Machine-Assisted Proofs of Properties of Avalon Programs

Jeannette M. Wing and Chun Gong

24 August 1989
CMU-CS-89-171$_2$

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Proving the correctness of programs by hand is hard and error-prone. How can mechanical theorem proving aids such as the Larch Prover (LP) help in the proofs of complex programs? We address this question by applying LP, a proof checker based on rewrite-rule theory, to the proof of an Avalon/C++ program. Avalon/C++ is a programming language that supports concurrency and fault-tolerance through transaction-based computations. Since reasoning about an Avalon/C++ program requires reasoning about histories, i.e., sequences of operations, and not just initial and final states, proofs of correctness are non-trivial. For the Avalon/C++ queue example, we present a formal Larch Shared Language specification, which we also used as input to LP. We discuss the LP-assisted proofs we performed of the representation invariant and the queue's key correctness condition, give detailed statistics of our proofs, and draw some conclusions based on our experience with LP.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory Under Contract No. F33615-87-C-1499. Additional support was provided in part by the National Science Foundation under grant CCR-8620027.

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

Machine-Assisted Proofs of Properties of Avalon Programs

Jeannette M. Wing and Chun Gong

August 24, 1989

1. Introduction

Many people have argued the importance of mechanical theorem-proving for reasoning about programs. Proving the correctness of programs by hand is usually hard and error-prone. One often misses boundary cases or forgets to state hidden assumptions. On the other hand, can current mechanical theorem provers deal with a wide scope of non-trivial problems? Here, the question of scale is in diversity of problems as well as in complexity of each problem. Some provers are more suitable for one class of problems than others and all provers have space and time bounds that set practical limits on the size of an individual problem that can be handled.

The specific purpose of this paper is to report on our experience using the Larch Prover (LP) [8] as a mechanical aid for proving properties of Avalon/C++ programs. Avalon/C++ is a programming language [6] that deals with concurrency and faults. Its semantics are based on a client/server model of distributed transactions. The Larch Prover is a proof checker based on rewrite-rule theory. It is more than a rewrite-rule engine, but not quite a general-purpose first-order logic theorem prover. [1]

We view our application of LP to Avalon/C++ from two ways. From the Avalon/C++ viewpoint, we consider how LP can help in the proofs of non-trivial properties like atomicity (*q.v.* Section 2.1.). From the LP viewpoint, we consider how LP fares on a non-trivial example like an Avalon/C++ program. Our example is different from those which LP-like checkers are traditionally good at or designed for (e.g., groups, sets, and other algebraic structures), and from those drawn from domains, such as hardware and operating system kernels, addressed before by LP and other checkers such as Gypsy [11], LCF [13], HOL [12], and Clarke's model checker [3].

Thus, we began this specification and proof exercise with the following general goals in mind:

1. To see how amenable Avalon-like properties are to specification and proof within the Larch framework;
2. To see what can be gained in our understanding of Avalon through the use of machine aids; and
3. To determine the limitations of one of the state-of-the-art mechanized proof checkers.

As a quick summary, we conclude that the Larch specification language is best suited for describing theories of underlying Avalon/C++ data types, but less suited for describing global properties of Avalon/C++ computations. Though we did not gain a deeper understanding of Avalon/C++ with our use of LP, we were forced to be extremely explicit about Avalon/C++'s model of computation and, sometimes more than we felt necessary, about certain equality and membership relations among objects. Finally, LP's only major technical limitation is its inability to handle explicit existential quantification. Its pragmatic limitation is that its users still have to be fairly sophisticated. In its favor, LP is a robust, efficient and well-engineered proof checker.

[1] We have implemented Avalon/C++ at CMU as an extension of C++ [22], using the Camelot transaction facility [21] for its runtime system. The Larch Prover was implemented at MIT.

One concrete result from this work is a formal specification of the well-worn Avalon/C++ queue example[2]. This specification includes an encoding of Avalon/C++'s history-based model of computation specialized for the queue. Another concrete result is a set of proofs of properties, ranging from simple, but general properties about sequences to more complex and very specific properties about the queue implementation. We stress both specification and verification in this paper.

We assume the reader has some familiarity with formal specifications, proof checkers, and some of their underlying theory. Though LP is based on rewrite-rule theory, the reader, just as LP's users, need not have a deep understanding of it. We do not assume the reader is familiar with Avalon/C++, so we will review its main ideas. In Section 2 we present informally some background on the specificand domain, our verification technique, and the Larch language and tools. Where appropriate we cite other papers that give more formal and complete descriptions. Section 3 gives the Avalon/C++ code for the queue example. Section 4 walks through the Larch specification of the queue representation and properties of the abstraction function that are needed to prove the key correctness condition for the queue. Section 5 walks through proof steps of one of the representation invariants, one helping lemma, and the basis case for the correctness condition; it also shows the entire proof outline for this condition. Section 6 presents some performance statistics about the proofs and compares LP with two other proof checkers. Finally, we state some general conclusions and point to future work in Section 7.

2. Background

2.1. Specificand Domain: Reliable Distributed Systems

To build a reliable distributed system, a widely-accepted technique for preserving consistency in the presence of failures and concurrency is to organize computations as a set of sequential processes called *transactions*. A key property of a transaction is *atomicity*, that is, *serializability* and *recoverability* [6]. *Serializability* means that concurrent transactions appear to execute in some sequential order, and *recoverability* means that a transaction either succeeds completely or has no effect. A transaction's effects become permanent when it *commits*, its effects are discarded if it *aborts*, and a transaction that has neither committed or aborted is *active*.

Objects contain the state of the system. Each object has a *type*, which defines a set of possible *values* and a set of *operations* that provide the only means to create and manipulate objects of that type. Typically, a transaction executes by invoking an operation on an object, receiving results when the operation terminates, then invoking another operation on a possibly different object, receiving results when it terminates, etc. After performing a sequence of such operation executions, a transaction then commits or aborts.

Atomicity is a global correctness property of the entire system. There are, however, restricted versions of atomicity that are "local" properties. If each object in a system satisfies a local atomicity property, then the entire system is guaranteed to be atomic [23]. Hence, to achieve atomicity, languages like Avalon/C++ provide *atomic objects*, which are typed objects that satisfy a local atomicity property, thus guaranteeing the atomicity of the transactions that access them.

The local atomicity property, i.e., the fundamental correctness condition, that all Avalon/C++ objects must satisfy is called *hybrid atomicity* [23]. *Hybrid atomicity* requires that all transactions must appear to execute sequentially in the order they commit. In Avalon/C++, when a transaction commits, it is assigned a logical timestamp [19]. These timestamps determine the transactions' serialization order. Avalon/C++ moreover supports a "pessimistic" variation of atomicity: an active transaction with no pending operation invocation is always allowed to commit. Operations are delayed when necessary to ensure this property.

Avalon/C++ provides ways to enable programmers to define abstract atomic types. For example, if one were to define an `atomic_array` type, one would define a new class, `atomic_array`, which perhaps provides `fetch` and `store` operations. (Syntactically, a *class* is a collection of *members*, which are the components

[2]We promise the community that this paper will be the penultimate word on the queue example.

of the object's representation, and a collection of operation implementations.) The intuitive difference between an `atomic_array` type and a conventional `array` type is that objects of `atomic_array` type will ensure serializability and recoverability of the transactions that access them, whereas objects of `array` type in general will not. However, the programmer who defines the abstract atomic type must prove that the new type is "correct," i.e., that all objects of the newly defined type are (hybrid) atomic. By providing language support for constructing atomic objects, we gain the advantage that this proof is done only once per class definition, not each time an object is created or accessed.

We present other details of Avalon/C++ as necessary. See [23] for formal definitions of our model of computation and of global and local atomicity properties. See [6] for a more complete description of Avalon/C++.

2.2. The Verification Method

To show the correctness of a user-defined atomic type implementation, we generalize techniques from the sequential domain. Our method requires that the user provide: (1) a representation invariant, (2) an abstraction function, and (3) a type-specific correctness condition. The representation invariant defines the domain of the abstraction function. The abstraction function maps a representation value to a set of sequences of abstract operations. The type-specific correctness condition determines which of those sequences are *legal*. The only unusual aspect of any of these provisions is the range of the abstraction function: it is not a set of abstract values as in the sequential domain, but a powerset of sequences of abstract operations.

Let *Rep* be the implementation object's set of values, *Abs* be the set of values of the abstract data type being implemented, and *OP* be the abstract object's set of operations. The subset of *Rep* values that are valid values is characterized by a predicate called the *representation invariant*, I : *Rep* — *bool*. The meaning of a valid representation is given by an *abstraction function*, $\mathcal{A}$: *Rep* — 2^{OP^*}, defined only for values that satisfy the invariant.

Our verification method requires one to show inductively that the following properties hold:

1. $\forall S \in \mathcal{A}(r)$, S is a legal sequence of abstract operations, and
2. The set of possible *serializations* is contained in $\mathcal{A}(r)$. (Any active transaction can commit.)

This paper will focus on showing only the first property for the queue example. The second property, though trivial to show, requires defining many terms (like *serialization*) that would distract the reader from the focus of this paper. See [24] for more details and justification of this verification method, and in particular for a hand-proof of both properties for the queue example we present herein.

2.3. The Specification and Verification Tools: LSL and LP

Traditional syntax-directed program verification requires that one annotate the program text with assertions, e.g., loop invariants, and use the programming language's proof rules to generate verification conditions (VC's) that must be shown to hold. VC's are typically expressed in the same assertion language as one's specification language and moreover, in the proof checker's language. However, we do not have formal proof rules for Avalon, C++, nor C; we doubt devising them would even be a worthwhile exercise.

Thus, to carry out a program proof with a proof checker, we take the approach of "encoding" an Avalon program into a high-level specification language that we can also use as an assertion language and proof checker language. We use the Larch Shared Language (LSL) for these purposes [14]. Syntactically, an LSL specification, called a *trait*, looks like a traditional algebraic specification; semantically, it defines a first-order theory. LSL provides ways to construct a large and complex theory from some smaller and simpler theories either by combining those small theories together or by adding induction rules.

To a first approximation, LSL specifications serve as input to the Larch Prover[3], and thus LSL serves as a subset of the proof checker's language. A proof of correctness amounts to stating a theorem in LP's language and proving it given the specification, i.e., the encoding of the program text. Roughly speaking, what we are really showing is that we have encoded enough in an LSL theory to prove some theorem, e.g., a correctness condition.

