Using Larch to Specify Avalon/C++ Objects

JEANNETTE M. WING, MEMBER, IEEE

Abstract—This paper gives a formal specification of three base Avalon/C++ classes; recoverable, atomic, and subatomic. Programmers derive from class recoverable to define persistent objects, and from either class atomic or class subatomic to define atomic objects. The specifications, written in Larch, provide the means for showing that classes derived from the base classes implement objects that are persistent or atomic, and thus exemplify the applicability of an existing specification method to specifying "nonfunctional" properties. Writing these formal specifications for Avalon/C++'s built-in classes has helped to clarify places in the programming language where features interact, to make unstated assumptions explicit, and to characterize complex properties of objects.

Index Terms—Atomicity, Avalon, C++, distributed systems, fault-tolerance, formal methods, Larch, object-oriented programming, specification, transactions.

I. INTRODUCTION

FORMAL specification languages have matured to the point where industry is receptive to using them and researchers are building tools to support their use. They have been used successfully for specifying the input-output behavior, i.e., functionality, of programs, but less often for specifying a program's "nonfunctional" properties. For example, the functionality of a program that sorts an array of integers might be informally specified as follows: given an input array A of integers, an array B of integers is returned such that B's integers are the same as A's, and B's are arranged in ascending order. Nothing is said about the program's performance, such as whether the algorithm for sorting should be \( O(n) \) or \( O(n^2) \). Performance is an example of a "nonfunctional" property. Other "nonfunctional" properties are degree of concurrency, reliability, safety, and security.

This paper demonstrates the applicability of formal specifications to the "nonfunctional" properties, persistence and atomicity. Atomicity, which subsumes persistence, requires that an object's state be correct in the presence of both concurrency and hardware failures. The correct behavior of these objects is fundamental to the correctness of the programs that create, access, and modify them.

Our basic approach is to use standard first-order logic to specify properties of atomic objects, and to specify an object's "nonfunctional" properties indirectly in terms of its functionally defined properties. This approach is similar to that taken in the work done on specifying a fault-tolerant flight control system [20], safety-critical nuclear control software [5], [23], and secure operating systems [2], [22], respectively, for the "nonfunctional" properties of fault-tolerance, safety, and security. As for these other examples, general-purpose methods are used to specify properties of specific systems. An alternative approach is to use or devise a method for a particular property, much like probability and queueing theory are used to model hardware reliability.

Section II describes in more detail a context in which atomic objects are used: fault-tolerant distributed systems. Sections III, IV, and V present a concrete programming language interface to such objects and formal specifications of this interface. Section VI summarizes the lessons learned from writing these specifications formally. The results are gratifying: they provide evidence that an existing specification method is suitable for describing a new class of objects; they validate the correctness of the design and implementation of a key part of a large software development project; and not surprisingly, they demonstrate that the process of writing formal specifications greatly clarifies our understanding of complex behavior. Finally, Section VII concludes with remarks about current and future work.

II. BACKGROUND

A. Abstract Context: Transaction Model of Computation

A distributed system runs on a set of nodes that communicate over a network. Since nodes may crash and communications may fail, such a system must tolerate faults; processing must continue despite failures. For example, an airline reservations system must continue servicing traveling agents and their customers even if an airline's database is temporarily inaccessible; an automatic teller machine must continue dispensing cash even if the link between the ATM and the customer's bank account is down.

A widely accepted technique for preserving data consistency and providing data availability in the presence of both concurrency and failures is to organize computations as sequential processes called transactions. A transaction is a sequence of operations performed on data objects in the system. For example, a transaction that transfers $25 from a savings account S to a checking account C might be performed as the following sequence of three opera-
tions on $S$ and $C$ (both initially containing $\$100$):
$$\{ S = \$100 \land C = \$100 \}$$

Read($S$)

Debit($S$, $\$25$)

Credit($C$, $\$25$)

$$\{ S = \$75 \land C = \$125 \}$$

In contrast to standard sequential processes, transactions must be atomic, that is serializable, transaction-consistent, and persistent. Serializability means that the effects of concurrent transactions must be the same as if the transactions executed in some serial order. In the above example, if two transactions, $T1$ and $T2$, were simultaneously transferring $\$25$ from $S$ to $C$, the net effect to the accounts should be that $S = \$50$ and $C = \$150$ (that is, as if $T1$ occurred before $T2$ or vice versa). Transaction-consistency means that a transaction either succeeds completely and commits, or aborts and has no effect. For example, if the transfer transaction aborts after the Debit but before the Credit, the savings account should be reset to $\$100$ (its balance before the transfer began). Persistence means that the effects of committed transactions survive failures. If the above transfer transaction commits, and a later transaction that modifies $S$ or $C$ aborts, it should be possible to "roll back" the state of the system to the previous committed state where $S = \$75$ and $C = \$125$.

Wehl probes that the (global) atomicity of the entire system is guaranteed if each object accessed within transactions is (locally) atomic [27]. An atomic object is an instance of an abstract data type with the additional property that it ensures the serializability, transaction-consistency, and persistence of all the transactions that use its operations. For example, if the bank account is represented by an atomic object, then any set of transactions that accesses the object is guaranteed to be serializable, transaction-consistent, and persistent. The advantage of constructing a system by focusing on individual objects instead of on a set of concurrent transactions is modularity: we need only ensure that each object is atomic to ensure the global atomicity of the entire system. Thus, we transform the problems of specifying, designing, implementing, and reasoning about an entire distributed system into the more manageable problems of specifying, designing, implementing, and reasoning about each of the objects in the system.

