Avalon/C++:

C++ Extensions for Transaction-Based Programming

David Detlefs, Maurice Herlihy, Karen Kietzke, Jeannette Wing[1]

Abstract

Avalon extends C++ to support transaction-based programming. Avalon allows programmers to implement classes of *atomic objects*, objects that provide *failure atomicity, permanence*, and ensure the *serializability* of the transactions that perform operations on them. Classes gain these properties by inheritance from a set of built-in classes. Atomic objects are encapsulated in processes called *servers;* Avalon provides syntactic constructs for defining and executing servers. Constructs are also provided for beginning, ending, and aborting transactions, and for executing transactions concurrently.

Avalon is being implemented as a preprocessor which translates Avalon code into C++. This preprocessor is being built by modifying the C++ preprocessor. The code we generate makes extensive use of the Camelot transaction processing system [9], which, in turn, relies on the Mach operating system [1].

1. Introduction

This paper has four parts. First, we motivate the need for Avalon, as language support for programming reliable distributed systems (section 2), and highlight its key features (section 3). Avalon is interesting in its own right as an example of the extension of object-oriented programming to a new application domain. Second, we describe our experience extending C++ (section 4), which may be of interest to others considering extending C++ for other uses. We briefly give some comments on our experience in using C++ (section 5), and close with an outline of our future plans (section 6).

---

[1]This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory Under Contract F33615-84-K-1520. Additional support for J. Wing was provided in part by the National Science Foundation under grant CCR-8620027.

2. Motivations for Avalon

Concurrency and fault-tolerance are the two key properties of large software systems that Avalon addresses. The object-oriented programming paradigm allows concurrency to be easily introduced into systems. For example, the ''message-passing'' metaphor used by systems such as Smalltalk corresponds well to the communication techniques used by concurrent languages like CSP [7]. However, few object-oriented languages actually support concurrency directly. Some significant exceptions include Orient84K [8], Concurrent Smalltalk [10], and Actors [2]. We believe that it is important to realize that adding concurrency to an object-oriented language must be more than simply adding a ''cobegin'' statement and locking primitives. In an object-oriented language, concurrency control can and should reside in the objects. Avalon provides facilities that allow objects to synchronize processes that are attempting to execute operations on them concurrently. Avalon supports concurrency both at the level of multiple processors on a single machine and at the level of multiple machines in a network.

We address fault-tolerance, particularly in conjunction wich concurrency, by relying on known serialization and recovery techniques from the database community. Databases themselves are logical candidates for being recast in the object-oriented paradigm. The view is that databases are objects supporting various ''lookup'' and ''insert'' operations, and that the records in the database are themselves other objects of the appropriate record type. Databases systems, however, often have very stringent reliability requirements, which are commonly satisfied by grouping operations on the database into sequences called *transactions*. Modifications made by a transaction become permanent if and only if the transaction completes; if something goes wrong (the transaction *aborts*) the modifications are undone. Avalon provides transaction semantics through the objects that transactions manipulate. These *atomic objects* provide for their own recovery in the event of a transaction abort. Avalon provides a set of built-in atomic classes, as well as facilities that allow users to define new ones.

Taken together, network- and processor-level concurrency and database-style fault-tolerance form an application domain called *reliable distributed computing*. This domain is already important; systems such as automated teller machine networks and airline reservation systems are real-world examples. We believe that many more systems will be built this way in the future, especially if it becomes easier to implement them. A high-level programming language such as Avalon, which supports this domain, will greatly reduce implementation complexity. Providing features in high-level constructs will avoid errors, by eliminating the need for hand-crafting complicated operating system-level code for each instance of the construct. Also, a high-level language may be able to recognize opportunities for special-case optimizations and apply them more uniformly.

3. Avalon Language Features

Avalon is a superset of C++. The main features Avalon adds to C++ are transactions and atomic objects, as summarized above, and *servers*.