Given an Avalon program, there is by no means an obvious encoding scheme that determines an LSL trait. The specification that we present in Section 4 is a result of a long series of attempts, some of which led us astray, got too complicated for us to understand, or were unsuitable for LP to handle. The resulting specification is a compromise between being "user-friendly" and "machine-friendly." In short, it is what we were able to use in order to get the proof to go through in a reasonably natural way.

A note on LSL expressibility

Our most constraining limitation in using LSL as our specification language was in not having explicit existential quantification. Having to express first-order assertions in equational logic makes some of our specifications look awkward. In principle, LSL has the expressive power of full first-order predicate logic plus induction schema. However, all variables in LSL equations are implicitly universally quantified and all function symbols are implicitly existentially quantified. In order to express an existentially quantified variable, we have to use one of two skolemization techniques [7], depending on which side of the universal quantifier the existentially quantified variable appears.

Suppose f and g are functions symbols and $x1.....xn$ are variables. An LSL equation that appears as:

$$f(x1.....xm) == g(x1.....xm)$$

means:

$$\exists f.g\ \forall x1.....xm\ \ f(x1.....xm) = g(x1.....xm)$$

So, if one needs to say:

$$\exists c1.....cn\ \exists f.g \forall x1.....xm\ \ f(x1.....xm.\ c1.....cn) = g(x1.....xm.\ c1.....cn)$$

then the $c1.....cn$ are Skolem *constants*. In LSL, $c1.....cn$ can be introduced as nullary functions, and hence, treated like any other function symbols in LSL. LP can handle Skolem constants; in the specifications below, we use "c_" to prefix "variables" that are Skolem constants; all other user-declared regular variables are prefixed by "x" or "y"[4].

On the other hand, if one needs to say:

$$\exists f.g\ \forall x1.....xm\ \exists h1.....hn\ \ f(x1.....xm.\ h1.....hn) = g(x1.....xm.\ h1.....hn)$$

then the $h1.....hn$ are Skolem *functions*, each over $x1.....xm$. In LSL, $h1.....hn$ would have to be declared as m-ary functions, and hence be treated as the other existentially quantified function symbols ($h1.....hn$ are "carried over" the universally quantified xi's and get captured by the implicit existential quantifier).

We give other details about LSL and LP as needed; see [15] for a more complete description of LSL and [9] for another example use of LP.

---

[3]To some LSL terms, we needed to add signature information to disambiguate operator names.

[4]LP actually lets the user declare whether a symbol is a variable or constant, and thus the prefix convention is unnecessary. We adhere to this convention as an aid to the reader.

3. The Code for the Queue Example

We now present the Avalon/C++ code, taken from [16], for a highly concurrent atomic FIFO queue implementation. Our implementation is interesting for two reasons. First, it supports more concurrency than commutativity-based concurrency control schemes such as two-phase locking. For example, it permits concurrent enqueuing transactions, even though enqueuing operations do not commute. Second, it supports more concurrency than any locking-based protocol, because it takes advantage of state information. For example, it permits concurrent enqueuing and dequeuing transactions while the queue is non-empty. Most other transaction-based systems use read-write locking protocols for synchronizing access to shared data, thereby precluding concurrent enqueues and concurrent enqueues and dequeues (both enqueue and dequeue would be classified as "writers").

To support such high degrees of concurrency, Avalon/C++ provides programmers the means to test the serialization order of transactions at run-time. Indeed, one of Avalon/C++'s novelties is its built-in class `trans_id`, used for creating and comparing transaction identifiers. For any two transaction identifiers *t1* and *t2* a successful comparison of *t1* < *t2* implies that *T1*, the transaction that generated *t1*, is committed and serialized before *T2*, the transaction that generated *t2*. If *t1* and *t2* were generated by the same transaction, then *t1* was generated first. If the comparison evaluates to false, then the `trans_id`'s may have the reverse ordering, or their ordering may be unknown (e.g., because their associated transactions are both active).

3.1. Representation

We record information about each `enq` operation in the following `struct`:

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

The `item` component is the enqueued item. The `enqr` component is a new transaction identifier generated by the enqueuing transaction. It is used for uniquely tagging each `enq` operation. The last component defines a constructor operation for initializing the `struct`.

We record information about `deq` operations similarly, where the `deqr` component is a new transaction identifier generated by the dequeuing transaction:

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

We represent the queue itself as follows:

```
class atomic_queue : public subatomic {
  deq_stack deqd;                  // Stack of dequeued items.
  enq_heap enqd;                   // Heap of enqueued items.
 public:
  atomic_queue() {};               // Create empty queue.
  void enq(int item);              // Enqueue an item.
  int deq();                       // Dequeue an item.
```

```
    void commit(trans_id& t);    // Called on commit.
    void abort(trans_id& t);     // Called on abort.
};
```

The deqd component is a stack of deq_rec's used to undo deq operations of aborted transactions. The enqd component is a partially ordered heap of enq_rec's ordered by their enqr fields. A partially ordered heap provides operations to enqueue an enq_rec, to test whether there exists a unique oldest enq_rec, to dequeue it if it exists, and to discard all enq_rec's inserted by (aborted) transactions.

Recall that in Avalon/C++ a transaction's effect is determined only after it has committed and we must serialize transactions according to their commit time. Thus, while there are active transactions, we cannot get a totally ordered queue. However, we can test whether there is a unique "oldest" element in the heap component; if so, a concurrent dequeueing transaction can be permitted to proceed, subject to other constraints discussed below.

3.2. The Operations

Besides the constructor, the queue has four operations: enq, deq, commit, and abort. The queue has control over the operation-level synchronization of transactions that desire access to it. These synchronization conditions are encoded in the boolean test of the Avalon when statement. The when(TEST) BODY statement is a conditional critical region: BODY is executed only when TEST evaluates to true. Avalon associates a short-term lock with each object to guarantee mutual exclusion between transactions entering and exiting their critical regions. TEST typically contains a comparison (<) between two trans_id's.

If *B* is an active transaction, then we say *A* is *committed with respect to B* if *A* is committed, or if *A* and *B* are the same transaction. Enq and deq must satisfy the following synchronization constraints to ensure atomicity. Transaction *A* may dequeue an item if (1) the most recent transaction to have executed a deq is committed with respect to *A*, and (2) there exists a unique oldest element in the queue whose enqueuing transaction is committed with respect to *A*. The first condition ensures that *A* will not have dequeued the wrong item if the earlier dequeuer aborts, and the second condition ensures that there is something for *A* to dequeue. Similarly, *A* may enqueue an item if the last item dequeued was enqueued by a transaction committed with respect to *A*.

Given these conditions, here is the code for enq:

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

Enq checks whether the item most recently dequeued was enqueued by a transaction committed with respect to the caller. If so, the new trans_id and the new item are inserted in enqd. Otherwise, the transaction releases the short-term lock and tries again later (guaranteed by the implementation of the when statement).

Here is the code for deq:

```
int atomic_queue::deq() {
  trans_id tid = trans_id();
  when ( (deqd.is_empty() || deqd.top()->deqr < tid)
        && enqd.min_exists() && (enqd.get_min()->enqr < tid)) {
    enq_rec* min_er = enqd.delete_min();
    deq_rec dr(*min_er, tid);  // Move from enqueued heap...
    deqd.push(dr);             // to dequeued stack.
    return min_er->item;
  }
}
```

Deq tests whether the most recent dequeuing transaction has committed with respect to the caller, and whether enqd has a unique oldest item. If the transaction that enqueued this item has committed with respect to the caller, it removes the item from enqd and records it in deqd. Otherwise, the caller releases the short-term lock, suspends execution, and tries again later.

The queue also manages the commit and abort processing of transactions as it learns of their commit or abort status. This processing is done through the commit and abort operations. The Avalon/C++ run-time system calls the commit (abort) operations of all objects a transaction accesses after it commits (aborts). The commit operation looks like:

```
void atomic_queue::commit(trans_id& committer) {
  when (TRUE)                    // Always ok to commit.
    if (!deqd.is_empty() && descendant(deqd.top()->deqr, committer)) {
      deqd.clear();              // Discard all dequeue records.
    }
}
```

When a transaction commits, the queue discards deq_rec's no longer needed for recovery. The implementation ensures that all deq_rec's below the top are also superfluous, and can be discarded. We state this property formally when giving the representation invariant in Section 5.1.

The abort operation looks like:

```
void atomic_queue::abort(trans_id& aborter) {
  when (TRUE) {                  // Always ok to abort.
    while (!deqd.is_empty()      // Undo aborted dequeue by...
      && descendant(deqd.top()->deqr, aborter)) { // aborting transaction.
      deq_rec* d = deqd.pop(); // Undo aborted dequeue.
      enqd.insert(d->item, d->enqr);  // Put it back.
    }
    enqd.discard(aborter);       // Undo aborted enqueues.
  }
}
```

Abort undoes every operation executed by a transaction that is a descendant of the aborting transaction. It interprets deqd as an undo log, popping records for aborted operations, and inserting the items back in enqd heap. Abort then flushes all items enqueued by the aborted transaction and its descendants.

4. The Specification of the Queue

Recall that our verification method required that we give a representation invariant, abstraction function, and type-specific correctness condition. In the following two subsections, we give a specification of the representation, which suffices to state the representation invariant given in Section 5.1, and a specification of the properties of the abstraction function that are necessary to prove the type-specific correctness condition.

Informally, the correctness condition for the queue is:

Correctness Condition (informal): For all queues, the sequence of items dequeued must be a prefix of the sequence of items enqueued.

This condition intuitively captures the FIFO property of queues. We refer to it as the "prefix" property and state it formally in Section 5.2.

4.1. Queue Representation

We begin by defining the theory of the queue representation. In what follows, we make use of some basic traits, Pair, Triple, Set, and Stack, all given in Appendix A.