B. Concrete Context: Avalon

The Avalon Project, conducted at Carnegie Mellon University, provides a concrete context for this work. We have implemented language extensions to C++ [26], [8] and Common Lisp [25], [6] to support application programming of fault-tolerant distributed systems. Avalon relies on the Camelot System [24], also developed at CMU, to handle operating-system level details of transaction management, internode communication, commit protocols, and automatic crash recovery.

A program in Avalon/C++ 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.

Avalon/C++ includes a variety of primitives (not discussed here) for creating transactions in sequence or in parallel, and for aborting and committing transactions. Each transaction is identified with a single process (thread of control). Typically, a transaction executes by invoking an operation on an object (encapsulated by some server) receiving results when the operation terminates, then invoking another operation on a possibly different object (encapsulated by a possibly different server), receiving results when it terminates, etc. It then commits or aborts.

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. Each transaction has a unique parent, a (possibly empty) set of siblings, and sets of ancestors and descendants. A transaction is considered its own ancestor or descendant.

Avalon/C++ provides transaction semantics by requiring that all objects shared by transactions be atomic. The Avalon/C++ base hierarchy consists of three classes (Fig. 1), each of which provides primitives for implementors of derived classes to ensure the nonfunctional properties of objects of the derived classes. Programmers derive from either class atomic or class subatomic to define their own atomic objects. In practice, 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 nonatomic objects, those which guarantee only persistence. Programmers need only derive from class recoverable to define persistent objects.

In Avalon/C++ when a transaction commits, the runtime system assigns it a timestamp generated by a logical clock [18]. Atomic objects are required to ensure that all transactions are serializable in the order of their commit timestamps, a property called hybrid atomicity [27]. This property is automatically ensured by two-phase locking protocols [9], as obeyed by objects derived from class atomic. However, objects derived from class subatomic obtain additional concurrency by testing timestamp ordering at runtime. The key difference between class atomic and class subatomic is that class subatomic gives
programmers a finer-grained control over synchronization and crash recovery.

The main technical contribution of this paper is the formal specification of the interfaces of the three base Avalon/C++ classes, presented in Sections III, IV, and V. There are two purposes for writing this specification: to help Avalon/C++ programmers and to help Avalon/C++ implementors.

- **Clients** of these base classes need to know the effects of each of the classes’ exported operations in order to ensure correct usage. Instead of reading the code or in addition to reading informal commentary, clients can read these formal specifications and know what they must establish when invoking an operation and what is guaranteed to hold when it returns. Clients need not know at all how these effects are achieved.

Moreover, clients can use these specifications to reason, informally or formally, about their code that uses these interfaces. They can perform such reasoning locally and independently of the proof that the implementation satisfies the specification.

- **Implementors** of these base classes need to know what they must guarantee to their clients. They need not know who their clients are or why their clients will use these classes. They are free to focus on implementing these interfaces.

Implementors can also take these specifications as the “contract” they must satisfy. They prove their implementations satisfy the specifications of these classes independently of any of the classes’ uses.

These remarks are applicable in general to the use of any interface specification. For Avalon/C++, the most important audience is the clients. The base classes are implemented once and for all (unless there is to be more than one implementation of Avalon/C++ or a similarly designed language), but are used over and over again, through C++ inheritance, as the means to define new classes. By specifying the base classes’ interfaces, we provide the means for showing that classes derived from the base classes define objects that are persistent or atomic.

### C. Specification Language: Larch

The formal specification language used in this paper is Larch [16], though others such as VDM [4], Z [1], or OBJ [12], might also be suitable. Larch was designed to specify the functionality of sequential programs, in particular, properties of abstract data types. A Larch specification has two parts: 1) an interface, written in a predicative language using pre- and postconditions, describes the effects on program state as operations are executed (e.g., an object’s change in value or allocation of new storage); and 2) a trait, written in the style of an algebraic specification, describes intrinsic properties that are independent of the model of computation (e.g., elements in sets are unordered and not duplicated). The advantage gained in using Larch is this explicit separation of concerns between state-dependent and state-independent properties. Readers familiar with Larch can skip the next subsection and go to Section II-C-2 in which we add extensions to deal with concurrency. Two other papers in this issue also discuss aspects of Larch [15], [11].

1) **Overview of Larch/C++ and the Larch Shared Language:**

Larch interfaces describe the effects of a C++ class’s operations. For example, Larch/C++ interfaces for a constructor, insertion, and deletion operations are given for a C++ intset class shown in Fig. 2. Aside from the header, an operation’s interface specification can have three clauses: requires, modifies, and ensures.

A requires clause states the precondition that must hold when an operation is invoked. We interpret an omitted requires clause as equivalent to "requires true." None of the operations in the intset example have explicit requires clauses, which means they can be invoked in any state.

A modifies (object_list) clause asserts that an operation may possibly change the value of any of the objects listed in object_list; it is a strong indirect assertion about which objects may not change in value. This assertion is implicitly conjoined to the operation’s postcondition in the ensures clause. We borrow the reserved C++ symbol this to denote the object at which a class operation is invoked; as in C++ this is an implicit argument formal to each operation of a class. An omitted modifies clause is equivalent to the assertion modifies nothing, meaning no objects are allowed to change in value.