3.1. Transactions

Transactions are sequences of operations on atomic objects. Transactions are characterized by three properties: *failure atomicity, permanence,* and *serializability*. Failure atomicity is the "all-or-nothing" property described above; if a transaction fails to complete (aborts), none of its partial effects become visible. Permanence means that when a transaction has committed, subsequent failures cannot erase its effects. *Serializability* means that the effect of running multiple transactions concurrently must be the same as if they had been run in some serial order. This makes it easier to reason, both formally and informally, about transactions, since each can be thought of as executing without interference. These properties also make it easier to recover a consistent state for a system after a crash: the recovered state should be the same as if all committed transactions executed in their serialization order. We summarize these properties by saying that transactions are *atomic*.

Avalon provides constructs for delimiting transactions (the **start** statement), for starting concurrent transactions (**costart**), and aborting transactions (**undo**).

3.2. Atomic Objects

The atomicity properties of transactions are ensured in Avalon by the implementations of the atomic objects they manipulate. These objects must ensure that enough state is saved to recover from transaction aborts; that committed modifications are written to stable storage to ensure permanence; and that the transactions that execute operations on an object do so in a way that guarantees the serializability of those transactions.

Avalon provides a number of features that make it easy for implementors of classes of atomic objects to meet these requirements. Most of these features are available through inheritance from classes in the Avalon class hierarchy. This hierarchy consists of three classes:

- RESILIENT provides PIN and UNPIN operations that derived classes can use to guarantee failure atomicity. Roughly speaking, these record snapshots of the object before and after modifications. Because these must be called in pairs, Avalon also provides a **pinning** block that brackets its body with a PIN/UNPIN pair. The Avalon run-time system guarantees that after a failure, subsequent transactions will observe resilient objects in the state they were in when last UNPIN'ed by a transaction that committed.
- ATOMIC is a subclass of RESILIENT. In addition to PIN and UNPIN, it provides (protected) operations that derived classes can use to do short-term synchronization of processes concurrently executing operations

at the same object. ATOMIC objects can also provide user-written operations that are invoked by the run-time system when transactions that modified the object commit or abort. This facility can be very useful in the implementation of atomic objects that allow a high degree of transaction-level concurrency.

- DYNAMIC is a subclass of ATOMIC that provides the traditional method of ensuring serializability: strict two-phase read/write locking [4].

3.3. Servers

Transactions are ephemeral; they execute for a relatively short time and are finished. Atomic objects, however, may have long lives. Therefore, they are encapsulated in processes called servers. A server is textually similar to a class. The private members of a server (may) include atomic objects. If a server crashes, it is automatically restarted, and its atomic members are recovered in the state seen by the last completed transaction. The public members of a server are operations that can be invoked from other processes by remote procedure calls (RPC's). The server process consists of a loop that listens for an RPC, and forks a lightweight process to handle each request that arrives. Thus, concurrency is allowed within servers. This level of concurrency could be effectively utilized by a shared-memory multiprocessor.

3.4. For Further Details...

This section provided only a brief outline of the Avalon language. For more details, interested readers may refer to [5] or [3].

4. Avalon as an Exercise in Extending C++

In this section, we will discuss some of the engineering decisions that went into (and are still going into) our implementation of Avalon. We hope that our experience will be of interest to others considering extensions to C++ and other languages.

4.1. Levels of Engineering Complexity

When modifying a language, one should obviously attempt to make changes in as simple and modular a way as possible. In our case, our modifications came in three forms, of successively greater complexity:

- *Working within the language.* One of the touted advantages of object-oriented languages and C++ in particular is that classes and operator overloading are supposed to provide the ability to create "virtual languages" customized to particular applications. To a large extent, we found this claim to be well-founded, and used this ability in the class hierarchy described above.
- *Run-time support.* Many of the features of Avalon are provided by a run-time system. Camelot

comprises most of the Avalon run-time system; it provides operating system-level support for transaction management, stable-storage logging, crash recovery, and the like. Avalon-specific elements of the run-time system include a package for dynamically allocating recoverable storage, a server for recording and comparing transaction identifiers for transaction-level synchronization, and a server for registering types and instances of other servers.

- *Language processor.* Avalon is a superset of C++, with a number of new syntactic (and semantic) features. We had to somehow take care of compiling these new constructs into appropriate code. We chose to do this using a preprocessor that translates Avalon into C++.

Implementing a processor for a new language is a complicated undertaking, and we had said previously that we would strive to avoid complication whenever possible. Therefore, we will explain the considerations that went into this decision.