First we build a tiny theory about `trans_id`'s, introducing only the partial order, <, and a Skolem constant *c_xt*, which we will find useful in subsequent specifications. In LSL "A => B" stands for the equation "A => B == true."

```
TransID(Tid): trait
  introduces
    __<__: Tid, Tid -> Bool
    c_xt: -> Tid
  asserts for all(xt, xt1, xt2: Tid)
    ((xt < xt1) & (xt1 < xt2)) => (xt < xt2),
    ((xt < xt1) & (xt1 < xt)) => (xt = xt1)
  end
```

Next, we build theories about enq_rec's and deq_rec's in terms of pairs and triples.

```
Enq_Rec(EL, enq_rec): trait
  includes TransID, Pair(EL, Tid, enq_rec, element for first, enqt for second)
  introduces
    e_before: enq_rec, enq_rec -> Bool
  asserts enq_rec partitioned by (element)
  for all(x, x1: enq_rec)
    e_before(x, x1) == enqt(x) < enqt(x1)
  end

Deq_Rec(EL, deq_rec): trait
  includes TransID, Enq_Rec,
           Triple(EL, Tid, Tid, deq_rec, what for first,
                  enqr for second, deqr for third)
  introduces
    d_before: deq_rec, deq_rec -> Bool
    convert: deq_rec -> enq_rec
  asserts for all (x, x1: deq_rec)
    d_before(x, x1) == deqr(x) < deqr(x1),
    convert(x) == pair(what(x), enqr(x))
  end
```

LSL's **includes** clause lets one build theories from other theories. For example, the Enq_Rec trait **includes** the Pair trait with a renaming of sort and function identifiers. The first three identifiers rename the sorts for the first component of the pair, the second component, and the pair itself. The other two identifiers (*element* and *enqt*) rename the two selector functions for pairs. *e_before* is an ordering between two enq_rec's defined in terms of the *Tid* (`trans_id`) ordering (<) and the *enqt* components (*Tid*'s) of each. *d_before* is similarly defined for deq_rec's. The *convert* function simply ignores the *deqr* component of a deq_rec and returns a corresponding enq_rec. Enq_Rec's **partitioned by** clause asserts that two enq_rec's with the same *element* components are indistinguishable. This rather strong assertion simplifies the proof of correctness of the queue implementation by letting us assume that all elements in a queue are distinct.[5]

The heap and stack components of the queue representation are defined using set and stack theories.

[5]Herlihy and Wing make the same assumption in their hand-proof [16], though the correctness of the Avalon/C++ code does not depend on it. It can be relaxed by tagging each item in the queue with a timestamp.

```
Enq_Heap(enq_heap): trait
  includes Enq_Rec, Set(enq_rec, enq_heap)
  introduces
    in_heap: enq_rec, enq_heap -> Bool
    e_in_heap: EL, enq_heap -> Bool
    least: enq_rec, enq_heap -> Bool
    is_top: enq_rec, enq_heap -> Bool
  asserts for all (xp: enq_heap, y, y1:  enq_rec, xt: Tid, xe: EL)
    in_heap(y, xp) == in(y, xp),
    e_in_heap(xe, emptyset) == false,
    e_in_heap(xe, insert(xp, y)) == (element(y)=xe) | e_in_heap(xe, xp),
    least(y, emptyset) == true,
    least(y, insert(xp, y1)) == (enqt(y)<enqt(y1)) & least(y, xp),
    is_top(y, xp) == in_heap(y,xp) & least(y, xp)
  end

Deq_Stack(deq_stack): trait
  includes Deq_Rec, Stack(deq_rec, deq_stack)
  introduces
    deq_before: deq_rec, deq_rec, deq_stack -> Bool
    in_stack: deq_rec, deq_stack -> Bool
    e_in_stack: EL, deq_stack -> Bool
  asserts for all (xk: deq_stack, y, y1, y2: deq_rec, xt: Tid, xe: EL)
    deq_before(y, y1, new) == false,
    deq_before(y, y1, push(xk, y2)) == ((y1=y2) & (in_stack(y, xk))) |
                                       deq_before(y, y1, xk),
    in_stack(y, new) == false,
    in_stack(y, push(xk, y1)) == if y = y1
                                 then true
                                 else in_stack(y, xk),
    e_in_stack(xe, new) == false,
    e_in_stack(xe, push(xk, y)) == (what(y)=xe) | e_in_stack(xe, xk)
  end
```

The Enq_Heap trait introduces functions that test if an enq_rec is in an enq_heap (*in_heap*), if an element has been enqueued (*e_in_heap*), and if a given enq_rec is the minimal in an enq_heap (*is_top* via *least*). The Deq_Stack trait introduces functions that test the relative ordering between two deq_rec's that are in the stack (*deq_before*), if a deq_rec is in a deq_stack (*in_stack*), and if an element is in a deq_stack (*e_in_stack*).

The State trait in Figure 1 defines a theory about the representation states of the queue. *St* is the sort name for queue states; the queue itself is a sequence of operations, each applied to some state to yield a new state. *init* stands for some actual initial state. Every state has two components: *enqd* and *deqd*, just as in the implementation. There are four functions (operations) with range *St*: *enq*, *deq*, *commit*, *abort*. *enq(xst,xt,xe)* represents the state after transaction *xt* enqueues an element *xe* in state *xst*; *deq(xst,xt,xr)*, the state after *xt* dequeues enq_rec *xr* in state *xst*; *commit* and *abort* are defined similarly. *when_enq* and *when_deq* are two boolean functions that capture the conditions under which the enq and deq operations can proceed to change the state of the queue.

The clause *St* **generated by** *(init,deq,enq,commit,abort)* means that if *xst* is a term of *St* then either *xst=init* or *xst* is produced by applying one of *enq*, *deq*, *commit*, and *abort* to a term *xst'* of *St*. Each term represents a history of operations, e.g., *deq(enq(enq(init,...),...),...)* represents two enq operations followed by a deq. Defining a set of generator functions implicitly defines a structural induction rule. Thus, we can easily prove some properties about queue states using structural induction over terms of sort *St*. To prove $\forall x : St.P(x)$, we need to prove only the following:

$$P(init) \wedge \{P(xst) \Rightarrow (P(enq(xst, xt, xe)) \wedge P(deq(xst, xt, xr)) \wedge P(commit(xst, xt)) \wedge P(abort(xst, xt)))\}$$

LP can prove such theorems automatically for any given property *P*.

The clause **St partitioned by** *(deqd,enqd)* adds the following theorem to the theory:

$$\forall x, x_1 : St.(x = x_1) \Leftrightarrow (enqd(x) = enqd(x_1) \wedge deqd(x) = deqd(x_1))$$

Thus, we define a queue state x of sort St by its two components, *enqd(x)* and *deqd(x)*. Two states are equal if and only if they are component-wise equal.

The State trait is an instance of a more generic theory about Avalon/C++'s model of computation. In general, we specify an Avalon/C++ operation by specifying the changes it causes on a state (e.g., the queue representation's stack and heap components) and the condition (e.g., *when_enq*) that must be satisfied before the operation can proceed to effect those changes. Here we exploit our knowledge that each operation in Avalon/C++ is guaranteed to occur indivisibly; hence, since transactions can call operations only one at a time, we need only be concerned about those states before or after an operation.

Moreover, assuming that Avalon/C++'s implementation of the `when` statement is correct, (in particular, that short-term locks are obtained and released properly), not all the states defined by a State trait stand for "reachable" states, states that would occur in the execution of the program. For example, if a given state *c_xst* does not satisfy the condition *when_enq*, then *enq(c_xst, xt, xe)* denotes a state that is not reachable. When trying to prove a property about an Avalon/C++ program, we need be concerned with proving it for only the reachable states. Assuming that the `when` statement's condition is met, as encoded in *when_enq* and *when_deq* for the queue, is analogous in sequential programming to assuming an operation's pre-condition is met.

4.2. Abstraction Function

We define an abstraction function *af* which maps a representation of queue to a set of sequences of operations, i.e., set of histories. For any valid, reachable representation value r we want to prove that for every history H in *af(r)*, H is legal, namely H is a behavior of a FIFO queue.

The following traits define a theory about Avalon/C++'s history-based model of computation, and describe some properties of the abstraction function. We start with a theory about sequences:

```
Sequence (EL, Seq): trait
  introduces
    null: -> Seq
    cons: Seq, EL -> Seq
    append: Seq, Seq -> Seq
    prefix: Seq, Seq -> Bool
    sub: Seq, Seq -> Seq
  asserts Seq generated by (null, cons)
  for all (xs, xs1: Seq, xe, xe1: EL)
    cons(xs,xe)=cons(xs1,xe1) == (xs=xs1) & (xe=xe1),
    append(xs, null) == xs,
    append(null, xs) == xs,
    append(xs, cons(xs1, xe)) == cons(append(xs, xs1), xe),
    prefix(null, xs1) == true,
    prefix(cons(xs, xe), null) == false,
    prefix(cons(xs, xe), cons(xs1, xe1)) == ((xe=xe1) & (xs=xs1)) | prefix(cons(xs, xe), xs1),
    sub(null, xs) == null,
    sub(xs, null) == xs,
    sub(cons(xs, xe), cons(xs1, xe1)) == if ((xs=xs1) & (xe=xe1))
                                         then null
                                         else cons(sub(xs, cons(xs1,xe1)), xe),
    ~(null = cons(xs,xe))
  end
```

Null and *cons* as the constructors. *Append* concatentates two sequences together; *prefix* detemines whether its first argument is a prefix of its second; *sub* returns the subsequence of its first argument with elements in its second removed.

