]
```

Now, suppose we start up another instance of **av_jack** (in another window, perhaps). In this Jack, we start a transaction, and write into location 10. Then we attempt to read the value we have written into location 7.

```
                                        % av_jack
                                        Type ? for a list of commands.
                                        Jack[1] w
                                        Location to write: 10
                                        Value to write: 10
                                        Write succeeded.
                                        Jack[1] r
                                        Location to read: 7
```

The other Jack ("Jack B") does not immediately return an answer. This is because the first Jack ("Jack A") obtained a *write lock* on location 7. This lock excludes all other transactions from observing the value written there. This is needed to ensure serializability: Jack A's transaction may either commit or abort. If it commits, then Jack B's query should return 7; if it aborts, then Jack B should inform the user that location 7 is still uninitialized. Thus, Jack B cannot return anything until Jack A's top-level transaction terminates. Let's commit Jack A's transaction:

```
Jack[1] c
Transaction committed.
(Transaction was top level.)            Value at location 7 is 7.
Jack[1]                                 Jack[1]
```

Committing Jack A's transaction allowed Jack B's transaction to proceed with the completion of the read operation. Now let Jack A start a new transaction. If we attempt to write a new value into location 7 in this transaction, we are also suspended, for similar reasons:

```
Jack[1] w
Location to write: 7
Value to write: 70
```

Jack A cannot write into location 7, because Jack B's transaction has already observed a value there. Jack A must wait for Jack B's transaction to terminate before it can invalidate this observation. Let's terminate Jack B's transaction with an abort:

```
                                        Jack[1] a
                                        Transaction aborted as per request.
Write succeeded.                        (Transaction was top level.)
Jack[1] r                               Jack[1]
Location to read: 7
Value at location 7 is 70.
Jack[1]
```

Note that in this particular situation, even if Jack B had committed, Jack A still reads a 70 at location 7 since Jack A's write would still be serialized after Jack B's read. This scenario has shown how the Jack application can manipulate the atomic integers contained in a Jill server. In doing so, it has demonstrated some of the properties of transactions, nested transactions, and atomic objects.

The next two sections describe the declaration and definition of the Jill server, all the way down to the level of the Avalon built-in **atomic_int** type; then the following section describes the Jack application program.

2.1.2. The Jill Server Declaration

A C++ class has a *declaration* and a *definition*. A class declaration is generally put in an include file, so that all files that need to use the class can have access to the necessary information. The class definition (the bodies of the class operations) is put in one or more files, each of which includes the declaration. An Avalon server should be written av_jill.h:

```
#include <avalon.h>

// Error return codes from operation procedures.
const int INDEX_OUT_OF_BOUNDS = 1; // Attempt to access a location out of bounds.
const int ILLEGAL_VALUE = 2;       // Attempt to insert a negative number.

// System Constants.
const int ARRAY_SIZE = 1000;       // Number of cells in the array.

server jill {
  stable atomic_int data[ARRAY_SIZE];
 public:
  int read(int index);
  void write(int index, int value);
  jill (x_string cmdline, x_string host) : (cmdline, host);
  void main ();
};
```

Figure 2-1: Declaration of Jill Server

using the same conventions. Thus, we will first examine Figure 2-1, the include file that declares the Jill server.

The first line of this file includes the file **avalon.h**. All Avalon programs must include this file before all others. The next three statements in the file declare and initialize constants used in the program. We follow the C++ recommendation against using preprocessor macros whenever possible. The first two constants, **INDEX_OUT_OF_BOUNDS** and **ILLEGAL_VALUE**, are used as error codes. The third, **ARRAY_SIZE**, determines the size of the array.

Next, we come to the declaration of the Jill server. This is textually identical to a C++ class declaration, with the keyword **server** substituted for **class**. A Jill server contains one data member, **data**, and four operations, which are the only means of accessing the server's data. A server differs slightly from a class in that all data members of a server must be private. Here, **data** is also declared to be **stable**, which asserts that it is persistent, i.e., will survive crashes. Avalon guarantees persistence of the built-in atomic data type, **atomic_int**; in general, the programmer must correctly implement any user-defined type of **stable** variables to ensure their persistence. Though the Jill server does not, a server could also have data members that are volatile, that is, not stable. Volatile data are often useful for efficiency, but care should be taken to ensure that all important data is stable. For example, a server might represent a database as set of records, and maintain a volatile index that allows operations to look up records based on different fields of the record. The index would speed up the server during normal operation, but could always be reconstructed after a crash.

The four operations of the Jill server come in two categories: *user operations* and *server operations*. **Read**, **write**, and the constructor, **jill**, are user operations, the ones that clients can invoke. **Read** returns the integer stored at the given index, and **write** writes the given value at the given index. The intent of these should be fairly clear; we will go over their implementations shortly. The constructor is a special user operation invoked to initialize the Jill server. A server will not accept any calls to other user operations until it has received a constructor call, and it will not accept any constructor calls once it has started accepting calls to other user operations. The remaining operation, **main**, is invoked automatically by the server. For implementation reasons, every server must have a **main** operation, even if it has no body. (The definition of **main** serves as a marker, so the Avalon preprocessor can decide where to put the C++ **main** procedure for the server.) If the **main** operation does have a body, it is executed in the background, concurrently with user operations. Another kind of server operation (not shown here), invoked automatically by the system, is an optional **recover** operation. If defined, it is executed whenever the server is started after any crash. A typical **recover** operation might reinitialize volatile data.

2.1.3. The Jill Server Definition

2.1.3.1. Jill's Data Member

Jill's data member, **data**, is a **stable** array of **ARRAY_SIZE atomic_int**'s. An **atomic_int** is an atomic integer, an integer specially implemented so that it ensures the serializability of transactions that access it, and is recovered after a crash with the value observed by the last committed transaction that accessed it. These properties are quite easy to achieve in Avalon. Figure 2-2 shows the declaration and definition of the **atomic_int** class.

atomic_int.h:

```
// Declares the atomic integer class.

#include <avalon.h>

class atomic_int: public atomic {
  int val;
 public:
  int operator=(int rhs);
  operator int();
};
```

atomic_int.av:

```
// Defines the atomic integer class.

#include <avalon.h>

int atomic_int::operator=(int rhs) {
  write_lock();
  pinning () return val = rhs;
}

atomic_int::operator int() {
  read_lock();
  return val;
}
```

Figure 2-2: The atomic_int Class

The file **atomic_int.h** declares the **atomic_int** class. This is *derived* from the class **atomic**, which provides operations that are used to make integers appear atomic. In particular, class **atomic** has two operations, **read_lock** and **write_lock**, which can be used in implementing operations of derived classes.

The class **atomic_int** has one data member, an integer called **val**, which holds the value of the atomic integer. We show two operations of **atomic_int**'s, both of which are C++ overloaded operators. One is the assignment operator, and the other is the coercion operator that converts an **atomic_int** into an **int**. The assignment operator is the only way to change the value of an **atomic_int**, and the coercion to **int** is the only way of using that value in a program. Thus, these operators mediate all access to the atomic integer.

In the file **atomic_int.av**, we see that the implementations of these operations are quite simple. Taking them in reverse order, we see that the **operator int()** simply calls **read_lock** and returns the current value. The assignment operator gets a write lock on the **atomic_int**, and then, within a **pinning** block, it sets the value to a new value, and returns the new value. The pinning block informs the Camelot system that the change must be logged permanently (i.e., to stable storage) so that in the event of crash recovery, the value of an atomic integer is consistent. Modifications to any atomic object should always be made from within a pinning block. The use of read and write locks guarantees that if a transaction observes the value of an atomic integer, then no other transaction may change it until the observer terminates. (Note that data type induction is needed to really make this guarantee; we can prove that this is true only if these two operators are the only ways of accessing **`atomic_int`**'s.)

2.1.3.2. Jill's Operations

Now that we understand atomic integers, we can consider the implementation of the operations of the Jill server. Figure 2-3 shows the contents of the file **`av_jill.av`**, which contains the definitions.

av_jill.av:

```
// The body of the "av_jill" server.

#include "av_jill.h"

int jill::read(int index) {
  // If index is out of bounds, return an error code.
  if (index < 0 || index >= ARRAY_SIZE) undo (INDEX_OUT_OF_BOUNDS) leave;
  return data[index];
}

void jill::write(int index, int value) {
  // If index is out of bounds, return an error code.
  if (index < 0 || index >= ARRAY_SIZE) undo (INDEX_OUT_OF_BOUNDS) leave;

  // If value is negative, return an error code.
  if (value < 0) undo (ILLEGAL_VALUE) leave;

  data[index] = value;
}

jill::jill(x_string cmdline, x_string host) {
  for (int i = 0; i < ARRAY_SIZE; i++) data[i] = -1;
}

void jill::main() {}
```

Figure 2-3: Definition of the Jill Server

`Read` takes an index, and returns the value at that index. **`Read`** assumes that it is being invoked by a client that is executing within a transaction. If the index is not within the array bounds, **`read`** executes the statement:

```
undo (INDEX_OUT_OF_BOUNDS) leave;
```

This aborts the client's transaction. The abort code **`INDEX_OUT_OF_BOUNDS`** can be used in an **`except`** clause, as we will see when we examine the Jack application. If the index passes this test, then we simply return the value in the **`data`** array at the index. Actually this is a little more subtle than that: the elements of **`data`** are **`atomic_int`**'s, and read returns an **`int`**. Thus, the C++ automatic coercion mechanisms call the coercion operator on the indexed element before returning it. The coercion operator gets a read lock on the element before returning its value. **`Write`** is very similar. It checks that the index is within the proper range, and that the value to be written is not negative; if so, it assigns the new value to the element. Again, the overloaded assignment operator of **`atomic_int`** takes care of getting the write lock on the atomic integer and logging its new value. The important lesson to learn from the Jill server is how the right implementation of **`atomic_int`** made it possible to treat **`atomic_int`**'s almost as if they were regular **`int`**'s within the bodies of the server's operations.

The constructor, **jill**, sets all the elements of **data** to -1, as we specified in the description of Jill. Since all servers implicitly inherit from the **server_root** class, the colon syntax tells the server_root constructor where to find the server executable (first argument) and what machine to start it on (second argument). Finally, the server operation **main** has no body but, as we have explained, every server must have a **main** operation.

2.1.4. The Jack Application

This section shows the code for the Avalon application, "Jack," which uses a Jill server. Most Avalon applications look very similar to Jack so in subsequent examples, we will omit the application-side code. When Jack starts, it enters a transaction. It then executes user commands until the user enters the command to exit the program. The user may read or write array elements, start nested transactions, and commit or abort transactions. Figure 2-4 shows the first part of the code in **av_jack.av**.

Like all Avalon programs, **av_jack.av** starts by including **avalon.h**. It also includes **stream.h** and **ctype.h** from the C++ library, and **av_jill.h** to get the declaration of the Jill server. After the includes, **av_jack.av** declares two more constants used as abort codes within this file and declares the two functions defined in this file so that they can be used before they are defined. The next statement declares a global variable of the Jill server type. The client program can invoke operations on this server object just as if it were a class object.

The **main** procedure prints out an initial message and locates the jill server. If it cannot find it, it calls the **jill** constructor with the names of the executable ("av_jill") and local host as arguments. It then repeatedly calls **jill_transaction** until the value of **quit_flag** indicates that the user wants to exit the program. Finally, the **print_help** procedure prints out a help message.

Now we consider the heart of the Jack application, the **jill_transaction** function. **jill_transaction** begins (Figure 2-5) by **starting** a **transaction**. It then enters a command loop, in which it remains until the user decides to quit the program, or terminate (commit or abort) the current top-level transaction. It prints out a prompt (which contains the current transaction nesting level, which it is given as an input.) Next, it gets an input command, and enters a **switch** statement that processes that input. The 'r' and 'w' commands should be fairly self-explanatory. Note that the **read** and **write** operations are invoked on the object denoted by the **jill_srv** variable exactly as if it were a normal class object. The 'c' command uses the **leave** statement to commit and exit the current transaction. The 'a' command aborts the innermost transaction, using the **undo leave** statement. We pass an abort code that indicates that the user aborted the transaction. The 'A' command aborts the current top-level transaction. This is implemented by first aborting the innermost transaction, using a special abort code. We will see in a moment how this code is processed. The 'q' command exits the program. To do this, we set the **quit_flag**, and exit **jill_transaction.** We use the special **undo return** statement to indicate that we not only want to return from the current procedure, but also to abort any transactions started by that procedure. The 'b' command starts a nested transaction by making a recursive call to **jill_transaction** (with **level** incremented by one.) An input of '?' causes the help message to be printed, and if the input command is none of these, a message to that effect is printed.

The rest of **jill_transaction** is shown in Figure 2-6. The first statement in this figure is just after the body of the loop that waited for the **quit_flag** to be set (by a nested transaction.) If we reach here, we do the same thing we did when the user entered a 'q': **undo return.** The next scope we leave is that of the transaction. This transaction block has an **except** clause appended to it. An **except** clause allows access to the abort codes provided in **undo leave** statements. If a transaction with an except clause aborts, the abort code, if there is one, is assigned to the variable named after the **except.** The rest of the **except** statement is exactly like a **switch** on this value. In **jill_transaction,** the first two cases handle user-requested aborts. In either case, we print out av_jack.av:

```
#include <avalon.h>
#include <stream.h>
#include <ctype.h>
#include "av_jill.h"

// Abort codes.
const int USER_REQUESTED_ABORT = 100;
const int TOP_LEVEL_ABORT = 101;

// Forward declarations.
void jill_transaction(int, int*);
void print_help();

// Global server variable.
jill *jill_srv;

void main() {
  int quit_flag = 0;

  cout << "Looking for jill...\n";
  jill_srv = (jill*) &locate_server ("jill");
  if (jill_srv == NULL){
    cout << "Couldn't find jill.  Starting a new jill...\n";
    jill_srv = new jill ("av_jill", "localhost");
  }else cout << "Found jill\n";

  cout << "Type ? for a list of commands.\n";
  while (quit_flag < 2) {
    quit_flag = 0;
    jill_transaction(1, &quit_flag);
    cout << "(Transaction was top level.)\n";
  }
  exit(0);
}

// print_help -- Prints the commands.

void print_help() {
  cout << "\n\
Commands are: \n\
    r   Read array element.\n\
    w   Write array element.\n\
    b   Begin nested transaction.\n\
    c   Commit innermost transaction.\n\
    a   Abort innermost transaction.\n\
    A   Abort top level transaction.\n\
    q   Abort top level transaction and quit program.\n\n";
}
```

Figure 2-4: First Part of the Jack Application

a message and return. If a top-level abort has been requested, then we set the `quit_flag` to exit all enclosing `jill_transaction` calls. The third and fourth cases handle transactions that were aborted by server operations because of improper inputs. They both print an appropriate message and return from `jill_transaction`. Finally, if the transaction aborted but the code is none of the above, then the abort must have been caused by the underlying system. We can find out why by calling the routine `avalon_abort_code_to_string`, which takes an integer argument (Section 3.4.6). All arms of the `except` statement return from `jill_transaction`, so if we exit the transaction and reach the last line of the procedure, the transaction must have committed. We print a message to that effect.