One alternate approach is to implement the new features of Avalon as a macro package. We saw three drawbacks with this approach:

- Using a macro preprocessor would restrict us somewhat in the syntax we could use. While we do not think that syntax is overwhelmingly important, we do believe that a consistent syntax enhances the understandability of a language. Using a separate preprocessor allowed us to choose whatever syntax we wished.
- Macro processors are notorious for producing unintelligible error messages, or worse, for allowing incorrect invocations to pass through to become run-time bugs. The semantics of the new features we provide are complicated enough that many checks must be done to ensure that they are being used correctly. This type of error checking increases the usability of the system by increasing the user's confidence. We also felt that it was important that, as in the C++ preprocessor, all compile-time errors should be detected in the first pass, and later passes (C++ and C, in our case) should function only as code generators. Without this quick feedback, the time required to rid a program of compile-time errors might, to some users, outweigh the advantages of compile-time type-checking.
- Finally, the implementations of many of the features we provide must use information (such as type information) that is available only during the compilation process. It would have been impossible to implement these features using macros.

Given that we had decided to implement a preprocessor, we had the further choice of implementing one from scratch, or of modifying the existing `cfront`. The first option could be rejected easily: it would have been too much work to implement a new preprocessor for the entire Avalon language. Another choice might have been to implement a new preprocessor that handled only the added features of the language. It turned out that the added features interacted extensively enough with the existing features to make this impossible; also, this would have violated the ''quick error turnaround'' goal. Thus, we decided to work from the existing `cfront`.

For the most part, this decision has been a good one. Working from a stable base, we have been able to concentrate on problems in our extensions, rather than problems in processing C++. However, we have found `cfront` somewhat difficult to work with because it uses a table-driven parser (`yacc`), rather than a recursive descent parser. On a number of occasions we have found it difficult to debug problems with our grammar from the diagnostics provided by `yacc`. With a recursive descent parser, one can at least locate the problem with a debugger. Our other reason for preferring a recursive descent parser is more subtle. In the Avalon preprocessor, we take the parse tree generated by our modified `cfront` and modify it to implement our extensions. These modification often require the insertion of fairly large amounts of new code. In some cases, we can insert this new code as text, and allow the next pass to compile it, but in others, later code must refer to the generated code for type-checking purposes. For these, we must splice the parse tree for the new code into the existing parse tree. Presently, we must create these parse trees ''by hand,'' by invoking the constructors of the various parse tree node types. It would be much easier if we could construct the string form of the expression or statement we wished to insert, parse that ''in place,'' and insert the resulting subtree into the parse tree. With the current parser, the only syntactic constructs that can be parsed are top-level declarations, so we can not usually use it for this purpose. With a recursive descent parser (or perhaps a differently structured table-driven parser) there could be different routines for parsing declarations, statements, expressions, etc., that we could invoke as needed.

5. Our Experience in Using C++

In this section we discuss our experience in using C++ on a moderately large program that uses a large base of existing C code. We have had a number of problems that we thought it would be interesting to share with our readers.

5.1. System-wide C++ Compatibility

Avalon programs make extensive use of previously written C code, specifically, the Mach operating system and the Camelot transaction-processing system. We needed to cause the include files that declare the routines we use from these systems to do so in a manner compatible with the C++ argument declaration syntax. We could have done this by making private copies of these files, modifying them, and attempting to track any changes (as is the strategy in the C++ versions of the Unix system include files sent out in the C++ distribution.) This strategy can obviously lead to problems when changes are not accurately and promptly tracked.

The obvious solution to this problem is to maintain one set of include files compatible with both C++ and (non-ANSI standard) C function declaration syntax. Technically, this is simple -- we use a macro developed by Mike Jones at CMU:

```
/* c_args.h:
 * A standard convention for declaring C and C++ functions in header files.
 * Author: Mike Jones, CMU.
 */
#if     c_plusplus
#define C_ARGS(arglist) arglist
#else   c_plusplus
#define C_ARGS(arglist) ()
#endif  c_plusplus
```

After inclusion of this definition, a function `foo` that takes an integer argument `k` may be declared as

```
void foo C_ARGS((int k));
```

Ensuring that these conventions are followed is critical, but difficult, since C++ declarations in C include files are not checked during C compilations. Therefore, inconsistencies are often not noticed until we detect them in our code. We hope to solve part of this problem soon on CMU machines within our department by modifying the standard system include files for C++ compatibility.

