On the use of Specification Styles in the Design of Distributed Systems

Chris A. Vissers, Giuseppe Scollo, Marten van Sinderen, Ed Brinksma

University of Twente, Fac. Informatics
7500 AE Enschede, NL

Abstract

The paper uses the term architecture to denote an abstract object that is defined in terms of a set of requirements for a product and that is used to derive various concrete objects, called implementations, from it. It is assumed that an architecture is expressed in a formal description language. The paper argues that in practice any architecture of more than elementary complexity, and thus its formal description, needs to be structured to keep it comprehensible and to efficiently express its functionality. This structuring introduces implementation oriented elements in the architecture, despite the fact that in principle the architecture should be implementation independent: i.e. it should be just a definition of the abstract object's external functionality.

The necessity of structuring formal descriptions implies a de facto responsibility of the architect for the equality of the implementation. To exploit this responsibility the architect should should obey qualitative design principles like ortogonality, generality, generality and open-endedness. Substanctial experience with the development of distributed system architecture has shown, however, that the task of finding an appropriate structure is seriously underestimated in terms of complexity and time required.

The paper, therefore, advocates the use of well defined specification styles that allow to structure formal specifications and can be used at advantage to pursue qualitative design principles. The establishment of common and related specification styles is also considered paramount to preserve homogeneity of large specifications developed by teams of specifiers. Such specification styles would enable the designer to control better the design trayectory and thus to produce higher quality designs in shorter timescales.

The paper introduces a monolithyc, a constraint oriented, a state oriented, and a resource oriented specification style. For each style it is globally indicated which design objectives may be supported. The styles can be used to support the complete design trayectory from architecture to implementation, as is also formally demonstrated by a verification exercise in the context of a simple example. The formal background of the constraint oriented style is explained in more detail. The style defines an object in terms of a set of constraints, each of which can be chosen so as to correspond closely to a natural element of human reasoning about the behaviour of an object. The constraint oriented style, therefore appears to be very effective in initial design phases when requirements capturing is the dominant objective.

The reasoning in the paper is using the specification language LOTOS to convey ideas and to present examples, but can be applied to other languages that support this styles.

Specifying Avalon Objects in Larch

Jeannette M. Wing[1]

Invited Paper

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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 "non-functional" properties. Writing these formal specifications for Avalon/C++'s built-in classes has helped clarify places in the programming language where features interact, make explicit unstated assumptions, and make precise complex properties of objects.

1. 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. People use these languages for specifying the input-output behavior, i.e., *functionality*, of programs, but have largely ignored specifying a program's "non-functional" 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 like whether the algorithm for sorting should be $O(n)$ or $O(n^2)$. Performance is one example of a non-functional property. Other non-functional properties are degree of concurrency, reliability, and security.

In this paper, we demonstrate the applicability of formal specifications to the non-functional 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. Section 2 describes in more detail the context in which atomic objects are used: fault-tolerant distributed systems. Sections 3, 4, and 5 present a concrete programming language interface to such objects and the formal specifications of this interface.

Section 6 summarizes the lessons learned from writing these specifications out formally. The results are extremely 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 an ongoing software development project; and not surprisingly, they demonstrate that the process of writing formal specifications greatly clarifies one's understanding of complex behavior. Finally, Section 7 concludes with remarks about current and future work.

[1]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 and 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 US Government.

2. Background

2.1. 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 travel 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 operations on S and C (both initially containing \$100):

$$\{S = \$100 \wedge C = \$100\}$$

Read(S)
Debit(S, \$25)
Credit(C, \$25)

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

In contrast to standard sequential processes, transactions must be *atomic*, that is serializable, transaction-consistent, and persistent.[2] *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.

It can be shown [17] that the atomicity of the entire system is guaranteeed if each object accessed within transactions is *atomic*. That is, each object is an instance of an abstract data type with the additional requirement that it must ensure the serializability, transaction-consistency, and persistence of all the transactions that use its operations. For example, as long as the bank account's Read, Debit, and Credit operations are implemented "correctly," then any set of transactions that access the account will 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: one need only ensure that each object is atomic to ensure the more global atomicity property of the entire system.