```
// Interactively construct and perform a transaction utilizing the jill
// server. Can be called recursively to construct nested transactions.

void jill_transaction(int level, int* quit_flag_ptr) {
  start transaction {
    char cmd;

    while (!*quit_flag_ptr) {
      int index = 0;
      int value = 0;

      cout << "Jack[" << level << "] ";
      while(isspace(cmd = getchar()));

      switch(cmd) {
       case 'r':                  // Read an array element
        cout << "Location to read: ";
        cin >> index;
        value = jill_srv.read(index);
        if (value == -1)
          cout << "Location " << index << " is uninitialized.\n";
        else
          cout << "Value at location " << index << " is " << value << ".\n";
        break;

       case 'w':                  // Write an array element
        cout << "Location to write: ";
        cin >> index;
        cout << "Value to write: ";
        cin >> value;
        jill_srv.write(index, value);
        cout << "Write succeeded.\n";
        break;

       case 'c':                  // Commit this transaction
        leave;

       case 'a':                  // Abort this transaction
        undo (USER_REQUESTED_ABORT) leave;

       case 'A':                  // Abort top-level transaction
        undo (TOP_LEVEL_ABORT) leave;

       case 'q':                  // Abort to top level transaction and quit.
        *quit_flag_ptr = 2;
        undo return;

       case 'b':                  // Begin a subtransaction
        jill_transaction(level+1, quit_flag_ptr);
        continue;

       case '?':                  // Print short help message
        print_help();
        break;

       default:
        cout << "Unknown command. Type ? for a list of commands.\n";
      }
    } // ...continued...
```

Figure 2-5: Beginning of the jill_transaction Function

```
    // ...rest of jill_transaction...
    // Quit_flag from nested transaction is non-zero, so we must undo return.
    undo return;
  } except (trans_status) {
   case TOP_LEVEL_ABORT:
    *quit_flag_ptr = 1;
   case USER_REQUESTED_ABORT:
    cout << "Transaction aborted as per request.\n";
    return;
   case INDEX_OUT_OF_BOUNDS:
    cout << "Transaction aborted: Array index out of bounds.\n";
    return;
   case ILLEGAL_VALUE:
    cout << "Transaction aborted:  Attempt to write a negative value.\n";
    return;
   default:
    cout << avalon_abort_code_to_string(trans_status) << "\n";
    return;
  }
  // Otherwise, we committed.
  cout << "Transaction committed.\n";
}
```

Figure 2-6: End of the jill_transaction Function

2.2. FIFO Queue

Let us consider how one would implement an *atomic first-in-first-out (FIFO) queue*. The easiest way to define such a queue is to inherit from **atomic**. A limitation of this approach is that **enq** and **deq** operations would both be classified as writers, permitting little concurrency. Instead, we show how a highly concurrent atomic FIFO queue can be implemented by inheriting from **subatomic**. Our implementation is interesting for two reasons. First, it supports more concurrency than commutativity-based concurrency control schemes such as two-phase locking. For example, it permits concurrent **enq** operations, even though **enq**'s do not commute. Second, it 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 non-empty.

In order to permit such concurrency it is necessary to provide:

1. A way to compare whether one transaction has committed with respect to another. In particular, suppose A and B are concurrent transactions:
   - If it is known that A has committed with respect to transaction B, then B should be allowed to observe the effects of A's operations. Thus, B need not wait and may proceed.
   - If it is not known that A has committed with respect to B, then B must not do anything that depends on A's effects, since A may still commit or abort. B should also not invalidate any results that A may have observed, since B may commit before A. Thus, B might have to wait till A completes.
2. Exclusive access to an object per operation. That is, while transactions may go on concurrently, we need to prevent individual operations from interfering with each other.

Fortunately, Avalon provides the first capability with the class **trans_id**, which gives us a way to test transaction-commit order, and the second with the class **subatomic**, which gives us a way to provide mutual exclusion per object.

In Avalon when a transaction commits, the run-time system assigns it a timestamp generated by a logical clock [8]. Atomic objects are expected to ensure that all transactions are serializable in the order of their commit timestamps, a property called *hybrid atomicity* [15]. This property is automatically ensured by two-phase locking protocols [3], such as that used for the atomic_int's in Jill's array. However, additional concurrency can be achieved by taking the timestamp ordering explicitly into account. The **trans_id** class provides operations that permit run-time testing of transaction-commit order, and thus of serialization order. In particular, **trans_id** provides a partial-ordering function <: for transactions with trans_id's t1 and t2, if t1 < t2 evaluates to true, then if both transactions commit, t1 is serialized before t2. Note that < induces a partial order on trans_id's; as transactions commit they become comparable. Section 4.1.2 describes this type in more detail.

Class **subatomic** provides operations that give transactions exclusive access to objects. Each subatomic object has a short-term lock, similar to a monitor lock, used to ensure that concurrent operations do not interfere. Avalon's special control construct, the **when** statement, is used as a kind of conditional critical region:

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

The calling process atomically acquires the object's short-term lock, blocks until the condition becomes true (releasing the lock if it is not), and then executes the body. The lock is released after the body is executed. Any changes made to the object while the lock is held will not be backed up to stable storage until sometime after the lock is released. A transaction's changes are guaranteed to be backed up before it commits.

2.2.1. The Queue Representation

Figure 2-7 shows that information about **enq** invocations is recorded in a **struct**. 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 in **deq_rec**'s.

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

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

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

Figure 2-7: Queue Representation

The queue is represented as follows: The **deqd** component is a stack of **deq_rec**'s used to undo aborted **deq** operations. The **enqd** component is a *partially ordered heap* of **enq_rec**'s, ordered by their **enq_tid** fields. A partially ordered heap provides operations to enqueue an **enq_rec**, to test whether there exists a unique oldest **enq_rec**, to dequeue it if it exists, and to keep and discard all **enq_rec**'s committed with respect to a particular transaction identifier.

Our implementation satisfies the following representation invariant: First, assuming all enqueued items are distinct, an item is either "enqueued' or "dequeued," but not both: if an **enq_rec** containing **[item, enqr]** is in the **enqd** component, then there is no **deq_rec** containing **[item, enqr, deqr]** in the **deqd** component, and vice-versa. Second, 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**. Finally, any dequeued item must previously have been enqueued: for all **deq_rec**'s **d, d->enqr < d->deqr**.

2.2.2. The Queue Operations

Enq and **deq** operations (Figure 2-8) may proceed under the following conditions: A transaction A may dequeue an item if (1) the most recent dequeuing transaction is committed with respect to A, and (2) there exists a unique oldest element in the queue whose enqueuing transaction is committed with respect to A. The first condition ensures that A will not have dequeued the wrong item if the earlier dequeuer aborts, and the second condition ensures that there is something element for A to dequeue. Similarly, A may enqueue an item if the last item dequeued was enqueued by a transaction B committed with respect to A. This condition ensures that A will not be serialized before B, violating the FIFO ordering.

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

int atomic_int_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 2-8: Queue Operations

Both **enq** and **deq** first obtain a new, unique trans_id for the calling transaction. The constructor creates and commits a "dummy" subtransaction, returning the subtransaction's trans_id to the calling transaction (i.e., parent). Since this constructor call returns a unique trans_id, a parent transaction can thus generate multiple trans_id's ordered in the serialization order of their creation events. We exploit this property here by using this trans_id to tag the current **enq** (**deq**) operation.

As for the atomic_int example, the modifications done by **enq** and **deq** must be wrapped in a **pinning** construct to ensure persistence (that is, changes are made to stable storage).

We use the **when** statement to guard against simultaneous access to the queue object itself. **Enq** checks whether the item most recently dequeued was enqueued by a transaction committed with respect to the caller. If so, the new 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.

2.2.3. Commit and Abort

Avalon lets programmers define type-specific **commit** and **abort** operations for atomic data types inheriting from class **subatomic**. They each take a **trans_id** as an argument. The Avalon run-time system automatically calls an object's **abort** operation whenever a transaction that may have modified the object aborts. Whenever a top-level transaction commits, the system calls the **commit** operation on all subatomic (and atomic) objects that the transaction (or any of its descendants) may have modified. We make no guarantee about the arrival times of commit operations, i.e., when the run-time system is informed of a transaction's commit. In particular, if T1 commits before T2, the run-time might execute T2's commit before T1's. In addition, the order in which commit (abort) operations for a given transaction are applied to multiple objects is left unspecified.

Figure 2-9 gives the code for the queue's **commit** and **abort** operations. When a top-level transaction commits, it discards **deq_rec**'s no longer needed for recovery. The representation invariant ensures that all **deq_rec**'s below the top are also superfluous (they have all committed with respect to the top), and can be discarded. **Abort** has more work to do. It undoes every operation executed by a transaction committed with respect to 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_int_queue::commit(trans_id& committer) {
  when (TRUE)
    if (!deqd.is_empty() && descendant(deqd.top()->deqr, committer)) {
      deqd.clear();
    }
}

void atomic_int_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 2-9: Queue's Commit and Abort

Notice that **commit** and **abort** for the queue example use the **descendant** operation of **trans_id**'s rather than the < operation. For example, when we are aborting, we want to remove all items enqueued by transactions that we know are aborting, i.e., the aborting transaction (**abort**'s argument) and all of its descendants. If we were to use <, an item enqueued by a separate top-level transaction that committed before the aborting transaction would be incorrectly deleted.

2.2.4. Enq and Deq Synchronization Revisited

Let us look more carefully at the synchronization conditions on **enq** and **deq**. Consider why **enq** must wait for the enqueuer of the last dequeued item to commit. If it does not wait, then it is possible that a dequeuer may get the wrong head of the queue as a result of the commit of some concurrent enqueue. For example, suppose a transaction A starts two subtransactions A1 and A2. A1 enqueues 5 and commits. A2 does a dequeue (A2 can proceed because A1 has committed with respect to A2), gets a 5, but does not yet commit. Now suppose another top-level transaction B starts and tries to enqueue 7. (B and A2 are both concurrent.) If B does not wait then it proceeds to put 7 at the head of the queue (A2 has temporarily claimed the 5). If B commits before A (the parent transaction of A1 and A2), then B is serialized before A, implying that A2 should get a 7, not a 5. In short, the FIFO behavior of the queue is violated because B did not wait for A to commit.

The condition on **enq** is sufficient as well. In particular, an enqueuing transaction does not need to wait for the dequeuer of the last dequeued item to commit because in some circumstances it can proceed even if the dequeuer has not finished. For example, suppose transactions A, B, and C are top-level transactions. A enqueues 5 and commits. B dequeues 5, but remains active. If C wants to enqueue, it should be allowed to proceed even though B (the dequeuer of the last dequeued item) has not completed. Here, if B commits, it does not matter whether B commits before or after C; B will correctly see 5 as the head of the queue and C will correctly place 7 as the new head. If B aborts, then C will correctly place 7 after 5, which remains at the head of the queue. Thus, C can proceed without waiting for B to complete because there is no way C can be serialized before A and it does not matter in which order B and C are serialized.

It is easier to see why a dequeueing transaction, B, must wait for the dequeuer, A, of the last dequeued item to be committed with respect B. If B proceeds to dequeue without waiting for A to complete, then it will have dequeued the wrong item if A aborts.

2.3. Atomic Counters

As our final example, suppose we wish to implement an *atomic counter* with operations to increment (**inc**), decrement (**dec**), and test for zero (**is_zero**). This counter could be used to represent a joint checking account: One party might be depositing money at one branch, another party may be withdrawing money from somewhere else, and a third party, perhaps an auditor, may be searching for depleted accounts. This is not quite realistic since one could not find out the exact balance of the account (there is no read operation), but adding that function would complicate our example.

By deriving from class **atomic**, we can easily implement the atomic counter as shown in Figure 2-10. (Recall that class **atomic** provides **read_lock** and **write_lock** operations.) The counter is represented by a **nonnegative_int**, a class supporting all the usual arithmetic operations on integers, with the property that a non-negative integer can have a value only greater than or equal to zero. (The overloaded subtraction operation is a "monus" operation.) Again, one can see that building a new atomic class from class **atomic** is fairly straightforward: Before performing its real work, an accessing operation ("reader") should first obtain a read lock; a modifying operation ("writers") should first obtain a write lock and then pin the object.

This implementation, however, does not realize the greatest possible concurrency. From the abstract viewpoint of our atomic counter, incrementing and decrementing transactions can go on concurrently (**inc** and **dec** are "blind" writes since they do not return any results); moreover, under certain conditions, it should be possible to return a result to **is_zero** even before all incrementing and decrementing transactions have completed. The implementation in Figure 2-10 does not support this degree of concurrency since it is based on standard two-phase read/write locking.