An event is the execution of an `enq` or `deq` operation:

```
State (St): trait
  includes Deq_Stack, Enq_Heap
  introduces
    init: -> St
    deqd: St -> deq_stack
    enqd: St -> enq_heap
    when_enq: St, enq_rec, deq_rec, Tid, EL -> Bool
    enq: St, Tid, EL -> St
    when_deq: St, deq_rec, Tid, enq_rec -> Bool
    deq: St, Tid, enq_rec -> St
    commit: St, Tid -> St
    abort: St, Tid -> St
  asserts St generated by (init,deq,enq,commit,abort)
          St partitioned by (deqd, enqd)
  for all (xst: St, xt : Tid,  xe: EL, xk: deq_stack, xp: enq_heap,
           x, y, w: deq_rec, xl, z, xn: enq_rec)
    deqd(init) == new,
    enqd(init) == emptyset,
    when_enq(xst,z,w,xt,xe) == ((deqd(xst)=new)|(enqr(top(deqd(xst)))<xt)) &
                               (~(in_heap(z, enqd(xst)) & (element(z)=xe))) &
                               (~(in_stack(w, deqd(xst))&(what(w)=xe))),
    deqd(enq(xst, xt, xe)) == deqd(xst),
    enqd(enq(xst, xt, xe)) == insert(enqd(xst), pair(xe, xt)),
    when_deq(xst,x,xt,xn) == ((deqd(xst)=new) | ((deqr(top(deqd(xst))) < xt) &
                                                 (enqr(top(deqd(xst))) < enqt(xn))) &
                                is_top(xn, enqd(xst)) & (enqt(xn) < xt) &
                              (~(in_stack(x,deqd(xst)) & (what(x)=element(xn))))),
    deqd(deq(xst, xt, xn)) == push(deqd(xst),trip(element(xn),enqt(xn),xt)),
    enqd(deq(xst, xt, xn)) == delete(enqd(xst), xn),
    deqd(commit(xst,xt)) == if (~(deqd(xst)=new)) & (deqr(top(deqd(xst))) < xt)
                            then new
                            else deqd(xst),
    enqd(commit(xst, xt)) == enqd(xst),
    in_stack(x,deqd(abort(xst,xt))) == in_stack(x,deqd(xst)) & (~(deqr(x)=xt)),
    deq_before(x,y,deqd(abort(xst,xt))) => deq_before(x,y,deqd(xst)),
    in_heap(xl,enqd(abort(xst,xt))) => ((~(enqt(xl)=xt)) &
                                        (in_heap(xl,enqd(xst)) |
                                        (in_stack(trip(element(xl),enqt(xl),xt),deqd(xst))) &
                                        ~(in_stack(x,deqd(abort(xst,xt))) &
                                        (what(x)=element(xl))))))
end
```

Figure 1: State Trait

```
Event (Ev): trait
  includes Enq_Rec, Deq_Rec,
  introduces
    E: enq_rec -> Ev
    D: deq_rec -> Ev
  asserts Ev generated by (E, D)
          enq_rec partitioned by (E)
          deq_rec partitioned by (D)
  for all (x,x1: enq_rec, y,y1: deq_rec)
    (x=x1)=>(E(x)=E(x1)),
    (y=y1)=>(D(y)=D(y1)),
    ~(E(x)=D(y))
  end
```

A history is simply a sequence of events:

```
History (H): trait
  includes Event, Sequence, Sequence(Ev, H)
  introduces
    c_h1: -> H
    c_h2: -> H
    DEQ: H -> Seq
    ENQ: H -> Seq
    max: Tid, H -> Bool
    min: Tid, H -> Bool
    ordered: H -> Bool
    discard: Tid, H -> H
  asserts for all (xh: H, u:enq_rec, v:deq_rec, xt:Tid)
    ENQ(null) == null,
    ENQ(cons(xh, E(u))) == cons(ENQ(xh),element(u)),
    ENQ(cons(xh, D(v))) == ENQ(xh),
    DEQ(null) == null,
    DEQ(cons(xh, E(u))) == DEQ(xh),
    DEQ(cons(xh, D(v))) == cons(DEQ(xh), what(v)),
    max(xt,null),
    max(xt,cons(xh,E(u))) == max(xt,xh) & (~(enqt(u)<xt)),
    max(xt,cons(xh,D(v))) == max(xt,xh) & (~(deqr(v)<xt)),
    min(xt,null),
    min(xt,cons(xh,E(u))) == min(xt,xh) & (~(xt<enqt(u))),
    min(xt,cons(xh,D(v))) == min(xt,xh) & (~(xt<deqr(v))),
    ordered(null),
    ordered(cons(xh,E(u))) == ordered(xh) & min(enqt(u), xh),
    ordered(cons(xh,D(v))) == ordered(xh) & min(deqr(v), xh),
    discard(xt, null) == null,
    discard(xt, cons(xh,E(u))) == if enqt(u)=xt
                                  then discard(xt,xh)
                                  else cons(discard(xt,xh),E(u)),
    discard(xt, cons(xh,D(v))) == if deqr(v)=xt
                                  then discard(xt,xh)
                                  else cons(discard(xt,xh),D(v))
  end
```

Note the two uses of the Sequence trait in the **includes** clause. The first brings in the theory of sequences with sort *Seq* for sequences of items (in the queue); the second brings in the theory of sequences, introducing the sort *H* for histories, i.e., sequences of events. *c_h1 and c_h2* will be used as two Skolem constants. *ENQ(h)* is the sequence of items enqueued by the subhistory of enq events in *h* and *DEQ(h)* is the sequence of items dequeued by the subhistory of deq events. *ENQ* and *DEQ* will be used to define the type-specific correctness condition. The events in a history are partially *ordered* by their invoking transaction. *discard(t, h)* is the subhistory of events of *h* with all events associated with transaction *t* removed. It will be used to define the effects of an abort event.

The abstraction function maps a queue representation state into a set of histories:

```
Abstraction (A): trait
 includes State, History, Set(H, A)
 introduces
   in_state: H, St -> Bool
   af: St -> A
 asserts for all(xst: St, xh: H, xn,ue: enq_rec, xd,vd: deq_rec, xe: EL,
                 xt: Tid)
   in_state(null, xst) == true,
   in_state(cons(xh, E(ue)), xst) => in_state(xh,xst)&(in_heap(ue, enqd(xst)) |
                          in_stack(trip(element(ue),enqt(ue),c_xt),deqd(xst))),
   in_state(cons(xh, D(vd)), xst) => in_state(xh,xst) & in_stack(vd,deqd(xst)),
   in(xh, af(xst)) => (ordered(xh) & in_state(xh, xst)),
   in(xh,af(enq(xst,xt,xe))) => (in(append(c_h1,c_h2), af(xst)) &
                                  (xh=append(cons(c_h1,E(pair(xe,xt))), c_h2))),
   in(xh,af(deq(xst,xt,xn))) => (in(append(c_h1,c_h2), af(xst)) &
                                  (xh=append(cons(c_h1,
                                  D(trip(element(xn),enqt(xn),xt))), c_h2)) &
                                  (DEQ(c_h2)=null:->Seq)),
   in(xh,af(commit(xst,xt))) => (DEQ(xh)=null:->Seq),
   in(xh,af(abort(xst, xt))) => (in(c_h1, af(xst)) & (xh=discard(xt,c_h1))),
   (prefix(DEQ(append(xh1,xh2)),ENQ(append(xh1,xh2)))&in(append(xh1,xh2),af(xst))&
    not(prefix(DEQ(append(xh1,xh2)),ENQ(append(cons(xh1,E(pair(xe,xt))),xh2))))
    & ordered(append(cons(xh1,E(pair(xe,xt))),xh2))) =>
                                                  not(enqr(top(deqd(xst)))<xt),
  (in(xh,af(xst))&prefix(DEQ(xh),ENQ(xh))&in(xn,enqd(xst))&least(xn,enqd(xst)))
                    => prefix(cons:Seq,EL->Seq(DEQ(xh),element(xn)),ENQ(xh))
end
```

For any queue state *xst*, *af(xst)* is a set of histories. The first three equations define a helping function for *in* which determines when a history is in *af(xst)*. *in_state* "parses" a term representing an arbitrary history and checks that it represents a history in which items have been enqueued and dequeued and are in the appropriate components of the queue representation. Note that *in_state(xh, xst)* does not imply *in(xh,af(xst))*. The fifth equation illustrates how we use "constant" histories to introduce existentially quantified variables in our equations. It states that *xh* is a history in *af(enq(xst,xt,xe))*–the state after a transaction *xt* enqueues an item *xe* in state *xst*–if there are two histories *c_h1,c_h2* such that *append(c_h1,c_h2)* is a history in *af(xst)* and *xh* is the result of inserting the enqueue event in between *c_h1* and *c_h2* (*xh=append(cons(c_h1,E(pair(xe,xt))), c_h2)*). The sixth equation is similarly defined for a dequeue event. These two equations (the fifth and sixth) capture the property that determines when it is legal to insert an operation in the middle of a legal history, or more precisely, when two operations commute. The seventh and eighth equations respectively capture the effects of applying a commit and abort operation to the queue state: committing reinitializes the sequence of pending dequeued items to the null sequence, and aborting discards all effects of the aborting transaction. The equation for abort uses the *discard* function from the Abstraction trait.

The second-to-last equation captures the conditions under which it is not safe to let an enqueue operation proceed. If the prefix property would not hold of the history with the new enqueue operation, then the transaction must wait. The last equation states that if there is a least element in the heap then dequeueing it will preserve the prefix property of the corresponding history.

5. Verification of the Queue

We translate the Larch traits, all checked by the Larch Checker (LC), into input acceptable to LP, which LP calls "theories." The translation from an LSL trait to an LP theory is mechanical. Once a trait has been transformed into an LP theory, users can use the trait name to access the corresponding theory. We used the boolean theory given in Appendix B. The proof procedure is sensitive to the boolean theory that one uses; with a weaker one, we would be unable to prove certain obvious theorems about booleans.

5.1. Representation Invariant

Given the queue specification of the State trait, we first prove that the queue operations preserve the following representation invariants. For all representation values *r*:

1. No item is present in both *r*'s deqd and enqd components;
2. Items are ordered in *r*'s deqd component by their enqueuing and dequeuing trans_id's;
3. Any dequeued item must previously have been enqueued.

which are formally expressed in LP's input language as:

% $\forall$ *reachable xst* $\in$ *State*.$\forall x \in deq\_rec.\forall y \in enq\_rec.$
$Inv1(xst.x.y) == (in\_stack(x.deqd(xst))\&in\_heap(y.enqd(xst))) \Rightarrow not(what(x) = element(y))$

% $\forall$ *reachable xst* $\in$ *State*.$\forall x.x1 \in deq\_rec.$
$Inv2(xst.x.x1) == deq\_before(x.x1.deqd(xst)) \Rightarrow (enqr(x) < enqr(x1))\&(deqr(x) < deqr(x1))$

% $\forall$ *reachable xst* $\in$ *State*.$\forall x \in deq\_rec.$
$Inv3(xst.x) == in\_stack(x.deqd(xst)) \Rightarrow (enqr(x) < deqr(x))$