[2]Unfortunately, no standard terminology is used for the terms *transaction-consistent* and *persistent*. Transaction-consistent is sometimes called *failure atomic*, *total*, or simply *atomic*. Persistent is sometimes called *recoverable*, *permanent*, or *resilient*. In this paper, we use terminology consistent with Avalon terminology as published in [5].

[3]Again, we use terminology from [5], but more standard terminology would call this property *atomic*, as in *atomic operation* or *atomic action* [2]. Since we use "atomic" for transactions, we needed to introduce a different term. Note that since a transaction is a sequence of operations, operation-consistency is a weaker property than transaction-consistency; it permits the partial effects of aborted transactions to be observed, while transaction-consistency does not.

2.2. Concrete Context: Avalon

The Avalon Project [5], conducted at Carnegie Mellon University, provides a concrete context for this work. We have implemented language extensions to C++ [16] to support application programming of fault-tolerant distributed systems. Avalon relies on the Camelot System [15], also being developed at CMU, to handle operating-systems level details of transaction management, inter-node communication, commit protocols, and automatic crash recovery.

A program in Avalon consists of a set of *servers*, each of which encapsulates a set of objects and exports a set of *operations* and a set of *constructors*. A server resides at a single physical node, but each node 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).

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 (Figure 2-1), each of which provides primitives for implementors of derived classes to ensure the "non-functional" properties of objects of the derived classes. Programmers derive from either class *atomic* or class *subatomic* to define their own atomic objects.

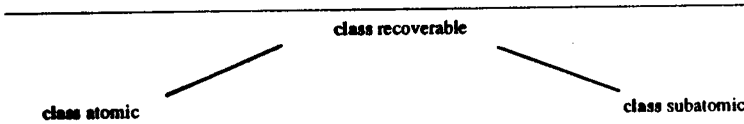

Figure 2-1: Inheritance Hierarchy of the Three Avalon/C++ Base Classes

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 non-atomic 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 run-time system assigns it a timestamp generated by a logical clock [12]. Atomic objects are expected to ensure that all transactions are serializable in the order of their commit timestamps, a property called *hybrid atomicity* [17]. This property is automatically ensured by two-phase locking protocols [6], used by objects derived from class *atomic*. However,

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.

```
class recoverable based on R from RecObj

  recoverable() returns (recoverable x)
     post x'.count = 0 ∧ new x

  pin(recoverable x) signals (already_claimed)
     modifies x
     post x' = pn(x, self) ∧
          x.pinner ≠ self ⇒ signal already_claimed

  unpin(recoverable x)
     pre  pinned(x) ∧ x.pinner = self
     modifies x
     post x' = un(x, self)


RecObj: trait
  includes
    Triple (R for T, Memory for T1, Tid for T2, Card for T3,
          value for .first, pinner for .second, count for .third)
    Pair (Memory for T, M for T1, M for T2, volatile for .first, stable for .second)
  introduces
    pn: R, Tid → R
    un: R, Tid → R
    pinned: R → Bool
  asserts for all (m: Memory, m1, m2: M, c: Card, t1, t2: Tid)
    pn(<m, t1, c>, t2) =
       if c > 0                           // is already pinned?
       then if t1 = t2                    // by same transaction
              then <m, t1, c+1>           // increment count
              else <m, t1, c>             // otherwise, leave unchanged
       else <m, t2, 1>                    // initialize it
    un(<<m1, m2>, t1, c>, t2) =
       if c = 1                           // if last unpin
       then <<m1, m1>, t1, 0>             // write to stable storage
       else <<m1, m2>, t1, c-1>           // or just decrement count
    pinned(r) = r.count > 0
```

Figure 3-2: Larch Specification of Class Recoverable

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*.[4] The bottom part contains the Larch trait *RecObj*, which gives meaning to the assertion language of the interface specifications.