```
class atomic_counter: public atomic {
  nonnegative_int count;
 public:
  atomic_counter()  {pinning() count = 0;} // initialize counter
  void inc();
  void dec();
  bool is_zero();
}

void atomic_counter:: inc() {
  write_lock();
  pinning () count += 1;
}

void atomic_counter:: dec() {
  write_lock();
  pinning () count -= 1;                // will return max of count-1 and 0
}

bool is_zero(); {
  read_lock();
  return (count == 0);
}
```

Figure 2-10: Atomic Counter Derived from Class Atomic

Thus, as for the queue example, we will use trans_id's and subatomic objects as an alternative way to build atomic objects.

2.3.1. Counter Representation

Let us walk through the representation of the atomic counter by beginning with some auxiliary structures shown in Figure 2-11. A **counter_range** will keep track of the range of possible values of the counter in order to permit **is_zero** to return possibly before transactions have completed. We will record in a log information about each transaction's sequence (**op_seq**) of **inc** and **dec** operations. Each **log_entry** consists of a transaction's trans_id and the sequence of its operations. Assume we have defined elsewhere (**recov_sorted_alist.h**) types for a recoverable sorted association list (**recov_sorted_alist**), parameterized over the tag type (e.g., trans_id) and value type (e.g., pointer to log_entry's) of the pairs to be inserted in the list, an equality function (e.g., on trans_id's) used for list insertion, lookup, and removal, and a comparison function (e.g., < on trans_id's) used for ordering the elements in the list. Its iterative version, (**recov_sorted_alist_ittr**), similar to that used in the C++ Manual (p. 183 of [14]), provides a method for looping over all elements in the list, guaranteeing that elements are yielded in sorted order. Our (recoverable sorted association) list will be sorted by trans_id's partial order < so that we can iterate over transactions in commit-time order.

Finally, we represent the counter by a non-negative integer (**count**) and a transaction log (**log_t**) (Figure 2-12). The value of the non-negative integer will be determined by operations of only top-level committing transactions.

2.3.2. Counter Operations

Implementations of the **inc** and **dec** operations are shown in Figure 2-13. They use the internal auxiliary functions shown in Figure 2-14. **Inc** and **dec** attempt to record themselves in the log. **Add_op_to_log** first calls the **trans_id** constructor with the value CURRENT to obtain the trans_id of the calling transaction (compare this to a different call with no argument in the **enq** operation of Figure 2-8). If the addition of the operation would not

```
#include <nonnegative_int.h>

struct counter_range {
  nonnegative_int lo;
  nonnegative_int hi;
  counter_range(counter_range&, op_seq*);
  counter_range(counter_range& cr)      { lo = cr.lo, hi = cr.hi; }
  counter_range(int l, int h)           { lo = l; hi = h; }
  counter_range(int i)                  { lo = hi = i; }
  counter_range()                       { init(); }
  void init()                           { lo = 1; hi = 0; }
  bool unset()                          { return (lo && !hi); }

  counter_range& operator+= (int i)     { lo = lo + i; hi = hi + i;
                                          return *this; }
  counter_range& operator= (counter_range& cr)
    { lo = cr.lo; hi = cr.hi; return *this; }

};


struct op_seq : public recoverable {
  bool to_inc;
  op_seq* ops;

  op_seq(bool b);
  ~op_seq()                             { delete ops; }
  op_seq& operator<<(op_seq*);
};


struct log_entry :public recoverable {
  trans_id common_id;
  counter_range query_range;
  op_seq* ops;

  log_entry(trans_id&);
  log_entry(trans_id&, bool);
  log_entry(trans_id&, counter_range&);

  ~log_entry()                          { delete ops; }

  bool operator<(log_entry& le)         { return (common_id < le.common_id); }
  bool operator>(log_entry& le)         { return (common_id > le.common_id); }
  log_entry& operator=(log_entry& le);

};


// Load recoverable list from library
#include "recov_sorted_alist.h"
recov_sorted_alistdeclare(trans_id,Plog_entry,tid_eq,tid_lt);
recov_sorted_alistittrdecl(trans_id,Plog_entry,tid_eq,tid_lt);

typedef recov_sorted_alist(trans_id,Plog_entry,tid_eq,tid_lt) log_t;
typedef recov_sorted_alist_ittr(trans_id,Plog_entry,tid_eq,tid_lt) logittr;
```

Figure 2-11: Auxiliary Structures for Counter

```
class atomic_counter : public subatomic {
  nonnegative_int count;
  log_t log;
  // internal functions
  counter_range* is_zero_work(trans_id&);
  bool is_zero_in_range(counter_range&);
  bool is_zero_value(counter_range&);
  void add_op_to_log(bool);
  bool add_op_to_log_work(trans_id&, bool, log_entry*&);
public:
  void inc();
  void dec();
  bool is_zero();
  atomic_counter()                        { count = 0; }
  void commit(trans_id& t);
  void abort(trans_id& t);
};
```

Figure 2-12: Atomic Counter Derived from Class Subatomic

```
// Add increment operation to log
void atomic_counter::inc()        { add_op_to_log(TRUE); }

// Add decrement operation to log
void atomic_counter::dec()        { add_op_to_log(FALSE); }
```

Figure 2-13: Counter's Inc and Dec Operations

change the possible view of the counter as seen by other active transactions, the operation proceeds. Otherwise, the operation is forced to wait until all interfering transactions terminate (by either committing or aborting).

An example of a blocked case is as follows: Assume a transaction tests for the zero state of the counter and receives a positive (i.e., TRUE) result. Until that transaction commits (or aborts) no other transaction can increment the counter, since that would change its state from zero to non-zero. Other transactions are free to decrement the counter, however, as this does not alter the visible state of the counter.

The **add_op_to_log** routine uses a **when** construct to ensure exclusive access to the log during the operation insertion. Prior to that, however, it verifies that the insertion of the operation record is possible by calling **add_op_to_log_work**, which examines the counter from views by all active transactions whose entries are present in the log. The **add_op_to_log_work** returns FALSE if the operation cannot be added at this time, causing the when construct to pause and be reactivated at a later time when the situation changes. When the condition in the **when** statement succeeds, **add_op_to_log** adds the operation to an existing log record (indexed by the current trans_id) if possible, creating a new record otherwise.

Much of the work for the **is_zero** predicate (Figure 2-15) is done by the auxiliary function **is_zero_work** (Figure 2-16), which constructs a range of possible values for the counter, given the committed value and the log. **Is_zero** first obtains the trans_id of the calling transaction. Then **is_zero_work** iterates over all log entries, constructing the range of counter values. For each log entry, the logged operations are added together to determine what the net effect of committing the transaction represented by the trans_id would be. Then, the net value is added to the high bound or subtracted from the low bound, as appropriate. Operations of uncommitted transactions enlarge the range of possible values. If the low end of the range is bounded below by a positive integer, **is_zero** returns -1. If the range starts and ends at zero, then it returns 1. In all other cases (the range starts at zero and ends at a non-zero integer), it returns 0.

```
// Add (inc/dec) operation to log, by adding it to operation sequence (op_seq) of existing
// log record, or by making a new one.  Log entries are keyed by current trans_ids.
void atomic_counter::add_op_to_log(bool b) {
  trans_id current_id = trans_id(CURRENT);
  log_entry* entry = NULL;
  bool new_entry_needed = FALSE;

  when (add_op_to_log_work(current_id, b, entry)) {
    if (entry == NULL) {
      entry = new log_entry(current_id,b);
      log.insert(current_id, entry);
    } else {
      if (entry->ops) *(entry->ops) << new op_seq(b);
      else pinning (entry) entry->ops = new op_seq(b);
    }
  }
}

inline bool atomic_counter::is_zero_in_range(counter_range range)
{ return ((int) range.lo <= 0); }

inline bool atomic_counter::is_zero_value(counter_range& range)
{ return ((int) range.hi == 0); }

bool atomic_counter::add_op_to_log_work(trans_id& id, bool to_inc, log_entry*& this_entry) {
  log_entry** entry;
  log_entry** found_entry;
  logittr next_entry(log);

  found_entry = log.lookup(id);
  if (found_entry == NULL) {
    this_entry = NULL;
  } else {
    this_entry = *found_entry;
  }

  for (entry = next_entry(); entry; entry = next_entry()) {
    if ((*entry)->common_id == id) { // We've already seen this guy, so ignore it
    } else if (((*entry)->common_id < id) ||
               (descendant(id, (*entry)->common_id))) {
      // committed wrt to me, so not a problem

    } else { // uncommitted, the tough one.  Must ensure there is no active transaction
      // whose termination state (commit or abort) could change the visible state
      // (zero or nonzero) of the counter.
      counter_range old_range = (*entry)->query_range;
      if (!old_range.unset()) {
        counter_range range((*entry)->query_range,
                            (found_entry) ? (*found_entry)->ops : NULL);
        counter_range new_range =
          counter_range(range.lo - ((to_inc) ? 0 : 1),
                        range.hi + ((to_inc) ? 1 : 0));
        if ((is_zero_value(old_range)    != is_zero_value(new_range)) ||
            (is_zero_in_range(old_range) != is_zero_in_range(new_range)))
          {
            return FALSE;                              // bad news
          }
      }
    }
  }
  return TRUE;
}
```

Figure 2-14: Counter's Inc and Dec Auxiliary Operations

```
// Public is_zero() predicate
bool atomic_counter::is_zero() {
  counter_range* result;
  trans_id current_id = trans_id(CURRENT);

  when (result = is_zero_work(current_id)) {
    log_entry** entry = log.lookup(current_id);
    if (entry) {
      pinning(*entry) (*entry)->query_range = *result;
    } else
      log.insert(current_id, new log_entry(current_id, *result));

    if (is_zero_in_range(*result))
      return TRUE;
    else
      return FALSE;
  }
}
```

Figure 2-15: Counter's Is_zero Operation

The predicate **is_zero** uses the result (-1, 1, or 0) of **is_zero_work** to determine whether it can return immediately (cases -1 or 1) or not. If it cannot, it exits the **when** block (thereby releasing the short-term lock), waits for more transactions to commit (or abort), and tries again later. This process repeats indefinitely, until one of the two cases for returning from **is_zero** holds.

2.3.3. Counter's Commit and Abort

The **commit** and **abort** operations (Figure 2-17) must clean up the log. The **commit** operation additionally updates the value of the counter by going through the log, finding all the entries for transactions committed with respect to it, applying these in serialization order, and then applying its own operations. Log entries for transactions relatively committed to the committing transaction can be discarded. Notice that we need to use the < operation because we cannot assume anything about the order in which commit operations are executed. Suppose A and B are transactions and the committed value before either transaction commits is 2. Suppose A does 1 **inc** and then 5 **dec**'s; B does 3 **inc**'s. If A commits, followed by B, the counter's committed value after A's **commit** operation is executed should be 0 (a **dec** has no effect on the counter if its value is 0 already); then after B commits, the counter's value changes to 3. However, if we were to execute B's **commit** operation before A's, then B would update the counter to 5, and A would change it to a final value of 1, which is wrong. By using <, the **commit** operator makes sure it installs all changes of transactions that have committed with respect to the committing transaction, not just its descendants.

On the other hand, the **abort** operation throws away only transactions that are **descendants** of the aborting transaction; it would be incorrect to throw away transactions that are not descendants but have relatively committed with respect to the aborter.