An ensures clause states the postcondition that the operation must establish upon termination. We use the reserved symbol return as an implicit result formal to denote the object returned as a result of executing an operation. An unprimed argument formal, e.g., this and e, in a postcondition stands for the value of the object when the operation begins. A primed argument formal, e.g., this', or primed result formal, i.e., return', stands for the value of the object at the end of the operation.

Finally, a new (object_list) predicate, which typically appears in a constructor’s postcondition, asserts that fresh storage is allocated for each object listed in object_list.

\(C++\)’s this denotes the pointer to the object; we use it to denote the object itself since we are almost always interested in the object being pointed to, not the pointer itself.
For example, one effect of calling the `inset` constructor operation is that an `inset` object that did not exist upon invocation now exists upon return. Its value is the empty set \( \{ \} \).

We use the vocabulary of traits to write the assertions in the pre- and postconditions of an object's operations; we use the meaning of equality to reason about its values. Hence, the meaning of `add` and `=` in `inset`'s postcondition is given by the `Set` trait of Fig. 3. In a trait, the set of operators and their signatures following `introduces` defines a vocabulary of terms to denote values. For example, \( \{ \} \) and `add` \( \{ \}, 5 \) denote two different `inset` values. The set of equational axioms following the `asserts` clause provides an equivalence relation on the terms, and hence on the values they denote. For example, from `Set`, we could prove that `rem` \( \{ \text{add}(\text{add}(\text{emp}, 3), 4), 3\} \) \( = \text{add}(\text{emp}, 4) \). The `generated by` clause of `Set` asserts that \( \{ \} \) and `add` are sufficient operators to generate all values of sets. Formally, it introduces an inductive rule of inference that allows us to prove properties of all terms of sort `S`. The `partitioned by` clause adds more equivalences between terms. Intuitively it states that two terms are equal if they cannot be distinguished by any of the operators listed in the clause. For example, sets are partitioned by `e` because sets are equal if and only if their members are the same: we can use this property to show that order of insertion of elements in a set does not matter.

A trait `T1` can include another trait `T2`, thereby adding to `T1` what appears in `T2`. Larch also supports renaming of sort and operator identifiers through for clauses. Our specifications of the Avalon/C++ classes will use the `Set` trait through trait inclusion, usually with some renaming of the sort identifiers `E` and `S`; we will also show uses of the subset operator \( \subseteq \) defined as the `Set` trait.

Further details of Larch are provided as necessary. See [17] for a more complete discussion.

2) Extensions for ConcURRENCY:

Following [3], we make three extensions to Larch interfaces for the transaction model of computation.

- A `when` clause states a condition on the state of the system that must hold before the operation proceeds. Specifying this condition is often necessary since the state of the system may change between the point of invocation (when the precondition must hold) and the actual point of execution of the operation (when the when-condition must hold).

  - It is implicit that each operation must be operation-consistent, that is, it completes entirely or has no observable effect. No intermediate states are observable between a state in which the when-condition holds and a state in which the postcondition holds. For an operation `op` that is a sequence of other operation-consistent operations `op`, that may be interleaved with operations of other transactions and have observable effects, we specify `op`'s effects as the composition of named operations `op`'s, each of which is specified as any operation-consistent operation. The only example of this kind of operation in this paper is the `pause` operation on `subatomic` objects found in Section V.

- `Self` is used to denote the transaction invoking the operation.

III. CLASS RECOVERABLE:

Conceptually, there are two kinds of storage for objects: volatile storage whose contents are lost upon crashes, and stable storage whose contents survive crashes with high probability. (Stable storage may be implemented using redundant hardware [19] or replication [7].) Recoverable objects are allocated in volatile storage, but their values are logged to stable storage so that recovery from crashes can be performed. If every recoverable object is written to stable storage after modifying operations are performed on it in volatile storage, then its state may be recovered after a crash. Recovering an object's state simply requires "replaying" the log, which is a sufficient method for recovering an object's state.

However, recovering an object's state entirely from the log is a time-consuming process. Camelot speeds up crash
recovery by dividing local storage into two classes, volatile storage and nonvolatile storage, and by distinguishing between two crash modes, node failures and media failures. In a media failure, both volatile and non-volatile storage are destroyed, while in a node failure, only volatile storage is lost. In practice, node failures are far more common than media failures. To optimize recovery from node failures, a protocol known as write-ahead logging [14] is used. An object is modified in the following steps:

1) The page(s) containing the object are pinned in volatile storage; they cannot be returned to nonvolatile storage until they are unpinned.

2) Modifications are made to the object in volatile memory.

3) The modifications are logged on stable storage.

4) The page(s) are unpinned.

The first step of the protocol ensures that the pages containing the object are not written to nonvolatile storage while a modifying operation is in progress. This protocol ensures that a recoverable object can be restored to a consistent state quickly and efficiently. Upon crash recovery, the status of each transaction is determined, and by comparing what is in nonvolatile storage to what is in stable storage, we can "redo" the effects of committed transactions and "undo" the effects of aborted ones. (For more details, see [24].) Notice the modifications must still be logged to stable storage to protect against the occurrence of a media failure.

A. Avalon Class Definition

The programmer's interface to a recoverable object is through the Avalon/C++ class header shown in Fig. 4.

