Specifying Avalon Objects in Larch

Chris A. Visser, Giuseppe Scollo, Martin von Sinderen, Ed Brinkman

University of Twente, Faculty of Informatics
7600 AE Enschede, NL

Abstract

This paper presents a formal specification of the Avalon interface using Larch. The interface is specified in a way that is consistent with the Avalon specification and allows for verification of properties of the specification.

1. Introduction

The Avalon interface provides a standard way for components to communicate with each other. The formal specification presented in this paper allows for verification of properties of the interface, which can help ensure that the interface behaves as intended.

2. Formal Specification

The specifications are written in Larch, a formal specification language. The specifications include both the structural and behavioral aspects of the Avalon interface.

3. Verification

The specifications are verified using a tool called PVS. Verification of properties such as liveness, safety, and atomicity is performed.

4. Conclusion

The formal specification and verification process presented in this paper provides a rigorous way to ensure the correctness of the Avalon interface.

Sections 2, 3, and 4 present a formal specification of the Avalon interface. Section 5 concludes with remarks about current work and future research.
2. The two transaction commands can run the same operation in parallel.

3. Only one transaction can run in one process at a time.
We see a rectangular object, 'rectangle', in the image. The rectangle is a graph in memory, as indicated by the notation and operations performed on it.

We now apply the same techniques to the rectangle in detail. The fill-part of the specification contains figures and notations indicating the operations performed on the rectangle.

Figure 2.2: Latch Specification of a Data Register

```
Latch = (Latch | Register) & ~Register
```

We can see that the expression is a combination of the Latch and Register, with the Register being inverted. This indicates how the latch is updated when the register is cleared.

However, the expression does not directly indicate the operations performed on the rectangle, which is a graph in memory. It seems to be a simplification for understanding the concept.
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:

```java
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:

```java
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.

```java
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:

```java
class atomic based on A from AtomObj
    atomic() returns (atomic x)
    post x'.rs = \{\} \land x'.ws = \{\} \land new x
    read_lock(atomic x)
    when x.wr \subseteq ancestors(ts, self)
    modifies x
    post x' = add_reader(x, self)
    write_lock(atomic x)
    when x.rs \subseteq ancestors(ts, self) \land x.ws \subseteq 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 \rightarrow A
    add_writer: A, Tids \rightarrow A
asserts for all (a: A, tid: Tid)
    add_reader(a, tid) = rs.get(a, add(a.rs, tid))
    add_writer(a, tid) = ws.get(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 `ts` of type tidTree is global information:
```
class tidTree based on TransId 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 Mom's locking rules for nested transactions [14], which are implemented in Avalon/C++.
Since our current options do not allow us to make decisions about the content or structure of our text, the following content will be generated based on a hypothetical scenario.

6. About Last Night

7. From Here?

In a scenario where we have the freedom to choose content and structure, we could potentially explore the following:

- The impact of recent events on our lives and society.
- The evolution of technology and its role in shaping our future.

These topics could serve as a foundation for further discussion and exploration in future sessions.

We are currently unable to provide a detailed discussion on the specific content mentioned in the image.
II. Auxiliary Truths

I. Transactions and the Transaction Tree

In particular, Apple's design brief and David's design, to each their own. We've
intentionally left this open to be filled in with something meaningful.

Acknowledgments

explicit material treatment and mere facade non-functional properties of objects.
References


For Term Rewriting Systems

A Self-Applicable Partial Evaluation
Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

352

J. Díaz  F. Orejas (Eds.)

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