```
// Returns the range of possible counter values as seen by the trans_id.
// Committed transactions operate directly on counter value,
// while (as yet) uncommitted ones increase range.
counter_range* atomic_counter::is_zero_work(trans_id& id) {
  log_entry** entry;
  op_seq* op;
  logittr next_entry(log);
 // Begin with committed value, and a sequence of op_seqs of uncommitted operations.
  nonnegative_int committed_value(count);
  struct op_seq_seq {op_seq* ops; op_seq_seq* next;};
  op_seq_seq* uncommitted_op_seqs = new op_seq_seq;
  op_seq_seq* last_uos = uncommitted_op_seqs;

  for (entry = next_entry(); entry; entry = next_entry()) {
    if ((id == (*entry)->common_id) ||
        (descendant(id, (*entry)->common_id)) ||
        (id > (*entry)->common_id)) {
      // Install relatively-committed operations
      for (op = (*entry)->ops; op; op = op->ops)
        committed_value += ((op->to_inc) ? 1 : -1);
    } else { // Cache uncommitted operation until all committed one are "in"
      if (last_uos != uncommitted_op_seqs) { // not first one
        last_uos->next = new op_seq_seq;
        last_uos = last_uos->next;
      }
      last_uos->ops = (*entry)->ops;
    }
  }

  // Delete uncommitted_op_seqs altogether if we haven''t seen any as yet
   if (uncommitted_op_seqs->ops == NULL) {
     delete uncommitted_op_seqs;
     uncommitted_op_seqs = NULL;
   }

  // Now, we can go through all (if any) uncommitted op_seqs.  Build range of
  // possible values from other operations "adding" it to range as appropriate.
   counter_range *range = new counter_range(committed_value);
   last_uos = uncommitted_op_seqs;
   while (last_uos) {
     counter_range possible_range(*range, last_uos->ops);
     range->hi = max(range->hi, possible_range.hi);
     range->lo = min(range->lo, possible_range.lo);
     op_seq_seq* temp = last_uos; last_uos = last_uos->next; delete temp;
   }

  // There are only two ways to produce a "result".  If the range does not
  // include zero, then we can safely return FALSE.  If the range includes a
  // single value, then we can determine with certainty whether it is zero.
  // If neither condition is attained, we pause (releasing the short-term lock
  // for another transaction), and then try the loop again.
   { int result;
     if (is_zero_in_range(*range))
       if (is_zero_value(*range)) result = 1;
       else result = 0;
     else
       result = -1;
     if (result != 0)
       return range;
     else
       return NULL;
    }
  }
```

Figure 2-16: Counter's Is_zero_work Operation

```
// Install (and remove) all descendants from log. They are all committed,
// by definition, since aborted ones have been previously deleted by the log.
// (See abort routine below.)

void atomic_counter::commit(trans_id& t) {
  log_entry** entry;
  op_seq* op;

  when(TRUE)
    pinning() {
      logittr next_entry(log);
      for (entry = next_entry(); entry; entry = next_entry()) {
        if (((*entry)->common_id == t) || ((*entry)->common_id < t)) {
          for (op = (*entry)->ops; op; op = op->ops)
            count += ((op->to_inc) ? 1 : -1);
          log.remove((*entry)->common_id);
          delete *entry;
        }
      }
    }
}

// Remove all descendants from log

void atomic_counter::abort(trans_id& t) {
  log_entry** entry;

  when(TRUE)
    pinning() {
      logittr next_entry(log);
      for (entry = next_entry(); entry; entry = next_entry()) {
        if (((*entry)->common_id == t) || descendant((*entry)->common_id, t)) {
          log.remove((*entry)->common_id);
          delete *entry;
        }
      }
    }
}
```

Figure 2-17: Counter's Commit and Abort

3. Reference Manual

3.1. Lexical Considerations

Avalon nonterminals are in roman face. C++ nonterminals are in italics, as in Section 14 of the C++ Reference Manual in [14]. Keywords are in bold typeface. C++ extended BNF is used. E.g., symbol_{opt} means an optional *symbol*. A C++ nonterminal followed by ":..." denotes an extension to that nonterminal.

The extended set of keywords is as follows:

costart
except
leave
pinning
process
server
stable
start
toplevel
transaction
undo
variant
when
whenswitch

3.2. Servers

aggr: ...
  server

decl-specifier: ...
  server-specifier

server-specifier:
  class-specifier

sc-specifier: ...
  stable

An Avalon server object is an instance of a **server** definition. A server definition, like a C++ class definition, encapsulates a set of objects, and exports to clients a set of operations that manipulate the objects and a set of operations that create and destroy instances of servers. A client invokes an operation on a server by calling a member function of a server object. Creating a new instance of a server causes a new server process to be started. When a server object is deleted, the server is killed.

3.2.1. Defining Servers

A server definition contains the following parts:

- Data declarations: Data declared to be **stable** in the server are restored following a failure. To be restored properly, **stable** data must be derived from one of Avalon's three base classes (Section 3.3): **recoverable**, **atomic** or **subatomic**. All data must be implemented to control concurrent access.
- A mandatory **main**. The **main** member function is executed as a background process when the server is started. This function can be used to provide code which needs to be run independently of the server's other operations. A printer server, for example, could use **main** for the code to run the printer. **Main** must exist, even if empty, because Avalon uses the existence of a **main** implementation to determine that the current compilation is for a server, rather than just for a client.
- An optional **recover** operation, which is executed whenever the server is restarted after a failure.
- Exported (and possibly internal) operations: The exported operations provide the clients the only means of accessing the server's data. Communication between clients and servers is through (hidden) remote procedure call with call-by-value transmission of data.
- A nonempty set of constructors: A server's constructor defines the parameters that a client must use when creating a new server and provides code to execute when the server is started. In contrast to constuctors for classes, a server's constructor must also specify to the run-time system the parameters

needed to start the server process; these parameters are specified in the declaration in a way similar to passing parameters to the constructor of a class's parent (see example below). When a client calls a server's constructor, the specified parameters are passed to the routines that start the server.[1]

- An optional destructor.

Example

Below is a simple server declaration:

```
server simple {
  stable atomic_int val;                         // Protected atomic integer
 public:
  simple(x_string p, x_string n) : (p,n);        // Constructor
  int  get();                                    // An exported operation
  void set(int i);                               // Another exported operation
  void recover();                                // Called upon server recovery
  void main();                                   // Background process
};
```

The parameters to the right of the colon in the constructor are passed to the run-time routines that start the server. The first parameter is the name of an executable file; if the full path name is not given, the user's path is used. The second parameter is the name of a node on which to start the server; If the value "localhost" or NULL is given, local machine is used; otherwise an x_string argument such as "wing.avalon.cs.cmu.edu" can be given to start the server on some remote machine.

3.2.2. Using Servers

For an Avalon program to make use of a server it must first obtain a reference to an instance of the appropriate server. As shown below the client may either create a new server object, starting a new server process:

```
(1)  printserver& p = *(new printserver(...));  // Start a new printserver
(2)  printserver p (...);                       // Start a temporary server
```

or it may, with the Avalon library **locate_server** function (see end of Section 4.3), obtain a reference to an existing server object representing a running server process:

```
(3)  printserver& p = (printserver&) locate_server(...); // Locate an existing printserver
```

Deleting a server object kills the associated server process. A declaration local to some block (as in (2) above) of a server object starts the server upon entering the block and kills it upon exit. Thus, (1) is recommended for ordinary starting of servers.

Once a server instance is found, operations are invoked on the object as for any C++ object:

```
p.spool ("myfile.txt");                         // Invoke an operation
```

Since server objects are really just C++ objects with special operations, they can be manipulated in the same manner as other C++ objects. In particular, server objects and references to servers can be passed as parameters to and returned as values from functions.

3.3. Base Classes

There are three base Avalon classes: **recoverable**, **atomic**, and **subatomic**. Users define their own recoverable types by deriving from **recoverable**. They define their own atomic types by deriving from **atomic** or **subatomic**, and are responsible for ensuring that the types they define are indeed atomic. If a type is not

[1] Unlike normal C++ usage, the startup parameters must be in the declaration seen by the client, rather than with the constructor definition. This information is useful only to the client, so it must appear in a place visible to the client, such as the server declaration.

atomic then transactions that use objects of that type are not guaranteed to be atomic. We expect most users to derive from class **atomic**, and more experienced (and demanding) users to derive from **subatomic**, especially if more control over the object's synchronization and recovery is desired. We refer the reader to Chapter 5, in particular Section 5.2, for correct usage of base classes, and [16] for a more formal description of their interfaces.

3.3.1. Class Recoverable

Class Definition

```
class recoverable {
 public:
   virtual void pin(int size);
   virtual void unpin(int size);
};
```

Operations

void pin(**int** size)

ensures Subsequent changes to the object will not be recorded to stable storage until a later matching unpin operation. Multiple pins (and their matching unpins) by the same transaction to the same object have no effect. If the object is already pinned by a transaction different from the calling transaction, a run-time error is signaled.

void unpin(**int** size)

modifies The value of the object in stable storage.
requires The calling transaction is currently pinning the object.
ensures If there is exactly one outstanding pin operation, the modifications to the object are logged to stable storage.

The **pin** and **unpin** operations, which should be called in pairs, are used to notify the run-time system that a modification to an object is to be made. In most cases, the integer argument to pin and unpin should be the size of the object being pinned. After a crash, a recoverable object will be restored to a previous state in which it was not pinned. The **pin** and **unpin** operations are usually not called explicitly by programmers; instead, Avalon/C++ provides a special control structure, the **pinning** block (Section 3.4.7), both for syntactic convenience and as a safety measure.

3.3.2. Class Atomic

Atomic is a subclass of **recoverable**, specialized to provide two-phase read/write locking and automatic recovery. Objects derived from class **atomic** should be thought of as containing *long-term locks*, used to ensure serializability. Each transaction obtains read (write) locks on all objects it accesses (modifies); locks are held until the transaction commits or aborts.

Class Definition

```
class atomic: public recoverable {
 public:
   // pin and unpin are inherited from recoverable.

   virtual void read_lock();
   virtual void write_lock();
}
```

Operations

void read_lock()

when No transaction other than the calling transaction has a write lock on the object.

ensures If the calling transaction already has a read lock on the object, there is no effect; otherwise, it obtains a read lock on the object. Many transactions may simultaneously hold read locks on the same object.

void write_lock()

when No transaction other than the calling transaction has a read or write lock on the object.

ensures If the calling transaction already has a write lock on the object, there is no effect; otherwise it obtains a write lock on the object, preventing other transactions from gaining any kind of lock on it.

Read_lock and **write_lock** suspend the calling transaction until the requested lock can be granted (i.e., when the **when** condition holds); this may involve waiting for other transactions to complete and release their locks.

The run-time system guarantees that for nested transactions, the following rules are obeyed in obtaining read and write locks:

- A child can get a read lock if all transactions holding write locks are ancestors.
- A child can get a write lock if all transactions holding read or write locks are ancestors.
- When a child commits, locks are inherited by parents.
- When a child aborts, locks are discarded.

The run-time system guarantees transaction-consistency of atomic objects, by performing special abort processing that "undoes" the effects of aborted transactions, including those aborted by crashes. Thus, implementors of atomic types derived from **atomic** need not provide explicit commit or abort operations. Finally, persistence is "inherited" from class **recoverable**; its **pin** and **unpin** operations should be used in the same way as described in Section 3.3.1.

3.3.3. Class Subatomic

Like **atomic**, **subatomic** provides the means for objects of its derived classes to ensure atomicity. While **atomic** provides a quick and convenient way to define new atomic objects, **subatomic** provides primitives to give programmers more detailed control over their objects' synchronization and recovery mechanisms. This control can be used to exploit type-specific properties of objects to permit higher levels of concurrency and more efficient recovery. A subatomic object must synchronize concurrent accesses at two levels: *short-term* synchronization to ensure that concurrently invoked operations are executed in mutual exclusion, and *long-term* synchronization to ensure that the effects of transactions are serializable. For short-term synchronization, each object derived from class **subatomic** should be thought of as containing a *short-term lock*, much like a monitor lock.

Class Definition

```
class subatomic: public recoverable {
 protected:
   void seize();
   void release();
   void pause();
 public:
   // pin and unpin are public, by inheritance from recoverable.

   virtual  void commit(trans_id& tid);
   virtual  void abort(trans_tid& tid);
}
```

Operations

`void` seize()

when No transaction holds the short-term lock on the object.
ensures The calling transaction obtains the short-term lock on the object.

`void` release()

requires The calling transaction holds the short-term lock.
ensures The calling transaction relinquishes the short-term lock.

`void` pause()

requires The calling transaction holds the short-term lock.
ensures The calling transaction releases the lock, waits for some duration, and reacquires the lock before returning.

The above operations ensure that only one transaction may hold the short-term lock at a time, thus allowing type implementors to ensure that transactions have mutually exclusive access to subatomic objects. 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. Like `pin` and `unpin`, the above operations are usually not called explicitly; instead, Avalon/C++ provides special control structures, the `when` and `whenswitch` statements (Section 3.4.8), which automatically seize, release, and pause on the short-term lock.

Since `commit` and `abort` are C++ virtual operations, classes derived from `subatomic` are allowed (and indeed, expected) to reimplement these operations. They each take a reference to a transaction identifier as an argument. (See the Avalon class `trans_id` of Section 4.1.2.) The typical effects of these operations are specified as follows:

`void` commit(trans_id& tid)

requires The transaction tid has committed.
ensures Non-idempotent undo information stored for transactions that have committed with respect to tid is discarded.

`void` abort(trans_id& tid)

requires The transaction tid has aborted.
ensures The effects of every transaction that has committed with respect to tid are undone.

Commit operations are called for only transactions that commit at the top-level. Whenever a top-level transaction commits (aborts), the Avalon run-time system calls the `commit` (`abort`) operation of all subatomic objects accessed by that transaction or its descendants. `Abort` operations are also called when nested transactions abort. When `commit` or `abort` is called by the system, 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 in implementing user-defined atomic types efficiently. Notice that users need not call `commit` and `abort` explicitly; the system automatically calls them when appropriate.

3.4. Control Structures

3.4.1. Start