Informally, the pin operation causes the pages in volatile storage containing the object to be pinned; unpin causes the modifications to the object to be written to stable storage, and unpins its pages. A recoverable object must be pinned before it is modified, and unpinned afterwards. For example if \( x \) is a recoverable object, a typical use of the pin and unpin operations within a transaction would be:

```c
start { //begin transaction
  ...
  x.pin();
  //modify x here
  x.unpin();
  ...
}; //end transaction
```

After a crash, a recoverable object is restored to a previous state in which it was not pinned. Transactions can make nested pin calls; if so, then the changes made within inner pin/unpin pairs do not become permanent, i.e., written to stable storage, until the outermost unpin is executed. Classes derived from recoverable inherit pin and unpin operations, which can be used to ensure persistence of objects of the derived class.

```c
class recoverable {
  public:
    void pin(); // Pines object in volatile storage.
    void unpin(); // Unpins and logs object to stable storage.
}
```

Fig. 4. Avalon class recoverable.

B. Larch Specification

The specification shown in Fig. 5 captures the following three properties of recoverable objects:

1) Only one transaction can pin an object at once.

2) The same transaction can pin and unpin the same object multiple times.

3) Only at the last unpin does the object's value get written to stable storage.

We now walk through the specification in detail. The top part of the specification contains Larch interface specifications for a constructor given by the class name, recoverable, and the two operations, pin and unpin. The bottom part contains the Larch trait RecObj, which gives meaning to the assertion language of the interface specifications.

We see in RecObj that the state of a recoverable object is a triple of its value in memory, a single transaction identifier, and a pin count:

```c
R tuple of value: Memory, pinner: Tid, count: Card
```

We use the built-in tuple schema, given in Appendix II, to introduce the sort \( R \) for terms denoting states of recoverable objects.

Memory itself is modeled as a pair of values, one each for volatile and stable storage:

```c
Memory tuple of volatile: M, stable: M
```

Now let us turn to the three operations defined for class recoverable. The constructor's postcondition ensures that new storage is allocated for the returned object and initializes its pin count to zero.

Pin's postcondition specifies how the state of a recoverable object changes. Pin might terminate with an error condition signaled to the invoker to indicate that the object to be pinned is already pinned by some other transaction. We use the reserved signal to denote the object whose value ranges over an enumeration of error conditions.5 Pin's postcondition makes use of the auxiliary function, pn, defined in the trait RecObj:

```c
pin(r, t) <=
if r.count > 0
  then if r.pinner = t
    then count gets(r, r.count + 1)
  else r
else count gets(pinner gets(r, t), 1)
```

It takes a recoverable object's state (of sort \( R \)) and a transaction identifier (of sort Tid ) and returns a (new) state for

5Using signal is suggestive of exception handling, which is not supported in C++.
class recoverable based on R from RecObj
recoverable()
ensures new (return) \& return:count = 0
void pin()
modifies this\(->\)pin,
ensures this\(->\)pin \& signal \& this\(->\)pin \& count \& already_claimed
void unpin()
requires pinned(this) \& this\(->\)pin \&= self
modifies this\(->\)pin \&= self
ensures this\(->\)pin = un(this, self)
RecObj: trait
includes
R tuple of: value: Memory, pin: Tid, count: Card
Memory tuple of volatile: M, stable: M
introduces
pin: R, Tid \rightarrow R
un: R, Tid \rightarrow R
pin: R \rightarrow Bool
asserts for all (r, t, ti: Tid, m: Memory, c: Card)
\[\text{\textit{pin}}(t, t) =
\begin{cases}
\text{\textit{true}} & \text{if } r \text{\textit{count}} > 0 \\
\text{false} & \text{otherwise}
\end{cases} \]
\[\text{\textit{un}}(m, t, c, t) =
\begin{cases}
\text{\textit{false}} & \text{if } c = 0, t = 1 \\
\text{\textit{false}} & \text{if } c = 1, t = 1 \\
\text{\textit{true}} & \text{otherwise}
\end{cases} \]
\[\text{\textit{pin}}(r) = \text{\textit{true}} \] Fig. 5. Larch specification of class recoverable.

a recoverable object. If the count \((r.count)\) is nonzero, then the object must be pinned. If the object is pinned by a transaction \((r.pinner)\) that is the same as the transaction \((t)\) attempting to pin the already pinned object, then the count is incremented; otherwise, the object is left unchanged. If the object is not already pinned, then its state is initialized with the pinning transaction’s identifier and a count of 1.

\textit{Unpin}’s preconditions requires that an object not be un-­pinned unless it is already pinned; moreover it must be pinned by the calling transaction. \textit{Un} is defined as follows:

\[\text{\textit{un}}(m, t, c, t) =
\begin{cases}
\text{\textit{false}} & \text{if } c = 1 \\
\text{\textit{true}} & \text{otherwise}
\end{cases} \]

Unlike \textit{pin}, it is unnecessary for \textit{un} to check if the object is already pinned and if the transaction \((t)\) that currently has the object pinned is the same as the unpinning transaction \((t)\); \textit{unpin}’s preconditions checks for this case. \textit{Un} simply checks if there is only one outstanding call to pin \((c = 1)\), in which case the value of the object in volatile storage is written to stable storage; otherwise, the count is decremented. We defer discussion of this asymmetry between \textit{pin} and \textit{unpin} to Section VI.

Note that \textit{pin} and \textit{unpin} each has a \texttt{modifies} clause, indicating that \texttt{this}, but no other object, may be modified.