Now let's see how we use LP to prove one of these theorems, the second invariant, *Inv2(xst, x, x1)*. In the following proof session, all lines preceded by -> are user-given LP commands. All others are LP-generated output. A line preceded by [] (read "box") is generated whenever LP discharges a proof step. As a hint to the reader who wants only a cursory view of the proofs, look for lines that begin with -> and [], and skip to our discussion following each proof transcript fragment. To save space, we present here only that output of LP that indicates what has been added to the system. See [10] for complete transcripts of all proofs.

```
-> thaw Inv
-> set name thm2
-> prove Inv2(xst,x,y) by induction xst St

   The basis step in an inductive proof of Conjecture thm2.1
       Inv2(xst, x, y) -> true
   involves proving the following lemma(s):
   thm2.1.1: Inv2(init, x, y) -> true
           [] Proved by normalization
   The induction step in an inductive proof of Conjecture thm2.1
       Inv2(xst, x, y) -> true
   uses the following equation(s) for the induction hypothesis:
   Induct.2: Inv2(c_xst, x, y) -> true
   The induction step involves proving the following lemma(s):
   thm2.1.2: Inv2(deq(c_xst, vi1, vi2), x, y) -> true
               which reduces to the equation
               ((deqr(x) < deqr(y)) & (enqr(x) < enqr(y)))
                  | not(((trip(element(vi2), enqt(vi2), vi1) = y)
                         & in_stack(x, deqd(c_xst)))
                  | deq_before(x, y, deqd(c_xst)))
           -> true
   thm2.1.3: Inv2(enq(c_xst, vi1, vi2), x, y) -> true
             [] Proved by normalization
   thm2.1.4: Inv2(commit(c_xst, vi1), x, y) -> true
             [] Proved by normalization
   thm2.1.5: Inv2(abort(c_xst, vi1), x, y) -> true
               which reduces to the equation
                ((deqr(x) < deqr(y)) & (enqr(x) < enqr(y)))
                | not(deq_before(x, y, deqd(abort(c_xst, vi1))))
                -> true
```

The **thaw** command makes the *Inv* theory be the "system" with which we are going to work. *Inv* is the name of the theory corresponding to the State trait plus the three equations for the invariants. The **set name** command gives a name (thm2) to the working system plus what will be added later. The third command tells LP to **prove** Inv2 **by induction** over the induction variable *xst* of sort *St*. In general, to prove an equation "e" by induction, LP first generates a set of equations to prove as the basis of the induction by substituting each basis generator of sort "S" for all occurrences of variable "x" in "e" (new variables are introduced, if necessary, as the basis generators' arguments). In this case there is only one basis generator (with no arguments), *init*, and its associated instantiated equation is normalized to true. After LP has proved the basis case (see the first boxed line), it generates a set of equations to serve as the induction hypothesis, substituting new constants (0-ary operators) for "x" in "e"; it then generates a set of equations to prove in the induction step by substituting each non-basis generator of "S" (applied systematically to these constants and to new variables) for "x" in "e." Since we have four non-basis generators (*deq, enq, commit, abort*), we need to prove four lemmas, two of which are proven automatically (by normalization). LP gives names with root "Induct" to the equations arising from the induction bases and induction hypothesis.

Now let's see how to prove thm2.1.5. Notice that LP rewrites the implication ($P \Rightarrow Q$) of *Inv2* to the logically equivalent ($Q \vee \neg P$).

```
-> resume by case deq_before(x,y,deqd(abort(c_xst,vil)))

Case.3.1
     deq_before(c_x, c_y, deqd(abort(c_xst, c_vil))) == true
involves proving Lemma thm2.1.5.1
     Inv2(abort(c_xst, c_vil), c_x, c_y) -> true
Lemma thm2.1.5.1 in the proof by cases of Lemma thm2.1.5
     Inv2(abort(c_xst, c_vil), c_x, c_y) -> true
     Case.3.1: deq_before(c_x, c_y, deqd(abort(c_xst, c_vil)))
is NOT provable using the current partially completed system.  It reduces to
the equation
    (deqr(c_x) < deqr(c_y)) & (enqr(c_x) < enqr(c_y)) -> true
Proof of Lemma thm2.1.5.1 suspended.
-> critical case with State
   deq_before(c_x, c_y, deqd(c_xst)) == true
-> crit thm2 with Induct
   (deqr(c_x) < deqr(c_y)) & (enqr(c_x) < enqr(c_y)) == true
Lemma thm2.1.5.1 in the proof by cases of Lemma thm2.1.5
    Inv2(abort(c_xst, c_vil), c_x, c_y) -> true
    Case.3.1: deq_before(c_x, c_y, deqd(abort(c_xst, c_vil)))
[] Proved by rewriting.
```

The **resume** command takes a proof method (e.g., **case**) and continues work on the current proof-in-progress using the specified method. The method **case** splits the proof into two cases (Case.3.1 above and Case.3.2 below) according to whether or not *deq_before(x,y,deqd(...))* is true. To prove something by **case** t_1. t_2. ... t_n. LP will first try to prove that $(t_1 | t_2 \ldots | t_n) == true$, or simply $(t_1 | not(t_1)) == true$ if n=1.

For the first case above, we first assume that *deq_before(x,y,deqd(...))* is true. Under this assumption, LP tries to prove thm2.1.5 by proving the lemma *Inv2(abort(c_xst, c_vil), c_x, c_y)==true*, where *c_xst, c_vil, c_x, c_y* are new LP-generated constants. Note that for each case, LP generates new constants not already in use and creates a specific instance of the general case by substituting the constants in for the variables (as in the third line above). We guide LP in proving the lemma by invoking the command **critical**[6] which causes an explicit use of the theory *State* and the induction hypothesis. Informally, **critical** *theory1* with *theory2* tells LP to try to prove that the theorem in progress is a consequence of *theory1* plus *theory2*. More formally, **critical** will find all critical pairs [18] between the two rewrite-rule theories given as its arguments. Doing **critical case with State**, LP finds $deq\_before(c\_x, c\_y, deqd(c\_xst)) == true$, which when added to Induct.2, implies thm2.1.5.

Let's continue with the proof:

[6] LP understands unambiguous prefixes of command names so we also use crit in our proofs.

```
Case.3.2
    not(deq_before(c_x, c_y, deqd(abort(c_xst, c_vil)))) == true
involves proving Lemma thm2.1.5.2
    Inv2(abort(c_xst, c_vil), c_x, c_y) -> true
Lemma thm2.1.5.2 in the proof by cases of Lemma thm2.1.5
    Inv2(abort(c_xst, c_vil), c_x, c_y) -> true
    Case.3.2: not(deq_before(c_x, c_y, deqd(abort(c_xst, c_vil))))
[] Proved by rewriting (with unreduced rules).

Lemma thm2.1.5 for the induction step in the proof of Conjecture thm2.1
    Inv2(abort(c_xst, vil), x, y) -> true
[] Proved by cases
    deq_before(x, y, deqd(abort(c_xst, vil)))
     | not(deq_before(x, y, deqd(abort(c_xst, vil))))
```

For the second case, LP tries to prove the lemma assuming:

$not(deq\_before(c\_x.c\_y.deqd(abort(c\_xst.c\_vil))))$.

The proof goes through immediately since *Inv2* becomes vacuously true (it is the "false" case for the antecedent of an implication). The last boxed line shows that the proof for the abort case for the second invariant (thm2.1.5) is completed.

The proof for thm2.1.2 follows a proof procedure similar to that for thm2.1.5; the proc or the other invariants (*Inv1* and *Inv3*) follow procedures similar to that for *Inv2*.

5.2. Correctness of Queue

Combining the theories of the queue representation and the abstraction function, we now prove the following correctness condition, what we have been calling the "prefix" property for the queue example:

% $\forall xh : H. \forall xst : St.$
Correctness Condition: $in(xh.af(xst)) \Rightarrow prefix(DEQ(xh).ENQ(xh))$

To simplify the proof procedure, we first prove three groups of lemmas shown in Figure 2.

Most of the lemmas in Lemma1 are basic properties of sequences and queue histories. Some of them are so obvious that we believe LP should have had them as built-in theorems. For example, LP could have a built-in meta theorem like $(x = y) \Rightarrow op(x) = op(y)$ where op is any user-defined function, thereby giving us lemma1.10 and lemma1.11 "for free." Since LP maintains a proof stack, we must prove all lemmas before we can use them.

The lemmas in Lemma2 and Lemma3 state the conditions under which one can conclude that the prefix property holds for a given history *xh*. Lemma 2.3 handles the case for when an enqueue operation is performed; Lemmas 3.2 and 3.3 inductively handle the case for when a dequeue is performed.

The following is the proof session for lemma1.3, where *theory* is the theory consisting of State, Abstraction and the invariants.

```
->thaw theory
->set name lemma1
->  prove prefix(x,cons(y,z))=>(prefix(x,y) | x=(y,z)) by induction x Seq
    The basis step in an inductive proof of Conjecture lemma1.3
        prefix(x, cons(y, z)) => ((cons(y, z) = x) | prefix(x, y)) -> true
    involves proving the following lemma(s):
    lemma1.3.1: prefix(null, cons(y, z)) => ((cons(y, z) = null) | prefix(null, y))
                -> true
                [] Proved by normalization
  The induction step in an inductive proof of Conjecture lemma1.3
     prefix(x, cons(y, z)) => ((cons(y, z) = x) | prefix(x, y)) -> true
```

```
Lemma1:
lemma1.1:  append(append(x, y), z) == append(x, append(y, z))
lemma1.2:  prefix(x, y) => (append(x, sub(y, x)) = y)
lemma1.3:  prefix(x, cons(y, z)) => (cons(y, z) = x) | prefix(x, y))
lemma1.4:  append(ENQ(x), ENQ(y)) == ENQ(append(x, y))
lemma1.5:  append(DEQ(x), DEQ(y)) == DEQ(append(x, y))
lemma1.6:  ENQ(append(cons(x, E(y)), z)) == append(cons(ENQ(x), element(y)), ENQ(z))
lemma1.7:  ENQ(append(cons(x, D(y)), z)) == ENQ(append(x, z))
lemma1.8:  DEQ(append(cons(x, E(y)), z)) == DEQ(append(x, z))
lemma1.9:  DEQ(append(cons(x, D(y)), z)) == append(cons(DEQ(x), what(y)), DEQ(z))
lemma1.10: (x = y) => (DEQ(x) = DEQ(y))
lemma1.11: (x = y) => (ENQ(x) = ENQ(y))
lemma1.12: in_state(x, init)) => (x = null)
lemma1.13: prefix(x, y) => prefix(x, cons(y, z))
lemma1.14: prefix(cons(x, z), y) => prefix(x, y)
lemma1.15: in_state(cons(xh, we), xst)) => in_state(xh, xst)
lemma1.16: prefix(x, append(x, y))
lemma1.17: (in_state(xh, xst) & prefix(DEQ(xh), ENQ(xh))) =>
               prefix(DEQ(discard(xt, xh)), ENQ(discard(xt, xh)))

Lemma2:
lemma2.1: ((deqd(xst) = new) & in_state(xh, xst)) => (DEQ(xh) = null)
lemma2.2: (xh = xh1) => (ordered(xh) <=> ordered(xh1))
lemma2.3: ((append(cons(xh1, E(pair(xe, xt))), xh2) = xh)
           & (ordered(xh)
               & (prefix(DEQ(append(xh1, xh2)), ENQ(append(xh1, xh2)))
                   & (in(append(xh1, xh2), af(xst)) & (enqr(top(deqd(xst))) < xt)))))
           => prefix(DEQ(xh), ENQ(xh))

Lemma3:
lemma3.1: (DEQ(append(xh, xh1)) = null) => ((DEQ(xh) = null) & (DEQ(xh1) = null))
lemma3.2: ((append(cons(xh1, D(trip(element(xn), enqt(xn), xt))), xh2) = xh)
             & ((DEQ(xh1) = null)
               & ((DEQ(xh2) = null)
                   & (in(append(xh1, xh2), af(xst))
                     & (in(xn, enqd(xst)) & least(xn, enqd(xst)))))))
          => prefix(DEQ(xh), ENQ(xh))
lemma3.3: ((DEQ(xh2) = null)
           &(append(cons(xh1, D(trip(element(xn), enqt(xn), xt))), xh2) = xh)
             & (in(append(xh1, xh2), af(xst))
                 & (in(xn, enqd(xst))
                     & (least(xn, enqd(xst))
                         & (prefix(DEQ(append(xh1, xh2)), ENQ(append(xh1, xh2)))
                             & (enqr(top(deqd(xst))) < enqt(xn)))))))
           => prefix(DEQ(xh), ENQ(xh))
```