```
statement: ...
    start trans-body
trans-body:
    trans-tag statement except-clause opt
trans-tag:
    toplevel
    transaction
```

Sequential transactions are created by means of a `start` statement. The `toplevel` qualifier causes the body of the **start** statement to execute as a new top-level (root) transaction. The **transaction** qualifier causes the body to execute as a subtransaction of the current transaction, if there is one; otherwise, it too begins a new top-level transaction. When the body terminates, the transaction either commits or aborts. Normal completion of the body results in a commit of the transaction. Control flow statements (**return**, **leave**, **break**, and **continue**) that transfer control outside the scope of the transaction normally commit it, unless they state otherwise via an **undo** qualifier (Sections 3.4.4, 3.4.3, 3.4.5). The **undo leave** statement can be used to pass an abort code that can be used as a switch value in an **except** clause (Section 3.4.6). **Goto** statements that transfer control outside a transaction are currently not supported. Future versions of Avalon will prohibit such transfers at compile-time; presently, the result of such a statement is undefined.

3.4.2. Costart

statement: ...
  costart { coarms }

coarms:
  coarm coarms$_{opt}$

coarm:
  trans-body

Concurrent transactions and processes are created by means of the **costart** statement. The process executing the **costart** is suspended; it resumes after the **costart** is finished. Execution of the **costart** consists of executing all the coarms concurrently. No guarantee is made about order of execution, or of initialization. Each coarm runs as a separate (lightweight) process. The **toplevel** or **transaction** qualifier indicates whether the coarm is a top-level transaction or subtransaction.

A coarm may terminate without terminating the entire **costart** either by normal completion of its body, or by executing a **leave** statement (Section 3.4.3). A coarm may also terminate by transferring control outside the **costart** statement. If an outside transfer occurs, the following steps take place:

1. All containing statements are terminated to the outermost level of the coarm, at which point the coarm becomes the *controlling* coarm.
2. Every other active coarm is terminated (and aborts if declared as a transaction). The controlling coarm is suspended until all other coarms terminate.
3. The controlling coarm commits or aborts.
4. The entire **costart** terminates. Control flow continues outside the **costart** statement.

3.4.3. Leave

statement: ...
  leave ;
  undo (*expression*)$_{opt}$ **leave** ;

Executing a **leave** statement terminates the (innermost) transaction that the **leave** occurs in. By itself. **leave** commits the transaction, but with the **undo** qualifier. it aborts it. An unqualified **leave** statement must occur textually within the scope of a transaction, or a compile-time error results. An **undo leave** statement need not occur within the textual scope of a transaction, but it it must occur within the dynamic scope of one, or a run-time error will occur. The optional integer *expression* in an **undo leave** statement can be used to pass a value that can be used in the **except** clause of the aborted transaction (see section 3.4.6.) The value of the expression must be greater than zero, but less than or equal to the constant **AVALON_SYS_USER_ABORT_MAX**[2], or a run-time error

[2] Currently equal to (2^{15})-1.

will result. If the expression can be evaluated at compile-time, this restriction will be enforced then.

3.4.4. Return

statement: ...
undo$_{\text{opt}}$ **return** *expression*$_{\text{opt}}$

The **return** statement terminates execution of the containing operation. If no **undo** qualifier is present, then all containing transactions (if any) terminated by this statement are committed. If the **undo** qualifier is present, then all terminated transactions are aborted. When a **return** statement in a coarm causes control to leave the **costart** statement, active sibling coarms are aborted. The **undo** qualifier can only be used within the lexical scope of a transaction, or a compile-time error will result.

3.4.5. Break and Continue

statement: ...
undo$_{\text{opt}}$ **break** ;
undo$_{\text{opt}}$ **continue** ;

Terminating a cycle of a loop (**while**, **do**, **for**), or a **switch** statement may also terminate one or more transactions within the loop or switch. If no **undo** qualifier is present, then all these terminated transactions (if any) are committed. If the **undo** qualifier is present, then all of the terminated transactions are aborted. When a **break** or **continue** in a coarm causes control to leave the **costart** statement, active sibling coarms are aborted. The **undo** qualifier can only be used within the lexical scope of a transaction, or a compile-time error will result.

3.4.6. Except Clauses

except-clause:
except (*identifier*)$_{\text{opt}}$ *statement*

An **except** clause, which may be appended to a transaction body, is used to handle different cases of an aborting transaction. After a transaction aborts, it allows some case-specific action to be taken. The *statement* in the clause is expected to be one or more **case** statements. If the transaction was aborted as a result of an **undo** (*expression*) **leave** statement, then the value of the integer *expression* (called the *abort code*) is used to determine which of the cases in *statement* are executed, just as in a **switch** statement. The Avalon run-time system may abort the transaction for a variety of other reasons; in this case, the abort code will be an integer greater then **AVALON_SYS_USER_ABORT_MAX**. If the optional *identifier* is present, then an integer variable of that name will be defined to have the value of the abort code within the scope of the **except** clause. The routine **avalon_abort_code_to_string** may be used to translate system abort codes to strings describing the reason for the abort:

char* avalon_abort_code_to_string(int ac)
ensures The returned string describes the reason for an underlying system-induced abort according to the integer abort code ac.

3.4.7. Pinning

statement: ...
pinning (*expression*$_{\text{opt}}$) *statement*

The **pinning** statement indicates that *statement* may modify *expression*. *Expression* must evaluate to be the address of a recoverable object (Section 3.3.1); if it is not provided, **this** will be used. All modifications to recoverable objects should be done within **pinning** statements. If a recoverable object is not "pinned" in memory while it is being modified, it may cease to be recoverable. If the object to be pinned is of variable size, then explicit calls to **pin** and **unpin** are necessary; the **pinning** statement cannot be used.

3.4.8. When

statement: ...
  when (*expression*) *statement*
  whenswitch (*expression*) *statement*

The **when** statement provides short-term synchronization for operations on **this**, which must be a subatomic object (Section 3.3.3). After a short-term lock on **this** is obtained, *expression* is evaluated; if true, *statement* is executed. If *expression* evaluates to false, execution pauses, temporarily relinquishing the lock, until it becomes true. The short-term lock is released after *statement* is executed.

The **when** statement can also be used to provide *operation consistency* of implementations of operations of subatomic objects. The operations done in a **when** statement are done atomically: either they all happen or none of them happen. If the implementation of a subatomic operation does all of its work in a **when** statement, operation consistency is guaranteed. **When**'s can be nested, but the use of more than one (non-nested) **when** statements in the implementation of an operation (e.g., two **when**'s in sequence) is strongly discouraged and will void this guarantee.

As its name suggests, the **whenswitch** statement is a combination of the **when** and **switch** statements. *Expression* and *statement* are handled just as they would be in a **switch** statement, with one difference: the **default** action is to pause execution until the value of *expression* equals the value of one of the **cases**. Since the **default** action is provided, it is illegal to include a **default** in *statement*.

3.5. Transmission of Data

Clients and servers communicate through remote procedure call. The arguments and return values of server member functions are passed by value. Hence, references and pointers to objects are not transmissible. Objects of any other C++ or Avalon fundamental type are transmissible. An array, struct, or variant (Section 4.1.4) is transmissible if and only if all its component types are transmissible. Unions cannot be transmitted, since their actual type is not known at compile time. The chart in Figure 3-1 summarizes which types are transmissible and which are not. Future releases of Avalon/C++ are likely to reduce the restrictions on transmissible types, and allow pointer indirection in structures to be transmitted (by copying) between server and client.

In most cases, users can rely on the Avalon/C++ compiler to determine automatically how to transmit a value as an argument to a server function. In the cases where the compiler fails to recognize a type as transmissible, or when the automatically generated transmission functions are inefficient, the user can define his or her own transmission functions as part of the class definition. Section 5.4.2 explains how this can be accomplished, and should be read on a need-to-know basis only.

Types	***Transmissible***	***Non-Transmissible***
C++ Fundamental	int, short int, *long int, unsigned int,* char, float, double, enum	pointers, references
Avalon/C++ Fundamental	bool, trans_id, x_string (character strings)	
C++ Derived (!)	servers, arrays, variants, classes (-), structs (-)	unions, functions, classes (+), structs (+)
Avalon/C++ Derived		recoverable, atomic, subatomic

Italics indicates that transmission of that type is not yet supported by the current implementation.

(!) Provided component types and inherited supertypes, if any, are transmissible.

(+) With union or bitfield component types.

(-) With no union or bitfield component types.

Figure 3-1: Transmissible and Non-Transmissible Types

4. Library

4.1. Non-atomic Avalon/C++ Types and Type Generators

4.1.1. Bools

Avalon defines a boolean type, **bool**, with exactly two values, **TRUE** and **FALSE**, and the usual C++ operations on booleans: !, &&, ||, ==, !=, and =.

4.1.2. Transaction Identifiers

The Avalon run-time system guarantees that the serialization order of transactions is the order in which they commit. The **trans_id** class defines operations on Avalon transaction identifiers to permit run-time testing of the transaction serialization order. There is a *trans_id server* at each site which keeps track of all the trans_id's at that site and handles sending trans_id's to other sites that need them.

Class Definition

```
class trans_id {
 public:
   trans_id(int = UNIQUE);
   ~trans_id();
   trans_id& operator=(trans_id& t);
   bool operator==(trans_id& t);
   bool operator<(trans_id& t);
   bool operator>(trans_id& t);
   bool done();
   friend bool both(trans_id& t1, trans_id& t2);
   friend bool descendant(trans_id& t1, trans_id& t2);
 };
```

Operations

trans_id(), trans_id(UNIQUE)

ensures A dummy subtransaction is created and committed and the subtransaction's identifier is returned to the calling transaction. Note that UNIQUE is the default argument to the trans_id constructor.

trans_id(CURRENT)

ensures Returns the trans_id of an operation's calling transaction.

~trans_id()

ensures The trans_id is deleted.

trans_id& **operator**=(trans_id& t)

modifies **this**
ensures **this** becomes identical to t.

bool **operator**==(trans_id& t)

ensures t1 == t evaluates to **TRUE** if t1 and t are equivalent: **FALSE**, otherwise. Note that trans_id's created by different operations within the same transaction are not equivalent.

bool **operator**<(trans_id& t)

ensures If t1 < t evaluates to **TRUE**, then if both t1 and t commit to the top level, t1 serializes before t. If the expression evaluates to **FALSE**, either t1 serializes after t, or t1 and t are incomparable.

bool **operator**>(trans_id& t)
: **ensures** If t1 > t evaluates to **TRUE**, then if both t1 and t commit to the top level, t1 serializes after t. If the expression evaluates to **FALSE**, either t1 serializes before t, or t1 and t are incomparable.

bool done()
: **ensures** Returns **TRUE** if **this** is committed to the top level; **FALSE**, otherwise.

bool both(trans_id& t1, trans_id& t2)
: **ensures** Returns **TRUE** if t1 and t2 are committed to their least common ancestor; **FALSE**, otherwise.

bool descendant(trans_id& t1, trans_id& t2)
: **ensures** Returns **TRUE** if t1 is a descendant of t2; **FALSE**, otherwise.

4.1.3. x_string: Transmissible Strings

Strings are normally declared in C++ in two subtly different ways: (1) as a fixed array of chars, whose size is known at compile time, and (2) as a char pointer, terminated by a \0, whose size is dynamic; its space is allocated at run-time. Whereas strings as arrays of characters can be trivially transmitted (Section 3.5), strings as char pointers cannot because pointers are not transmissible. The built-in Avalon/C++ class, **x_string**, provides for transmission of dynamically allocated strings.

Class Definition

```
class x_string {
 public:
  operator char*();
  x_string(char* c);
};
```

Operations

operator char*()
: **ensures** Coerces an x_string into a character array.

x_string (char* c)
: **ensures** Returns an x_string constructed from c.

Example

```
server nameList {
 public:
   add_member(x_string member_name);
   x_string pick_random_member();
};

main() {
   namelist nl;
   char* name = new char;

   nl.add_member("Stewart");
   name = nl.pick_random_member();
}
```

The constructor from **char*** to **x_string** will be automatically called in the case of calls to **nameList::add_member**. The coercion operator will transform the result value of **nameList::pick_random_member** into a **char***.

4.1.4. Variants

aggr: ...
  variant

decl-specifier: ...

variant-specifier

variant-specifier:
class-specifier

Avalon/C++ provides an aggregate data type generator, the `variant`, which is declared similarly to a structure or class. An object of variant type can contain a value from a set of types. A variant differs from a standard C++ structure in that it can be only one of its possible subtypes at any given time; it differs from the standard C++ union type in that it is transmissible, i.e., can be sent as an argument to or returned as a result from a server member function.

A variant is a tagged, discriminated union and is made up of two parts, a *tag* and a *value*. The *tag* field specifies which of the possible subtypes is stored in the *value* field, while the *value* field contains some instance of that specified type.

Operations

A variant declaration of the form:

`variant` VT {T_1 V_1; ... ; T_n V_n;};

automatically defines the following operations:

VT `operator=` (VT v)
modifies `this`
ensures Copies v into `this`. The operational effect is that `this`'s tag field changes to be v's, and `this`'s value field is assigned v's, using the the assignment operator defined on v's type.

bool `operator==` (VT v)
ensures v1 == v returns `TRUE` if v1 and v have the same tag, and their values are equal; `FALSE`, otherwise. Two *void* instances of the same variant type are equal.

bool is_void ()
ensures Returns `TRUE` if *this* has no value, and is of the special null-valued `void` type; `FALSE`, otherwise. The `void` type represents the state of a variant instance prior to its first assignment.

and the following operations for each type T_i and tag V_i:

void set_V_i(T_i val)
modifies `this`
ensures Sets the tag of `this` to V_i and its value to val.

T_i value_V_i()
ensures Returns the value of `this` if its tag is V_i; returns a run-time error otherwise.

bool is_V_i()
ensures Returns `TRUE` if the tag of `this` is V_i; `FALSE`, otherwise.

Restrictions

Variants are a special type of class, and can only be declared and defined at the top level, i.e., variants cannot be nested within declarations or definitions of other types, including variants. Variants cannot have member functions.

Example

```
enum PF {FAIL, PASS};
variant grade {
  char  letter;
  short percentage;
  PF    pass_fail;
};
```

In the above example, `grade::set_letter(char c)` would be defined to set the tag of the variant instance to `char`, and its value to `c`, `bool grade::is_letter()` returns `TRUE` if the tag of the variant instance is `char`, and **FALSE** otherwise, and **char grade::value_letter()** returns the **char** value of the instance if it contains a **char**, and produces a run-time error otherwise. Similar functions for **percentage** and **pass_fail** are provided as well.

4.2. Atomic Types

Each C++ fundamental type, **t**, has a derived Avalon atomic type counterpart, **atomic_t**, where **t** currently can be **int**, **char**, or **float**. There is also an Avalon atomic type for booleans, **atomic_bool**, and for (dynamically-sized) strings, **atomic_string** (Section 4.2.1). Each Avalon atomic type has the same sets of values and operations as its non-atomic counterpart. No atomic type is transmissible.

4.2.1. Atomic Strings

The **atomic_string** class is intended to be used in a manner similar to a **char***, as used to represent C++ strings. They should be used as components of atomic and subatomic objects to ensure their recoverability. An **atomic_string** can be of arbitrary, varying length.

Class Definition