We see in *RecObj* that a recoverable object is "modeled" as its value in memory, a single transaction

[4]We depart from C++ syntax for an operation's header by explicitly listing as an argument to each operation the implicit C++ object *this*, and listing the returned object and its type in a **returns** clause.

identifier, and a pin count:

```
Triple (R for T, Memory for T1, Tid for T2, Card for T3,
      value for .first, pinner for .second, count for .third)
```

Here we include the trait *Triple*, defined in Appendix II, and rename (through the for clauses) the sort and functions identifiers it introduces. For example, *R*, the sort identifier introduced for recoverable objects, renames the sort identifier *T* introduced in *Triple*. Memory itself is modeled as a pair of values, one each for volatile and stable storage:

```
Pair (Memory for T, M for T1, M for T2, volatile for .first, stable for .second)
```

The constructor's post-condition initializes the pin count to be zero and ensures that new storage is allocated for the returned object. An omitted pre-condition is interpreted as equivalent to **pre true**. Thus, the constructor can be called in any state.

Pin's post-condition specifies how the state of a recoverable object changes: *x* stands for the object's initial state (upon invocation) and *x'* stands for its final state (upon return). *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. *Pin*'s post-condition makes use of the auxiliary function, *pn*, defined in the trait *RecObj*:

```
pn(<m, t1, c>, t2) =
  if c > 0
  then if t1 = t2
         then <m, t1, c+1>
         else <m, t1, c>
  else <m, t2, 1>
```

It takes a recoverable object's state (of sort *R*) and a transaction identifier (of sort *Tid*) and returns a (new) state for a recoverable object. If the count (*c*) is non-zero, then the object must be pinned. If the object is pinned by a transaction (*t1*) that is the same as the transaction (*t2*) 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.

Unpin's pre-condition requires that an object cannot be unpinned unless it is already pinned; moreover it must be pinned by the calling transaction. *Un* is defined as follows:

```
un(<<m1, m2>, t1, c>, t2) =
   if c = 1
   then <<m1, m1>, t1, 0>
   else <<m1, m2>, t1, c-1>
```

Unlike for *pn*, it is unnecessary for *un* to check if the object is already pinned and if the transaction (*t1*) that currently has the object pinned is the same as the unpinning transaction (*t2*); *unpin*'s pre-condition checks for this case. *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.

Both *pin* and *unpin* have a **modifies** clause, which lists the set of objects in the state of the entire system whose values may possibly change. It is a strong indirect assertion about which objects may not change in value. This assertion is implicitly conjoined to the operation's post-condition. An omitted **modifies** clause is equivalent to the assertion **modifies nothing**, meaning no objects are allowed to change in value.

3.3. Deriving From Class Recoverable

A typical use of class *recoverable* is to define a derived class for objects that are intended to be persistent. For example, suppose we derive a new class, *recov_int*, from *recoverable*:

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

If *Int* is the sort identifier associated with values of recoverable integer objects, then the identifier *M* that appears in the *RecObj* specification would be renamed with *Int*. The header for the Larch interface specification for the recov_int class would look like:

```
class recov_int based on R from RecObj (Int for M)

    // ... specifications of recov_int's operations ...
```

4. Class Atomic

The second base class in the Avalon/C++ hierarchy is *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 *atomic* is used to ensure transaction-consistency. Persistence is "inherited" from class recoverable since *pin* and *unpin* are inherited through C++ inheritance.

4.1. Avalon Class Definition

Figure 4-1 gives the class header for atomic.

```
class atomic: public recoverable {
 public:
  void read_lock();                   // Obtain a long-term read lock.
  void write_lock();                  // Obtain a long-term write lock.
}
```

Figure 4-1: Avalon Atomic Class

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

4.2. Larch Specification

Figure 4-2 gives the Larch interfaces and trait for class *atomic*. As indicated in the trait *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:


```
class atomic based on A from AtomObj

  atomic() returns (atomic x)
     post x'.rs = { } ∧ x'.ws = { } ∧ new x

  read_lock(atomic x)
     when x.ws ⊆ ancestors(ts, self)
     modifies x
     post x' = add_reader(x, self)

  write_lock(atomic x)
     when x.rs ⊆ ancestors(ts, self) ∧ x.ws ⊆ ancestors(ts, self)
     modifies x
     post x' = add_writer(x, self)


AtomObj: trait
  includes
     RecObj, Set(Tid, Readers), Set(Tid, Writers)
     A record of (ob: R, rs: Readers, ws: Writers)
  introduces
     add_reader: A, TidS → A
     add_writer: A, TidS → A
  asserts for all (a: A, tid: Tid)
     add_reader(a, tid) = rs_gets(a, add(a.rs, tid))
     add_writer(a, tid) = ws_gets(a, add(a.ws, tid))
```