C. Deriving from Class Recoverable
A typical use of class \textit{recoverable} is to define a derived class for objects that are intended to be persistent. For example, suppose we derive a new class, \textit{rec_int}, from \textit{recoverable}:

class rec_int: public recoverable{
  // private representation
public:
  // operations on recov_int
}

If \texttt{Int} is the sort identifier associated with values of recoverable integer objects, then the identifier \(M\) that appears in the \texttt{RecObj} specification would be renamed with \texttt{Int}. The header for the Larch interface specification for the \texttt{rec_int} class would look like:

class rec_int based on R from RecObj (Int for M)
  // . . . specification of recov_int’s operations . . .

IV. Class Atomic

The second base class in the Avalon/C++ hierarchy is \textit{atomic}. Atomic is a subclass of recoverable, specialized to provide two-phase read/write locking and automatic recovery. Locking is used to ensure serializability, and an automatic recovery mechanism for objects derived from \textit{atomic} is used to ensure transaction-consistency. Persistence is “inherited” from class \textit{recoverable} since \textit{pin} and \textit{unpin} are inherited through C++ inheritance.

A. Avalon Class Definition

Fig. 6 gives the class header for \textit{atomic}.

Atomic objects should be thought of as containing long-term locks. Under certain conditions, \texttt{read_lock} (\textit{write_lock}) gains a read lock (write lock) for its caller. Transactions hold locks until they commit or abort. \texttt{Read lock} and \texttt{write_lock} suspend the calling transaction until the requested lock can be granted, which may involve waiting for other transactions to complete and release their locks. If \texttt{read_lock} or \texttt{write_lock} is called while the calling transaction already holds the appropriate lock on an object, it returns immediately.

B. Larch Specification

Fig. 7 gives the Larch interfaces and trait for class \textit{atomic}. As indicated in the trait \texttt{AtomObj}, an atomic object is a recoverable object, along with a set of transactions that hold read locks on the object and a set of transactions that hold write locks on it:

A tuple of \texttt{ob}: \texttt{R}, \texttt{readset}: Readers, \texttt{writeSet}: Writers

Even though only one writer can be modifying the state of an atomic object at once, we keep track of a set of transactions with write locks because a child transaction can get a write lock if its parent has one. The constructor for \textit{atomic} initializes both the sets of readers and writers to be empty.

The transaction tree \(ts\) of type \texttt{tidTree} is global information:

class tidTree based on TransIdS from TransIdTree
  // . . . TransIdTree defined in Appendix I . . .
global ts: tidTree
As for the previous `recov_int` example, when giving the
Larch interface specification for the `atomic_int` class, we
rename the sort identifier `M`, introduced in the `RecObj` trait
and included in the `AtomObj` trait:

```larch
class atomic_int based on A from AtomObj

atomic() ensures new (return) \land return'.readset = \{\} \land return'.writeiset = \{

void read_lock() when this.writeiset \subseteq ancestors(ts, self) modifies (this)
ensures this = add_reader(this, self)

void write_lock() when this.readset \subseteq ancestors(ts, self) \land this.writeiset \subseteq ancestors(ts, self)
modifies (this)
ensures this' = add_writer(this, self)
```

The second conjunct of the postcondition makes the cli-
mactic point: the value (of sort Int) of the returned item
object `return` is the *volatile storage's value* (of sort Int) of
the recoverable object component of the `atomic_int` `this`.
We retrieve the value from volatile storage because we
can assume the when-condition: if the object is pinned
(but not yet unpinned by `self`), then we want `this`'s most
recent value; if the object is unpinned, then the values in
volatile and stable storage would be identical.

Let us examine how the derived class uses the inherited
operations, relying on their specifications. An Avalon/C++
implementor of `atomic_int` can use `write_lock` and
`read_lock` of class `atomic` and `pin` and `unpin` of class
`recoverable` to ensure the serializability, transaction-consis-
tency, and persistence of `atomic_ints`. (Thus, `atomic_int`
class's clients can assume these properties hold for all
`atomic_ints`.) For example, here is how the coercion
operation would be implemented in Avalon/C++:

```larch
atomic_int::operator int() {
    read_lock(); // get read lock on
    // representation object
    return val; // return its value
}
```

Using the specification of class `atomic`'s `read_lock` opera-
tion, we can show 1) the coercion operation's when-condi-
tion trivially implies `read_lock`'s when-condition; and 2) `read_lock`'s postcondition guarantees the calling trans-
action has a read lock on the `atomic_int` object. These two
properties imply that `val`, the int representation of an
`atomic_int` will not be read and returned until the calling
transaction obtains a read lock on the `atomic_int` and,
moreover, no concurrent transactions have locks on it.

V. CLASS SUBATOMIC

The third, and perhaps most interesting, base class in
the Avalon/C++ hierarchy is *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, `sub-
atomic` provides more complex primitives to give pro-
grammers more detailed control over their objects' synchronization and recovery mechanisms by exploiting typespecific properties of objects. For example, a queue object with enqueue and dequeue operations can permit enqueuing and dequeuing transactions to go on concurrently, even though those transactions are both "writers." In defining an atomic queue class by deriving from class atomic, such concurrency would not be possible; deriving from class subatomic makes it possible. See 8 for details and other examples.

A. Avalon Class Definition

Fig. 8 gives the class header for subatomic.

A subatomic object must synchronize concurrent accesses at two levels: short-term synchronization ensures that concurrently invoked operations are executed in mutual exclusion, and long-term synchronization ensures that the effects of transactions are serializable. Short-term synchronization is used to guarantee operation-consistency of objects derived from subatomic.