```
class atomic_string {
 public:
  atomic_string();
  atomic_string(const char* str);
  atomic_string(atomic_string& astr);
  void operator=(const char* str);
  void operator=(const atomic_string& astr);
  operator char*();

  friend bool operator==(const atomic_string& astr, const char* str);
  friend bool operator==(const char* str, const atomic_string& astr);
  friend ostream& operator<<(ostream& s, atomic_string& astr);
};
```

Operations

atomic_string()
ensures Creates and returns a new, empty atomic_string.

atomic_string(**const** char* str)

atomic_string(**const** atomic_string& astr)
ensures Creates and returns a new atomic_string, initialized with the value of str (astr).

void operator=(**const** char* str)

void operator=(**const** atomic_string& astr)
modifies this
ensures Assigns str (astr) to an atomic_string, adjusting the amount of storage for the string if necessary.

operator char*()
ensures Coerces an atomic_string into a ''standard'' C string, char*, allowing atomic_strings to be used in standard C routines.

bool **operator==**(**const** atomic_string& astr, **const** char* str)

bool **operator==**(**const** char* str, **const** atomic_string& astr)
ensures Returns **TRUE** if astr and str contain the same characters in the same order; **FALSE**, otherwise. Equality is case-sensitive.

ostream& **operator**<<(ostream& s, atomic_string& astr)
  modifies s
  ensures astr is written to the output stream s.

Restrictions

The **char*** returned by the coercion operator must only be used as a **const char***, i.e., the contents of the string should not be changed. The returned **char*** is only valid until the next operation on an **atomic_string**. Thus, multiple coercions may return different **char*** addresses.

Example

```
server foo {
  stable atomic_string a_str;
         ...
};

a_str = "Hello";
if (a_str == "Hello") ...
ulstrcmp (a_str, "hello");
```

a_str is defined to be an **atomic_string**. When the server is started, **a_str** is created uninitialized. The first statement assigns the value "**Hello**" to **a_str**. The second statement uses the equality operator. The last statement shows a use of an **atomic_string** where a **char*** is expected; this use is only acceptable if the called routine does not attempt to modify the contents of the **char*** generated by the coercion. See 5.2 for other usage guidelines.

4.3. Catalog Server

The *catalog server* [9] is part of the Avalon run-time system It maintains a mapping of server attributes to unique server names, and services lookup requests. The current implementation of Avalon has exactly one catalog server since it is expected to be used relatively infrequently; hence, we do not expect it to be a bottleneck. If experience shows otherwise, however, we may decide to run one catalog server per node in future versions of Avalon.

When a server starts, it must check in its attributes. The required attributes (type name (**TYPE**), unique name (**UNIQUE_NAME**), and node (**NODE**)), are automatically registered when the server starts. If more attributes are desired, the server programmer can add them in the constructor code. For example, a printer server might add the identity of the printer it is servicing. When a client wants to locate a server, the **locate_server** function (see end of this section) calls the catalog operation **name** with a list of attributes and returns an object representing the described server. To avoid boot-strapping problems, Avalon ensures that all clients have a reference to the catalog server, which has a fixed unique name, catalogS.

Class Definition

```
server catalog {
 public:
  catalog(path p, node n, size s) : (p,n,s);
  ~catalog();
  int check_in(attr_list alist);
  void remove(int id);
  void set_attributes(int id, attr_list new_alist);
  void set_attribute(int id, x_string attribute, x_string new_value);
  void remove_attribute(int id, x_string attribute);
  attr_list get_attributes(int id);
  x_string get_attribute(int id, x_string attribute);
  int find(attr_list alist);
  x_string name(attr_list alist);
  void main();
};
```

Operations

catalog(node n, path p, size s) : (n,p,s)
ensures Starts and initializes the catalog server on node n, using an executable file p, and reserving s (Camelot) chunks of recoverable storage (where 5 is a reasonable value for s).

~catalog()
ensures Kills the catalog server.

int check_in(attr_list alist)
modifies catalog server
ensures Creates a new entry in the catalog server with the attributes specified in alist and returns a unique id to be later used to look at and modify the attributes of the new entry.

void remove(int id)
modifies catalog server
ensures Deletes the entry of the server identified as id.

void set_attributes(int id, attr_list new_alist)
modifies Attributes of *id*
ensures Replaces the attributed list of the server entry *id* with the new list *alist*.

void set_attribute(int id, x_string attribute, x_string new_value)
modifies *attribute's* value
ensures Replaces the value of *attribute* with *new_value* for the server *id* in the catalog server.

void remove_attribute(int id, x_string attribute)
modifies Attributes of *id*.
ensures The set of attributes for *id* no longer contains *attribute*.

attr_list get_attributes(int id)
ensures Returns a list of attributes for the server *id*.

x_string get_attribute(int id, x_string attribute)
ensures Returns the value associated with *attribute* for the server *id*.

int find(attr_list alist)
ensures Returns the unique id of a server whose attributes match *alist*.

x_string name(attr_list alist)
ensures Returns the value of the *unique name* attribute of a server whose attributes match *alist*.

void main()
ensures No effect.

Locating a Server

In order to locate an existing, running server, one uses one of the following functions:

server_root& get_server (char* unique_server_name)

ensures Returns a reference to a server object for the named server, for those cases where the unique name is fixed or otherwise known. For example, the single catalog server is known as ''catalogS.''

server_root& locate_server (char* typename, attr_list attrl==NULL)

requires Each instance of a type of server supplies identifying attributes when it is started.

ensures Returns a reference to a server of type type with attribute values that match those in attrl, if such a server exists; returns NULL otherwise. For multiple instances of a particular type of server, a specific instance may be selected by listing its unique attributes in attrl.

Note that since locate_server is a generic function, the resulting reference must be coerced to the appropriate type when received.

Example

```
attr_list alist;                  // a new attribute list
alist.push ("printer", "iron"); // CMU printers are named after gems and minerals

printserver& ps = (printserver&) locate_server ("printserver", alist);
if (&ps != NULL)                  // check for NULL return value
  ps.spool (filename);
```

This code obtains a reference to the printserver server object for the printer ''iron.'' If such a server exists, it invokes the server's **spool** operation.

5. Guidelines for Programmers

5.1. Choosing Identifiers

In most ways, Avalon hides the complexity of its underlying mechanisms. When choosing identifiers, however, it must be remembered that Avalon is a preprocessor that generates code for the underlying system, Camelot, which in turn is built on top of Mach. Fortunately between Mach, Camelot, Avalon, C++, and C, some valid identifiers remain.

Here are some guidelines:

1. Do not begin your identifiers with "`_avalon`". Except for names documented in this report, all identifiers inserted into the generated code by Avalon/C++ begin with this string.
2. Do not end your identifiers with "`_t`". All Camelot types end with "`_t`".
3. Do not end your **`struct`** names with "`_struct`". Again, Camelot uses these.
4. Beware of uppercase identifiers. There are many constants (**`#define`**, enums, etc.) and macros which use uppercase identifiers.

5.2. Using and Implementing Avalon Types

This section gives some guidelines for correct usage of the two Avalon built-in classes, **`recoverable`** and **`atomic`**. (Rules for **`subatomic`** are forthcoming.) The rules outlined here do not represent the only correct usage, but rather, a usage which is "guaranteed" to provide correct results. These rules, of course, do not address standard programming practices such as "*Do not free memory twice.*"

There are three kinds of programmers:

Client programmers:

These people write programs which invoke operations on servers. Their job is to ensure that the operations are called correctly. There is only one rule for client programmers to obey: ***All server operation invocations must be made within a transaction.***

Type users/Server programmers:

These people define servers, and use built-in or user-defined types. Their job is to declare, construct, and invoke operations properly on instances of these types.

Type implementors: These people define new types, *Avalon types*, derived from built-in or other user-defined types. Their job is to define and implement the member functions of the type such that, provided it is used correctly, it will exhibit a desired behavior. Note that, when creating a new Avalon type that uses another Avalon type, the programmer is both a type implementor (of the new type) and a type user (of the used type).

In the next four sections, we give rules for users of recoverable types, users of atomic types, implementors of recoverable types, and finally, implementors of atomic types.

5.2.1. Using a Recoverable Type

Allocation: All Avalon types are allocated from *recoverable memory* (a special heap). This is accomplished through an appropriate constructor provided by either the type implementor or generated by Avalon. Care must still be taken, however, not to force allocation of an Avalon type from other than recoverable memory (such as the stack). Thus:

1. Do not declare variables or functions of an Avalon type. Instead, use references or pointers to Avalon types.

2. Do not **new** an array of Avalon objects (e.g., **new** myatomic[10])[3].

3. Do not coerce a non-Avalon type to an Avalon type either explicitly, e.g.,

   ```
   str = (atomic_string)"string";
   ```

   or implicitly, e.g.,

   ```
   atomic_string::atomic_string (char* istr) {...} // constructor taking a char* argument
   void afunction (atomic_string& s) {...}         // function expecting an Avalon type
   afunction ("string");                           // BAD code!
   ```

   The trouble here is that C++ interprets a constructor of one argument as a coercion from the argument's type to the class type. In the example, C++ converts the **char*** "**string**" to an **atomic_string** reference by creating a temporary variable on the stack of type **atomic_string**.

Use: All usage of an Avalon type should be through member functions provided by the type.

5.2.2. Using an Atomic Type

Constructing Atomic Objects: When constructing an atomic object it is important that the creating transaction has exclusive access to the location which will hold the new object. Thus:

```
class myatomic : public atomic {
  atomic_int* i;
  ...
  void newint (int);
};

void myatomic::newint (int n) {
  (*this).write_lock();
  pinning () i = new atomic_int (n);
}
```

Before creating the new **atomic_int**, the function obtains exclusive access to the variable (**i**) which will hold the address of the object.

Destroying Atomic Objects: Similarly, when destroying an atomic object, the transaction must have exclusive access to all pointers to the object.

```
class myatomic : public atomic {
  atomic_int* i;
  ...
  void deleteint();
};

void myatomic::deleteint() {
  (*this).write_lock();
  delete i;
  pinning () i = 0;
}
```

5.2.3. Implementing Recoverable Types

Constructors and Destructors: Storage for all Avalon types must be allocated from recoverable memory. Avalon takes care of storage allocation and deallocation for types with constructors which do not make assignments to **this**. See the section **Assignment to This** for special rules concerning the proper use of such assignments.

Any initializations made to the object within a constructor must be within a **pinning** block or **pin** and **unpin** statements (see the section below on **Modifications**).

[3] This restriction should be temporary.

<u>Contents</u>: Avalon types may be constructed from only the following types:

1. In-line basetypes such as int, char, bool, etc.,
2. In-line Avalon types,
3. Pointers to Avalon types.
4. In-line arrays and structs of the preceding types.

All fields must be either **private** or **protected**.

<u>Modifications</u>: All modifications must be (dynamically) within a **pinning** block or a **pin/unpin** pair. There must be a matching **unpin** called for each **pin** and **unpin** may not be called without a prior call to a matching **pin**.

<u>Coercions</u>: Care should be taken against providing the user with a pointer directly into recoverable memory. All changes to a recoverable object should occur within only the object's member functions. For example, an **atomic_string** may have an **operator char*** function. This function should **malloc** volatile memory to hold the string rather than return a pointer to the array in recoverable memory. Otherwise, the user could modify it outside a **pinning** block with undefined results. Ideally, C++ would let you define an **operator const char***, but it does not.

<u>Overriding Member Functions</u>: If the type overrides the default **pin** and **unpin** operations, the new implementations must ensure that, if **pinning**, or **pin** and **unpin** are properly called, all changes will be made within calls to **recoverable::pin** and **recoverable::unpin**.

<u>Assignment to This (long section)</u>: C++ allows the programmer to manage the allocation of objects through special code in its constructors, particularly assignments to the variable **this**. Using assignments to **this**, the programmer can, for example, implement variable-sized objects, and objects which are allocated from a programmer maintained memory free store. When using an assignment to **this**, however, care must be taken not to interfere with Avalon's managing of the recoverable heap.

In what follows, we will describe the requirements for

- A simple constructor which explicitly allocates its memory,
- Variable-sized objects, and
- Objects which may be either allocated by the constructor or pre-allocated (such as when the object is an in-line part of a struct).

A simple constructor or destructor could look like this:

```
mytype::mytype() {
  int mysize = sizeof(mytype);
  this = (mytype*) REC_MALLOC (mysize);

  pinning() {
    // Initialize the fields of your type.
  }
}

mytype::~mytype() {
  pinning() {
    // Cleanup the fields of your type.
  }

  REC_FREE(this);
  this = 0;
}
```

In the constructor:

- All execution paths must make an assignment to **this**.
- To allocate memory for the object you must use **REC_MALLOC** rather than **new** or **malloc**. If you have reason to allocate another recoverable object, you may (and should) use **new**. For example:

  ```
  this = (mytype*) new atomic_int;
  ```
- You must compute the size correctly (use **sizeof(your_type)** so you include any space needed by the type's ancestors.)
- No member functions (e.g. **pin** and **unpin**) may be called before the assignment to this.

In the destructor:

- **REC_FREE** (rather than **delete** or **free**) must be used to deallocate the memory.
- After deallocation, **this** must be assigned the value 0 so that the ancestor's destructors will not be called.
- No member functions may be called after the deallocation of **this**.

The most common use of an assignment to **this** is to implement variable-sized objects[4]. However, any recoverable type for which **sizeof(yourtype)** may return an incorrect value must either call the functions **pin** and **unpin** with the correct size rather than use the **pinning** statement, or override these functions so that they use the correct size, allowing **pinning** to work properly (as shown here).