Figure 4-2: Larch Specification of Class Atomic

```
A record of (ob: R, rs: Readers, ws: 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 *atomic* initializes both the sets of readers and writers to be empty.

The transaction tree *ts* of type tidTree is global information:

```
class tidTree based on TransIdS from TransIdTree
  // ... TransIdTree defined in Appendix I ...

global ts: tidTree
```

Appendix I gives traits for defining a transaction tree, providing functions like *ancestors*, which returns the set of transactions that are ancestors of a given transaction (including itself). We declare the transaction tree global only for convenience since such objects could be passed as explicit arguments to each operation.

Read_lock's when-condition states that a transaction can get a read lock if all transactions holding write locks are ancestors; *write_lock*'s when-condition states that a transaction can get a write lock if all transactions holding read or write locks are ancestors. These two requirements reflect the conditions of Moss's locking rules for nested transactions [14], which are implemented in Avalon/C++.

As usual, the post-conditions look simple; the trait's *add_reader* and *add_writer* functions do the actual work, by adding the calling transaction to the appropriate set. Notice that since *rs* (*ws*) is a set, adding a transaction that already is in it has no effect. Thus, if the calling transaction already has a read (write) lock on the object, no change is made; otherwise, it obtains a read (write) lock.

4.3. Deriving From Class Atomic

Suppose we now define an *atomic_int* class as follows:

```
class atomic_int: public atomic {
  int val;                  // representation
 public:
  int operator=(int rhs);   // overloaded assignment
  operator int();           // overloaded coercion
}
```

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, :

```
class atomic_int based on A from AtomObj (Int for M)

    // ... specifications of atomic_int's operations ...
```

Now let us specify atomic_int's coercion operation, which takes an atomic_int, *x*, and returns a regular C++ int, *i*:

```
operator int(atomic_int x) returns (int i)
  when  x.ws ⊆ ancestors(ts, self) ∧ (~pinned(x.ob) ∨ x.ob.pinner = self)
  modifies x
  post  x' = add_reader(x, self) ∧ i' = x.ob.value.volatile
```

The second conjunct of the post-condition makes the climactic point: The value (of sort Int) of the int object *i* returned is the *value* (of sort Int) in *volatile* storage of the recoverable *object* component of the atomic_int *x*. We retrieve the value from volatile storage because we can assume that the when-condition held: if the object is pinned, but not yet unpinned (by *self*) then we want x'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++ implementator 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-consistency, 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++:

```
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* operation, we can show (1) the coercion operation's when-condition trivially implies *read_lock*'s when-condition; and (2) *read_lock*'s post-condition guarantees the calling transaction 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 write locks on it.

5. 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, *subatomic* provides more complex primitives to give programmers more detailed control over their objects' synchronization and recovery mechanisms by exploiting type-specific properties of objects. For example, a queue object with *enqueue* and *dequeue* operations can permit enqueuing and dequeueing 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 [5] for details and other examples.

5.1. Avalon Class Definition

```
class subatomic: public recoverable {
 protected:
  void seize();                        // Gains short-term lock.
  void release();                      // Releases short-term lock.
  void pause();                        // Temporarily releases short-term lock.
 public:
  virtual void commit(trans_id& t);    // Called after transaction commit.
  virtual void abort(trans_id& t);     // Called after transaction abort.
}
```

Figure 5-1: Avalon Subatomic Class

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.

To ensure transaction-consistency, *subatomic* provides *commit* and *abort* operations. Whenever a top-level transaction commits (aborts), the Avalon/C++ run-time 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.

6. Observations

6.1. About Avalon

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

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 of class *recoverable*. The Avalon/C++ implementation precludes the possibility of concurrent pins 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, one can give a meaning to persistence; without crash recovery, one can give a meaning to the correct synchronization of processes. But to support both, there are points where one must consider both persistence and synchronization together.