Figure 2: Lemmas for Correctness Proof

```
uses the following equation(s) for the induction hypothesis:
Induct.6: prefix(c_x, cons(y, z)) => ((c_x = cons(y, z)) | prefix(c_x, y))
          -> true
The induction step involves proving the following lemma(s):
lemma1.3.2: prefix(cons(c_x, vi1), cons(y, z))
            => ((cons(c_x, vi1) = cons(y, z)) | prefix(cons(c_x, vi1), y))
             -> true
            [] Proved by normalization
Conjecture lemma1.3
    prefix(x, cons(y, z)) => ((cons(y, z) = x) | prefix(x, y)) -> true
[] Proved by induction over 'x' of sort 'Seq'.
```

Here, we need only tell LP to **prove** the lemma using **induction**. LP is then able to prove the theorem automatically because we have defined the predicate *prefix* over *Seq* recursively and through structural induction, LP considers all possible terms of sort *Seq*.

Now let us look at the proof session for the **Correctness Condition**, where *theory3* is *theory* plus L[illegible]a1, Lemma2, and Lemma3:

```
->thaw theory3
-> set name sync
The name prefix is now 'sync'.
-> prove in(xh,af(xst))=>prefix(DEQ(xh),ENQ(xh)) by induction xst St
The basis step in an inductive proof of Conjecture sync.1
    in(xh, af(xst)) => prefix(DEQ(xh), ENQ(xh)) -> true
involves proving the following lemma(s):
sync.1.1: in(xh, af(init)) => prefix(DEQ(xh), ENQ(xh)) -> true
              which reduces to the equation
              (false <=> in(xh, af(init))) | prefix(DEQ(xh), ENQ(xh)) -> true
Proof of Lemma sync.1.1 suspended.
```

sync.1.1 is the base case for the initial state *init*.

```
-> resume by case in(xh,af(init))
Case.1.1
    in(c_xh, af(init)) == true
involves proving Lemma sync.1.1.1
    in(c_xh, af(init)) => prefix(DEQ(c_xh), ENQ(c_xh)) -> true
Lemma sync.1.1.1 in the proof by cases of Lemma sync.1.1
    in(c_xh, af(init)) => prefix(DEQ(c_xh), ENQ(c_xh)) -> true
    Case.1.1: in(c_xh, af(init))
is NOT provable using the current partially completed system.  It reduces to
the equation
    prefix(DEQ(c_xh), ENQ(c_xh)) -> true
Proof of Lemma sync.1.1.1 suspended.
-> crit case with Abstraction
  sync.2.1: in_state(c_xh, init) == true
  sync.2.2: ordered(c_xh) == true
-> crit sync with lemma1.12
  sync.3.1: c_xh == null
Lemma sync.1.1.1 in the proof by cases of Lemma sync.1.1
    in(c_xh, af(init)) => prefix(DEQ(c_xh), ENQ(c_xh)) -> true
    Case.1.1: in(c_xh, af(init))
[] Proved by rewriting.
```

Recall that in the Abstraction trait, instead of defining *af(xst)* explicitly, we just stated properties of histories in *af(xst)* for all *xst*. For LP to use these properties we need to assume the **case** *in(xh,af(init))* and tell LP to find the **critical** consequences of the case [illegible] *Abstraction* and lemma1.12. After both **crit**'s LP proves the positive case easily by rewriting.

Now we have to consider the ne[illegible]ve (*not(...)*) case, which, as before, goes through automatically because it is vacuously true:

```
Case.1.2
    not(in(c_xh, af(init))) == true
involves proving Lemma sync.1.1.2
    in(c_xh, af(init)) => prefix(DEQ(c_xh), ENQ(c_xh)) -> true
Lemma sync.1.1.2 in the proof by cases of Lemma sync.1.1
    in(c_xh, af(init)) => prefix(DEQ(c_xh), ENQ(c_xh)) -> true
    Case.1.2: not(in(c_xh, af(init)))
[] Proved by rewriting (with unreduced rules).

Lemma sync.1.1 for the basis step in the proof of Conjecture sync.1
    in(xh, af(init)) => prefix(DEQ(xh), ENQ(xh)) -> true
[] Proved by cases
    in(xh, af(init)) | not(in(xh, af(init)))
```

We are now left with the inductive steps. LP generates the induction hypothesis, and the four inductive cases:

```
The induction step in an inductive proof of Conjecture sync.1
    in(xh, af(xst)) => prefix(DEQ(xh), ENQ(xh)) -> true
uses the following equation(s) for the induction hypothesis:
Induct.1: in(xh, af(c_xst)) => prefix(DEQ(xh), ENQ(xh)) -> true
The induction step involves proving the following lemma(s):
sync.1.2: in(xh, af(deq(c_xst, vi1, vi2))) => prefix(DEQ(xh), ENQ(xh)) -> true
              which reduces to the equation
              (false <=> in(xh, af(deq(c_xst, vi1, vi2))))
               | prefix(DEQ(xh), ENQ(xh))
              -> true
sync.1.3: in(xh, af(enq(c_xst, vi1, vi2))) => prefix(DEQ(xh), ENQ(xh)) -> true
              which reduces to the equation
              (false <=> in(xh, af(enq(c_xst, vi1, vi2))))
               | prefix(DEQ(xh), ENQ(xh))
              -> true
sync.1.4: in(xh, af(commit(c_xst, vi1))) => prefix(DEQ(xh), ENQ(xh)) -> true
              which reduces to the equation
              (false <=> in(xh, af(commit(c_xst, vi1))))
               | prefix(DEQ(xh), ENQ(xh))
              -> true
sync.1.5: in(xh, af(abort(c_xst, vi1))) => prefix(DEQ(xh), ENQ(xh)) -> true
              which reduces to the equation
              (false <=> in(xh, af(abort(c_xst, vi1))))
               | prefix(DEQ(xh), ENQ(xh))
              -> true
Proof of Lemma sync.1.5 suspended.
```

The proofs of each case follow a familiar pattern. For each case, a number of crit's are performed, perhaps with some extra equations added to the working theory. Figure 3 shows the outline of the entire proof. One of LP's user amenities is a command for logging all user input (the "->"'s do not appear in the figure) and boxed lines into a file during a proof session, thus saving a proof outline, which can be used to replay a proof.

6. Discussion

6.1. Experimental Statistics

We used the 0.9 Beta Test (June 28, 1989) version of LP running on a DEC Microvax-3 with 16 megabytes of primary memory. We used a 2 megabyte heap size for LP.

Tables 1, 2, and 3 summarize some of the statistics taken from our usage of LP. The first table indicates how much user input to LP was required in specifying each of the theories. *Declarations* refer to the variable, constant, and non-generator function declarations. Since theories are built from previously defined ones, as one goes down the column, we indicate only the new declarations that each theory makes. The *Generators* and *Equations* columns

```
thaw theory3
set name sync
 prove in(xh, af(xst)) = prefix(DEQ(xh),ENQ(xh)) by induction xst St
 resume by case in(xh, af(init))
crit case with Abstraction
crit sync with lemma1.12
    [] % rewriting for case in(c_xh, af(init))
    [] % rewriting (with unreduced rules) for case not(in(c_xh, af(init)))
  [] % cases in(xh, af(init)) | not(in(xh, af(init))) for basis step
resume by case in(xh,af(abort(c_xst,vi1)))
crit case with Abstraction
crit induct with sync
 crit sync with lemma1.17
 crit sync.5.2 with Abstraction.5
 crit sync with sync
         [] % rewriting for case in(c_xh, af(abort(c_xst, c_vi1)))
         [] % rewriting (with unreduced rules) for case not(in(c_xh, af(abort(c_xst, c_vi1))))
       [] % cases in(xh, af(abort(c_xst, vi1))) | not(in(xh, af(abort(c_xst, vi1)))) for
induction step
 resume by case in(xh,af(commit(c_xst,vi1)))
crit case with Abstraction
        [] % rewriting for case in(c_xh, af(commit(c_xst, c_vi1)))
        [] % rewriting (with unreduced rules) for case not(in(c_xh, af(commit(c_xst, c_vi1))))
      [] % cases in(xh, af(commit(c_xst, vi1))) | not(in(xh, af(commit(c_xst, vi1)))) for
induction step
 resume by case in(xh,af(enq(c_xst,vi1,vi2::EL)))
 add when_enq(c_xst,z,w,c_vi1,c_vi2)
 crit case with Abstraction
 resume by case deqd(c_xst)=new
 crit case with lemma2.1
 crit sync with lemma1.8
 crit sync.16.2 with Abstraction.5
 crit sync with sync
        [] % rewriting for case deqd(c_xst) = new
crit induct with sync.16.2
instantiate xh1 by c_h1,xh2 by c_h2,xh by c_xh,xe by c_vi2,xt by c_vi1,xst by c_xst in lemma2.3
        [] % rewriting for case not(deqd(c_xst) = new)
      [] % cases (deqd(c_xst) = new) | not(deqd(c_xst) = new) for case in(c_xh, af(enq(c_xst,
c_vi1, c_vi2)))
      [] % rewriting (with unreduced rules) for case not(in(c_xh, af(enq(c_xst, c_vi1, c_vi2))))
    [] % cases in(xh, af(enq(c_xst, vi1, vi2))) | not(in(xh, af(enq(c_xst, vi1, vi2)))) for
induction step
resume by case in(xh,af(deq(c_xst,vi1,vi2::enq_rec)))
crit case with Abstraction
add when_deq(c_xst,x,c_vi1,c_vi2)
 crit induct with sync
resume by case deqd(c_xst)=new
 crit case with lemma2.1
 crit sync.25.3 with Abstraction.5
 crit sync.30.1 with sync.29
 crit sync.31.1 with lemma3.1
 instantiate xh1 by c_h1,xh2 by c_h2,xn by c_vi2,xt by c_vi1,xh by c_xh,xst by c_xst in lemma3.2
      [] % rewriting for case deqd(c_xst) = new
 instantiate xst by c_xst, xh by c_xh,xh1 by c_h1,xh2 by c_h2,xn by c_vi2,xt by c_vi1 in lemma3.3
      [] % rewriting for case not(deqd(c_xst) = new)
    [] % cases (deqd(c_xst) = new) | not(deqd(c_xst) = new) for case in(c_xh, af(deq(c_xst,
c_vi1, c_vi2)))
    [] % rewriting (with unreduced rules) for case not(in(c_xh, af(deq(c_xst, c_vi1, c_vi2))))
  [] % cases in(xh, af(deq(c_xst, vi1, vi2))) | not(in(xh, af(deq(c_xst, vi1, vi2)))) for
induction step
[] % induction over 'xst::St' of sort 'St'
q
```