```
void mytype::pin(int ignore_size) {
  int size = (*this).object_size;
  recoverable::pin(size);
}

void mytype::unpin(int ignore_size) {
  int size = (*this).object_size;
  recoverable::unpin(size);
}
```

These functions ignore the incorrect size which the **pinning** statement uses when it calls **pin** and **unpin** and instead, uses the real size of the object. This particular example assumes that the constructor stores the allocated size in the field **object_size**.

It is important to remember that, with C++, many uses of a type force the allocation of the object's memory prior to calling its constructor. These uses include: (1) construction of a derived type, (2) allocation of an array of objects of this type, and (3) in-line use of the type in a struct. If a type which handles its own allocation (assignment to **this**) is to be used in these situations, the constructor must be written such that:

1. Memory is allocated only if **this** is 0 upon entering.
2. If **this** is not 0, an assignment to **this** is still executed. The statement **this = this;** will suffice.
3. If memory is allocated, the function **(*this).on_heap** is called after the assignment to **this**. This tells the destructor that the memory was allocated and needs to be deallocated.

For example:

```
mytype::mytype() {
  if (this == 0) {
    int mysize = sizeof(mytype) + <whateverelse>;
    this = (mytype*) REC_MALLOC (mysize);
    (*this).on_heap();
```

[4] The last field of a struct is declared as an array of size 1. When you construct an instance of the type, however, you **REC_MALLOC** as much memory as needed for an array of the desired length (plus the initial fixed size portion of the struct and its ancestors). See [14] for examples.

```
    }
    else this = this;

    pinning() {
      // Initialize the fields of your type.
    }
  }
```

The destructor would then deallocate the memory only if the constructor allocated it:

```
  mytype::mytype() {
    pinning() {
      // Cleanup the fields of your type.
    }

    if ((*this).get_heap_bit() == TRUE) {
      REC_FREE (this);
      this = 0;
    }
  }
```

The functions **on_heap** and **get_heap_bit** are protected member functions exported by class **recoverable**. (Since these are used only in the rare instances in which programmers wish to pre-allocate objects, they are not described with the other exported functions.) The function **on_heap** simply sets a bit in the object which is checked by the function **get_heap_bit** (returning **TRUE** if it was set and **FALSE** otherwise).

5.2.4. Implementing an Atomic Type

Types derived from class **atomic** should follow the requirements outlined above. In addition, if the type is expected to exhibit atomic behavior (serializability, transaction-consistency, and persistence), the guidelines in this section should be followed.

Contents: Pointer fields in the type should point only to types which are atomic (derived from **atomic** or **subatomic**), or recoverable provided that concurrent access to a recoverable object is protected by an appropriate lock on the containing atomic object.

Modifications:

1. **read_lock** on the object should be called by a member function prior to accessing any data in the object. **write_lock** should be called prior to any modification to the data. Pointers to non-atomic (recoverable) objects should be treated the same as in-line non-atomic objects in that appropriate locks should be obtained on the enclosing atomic object prior to invoking member functions on the object. No locking is required when accessing atomic components (in-line or pointers) since the objects' member functions should acquire the necessary locks.

2. If it is intended that a non-in-line subcomponent of an object be protected through locks on the containing object, the subcomponent should be derived from **recoverable** rather than **atomic** (i.e., the object is persistent but relies on the caller for concurrency control).

Coercions: An atomic object should not be coerced to a non-atomic type.

Overriding Member Functions: If the type overrides the default **read_lock** and **write_lock** operations, the new implementations must ensure that, if the type user properly calls **read_lock** or **write_lock**, the appropriate calls to **atomic::read_lock** and **atomic::write_lock** are made.

5.3. Constructing an Avalon Program

5.3.1. Server Programs

A server program should be broken into files as follows:

<server>.h declares the server and includes any type definitions required by the server.

<server>.av provides the implementation for each of the server's member functions and any support functions not declared or included in <server>.h.

<other>.{av,o} provides the implementation for any functions declared in <server>.h other than the server's member functions.

A server program should be linked with the following libraries in order:

```
-lmisc -lava -lgen -lcamlib -lswitches -ltermcap \
-lthreads -lcam -lmach -lm -lnode
```

5.3.2. Client Programs

A client program includes the <server>.h file for each server it uses. Avalon ensures that implementations for the server's member functions are included. It is the responsibility of the programmer, however, to include the implementations of any other functions declared in <server>.h and any files it includes. In general, a client program must be linked with all of the .o files for each server it uses *except for <server>.o*. The libraries needed by the server should also be linked with the client program.

5.3.3. Example Templates

```
                          ---- myserver.h ----
                          #include <avalon.h>   // always first file included.
                          #include <mytype.h>   // defines types used by the server.

                          server myserver {
                            mytype mt;
                           public:
                            myserver (...);
                            ms_op1 (...);
                            ms_op2 (...);
                          };

---- myserver.av ----                                ---- myclient.av ----
#include <myserver.h>                                #include <myserver.h>
                                                     ...
int private_utility () {...}

myserver::myserver (...) {...}
myserver::ms_op1 (...) {... private_utility(); ...}
myserver::ms_op2 (...) {...}

---- mytype.av ----
#include <mytype.h>

mytype::mytype(...) {...}
mytype::mt_op1(...) {...}
mytype::mt_op2(...) {...}

---- server.make ----                                ---- client.make ----
acc -o myserver myserver.o mytype.o \                acc -o myclient myclient.o mytype.o \
        -lmisc -lava -lgen -lcamlib \                    -lmisc -lava -lgen -lcamlib \
        -lswitches -ltermcap -lthreads \                 -lswitches -ltermcap -lthreads \
        -lcam -lmach -lm -lnode                          -lcam -lmach -lm -lnode
```

The file **myserver.av** provides only the implementations of the server's member functions and the implementation of private_utility which is not defined in myserver.h and thus, will not be needed by the client. The object file generated for **myserver.av** is linked in with the server program but not the client program.

The file **mytype.av** provides implementations of the other functions defined in **myserver.h** through the **#include <mytype.h>**. Since the client includes this file, it also needs to be linked with **mytype.o**.

Finally, both the client and the server need to be linked with the standard set of libraries needed by Avalon. For complete examples, look at the servers, clients and makefiles in /afs/cs/project/avalon/src/avalon/bin/validate. See also the *acc* man pages (Appendix II) for appropriate flags with which to call *acc*.

5.4. For Experts Only

5.4.1. Undo and Destructors

When a transaction is aborted using an **undo leave (return, break, continue)** statement, control may be transferred directly to the textual end of the transaction using the C **longjmp** mechanism. This transfer of control will exit one or more blocks in which automatic variables may have been initialized by a constructor. These variables may be instances of a class that has a destructor, and, if so, this destructor would normally be called on these variables before the block was exited. When a transaction is aborted, however, these variables will not have destructors called for them. (Note that this is a problem shared with any use of the **setjmp/longjmp** mechanism in C++.) Normally, the constructor and destructor of a class only modify the object they are invoked on. In this case, this may not be a serious problem; the only result of not calling the destructor is that space on the free store is gradually lost. However, some classes are written so that the constructor and destructor modify some external data structures, and rely on the assumption that both the constructor and the destructor will be called for each object to maintain the integrity of those data structures. These kinds of classes would interact badly with **undo** statements that exit multiple blocks, and should probably be avoided. Future versions of Avalon/C++ may attempt to handle this interaction more gracefully.

5.4.2. User-Defined Transmission Functions

Before any class instance can be actually transmitted to another process, it must be translated into a special, built-in class called **_ava_message**. The _ava_message abstract representation is that of a queue. Objects are removed from the queue in the same order in which they were inserted.

Class Definition

```
class _ava_message {
 _ava_message();
 _ava_message& operator<<(_ava_message& msg);
 _ava_message& operator<<(_ava_msgfield& msg);
 _ava_message& operator>>(_ava_message& msg);
};
```

Operations

_ava_message()

ensures Creates and returns a new instance of an _ava_message.

_ava_message& **operator**<<(_ava_message& msg)

_ava_message& **operator**<<(_ava_msgfield& msg)

ensures Appends msg to the end of an _ava_message.

_ava_message& **operator**>>(_ava_message& msg)

ensures Extracts built-in base types from the message instance. Higher-order types are extracted using the class's *_recompose* function (see below) with the message instance as an argument.

To add user-defined transmission to a user-defined class, you must define two class member functions in order to be able to transmit a class instance:

operator _ava_message()

ensures Coerces a class instance into an _ava_message. It will typically need to call the transmission functions on other types. For each class, _ava_message instances are constructed by calling the class's coercion operator. For each built-in fundamental type (int, chars, floats), a special class, *_ava_msgfield*, with overloaded constructors, is provided. Since enumerations are represented in C++ as integer constants, they should be treated as if they were of type *int* for the purpose of transmission.

void _recompose(_ava_message& msg)

modifies ***this** (Obscure, but true.)
ensures Constructs a new instance of the class and overwrites the old one with the new.

Figure 5-1 gives a sample of transmission functions for a simple class.

5.4.3. Processes

Support for **processes** *has not yet been implemented and will not be soon.*

A coarm of a **costart** statement can also be a regular process with no transaction semantics:

coarm: ...
process *statement*

We make no guarantees as to giving any meaningful semantics to processes that run concurrently with transaction coarms, or processes that run within transactions.

5.4.4. Pragmas

Support for **pragmas** *has not yet been implemented and will not be soon.*

pragma:
@pragma@ pragma-list

pragma-list:
prag
prag , pragma-list

prag:
identifier
identifer = value

A pragma is used to convey information to the compiler. Use of pragmas is an appropriate escape mechanism to Camelot features.

For example, Camelot provides two different kinds of logging, *new-value/old-value* and *new-value only* and mechanisms to support various commit protocols. Different combinations are useful depending on the expected length of a transaction. Thus, we allow the user to specify via a pragma whether a newly started transaction will be "short" or "long." The standard default is "medium" and the following combinations are defined for each value:

Short	new-value only logging blocking protocol, e.g., two-phase commit
Medium	new-value/old-value logging blocking protocol, e.g., two-phase commit
Long	new-value/old-value logging non-blocking commit protocol

```
struct address {
  int      number;
  char     street[40];
  char     appt[8];
  char     city[20];
  char     state[3];
  int      zipcode;
};

class personnel {
  char     name[40];
  int      ss_number;
  float    salary;
  enum {WEEKLY, HOURLY, MONTHLY} payroll_type;
  address home_address;

  personnel(istream);                          // For data entry
  personnel(char* new_name, int new_ss, float new_sal, address new_add);
  operator _ava_message();
  void _recompose(_ava_message&);
};

// Definitions of constructors omitted

personnel:: _ava_message() {
  _ava_message msg = new _ava_message();
  int i;

  // this->name
  for (i = 0; i < 40; i++) *msg << _ava_msgfield(name[i]);

  *msg << _ava_msgfield(ss_number);               // this->ss_number
  *msg << _ava_msgfield(salary);                  // this->salary
  *msg << _ava_msgfield((int) payroll_type);      // this->payroll_type
  *msg << _ava_message(home_address);             // this->home_address

  return (*msg);
}

void personnel _recompose(_ava_message& msg) {
  int i;

  for (i = 0; i < 40; i++) msg >> name[i];          // this->name
  msg >> ss_number;                                 // this->ss_number
  msg >> salary;                                    // this->salary
  { int temp; msg >> temp; payroll_type = temp; } // this->payroll_type
  home_address._recompose(msg);                     // this->home_address
}
```

Figure 5-1: User-defined Transmission Functions

Default The default value is "Medium."

Notice that the combination of new-value only logging and a non-blocking commit protocol is not permitted.

Other pragma values will be determined to incorporate other meaningful combinations, e.g., to indicate using a "highly optimized" protocol for a local transaction.

Restrictions

In general, pragmas are only allowed at any place where the syntax rules allow a *declaration*. Currently, pragmas are treated exactly as comments, and thus, can appear anywhere a comment can appear. No interpretation of pragma values is currently done.

Appendix I
Grammar

The language this grammar defines is a strict superset of that presented in Section 14 of the Reference Manual in [14].

I.1. Expressions