Subatomic provides the seize, release, and pause operations for short-term synchronization. Each subatomic object contains a short-term lock, similar to a monitor lock or semaphore. Only one transaction may hold the short-term lock at a time. The seize operation obtains the short-term lock, and release relinquishes it. Pause releases the short-term lock, waits for some duration, and reacquires it before returning. Thus, these operations allow transactions mutually exclusive access to subatomic objects.

Seize, release, and pause are protected members of the subatomic class since it would not be useful for clients to call them. 6 To ensure transaction-consistency, subatomic provides commit and abort operations. Whenever a top-level transaction commits (aborts), the Avalon/C++ runtime system calls the commit (abort) operation of all objects derived from subatomic accessed by that transaction or its descendants. Abort operations are also called when nested transactions "voluntarily" abort. Since commit and abort are C++ virtual operations, classes derived from subatomic are expected to reimplement these operations. Thus, subatomic allows type-specific commit and abort processing, which is useful and often necessary in implementing user-defined atomic types efficiently.

B. Larch Specification

Figs. 9 and 10 give the Larch interfaces and trait for class subatomic. As indicated in the trait SubAtomObj, a subatomic object is a recoverable object, along with the transaction holding the short-term lock, and a set of transactions that are waiting to acquire it.

S tuple of ob: R, locker: Tid, waiters: Waitset

Initially, as specified in the constructor, no one holds the short-term lock on the object.

6 That is, only implementors of a new class derived from subatomic need to call seize, release, and pause—when implementing the operations of the new class, clients of the new class, however, never need to call seize, release, or pause explicitly.

Fig. 8. Avalon subatomic class.

Fig. 9. Larch specification of class subatomic (interfaces).

Fig. 10. Larch specification of class subatomic (trait).

Seize's when-condition states that a transaction must wait until no transaction holds the short-term lock on the object before acquiring the lock. The postcondition states that the calling transaction obtains the short-term lock on the object, and the object is now locked.
Release's precondition requires that the calling transaction be the one that has the lock on the object and that the object be locked. The postcondition states that either the object is no longer locked or if some other transaction is waiting to obtain the lock, it is given the lock.

Pause’s precondition is similar to release’s. The rest of its specification, however, is unlike all the others. Pause’s effects are specified in terms of the sequential composition of two operations, each of which can be interleaved with operations of other transactions. First, pause relinquishes the short-term lock as release does. However, relinquish’s postcondition differs from release’s in one critical way: the calling transaction is added to the waiting set of transactions upon relinquishing the lock. The second operation in the sequence, reacquire, is delayed until either some other transaction has released the lock and given it back to self or no one has a lock on the object at all. Its postcondition ensures that the original caller of pause again possesses the short-term lock upon return.

The specifications of commit and abort deserve special attention. Each is called with a trans_id argument denoting some transaction that has committed (aborted) in the given (global) transaction tree tx. The implicit "modifies nothing" assertion states that no change to the object is allowed. This seemingly strong assertion reflects the intention that commit and abort operations are to have only "benevolent" side effects on the object’s state, meaning that the abstract state of the object remains the same, though the representation state may change. Indeed, typical Avalon/C++ implementations of commit operations simply discard redundant state information stored in the representation object, not affecting the abstract state at all. Typical implementations of abort operations use this redundant state information to undo tentative changes performed by the aborting transaction (and any of its descendants that have committed with respect to it).

Deriving from class subatomic is similar to deriving from class recoverable or class atomic and is omitted for brevity.

VI. OBSERVATIONS

A. About Avalon

The exercise of formally specifying the Avalon/C++ classes revealed unstated assumptions about the actual implementation and made Avalon/C++'s fundamental semantics more precise.

One unstated assumption in the underlying operating system (Camelot) is reflected in the implementation, but was never made explicit until we wrote the formal specification for class recoverable. The Avalon/C++ implementation precludes the possibility of concurrent pins to the same object by different transactions; Camelot forbids this situation because it assumes that any transaction that pins an object intends to modify it. This assumption is one example of where crash recovery and concurrency cannot be separated when reasoning about Avalon programs. Without concurrency, we can give a meaning to persistence; without crash recovery, we can give a meaning to the correct synchronization of processes. But to support both, there are points where we must consider both persistence and synchronization together.

Another kind of unstated assumption discovered by writing this specification is implicit preconditions. For example, whereas pin has no precondition, unpin does. This asymmetry in the specifications reflect the asymmetry that exists in the actual implementation. An earlier version of the specification of unpin did not have a precondition, but not until the implementor was shown this (incorrect) version was the unstated precondition revealed. In fact, upon seeing the asymmetry in the current version of the specification, the implementor realized that the precondition on unpin could easily be removed by performing a run-time check, as is already done for pin, and signaling an exception instead. In general, there is a tradeoff between imposing a precondition on the caller and implementing a run-time check; if it can be proven that all uses of an operation are guaranteed to satisfy an implicit precondition, no run-time check is needed.

Specifying the class atomic helped make the rules for obtaining long-term locks more precise. It also made explicit, by modeling a set of writers, not just a single writer, the property that more than one transaction might hold a long-term write lock on an object at once. Recall this situation can arise because of nested transactions. On the other hand, the specification of class subatomic made explicit that only one transaction (the locker) can have the short-term mutual exclusion lock on an object at once.