Figure 3: Entire Proof Outline for Correctness Condition

Theory	*Declarations*	*Generators*	*Equations*	*Deduction rules*
Bool	10	0	21	4
TransID	3	0	2	0
Enq_rec	3	1	3	2
Deq_rec	0	1	5	1
Deq_Stack	1	2	10	0
Enq_Heap	1	2	17	1
State	1	5	13	0
Invariant	0	0	3	0
Event	0	2	3	2
Sequence	3	2	11	0
History	6	2	29	0
Abstraction	5	2	21	1

Table 1: User Input

Proofs	*Number of rules in system*	*Number of commands used during proof*	*Number of rules produced by LP*
Invariant 1	72	25	10+3+10
Invariant 2	72	20	5+16
Invariant 3	72	9	7
sync	176	31	3+12+5+15+20

Table 2: Numbers of Rules and Commands Used in Proofs

are self-explanatory. Finally, for each LSL **partitioned by** clause, LP defines a *deduction rule*, a general kind of a logical inference rule that gives one way for LP users to introduce implicit universal quantification. Applying a deduction rule will yield a set of equations and/or rewrite rules.

For proofs of the three invariants and the correctness condition (sync), Table 2 indicates how many input rules and commands we used and how many rules LP generated. The first column indicates the total number of user-defined rewrite rules and deduction rules. The third column shows the number of rules LP generated. Since a proof can be divided into several stages, e.g., by case analysis or induction steps, the notation $n+m+...$ expresses the maximum number of rules produced by LP during each of the different stages. To save space, LP discards rules once an equation is proven; hence, when proving one theorem LP may actually recompute rules it discarded during the proof of another.

Table 3 indicates the time and space usage by LP. The first column shows the number of successful applications of rules and the total amount of CPU time (in minutes:seconds) taken to perform the applications. The second column indicates the number of attempted (not successful) applications, and corresponding time. Given a set of rules, LP will go through the entire set, perhaps adding more rules in the process, and apply each rule in an attempt to reduce an equation. It will go through this modified set of rules multiple times, hence the reason that the total number of applications of rules far outnumber the actual number of rules. The heap size is the amount of space in bytes taken up at the end of each proof, and not the total amount taken during the proof, since garbage collection may occur during the proof. In fact, LP did 19 garbage collections during the proof of Invariant 1; 10 for Invariant 2; 4 for Invariant 3; and 111 for sync. The fourth column indicates the total amount of CPU time taken for the entire proof session. Of course, in real time, the first attempt at proving, say sync, was more on the order of hours; once the proof is known, re-running it takes about 20 minutes of real time.

Appendix C contains versions of Tables 2 and 3 for all proofs, including those of all the lemmas shown in Figure 2. We computed all time and space numbers using LP's **statistics** command.

Proofs	*Number of successful applications of rules, time*	*Number of attempted applications of rules, time*	*Heap size (bytes)*	*Total time (min:sec)*
Invariant 1	505, 1:20.62	8811, 38.94	351,312	2:49.77
Invariant 2	280, 17.60	5878, 18.79	347,528	1:16.06
Invariant 3	88, 5.53	2197, 6.01	341,240	24.77
sync	387, 3:22.64	30672, 2:15.45	623,804	14:06.52

Table 3: Time and Space Statistics on Proofs

6.2. Comparison between LP and Other Mechanical Proof Checkers

Mechanical theorem-proving tools can be divided into two categories: automatic theorem provers and proof checkers. The first kind have general built-in proof strategies, e.g., resolution, that find proofs without user intervention. The second kind largely depend on the user to guide the proof. In this section we compare LP with two other proof checkers, the Boyer-Moore System (BMS) [2] and LCF [13]. Since all three systems are interactive, they all provide both a meta-language for defining the proof strategy and a language for defining the theory of the problem. In the case of BMS, the distinction between these two languages is subtle. Below, we consider four dimensions of comparison.

Expressibility of Theory: Since LP limits the use of the existential quantifier, its expressibility is weaker than that of LCF which has the expressive power of full predicate calculus. What make LP more attractive is its support for modularity, i.e., the ability to let users construct a large theory from some smaller ones. BMS also limits the use of the existential quantifier. All three systems emphasize the role of recursively-defined functions.

Were we to define *af* over *St* recursively, LP (in theory) would be able to prove theorems using *af* automatically. However, such a definition would look unnaturally complex because it would depend on numerous existentially-quantified variables all of which would have to be skolemized in the ways described in Section 2.3. This complexity is inherent to the abstraction function, *af*, since its range is expressed in terms of mathematical concepts like sets and sequences. A term-rewriting system is best at manipulating syntactic representations of these concepts, yet such terms would would be hard to read and presumably bring LP "to its knees." So instead of defining *af* in a direct, constructive way[7] that would completely define all the elements in the set, we resorte just stating properties that these elements satisfy.

Expressibility of Strategy: By providing a meta-language, ML, for users to define t vn proof strategies, LCF is the most powerful and most flexible of the three. LCF designers claim that in pr le one can program almost any style of proof [6]. In contrast, LP provides some built-in proof strategies, i.e., t. et of LP commands such as **case** and **critical**, all of which have fixed meanings. Users are free to choose among these strategies, and indeed must wisely choose some sequence of commands to push a proof to completion.

BMS is more general than LP since it lets users define "shells," which are like meta-theories with induction schemata. By defining a theory through the "instantiation" of a shell, one simultaneously gets a proof strategy (in particular, an induction rule). Unlike LP, an induction rule can be defined over more than one variable (thus, one can perform simultaneous inductions in BMS instead of having to do nested inductions) and one can choose to do the inductive cases before the base cases.

Finally, LCF goes even further than LP and BMS by providing the ability to glue smaller strategies (called "tactics") together to construct a complete strategy.

Validity of Proof Strategy: While LCF has more flexible ways for devising proof strategies, it cannot guarantee the validity of a user-defined strategy. With LCF, it is possible for a user to prove something which is nonsense. The user is responsible for proving the validity of his or her strategy. To alleviate this problem, LCF encourages users to adopt a preferred style that uses only valid tactics. LP and BMS do not face this problem since everything

[7] For example, as given in [24].

proved must be a consequence of the defined theory.

Pragmatic Aspects: LCF supports any style of proof, though this expressibility is not achieved without cost; users themselves must go through the trouble and tedium of defining the proof strategies. At the other extreme, some argue that BMS is almost an automatic theorem prover since it has some well-defined built-in proof strategies, called *heuristic searches*, many of which it invokes without user intervention. In most cases, however, the user still guides BMS to find a proof with BMS providing some possible strategies from which the user can choose. During a proof session, if a choice point is encountered, BMS will propose a number of possible next steps and list the lemmas the user would need to prove in order to get to each of those next steps. Between these two extremes is LP, which provides a fixed set of proof strategies and leaves it up to the user to know when to invoke which as well as what additional lemmas are needed. The user also has more freedom moving around BMS's proof tree than LP's proof stack.

Using any of these three systems requires that one have a clear proof outline in mind. Using a prover like LCF requires not only having the theorem's proof outline in mind but also the particular proof strategies one would use in the proof itself. In this sense, LCF users need to be logicians.

7. Conclusions and Future Directions

When we began, we were familiar with and knowledgeable about both the specificand domain, Avalon/C++, and the specification language, Larch; one of us (JMW) was involved in the design of both. Our knowledge of LP at first was only superficial, but not naive. As our experiment progressed, one of us (CG) became much more proficient in using LP. Based on our experience, we conclude the following.

The specificand domain is complex. We knew this from the start. Going through the exercise of formally specifying a model of computation for Avalon/C++ and the specific queue example down to the level of detail that can be used as input to a proof checker made Avalon/C++'s intricacies painstakingly clear. Yes, the specificand is complex and no amount of machine assistance is going to make that less complex.

The prover is complex. We used only a small subset of the full functionality of LP. To use LP at its fullest and perhaps more effectively than we have illustrated here, the user needs to understand concepts from rewrite-rule theory, e.g., confluence, termination, convergence, termination orderings[8]. The user needs to know the theoretical and practical implications of invoking each of the related commands. For example, given a set of equations and rewrite rules, the **complete** command will attempt (by computing all critical pairs) to find a convergent set of rewrite rules that decides the equational theory of the original system. Instead of naively applying **complete** to our specifications, which would certainly exhaust heap space and probably not terminate, we chose the more conservative and more manageable strategy of computing specific sets of critical pairs at "critical" instances in our proofs.

Proving is like programming. Using LP is like programming since the user designs a proof and lets LP execute it. Getting a proof to go through requires iterations through specification (of the specificand), design (of the proof), and "implementation" (checking the proof). Debugging occurs at all phases. One changes the specification because not enough has been stated for the proof to go through. One changes the proof design because the proof path leads nowhere or because the specification has changed.