```
expression:
        term
        expression binary-operator expression
        expression ? expression : expression
        expression-list
expression-list:
        expression
        expression-list , expression
term:
        primary-expression
        unary-operator term
        term ++
        term --
        sizeof expression
        sizeof ( type-name )
        ( type-name ) expression
        simple-type-name ( expression-list )
        new type-name initializer_opt
        new ( type-name )
        delete expression
        delete [ expression ] expression
primary-expression:
        id
        : : identifier
        constant
        string
        this
        ( expression )
        primary-expression[ expression ]
        primary-expression ( expression-list_opt )
        primary-expression . id
        primary-expression -> id
id:
        identifier
        operator-function-name
        typedef-name :: identifier
        typedef-name :: operator-function-name
operator:
        unary-operator
        binary-operator
        special-operator
        free-store-operator
```

Binary operators have precedence decreasing as indicated:

binary-operator: **one of**

`* / %`

`+ -`

`<< >>`

`< >`

`== !=`

`&`

`^`

`|`

`&&`

`||`

assignment-operator

assignment-operator: **one of**

`= += -= *= /= %= ^= &= |= >>= <<=`

unary-operator: **one of**

`* & + - ~ ! ++ --`

special-operator: **one of**

`() []`

free-store-operator: **one of**

new **delete**

type-name:

decl-specifiers abstract-declarator

abstract-declarator:

empty

`*` *abstract-declarator*

abstract-declarator (*argument-declaration-list*)

abstract-declarator [$constant\text{-}expression_{opt}$]

simple-type-name:

typedef-name

char

short

int

long

unsigned

float

double

void

typedef-name:

identifier

I.2. Declarations

declaration:

$decl\text{-}specifiers_{opt}$ $declarator\text{-}list_{opt}$;

name-declaration

asm-declaration

pragma

name-declaration:

aggr identifier ;

enum *identifier* ;

aggr:

class

struct

union
server
variant

asm-declaration:
asm *(string)* ;

pragma:
@pragma@ pragma-list

pragma-list:
prag
prag , pragma-list

prag:
identifier
identifer = value

decl-specifiers:
decl-specifier decl-specifiers$_{opt}$

decl-specifier:
sc-specifier
type-specifier
fct-specifier
friend
typedef
server-specifier
variant-specifier

type-specifier:
simple-type-name
class-specifier
enum-specifier
elaborated-type-specifier
const

sc-specifier:
auto
extern
register
static
stable

fct-specifier:
inline
overload
virtual

server-specifier:
class-specifier

variant-specifier:
class-specifier

elaborated-type-specifier:
key typedef-name
key identifier

key:
class
struct
union
enum
server
variant

declarator-list:
init-declarator
init-declarator , declarator-list

init-declarator:
declarator initializer$_{opt}$

declarator:
dname
(*declarator*)
* **const**$_{opt}$ *declarator*
& **const**$_{opt}$ *declarator*
declarator (*argument-declaration-list*)
declarator [*constant-expression*$_{opt}$]

dname:
simple-dname
typedef-name :: simple-dname

simple-dname:
identifier
typedef-name
~ *typedef-name*
operator-function-name
conversion-function-name

operator-function-name:
operator *operator*

conversion-function-name:
operator *type*

argument-declaration-list:
arg-declaration-list$_{opt}$ ...$_{opt}$

arg-declaration-list:
arg-declaration-list , argument-declaration
argument-declaration

argument-declaration:
decl-specifiers declarator
decl-specifiers declarator = expression
decl-specifiers abstract-declarator
decl-specifiers abstract-declarator = expression

class-specifier:
class-head { *member-list*$_{opt}$ }

class-head:
aggr identifier$_{opt}$
aggr identifier : **public**$_{opt}$ *typedef-name*

member-list:
member-declaration member-list$_{opt}$

member-declaration:
decl-specifiers$_{opt}$ *member-declarator initializer*$_{opt}$;
function-definition ;$_{opt}$
decl-specifiers$_{opt}$ *fct-declarator base-initializer*$_{opt}$
private:
protected:
public:

member-declarator:
declarator
identifier$_{opt}$: *constant-expression*

initializer:
= *expression*
= { *initializer-list* }
= { *initializer-list* , }
(*expression-list*)

initializer-list:
expression
initializer-list , *initializer-list*
{ *initializer-list* }

enum-specifier:
enum *identifier*$_{opt}$ { *enum-list* }

enum-list:
enumerator
enum-list , *enumerator*

enumerator:
identifier
identifier = *constant-expression*

I.3. Statements

compound-statement:
{ *statement-list*$_{opt}$ }

statement-list:
statement
statement statement-list

statement:
declaration
compound-statement
expression$_{opt}$;
if (*expression*) *statement*
if (*expression*) *statement* **else** *statement*
while (*expression*) *statement*
do *statement* **while** (*expression*) ;
for (*statement expression*$_{opt}$; *expression*$_{opt}$) *statement*
switch (*expression*) *statement*
case *constant-expression* : *statement*
default : *statement*
undo$_{opt}$ **break** ;
undo$_{opt}$ **continue** ;
goto *identifier ;*
identifier : statement
start trans-body
costart { coarms }
leave ;
undo (*expression*) $_{opt}$ **leave** :
undo$_{opt}$ **return** *expression*$_{opt}$
pinning (*expression*$_{opt}$) *statement*
when (*expression*) *statement*
whenswitch (*expression*) *statement*
pragma

trans-body:
trans-tag *statement* except-clause $_{opt}$

trans-tag:
toplevel

transaction

coarms:
coarm coarms$_{opt}$

coarm:
trans-body
process *statement*

except-clause:
except (*identifier*) $_{opt}$ *statement*

I.4. External Definitions

program:
external-definition
external-definition program

external-definition:
function-definition
declaration

function-definition:
decl-specifiers$_{opt}$ *fct-declarator base-initializer*$_{opt}$ *fct-body*

fct-declarator:
declarator (argument-declaration-list)

fct-body:
compound-statement

base-initializer:
: member-initializer-list

member-initializer-list:
member-initializer
member-initializer , member-initializer-list

member-initializer:
identifier$_{opt}$ *(argument-list*$_{opt}$ *)*

I.5. Preprocessor

#define *identifier token-string*
#define *identifier(identifier , ... , identifier) token-string*
#else
#endif
#if *expression*
#ifdef *identifier*
#ifndef *identifier*
#include *"filename"*
#include *<filename>*
#line *constant "filename"*
#undef *identifier*

Appendix II

NAME

acc – an Avalon/C++ compiler

SYNOPSIS

acc [option] ... file ...

DESCRIPTION

acc is an Avalon/C++ compiler. File names that end with

.c, .c+, .h, .h+, .av
are taken to be Avalon/C++ source files. They are compiled, producing **.o** files, as in *cc* (1).

.s are taken to be *as* (1) source files.

.i are ignored.

File names that end with anything else are assumed to be object files or libraries and are handed directly to *cc*.

acc uses *cpp* to pre-process the input, *avfront* to process the Avalon extensions to C++, *cpp* to pre-process the avfront output, */usr/misc/.c++/lib/cfront* to process the C++ extensions to C, *cc* to compile the resulting C code, and */usr/misc/.c++/lib/munch* to find global variables with constructors and destructors. *acc* defines the macros **__STDC__, c_plusplus,** and **avalon** when running *cpp* the first time, **__STDC__** and **c_plusplus** when running *cpp* the second time. C++ include files are normally taken from */usr/misc/.c++/include*.

There are several options which tell *acc* which programs to run and where to put the output. These options are all prefixed by **+a** .

The following options tell *acc* to run a partial Avalon compile:

+aE Only *cpp* is run. The result is printed on *stdout*.

+aF Only *cpp* and *avfront* are run. The result is printed on *stdout*.

+aG Only *cpp*, *avfront*, and *cpp* are run. The result is printed on *stdout*.

+aH Only *cpp*, *avfront*, *cpp*, and *cfront* are run. The result is printed on *stdout*.

The following options tell *acc* to run all or part of a C++ compile:

+aI Only *cpp* is run. The result is printed on *stdout*. The *avalon* macro is not defined. This option is equivalent to **+aE +aK.**

+aJ Only *cpp* and *cfront* are run. The result is printed on *stdout*. The *avalon* macro is not defined. This option is equivalent to **+aH +aK.**

+aK All passes except *avfront* and the second pass of *cpp* are run. The *avalon* macro is not defined.

The following options tell *acc* to generate a list of *makefile* dependencies:

+aM *cpp* is run to generate a list of *makefile* dependencies. The macros **__STDC__, c_plusplus,** and **avalon** are defined. The result is printed on *stdout.*

+aN *cpp* is run to generate a list of *makefile* dependencies. The macros **__STDC__** and **c_plusplus** are defined. The **avalon** macro is not defined. The result is printed on *stdout*. This option is equivalent to **+aK +aM.**

The following options tell *acc* various other things about how to do the compile:

+a.*suffix*
: The *+aE, +aF, +aG, +aH, +aI, +aJ, +aK, +aM, +aN* and *+aP* options will send the output for each file to a corresponding file with the suffix *suffix*, rather than to *stdout*.

+af Files are used in the preprocessor stage instead of pipes. This may improve performance on machines that spend most of their time paging.

+ah Lines beginning with *#line* or *#number* will be removed from the output produced with the *+aE, +aF, +aG, +aH, +aI, +aJ, +aK, +aM, +aN* and *+aP* options.

+ai The output of *cfront* for each file is put in a file with the suffix "..c". These files are normally deleted, but the *+ai* option keeps them around.

+aP *cpp* and *avplain* are run. The result is printed on *stdout*. *avplain* is a version of *avfront* that parses but does not actually implement the Avalon extensions. It is useful only for maintainers of *avfront*.

+aT *acc* will print timing information.

+aV *acc* will print all the details about what it is doing.

The following options are passed on in various forms to the programs that *acc* runs. This is not an exhaustive list. Other options not listed in this man page are assumed to be *avfront* and *cfront* options if they begin with '+', *cc* options if they begin with '–', and files if they begin with anything else.

+d *avfront* and *cfront* will generate code that is more suitable for debugging. Inline functions will not be expanded.

+S Some run-time statistics for *avfront* and *cfront* will be printed on *stderr*.

+V *avfront* and *cfront* will accept old-style C declarations. Include files will be taken from */usr/cs/include* rather than */usr/misc/.c++/include*

–2D*name=value*
–2D*name*
: *Name* is defined for the second pass of the C preprocessor. If no *value* is given, *name* is defined to be 1.

–2U*name*
: The definition of *name* in the second pass of the C preprocessor is removed.

–D*name=value*
–D*name*
: *Name* is defined for the first pass of the C preprocessor. If no *value* is given, *name* is defined to be 1.

–I*dir* *dir* is added to the search path for include files. Directories given in –I options are searched before */usr/misc/.c++/include* and the directories in the *CPATH* environment variable. This option affects both passes of the C preprocessor.

–U*name*
: The definition of *name* in the first pass of the C preprocessor is removed.

–w *avfront, cfront*, and *cc* warning messages are not printed.

FILES

<some directory in $LPATH>/cpp
: The C preprocessor.

avfront The Avalon preprocessor.

/usr/misc/.c++/lib/cfront
The C++ preprocessor.

/usr/misc/.c++/lib/munch
Finds global variables with constructors and destructors.

cc The C compiler.

***..c** Output from *cfront.*

__ctdt.c
Output from *munch.*

SEE ALSO

as (1), *cc* (1), *ld* (1), *The Avalon Report*

BUGS

avfront sometimes prints names twice in its error messages. For example, "foo" might be printed as "foofoo". This behavior has been observed only when *avfront* was given incorrect code.

The error handling routines in *avfront* get confused easily, resulting in unintelligible error messages. This problem may also cause *avfront* to crash.

The code generated by *cfront* seems to be more likely to trigger bugs and overflow tables in the C compiler than normal C code. The code generated by *avfront* is more likely to do these things to the C++ compiler than normal C++ code.

References

[1] Joshua J. Bloch.
The Camelot Library.
In Alfred Z. Spector, Kathryn R. Swedlow (editors), *The Guide to the Camelot Distributed Transaction Facility: Release 1*, pages 29-62. Carnegie Mellon, 1988.

[2] D. L. Detlefs, M. P. Herlihy, and J. M. Wing.
Inheritance of Synchronization and Recovery Properties in Avalon/C++.
IEEE Computer :57-69, December, 1988.

[3] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger.
The Notions of Consistency and Predicate Locks in a Database System.
Communications of the ACM 19(11):624-633, November, 1976.

[4] J.V. Guttag, J.J. Horning, and J.M. Wing.
Larch in Five Easy Pieces.
Technical Report 5, DEC Systems Research Center, July, 1985.

[5] M. P. Herlihy and J. M. Wing.
Avalon: Language Support for Reliable Distributed Systems.
In *Proceedings of the 17^{th} Int'l Symposium on Fault-Tolerant Computing.* Pittsburgh, PA, July, 1987.

[6] M.P. Herlihy and J.M. Wing.
Reasoning About Atomic Objects.
In *Proceedings of the Symposium on Real-Time and Fault-Tolerant Systems.* Warwick, England, Sept., 1988.
Also available as CMU-CS-87-176.

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

[8] L. Lamport.
Time, clocks, and the ordering of events in a distributed system.
Communications of the ACM 21(7):558-565, July, 1978.

[9] Richard Allen Lerner.
Reliable Servers: Design and Implementation in Avalon/C++.
In *Proceedings International Symposium on Databases in Parallel and Distributed Systems*, pages 13-21. IEEE CS TC on Data Engineering, ACM SIG on Computer Architecture, IEEE Computer Society Press, Austin, TX, December, 1988.
Also published as CMU Tech. Report: CMU-CS-88-177.

[10] B. Liskov, M. Day, M. Herlihy, P. Johnson, G. Leavens, R. Scheifler, W. Weihl.
Argus Reference Manual.
Technical Report TR-400, MIT Laboratory for Computer Science, Cambridge, MA, November, 1987.

[11] 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.

[12] Alfred Z. Spector, Kathryn R. Swedlow, ed.
The Guide to the Camelot Distributed Transaction Facility: Release 1
0.98(51) edition, Carnegie Mellon University, Pittsburgh, PA, 1988.

[13] Alfred Z. Spector, Randy Pausch, and Gregory Bruell.
Camelot: A Flexible, Distributed Transaction Processing System .
In *Proceedings of Compcon 88.* February, 1988.

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

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

[16] J.M. Wing.
Specifying Avalon Objects in Larch.
In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT).* Barcelona, Spain, March, 1989.
To appear, invited paper.

Index