Specifying the class subatomic helped identify a subtle source of a potential deadlock situation. As specified in Fig. 9, if there are waiters, pause will not return until some transaction, tid, other than the calling one, self, grabs the short-term lock and returns, thereby releasing the lock. If tid does not return (perhaps it is waiting for some synchronization condition to become true), then self will not be able to return since it will be unable to reacquire the lock. In fact, this situation can arise in the current Avalon/C++ implementation and was discovered only through trial and error when debugging some simple examples. Had we done the specification beforehand, we could more easily have anticipated this problem.

B. About Larch

Since Larch uses a two-tiered approach, people often ask "What goes in the interfaces and what goes in the traits?" There are some guidelines, but no hard-and-fast rules; decisions are based purely on taste and readability. An interface must specify the post- and postconditions of an object's operation. Assertions in these conditions determine what operator symbols to use in defining under the traits. At the same time, traits are meant to be as abstract and general as possible, and not necessarily specific to a particular interface. For any given Larch specification,

\footnote{The astute reader may have noticed that no's second argument, a vestige of the earlier specification, was ignored in its definition; if the precondition for unpin is removed, then the second argument is necessary.}
there are some traits that are general, e.g., for specifying sets, groups, and partial orders, and hence, reusable by not just different interfaces but also by different interface languages (e.g., the Set trait we used is equally useful for a Larch/Ada interface language). There are also traits that will be closely tied to a particular interface, where we intentionally choose to be less general than possible, e.g., omitting an explicit check in the definition of un in the RecObj trait since the precondition for the unpin operation makes it unnecessary to do so.

Following this traditional spirit of Larch for the Avalon/C++ example, we relegated most of the complexity of a specification to the traits. The rule-of-thumb is: If the predicates in the pre- and postconditions become unwieldy then introduce a trait operator to capture the intended property. However, one place where that cannot easily be done is in specifying nondeterminism. Since traits define (deterministic) functions, interfaces are responsible for specifying nondeterministic behavior. For example, the use of the existential quantifier in the postconditions of release and pause is unavoidable.

Not surprisingly, Larch needed to be extended to deal with concurrency, as exemplified here for Avalon/C++ and in [3] for Modula-2+. The two most important extensions are: 1) the need to specify an operation’s effects through the specification of a sequence of other operations, and 2) the when clause used for stating a third kind of condition in addition to pre- and postconditions. As an aside, this when-condition influenced the Avalon/C++ designers who added a when statement to the language. This statement, which makes appropriate calls to seize, release, and pause, is akin to a conditional critical region.

One critical class of properties that cannot be stated in Larch/C++, even as currently extended, is liveness. For example, we cannot say that an object’s commit or abort operation will eventually be called. Unfortunately, many programs may be correct with respect to safety but can deadlock or livelock in practice. In particular, typical implementations of operations of classes derived from subatomic test at run-time whether some transaction has committed; obtaining the short-term lock often requires that this test succeed. So, sometimes no progress can be made until some transaction has committed. We have seen in the previous section that deadlock may arise in the implementation, and how the specification permits this behavior. Although Larch/C++ was never intended to address liveness properties, such properties are important for practical reasons, especially in the context of concurrent transactions.

VII. Final Remarks

The specifications presented here represent ongoing work. Though the specification of Avalon/C++ is incomplete, we have specified a critical piece of it since all user-defined classes derive from the built-in ones. Knowing that a fundamental part of Avalon/C++’s semantics is implemented correctly is a reassurance to us as Avalon implementors as well as to all Avalon programmers.

Further work on the specification of Avalon/C++’s intricacies would include: 1) Avalon/C++’s transaction model of state, which must include two kinds of store, volatile and stable. It must also include the entire transaction tree, the status of each transaction in the tree, and the sets of locks each transaction holds. 2) System-wide commit and abort operations, which must be defined on behalf of a transaction committing or aborting. For example, the system-wide commit operation would take a transaction identifier and a timestamp, modifying the status of some transaction in the transaction tree. 3) A system-wide recover operation, which would define the effects of recovering from a crash. We would need to modify the specification for a recoverable object by keeping track of the entire history of operations performed on it in order to capture the set of possible values such an object can have [28]. 4) Avalon/C++’s built-in class, trans_id, which has operations for creating transaction identifiers and testing whether two transactions are serialized with respect to each other. Appropriate trait functions would be added to the trait TransIdTree of Appendix I to facilitate the specification of trans_id.

As we generate specifications, we also would like to prove theorems about the objects being specified. For example, from the specification in Figs. 9 and 10 we can prove that the transaction (tid) given the lock upon return from release is different from the calling transaction (self). The proof of this property depends on the following property of subatomic objects: (∀ s: S) locked(s) ⇒ (s.locker ≠ s.waiters).

We have used the Larch Prover [10], whose input is similar to Larch traits, to prove the correctness of a nontrivial implementation of a highly concurrent FIFO queue [29], [13]. The queue derives from class subatomic and we proved it satisfies the hybrid atomicity property, required of all Avalon objects.

The results of this paper should be of interest to both the fault-tolerant distributed systems community and the formal methods community. For the former, our specifications are a first attempt at formally specifying interfaces to a transaction-based programming language. Properties of transactions have never before been studied from a formal specification viewpoint.

For the formal methods community, especially those interested in formal specifications, we close with a summary of this paper’s contributions:

1) Larch is grounded in standard first-order predicate logic with equality. We showed how to use Larch to specify indirectly some ‘nonfunctional’ properties of an object, persistence and atomicity, through its functionally defined properties. Our formal specification approach complements, but does not replace, alternative approaches, including informal methods and property-specific techniques (e.g., hazard analysis).