Another kind of unstated assumption discovered from this exercise is implicit pre-conditions. For example, whereas *pin* has no pre-condition, *unpin* does. This assymmetry in the specifications reflects the assymmetry that exists in the actual implementation. An earlier version of the specification of *unpin* did not have a pre-condition, but not until the implementor was shown this (incorrect) version was the unstated pre-condition revealed. In fact, upon seeing the assymmetry in the current version of the specification, the implementor realized that the pre-condition on *unpin* could easily be removed by performing a run-time check, as is already done for *pin*.[5]

Specifying the class *atomic* helped make the rules for obtaining long-term locks more precise. It also makes 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* makes 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 Figure 5-2, 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.

6.2. About Larch

In the traditional spirit of Larch, all the complexity of a specification is relegated to the traits. The rule-of-thumb is: If the post-condition becomes unwieldy then introduce a trait function 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

[5]The astute reader may have noticed that *un*'s second argument, a vestige of the earlier specification, was ignored in its definition; if the pre-condition for *unpin* is removed, then the second argument is necessary.

behavior. For example, the use of the existential operator in the post-conditions 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 [2] 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 post-conditions. 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, even as currently extended, is liveness. For example, one 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 depends on this test to succeed. So, sometimes no progress can be made until some transaction has committed. We have seen in the previous section where deadlock may arise in the implementation, and how the specification permits for this behavior. Though Larch was never intended to address liveness properties, in the context of concurrent transactions, such properties are important to state for practical reasons.

7. Final Remarks

The specifications presented here represent ongoing work. They continue to change as we continue to specify more of Avalon/C++'s intricacies, such as: (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 [11]. (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 these specifications, we would also like to prove theorems about the objects being specified. For example, from the specification in Figures 5-2 and 5-3 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: $(\forall\ x: S)\ x.locker \notin waiters(x)$. Our plan is to use the Larch Prover [7] to help with these proofs.

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 early on that a fundamental part of Avalon/C++'s semantics is implemented correctly is a tremendous reassurance to us as Avalon implementors as well as to all Avalon programmers. In conclusion, writing the formal specifications of Avalon/C++'s built-in classes has helped clarify places in the language where features interact, make explicit unstated assumptions, and make precise complex non-functional properties of objects.

Acknowledgments

Discussions with John Guttag and Jim Horning and the examples given in [2] inspired my on-the-fly interface language design, in particular the Larch extensions for concurrency. Chun Gong helped develop the traits and Rick Lerner helped check the interfaces. I am grateful to all members of the Avalon group, in particular, Maurice Herlihy and David Detlefs, who helped design Avalon/C++, and David who was instrumental in building it.

I. Transactions and the Transaction Tree

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

```
TidStatus: trait
  includes TimeStamp
  introduces
    co: Time → S
    ac: → S
    ab: → S
  asserts S generated by (co, ac, ab)

TransId: trait
  includes TidStatus, UniqueId
    Pair(Tid for T, Id for T1, S for T2, name for .first, status for .second)
  introduces
    create: Id → Tid
    commit: Tid, Time → Tid
    abort: Tid → Tid
    aborted: Tid → Bool
    committed: Tid → Bool
  asserts
    Tid partitioned by (name)
    for all (t: Tid, id: Id, ti: Time)
      create(id) = <id, ac>
      commit(<id, ac>, ti) = <id, co(ti)>
      abort(<id, ac>) = <id, ab>
      committed(t) = (status(t) ≠ ac ∧ status(t) ≠ ab)
      aborted(t) = (status(t) = ab)
  exempting for all (id: Id, ti: Time)
      ( commit(<id, ab>), commit(<id, co(ti)>), abort(<id, ab>), abort(<id, co(ti)>) )

TransIdTree: trait
  includes TransId, Tree(Tid, TransIdS)
  introduces
    committed: TransIdS, Tid → Bool
    aborted: TransIdS, Tid → Bool
  asserts for all (ts: TransIdS, t: Tid)
      committed(ts, t) = t ∈ ts ∧ committed(t)
      aborted(ts, t) = t ∈ ts ∧ aborted(t)
```



```
Tree (N, T): trait
  includes Set(N, Nodes)
  introduces
    emp: → T
    add_root: N → T
    add_node: T, N, N → T
    _∈_: N, T → Bool
    des: T, N, N → Bool
    ancestors: T, N → Nodes
  asserts
    T generated by ( emp, add_root, add_node )
    for all (n, n1, n2, n3: N, t, t1: T)
      add_node(emp,n,n1) = emp
      add_node(add_root(n),n1,n2) =
                    if n=n1
                    then add_node(add_root(n),n,n2)
                    else add_root(n)
      add_node(add_node(t,n,n1),n2,n3) =
                    if n1=n2
                    then add_node(add_node(t,n,n1),n1,n3)
                    else add_node(add_node(add_node(t,n2,n3),n,n1))

      n ∈ emp = false
      n ∈ add_root(n1)) = (n=n1)
      n ∈ add_node(t,n1,n2) = (n ∈ t) ∨ (n=n1) ∨ (n=n2)
      des(emp,n,n1) = false
      des(add_root(n),n1,n2) = false
      des(add_node(t,n,n1),n2,n3) =
                    if (n=n2 ∧ (n1=n3 ∨ n2=n3))
                    then true
                    else if (n=n2)
                        then des(t,n2,n3)
                        else if (n1=n3)
                            then des(t,n2,n1)
                            else des(t,n2,n3)
      n1 ∈ ancestors(t,n) = des(t,n1,n)
```