5.2. Debugging

The Avalon preprocessor is a simple single-thread-of-control program that does no complicated I/O. We are currently debugging the program using `gdb`, the Gnu debugger. For many reasons, this is unsatisfactory. `Gdb`, `dbx`, and most other Unix source-level debuggers, are oriented toward C. Thus, when one runs your program under the debugger, one actually debugs the C code generated by `cfront`. Though this is not as severe a shift as in a debugger such as `adb`, where one debugs the machine code generated by the compiler, it is similar. Several C++ constructs expand into many lines of generated code. Names of variables and structures change drastically, and in some cases in ways that cannot be predicted by looking at the original source (unions are a special problem in this regard.) Debugging C++ code would be greatly aided by C++-oriented debugger.

6. Future Directions for Avalon

6.1. Writing Test Applications

Once we think we have a working system, our first order of business will be to write some applications to test it. Our first efforts will probably involve the reimplementation of applications being constructed (in C) to test Camelot. These include a conference room reservation system intended to be used by our department, and an ordering system for a department cheese purchasing cooperative. Building these kinds of applications will allow us to gain experience in using our new language constructs, to test and debug the Avalon preprocessor and runtime system, and to explore more generally the topics described below.

6.2. Testing Distributed Systems

Testing and debugging a reliable distributed system is a difficult process. In a reliable distributed system, one not only has to worry about the sequential correctness of one's program, but also its interactions with concurrently executing programs, and the possibility of system failure at any time. This makes testing correspondingly harder. Our attitude is that it is impossible to test a complicated distributed system exhaustively enough to give any real confidence in its correctness, especially in application domains with extreme reliability requirements. We must instead rely on methodologies that limit the categories of bugs we can encounter. For instance, it should be possible to test individual Avalon atomic classes in isolation, and have confidence that they will work correctly when combined in a server. That is, once one has satisfied oneself that a class enforces atomicity, one can write a server operation that use instances of that class as if the operation were part of a normal sequential program. Thus, we have at least made testing more tractable. To aid testing at the level of atomic classes, we might investigate ways of automatically generating concurrent tests from a specification of sequential tests of the type [6].

6.3. Debugging Distributed Systems

Despite the efforts described above, it seems inevitable that bugs will be found when a distributed system is put together. There will be a need for methods and tools for debugging these systems. These go beyond the scope of conventional debugging aids because they extend across machines, and may involve many processes on each machine. For instance, we might want the ability to single step a thread of control as it migrates from one process to another via an RPC. We might also want the ability to query the system about what threads of control are waiting to perform an operation on an atomic object. We need to develop tools with these and other capabilities for monitering and debugging these large distributed systems.

References

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

[2] Agha, Gul and Hewitt, Carl.
Concurrent Programming Using Actors.
Object-Oriented Concurrent Programming.
The MIT Press, Cambridge, MA, 1987.

[3] D. Detlefs, M. P. Herlihy, and J. M. Wing.
Inheritance of Synchronization and Recovery Properties in Avalon/C++.
In *The Proceedings of the 21th Hawaii International Conference on System Sciences.* Kailua-Kona, Hawaii, Jan, 1988.

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

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

[6] Herlihy, Maurice and Wing, Jeannette.
Reasoning About Atomic Objects.
Technical Report CMU-CS-87-176, Carnegie-Mellon University, November, 1987.

[7] C.A.R. Hoare.
Communicating Sequential Processes.
Communications of the ACM 21(8):666-677, August, 1978.

[8] Ishikawa, Yutaka and Tokoro, Mario.
Orient84/K: An Object-Oriented Concurrent Programming Language for Knowledge Representation.
Object-Oriented Concurrent Programming.
The MIT Press, Cambridge, MA, 1987.

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

[10] Yokote, Yasuhico and Tokoro, Mario.
Concurrent Programming in ConcurrentSmalltalk.
Object-Oriented Concurrent Programming.
The MIT Press, Cambridge, MA, 1987.