2) Unlike most other literature on Larch, we have focused on interfaces, not traits. In particular, we have informally introduced a Larch/C++ interface language and Larch interface language extensions for dealing with con-
currency. We illustrated how to use Larch by specifying interfaces of Avalon/C++ modules, thereby providing a basis for reasoning about Avalon/C++ programs. People who write software reference manuals can write similar stylized specifications for their software interfaces.

3) We added to a small, but growing, set of nontrivial specification case studies. These examples show that the process of formalization can reveal a better understanding of what is being specified, in our case, Avalon/C++ base classes and more generally, the Avalon/C++ programming language itself. Writing these formal specifications made unstated assumptions explicit and helped clarify places in the language where features interact.

APPENDIX I

TRANSACTIONS AND THE TRANSACTION TREE

Below is a Larch trait that specifies a transaction’s state. We assume the existence of a \textit{TimeStamp} trait used for generating timestamps of sort \textit{Time}, and a \textit{UniqiuedId} trait used for generating unique identifiers of sort \textit{Id}. A transaction can be either committed, active, or aborted. Only committed transactions are given timestamps.

\begin{verbatim}
TidStatus: trait
    includes TimeStamp
    introduces
    co: Tid \rightarrow \textit{S}
    ac: \rightarrow \textit{S}
    ab: \rightarrow \textit{S}
    asserts \textit{S} generated by (co, ac, ab)

TransId: trait
    includes TidStatus, UniqiuedId
    Tid tuple of name: Id, status: S
    introduces
    create: Id \rightarrow Tid
    commit: Tid, Time \rightarrow Tid
    abort: Tid \rightarrow Tid
    t_committed: Tid \rightarrow Bool
    t_aborted: Tid \rightarrow Bool
    asserts for all (t, Tid, id, ti, ti: Time)
        create(id) == [id, ac]
        commit((id, ac), ti) == [id, co(ti)]
        abort((id, ac), ti) == [id, ab]
        abort((id, ab), ti) == [id, ab]
        t_committed(t) == t.status \neq ac \land t.status \neq ab
        t_aborted(t) == t.status = ab
    implies
        forall (id: Id, ti, ti: Time)
            converts (create, commit, abort, t_committed, t_aborted)
        exempting (commit((id, ab)), commit((id, co(ti), ti1), abort((id, co(ti)))))

TransIdTree: trait
    includes TransId, Tree (Tid for N, TransIds for T)
    introduces
        committed: TransIds, Tid \rightarrow Bool
        aborted: TransIds, Tid \rightarrow Bool
    asserts for all (ts: TransIds, t: Tid)
        committed(ts, t) == \exists i \in ts \land t_committed(t)
        aborted(ts, t) == \exists i \in ts \land t_aborted(t)

Tree: trait
    includes Set (N for E, Nodes for S)
    introduces
        emp: \rightarrow T
        add_node: T, N \rightarrow T
        add_branch: T, T \rightarrow T
        root: T \rightarrow N
        _e : T, N \rightarrow Bool
        des: T, N, N \rightarrow Bool
        ancestors: T, N \rightarrow Nodes
    asserts
        T generated by (emp, add_node, add_branch)
        for all (t, t1, t2: T, n, n1, n2: N)

\end{verbatim}
APPENDIX II

Tuples

Tuples are a shorthand for a trait defined as follows. For each tuple of the form

\[ \text{S tuple of } f_1, f_2, \ldots, f_n \text{: } S_n \]

Append to the function declarations of the enclosed trait:

\[
\begin{align*}
\text{introduces} \\
\{ - - \} : S_1, \ldots, S_n \rightarrow S \\
\text{- -} : S \rightarrow S \\
f_i \text{- - gets: } S, S_i \rightarrow S
\end{align*}
\]

for \( 1 \leq i \leq n \).

Append to the set of equations of the enclosing trait:

\[
\begin{align*}
\text{asserts} \\
\text{S generated by } \{ \cdot - \} \\
\text{S partitioned by } \{ f_1, \ldots, f_n \} \\
\text{for all } (x_1, y_1 ; S_1, \ldots, x_n, y_n ; S_n) \\
[x_1, \ldots, x_i, \ldots, x_n, f_i = x_i] \\
f_i \text{- - gets} [x_1, \ldots, x_i, \ldots, x_n, y_i] = \\
[x_1, \ldots, y_i, \ldots, x_n]
\end{align*}
\]

for \( 1 \leq i \leq n \).

ACKNOWLEDGMENT

Discussions with J. Guttag and J. Horning and the examples given in [3] inspired my on-the-fly interface language design, in particular the Larch extensions for concurrency. C. Gong and R. Lerner helped check the specifications. I am grateful to all members of the Avalon group, in particular, M. Herlihy and D. Detlefs, who helped design Avalon/C++, and D. Detlefs who was instrumental in building it. Finally, I thank G. Leavens, J. Horning, and the anonymous referees for their comments on this paper.

REFERENCES


Jeanette M. Wing (S’76–M’78) received the S.B., S.M., and Ph.D. degrees in computer science from the Massachusetts Institute of Technology, Cambridge.

She is an Associate Professor of Computer Science at Carnegie Mellon University, Pittsburgh, PA. Her research interests include formal specifications, programming languages, concurrent and distributed systems, visual languages, and object management. She continues to contribute to the design of the Larch family of specification languages and, among other research projects, directs the Avalon work at Carnegie Mellon.

Dr. Wing is a member of the Association for Computing Machinery.