II. Auxiliary Traits

```
Set (E, S): trait
  introduces
    { }: → S
    add: S, E → S
    rem: S, E → S
    _∈_: E, S → Bool
    _⊆_: S, S → Bool
  asserts
    S generated by ( { }, add )
    S partitioned by ( ∈ )
    forall (s, s1: S, e, e1: E)
      rem({ }, e) = { }
      rem(add(s, e), e1) = if e = e1 then rem(s,e1) else add(rem(s,e1),e)
      e ∈ { } = false
      e ∈ add(s,e1) = (e = e1) ∨ (e ∈ s)
      { } ⊆ s = true
      add(s,e) ⊆ s1 = e ∈ s1 ∧ s ⊆ s1
```

```
Triple: trait
  introduces
    <_, _, _>: T1, T2, T3 → T
    _.first: T → T1
    _.second: T → T2
    _.third: T → T3
  asserts
    T generated by (<_, _, _>)
    T partitioned by (.first, .second, .third)
    for all (a: T1, b: T2, c: T3)
      <a, b, c>.first = a
      <a, b, c>.second = b
      <a, b, c>.third = c
```

```
Pair: trait
  introduces
    <_, _>: T1, T2 → T
    _.first: T → T1
    _.second: T → T2
  asserts
    T generated by (<_, _>)
    T partitioned by (.first, .second)
    for all (a: T1, b: T2)
      <a, b>.first = a
      <a, b>.second = b
```

Records are a shorthand for a trait defined as follows. For each **record** of the form

S record of (f_1: S_1, ..., f_n: S_n)

Append to the function declarations of the enclosing trait:

introduces
  mk_S: S_1, ..., $S_n \rightarrow$ S
  __.f_i: S $\rightarrow S_i$
  f_i_gets: S, $S_i \rightarrow$ S

for $1 \le i \le n$.

Append to the set of equations of the enclosing trait:

asserts
  S **generated by** (mk_S)
  S **partitioned by** (.f_1, ..., .f_n)
  for all (x_1, y_1: S_1, ..., x_n, y_n: S_n)
    mk_S(x_1, ..., x_i, ..., x_n).f_i = x_i
    f_i_gets(mk_S(x_1, ..., x_i, ..., x_n), y_i) = mk_S(x_1, ..., y_i, ..., x_n)

for $1 \le i \le n$.

References

[1] J.R. Abrial.
The Specification Language Z: Syntax and Semantics.
Technical Report, Programming Research Group, Oxford University, 1980.

[2] A. Birrell, J. Guttag, J. Horning, R. Levin.
Synchronization Primitives for a Multiprocessor: A Formal Specification.
In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 94-102. ACM/SIGOPS, 1987.

[3] D. Bjorner and C.G. Jones (Eds.).
Lecture Notes in Computer Science. Volume 61: *The Vienna Development Method: the Meta-language.*
Springer-Verlag, Berlin-Heidelberg-New York, 1978.

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

[5] 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.

[6] 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.

[7] S.J. Garland and J.V. Guttag.
Inductive Methods for Reasoning about Abstract Data Types.
In *Proceedings of the 15th Symposium on Principles of Programming Languages*, pages 219-228. January, 1988.

[8] J.A. Goguen and J.J. Tardo.
An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications.
In *Proceedings of the Conference on Specifications of Reliable Software*, pages 170-189. Boston, MA, 1979.

[9] J.V. Guttag, J.J. Horning, and J.M. Wing.
The Larch Family of Specification Languages.
IEEE Software 2(5):24-36, September, 1985.

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

[11] M.P. Herlihy and J.M. Wing.
Reasoning About Atomic Objects.
Technical Report CMU-CS-87-176, Carnegie Mellon University Department of Computer Science, November, 1987.

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

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

[14] J.E.B. Moss.
Nested Transactions: An Approach to Reliable Distributed Computing.
Technical Report MIT/LCS/TR-260, Massachusetts Institute of Technology Laboratory for Computer Science, April, 1981.

[15] A. Spector, J. Bloch, D. Daniels, R. Draves, D. Duchamp, J. Eppinger, S. Menees, D. Thompson.
The Camelot Project.
Database Engineering 9(4), December, 1986.

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

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

A Self-Applicable Partial Evaluator for Term Rewriting Systems

Anders Bondorf
DIKU, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark (*)
uucp: anders@diku.dk

Abstract

This paper describes a fully self-applicable partial evaluator developed for equational programs in the form of term rewriting systems. Being self-applicable, the partial evaluator is able to generate efficient compilers from interpreters as well as a compiler generator automatically.

Earlier work in partial evaluation of term rewriting systems has not achieved self-applicability due to the problem of partially evaluating pattern matching. This problem is overcome by developing an intermediate language for being able to express pattern matching at an appropriate level of abstraction.

We describe the intermediate language and partial evaluation of it. Binding time analysis, a well-known preprocessing technique, is used. We introduce further preprocessing to deal efficiently with our intermediate language.

The system has been implemented and compilers for small languages as well as a compiler generator have been generated with satisfactory results.

Keywords

Decision trees, functional languages, pattern matching, elementary matching operations, binding time analysis, abstract interpretation, partially static structures.

(*) Until July 1989:
University of Dortmund, Lehrstuhl Informatik V
Postfach 50 05 00, D-4600 Dortmund 50
Federal Republic of Germany
uucp: anders@unidoi5.ls5.informatik.uni-dortmund.de

1. Introduction

The potential use of partial evaluation for doing compilation, compiler generation, and even compiler generator generation has been known since the early seventies [Futamura 71]. A few years ago these promising ideas were carried out for the first time in practice in the LISP based "Mix" project in the Copenhagen group around Neil D. Jones [Jones, Sestoft, & Søndergaard 85]. Various people have since then been working on partial evaluation in Copenhagen and other places. The aims have been to understand partial evaluation better and to develop stronger partial evaluators: to make them "as automatic as possible" and to use stronger languages.

Partial evaluation is a general program transformation which, given a *subject* program and *static* values of some but not all of its input parameters, produces a so-called *residual* program [Ershov 82]. This, when applied to the rest of the inputs, will yield the same result the original program would have yielded on all its inputs. Partial evaluation is thus *program specialization*: its effect is to yield a new program equivalent to the original on a certain subset of its input. We therefore also refer to a partial evaluator as a *specializer*.

To compile by partial evaluation, an interpretive specification of the language is needed. By specializing the interpreter with static input being the interpreted source program, a target program is produced (compilation). And by *self-application*, specialization of the specializer itself with static input being an interpreter, a stand-alone compiler is generated. Finally, by specializing the specializer with the static input being the specializer itself, a compiler generator is produced.

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

352

J. Díaz F. Orejas (Eds.)

TAPSOFT '89

TAPSOFT '89

Proceedings of the International Joint Conference on Theory and Practice of Software Development
Barcelona, Spain, March 13–17, 1989

Volume 2:
Advanced Seminar on Foundations of Innovative Software Development II and Colloquium on Current Issues in Programming Languages (CCIPL)

Springer-Verlag
Berlin Heidelberg New York London Paris Tokyo