Using a proof checker requires forethought, patience (human cycles), and machine cycles. Given mechanical tools for theorem proving, users may easily be lured into thinking or hoping that the tool will find the proof for them. A proof checker does not decrease the amount of thinking required on the user's part; it can alleviate some of the bookkeeping and symbol pushing, but no more.

These conclusions may all sound like platitudes, and are certainly familiar to those who have worked with proof checkers, but they are worth repeating. Harder questions to answer are how far has theorem proving technology

[8]There are nine commands alone that deal with orderings.

gone, where is it going, and where should it go? To what use can we put mechanical theorem proving tools in practice?

Although we have no definitive answers to such questions, from our experience using LP, we believe that current mechanical theorem proving tools can be used today for medium-sized, well-defined, domain-specific problems, e.g., hardware circuits [9,4], microprocessors [17,5], operating systems kernels [1], and secure systems [20]. We suggest two areas of research to push against our current technological limits:

1. To build parallel systems that exploit parallel architectures and parallelized versions of standard theorem-proving algorithms (like Knuth-Bendix [18]). In theory, it would have been more convenient to invoke the **complete** command to have LP produce all consequences by computing all the critical pairs of our entire Avalon/C++ queue specification. In practice, we would have paid significant performance penalties. A parallel proving system could instead support a proof strategy in which relatively independent calculations are performed in parallel, e.g., computing critical pairs in parallel with executing the main proof.
2. To build a library of theories that are relevant to computer science. We had to start from scratch (booleans, sets, sequences, stacks, etc.) before we could even state the queue's correctness condition. With the exception of the Larch Handbook of Traits [15], there is a lack of pre-defined reusable theories for standard mathematical concepts that programmers use or assume. Ideally such a library of theories would be reusable across different theorem-proving tools, but they at least should be general enough for a variety of applications. They should also be extensible so that users can specialize the general theories as well as add their own application-specific theories, as we did with the History trait for Avalon/C++.

Though it may be a long time before a powerful enough mechanical theorem proving tool is built such that software engineers can use it in practice, pursuing the above two lines of research may help get us there quicker.

Acknowledgments

We thank members of the Larch Project at MIT and DEC/SRC, in particular Steve Garland, John Guttag, and Jim Horning, for providing us with LP. All three were extremely helpful and patient in providing guidance and answering questions as we used LP. We also thank members of the Avalon Project for realizing Avalon/C++, and Maurice Herlihy for his work with us on the hand-proof of the queue example. Finally, we thank David Detlefs, Rick Lerner, and Amy Moormann Zaremski for their comments on this paper.

References

[1] W.R. Bevier. *A Verified Operating System Kernel.* Technical Report 11, Computational Logic, Inc., March 1987.

[2] R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, 1979. ACM monograph series.

[3] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2):244–263, 1986.

[4] E.M. Clarke and O. Grumberg. Research on automatic verification of finite-state concurrent systems. *Ann. Rev. Comput. Sci.*, 2:269–290, 1987.

[5] W.J. Cullyer. Implementing safety-critical systems: the viper microprocessor. In *VLSI Specification, Verification and Synthesis*, Kluwer, 1987.

[6] D. L. Detlefs, M. P. Herlihy, and J. M. Wing. Inheritance of synchronization and recovery properties in Avalon/C++. *IEEE Computer*, December 1988.

[7] Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972.

[8] 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.

[9] S.J. Garland, J.V. Guttag, and J. Staunstrup. Verification of vlsi circuits using lp. In *Proceedings of the IFIP WG 10.2, The Fusion of Hardware Design and Verification*, North-Holland, 1988.

[10] C. Gong and J.M. Wing. *Raw Code, Specification and Proof of the Avalon Queue Example.* Technical Report CMU-CS-89-172, CMU School of Computer Science, August 1989.

[11] D.I. Good, R.L. London, and W.W. Bledsoe. An interactive program verification system. *IEEE Transactions on Software Engineering*, 1(1):59–67, 1979.

[12] M. Gordon. Hol: a proof generating system for higher-order logic. In *VLSI Specification, Verification and Synthesis*, Kluwer, 1987.

[13] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF.* Volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, 1979.

[14] J.V. Guttag, J.J. Horning, and J.M. Wing. The larch family of specification languages. *IEEE Software*, 2(5):24–36, September 1985.

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

[16] M.P. Herlihy and J.M. Wing. Reasoning about atomic objects. In *Proceedings of theSymposium on Formal Techniques in Real-time and Fault-tolerant Systems (LNCS 331)*, pages 193–208, Springer-Verlag, 1988.

[17] W.A. Hunt. *The Mechanical Verification of a Microprocessor Design.* Technical Report 6, Computational Logic, Inc., 1987.

[18] Knuth and Bendix. *Simple Word Problems in Universal Algebras*, pages 263–297. Pergamon Press, Elmsford, NY, 1970.

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

[20] A.P. Moore. Investigating formal specification and verification techniques for comsec software security. In *Proceedings of the 1988 National Computer Security Conference*, October 1988.

[21] A.Z. Spector, R. Pausch, and G. Bruell. Camelot: a flexible, distributed transaction processing system. In *Proceedings of Compcon 88*, San Francisco, CA, February 1988.

[22] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 1986.

[23] W.E. Weihl. Local atomicity properties: modular concurrency control for abstract data types. *Transactions on Programming Languages and Systems*, 11(2):249–283, April 1989.

[24] J.M. Wing. Verifying atomic data types. In *Lecture Notes in Computer Science: Proceedings of the REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness*, Springer-Verlag, Berlin, 1989. CMU Technical Report CMU-CS-89-168.

Appendix A. Auxiliary Traits

```
Set (EL, C): trait
  introduces
    emptyset: -> C
    insert: C, EL -> C
    in: EL, C-> Bool
    notin: EL, C -> Bool
    U: C, C -> C
    insect: C, C -> C
    __-__: C, C -> C
    delete: C, EL -> C
    subseteq: C, C -> Bool
    isEmpty: C -> Bool
  asserts C generated by (emptyset, insert)
          C partitioned by (in)
    for all (y, y1: C, x, x1: EL)
      ~(in(x, emptyset)),
      in(x, insert(y, x1)) == (x = x1) | in(x, y),
      notin(x, y) == ~(in(x, y)),
      in(x, U(y, y1)) == in(x, y) | in(x, y1),
      in(x, insect(y, y1)) == in(x, y) & in(x, y1),
      in(x, (y - y1)) == in(x, y) & notin(x, y1),
      in(x, delete(y, x1)) == (x \= x1) & in(x, y),
      subseteq(emptyset, y1),
      subseteq(insert(y, x), y1) == subseteq(y, y1) & in(x, y1),
      isEmpty(emptyset),
      ~isEmpty(insert(y, x))
  end

Stack (EL, C): trait
  introduces
    new: -> C
    push: C, EL -> C
    top: C -> EL
    pop: C -> C
    isNew: C -> Bool
  asserts
    C generated by (new, push)
    for all (x: C, y: EL)
      top(push(x, y)) == y,
      pop(push(x, y)) == x,
      isNew(new),
      ~ isNew(push(x, y))
  end

Pair (T1, T2, T): trait
  introduces
    pair: T1, T2 -> T
    first: T -> T1
    second: T -> T2
  asserts
    T generated by (pair)
    T partitioned by (first, second)
    for all (x: T1, y: T2)
      first(pair(x,y)) == x,
      second(pair(x,y)) == y
  end

Triple (T1, T2, T3, T): trait
  introduces
    trip: T1, T2, T3 -> T
    first: T -> T1
    second: T -> T2
    third: T -> T3
  asserts
    T generated by (trip)
    T partitioned by (first, second, third)
```

```
  for all (x: T1, y: T2, z: T3)
    first(trip(x,y,z)) == x,
    second(trip(x,y,z)) == y,
    third(trip(x,y,z)) == z
end
```

Appendix B. Boolean Theory

```
set name bool
declare
  true:->bool
  false:->bool
  &:bool,bool->bool
  |:bool,bool->bool
  <=>:bool,bool->bool
  =>:bool,bool->bool
  not:bool->bool
  b::bool
  b1::bool
  b2::bool
..

op ac <=> & |
op prec <=> &
op prec <=> |

add
  true & b -> b
  false & b -> false
  b & b -> b
  not(b) -> false <=> b
  true <=> b -> b
  not(b) & b -> false
  true | b -> true
  false | b -> b
  b | b -> b
  not(b) | b -> true
  b => b1 -> not(b) | b1
  (b | b1) & b -> b
  % not(b) & not(b1) -> not(b | b1)
  not(b | b1) -> not(b) & not(b1)
  % not(b) | not(b1) -> not(b & b1)
  not(b & b1) -> not(b) | not(b1)
  b & (not(b) | b1) -> b & b1
  (b | b1) & not(b) & not(b1) -> false
  (b | b1) & (b | not(b1)) -> b
  (b & b1) | not(b1) -> b | not(b1)
  (b & b1) | (b & not(b1)) -> b
  b | (not(b) & b1) -> b | b1
  b | (b & b1) -> b
  (b <=> b1) | (b1 <=> b2) | (b <=> b2) -> true
..

add-ded
  when (b <=> false) == false
  yield b -> true
  when b <=> b1 == b <=> b2
  yield b1 == b2
  when if(b, b1, b2) == true
  yield b => b1
        b | b2
  when if(b, b1, b2) == false
  yield b1 => not(b)
        b2 => b

..
```

Appendix C. LP Usage and Performance Statistics

In the following two tables, Lemma.1, Lemma.2, and Lemma.3 refer to the following three (simple) lemmas that we also need to prove:

```
1. (x=pair(y,z))=>((element(x)=y)&(enqt(x)=z))
2. (x=trip(u,v,w))=>((what(x)=u)&(enqr(x)=v)&(deqr(x)=w))
3. in_stack(x,y)=>(deq_before(x,top(y),y) | (x=top(y)))
```

The first two were proven use the **case** command; the third by **induction**.

Proofs	*Number of rules in system**	*Number of commands used during proof*	*Number of rules produced by LP***
Invariant 1	72	25	10+3+10
Invariant 2	72	20	5+16
Invariant 3	72	9	7
lemma.1	147	1	1
lemma.2	148	1	1
lemma.3	149	1	1
lemma1.1	150	1	1
lemma1.2	151	4	2
lemma1.3	152	1	1
lemma1.4	153	2	1
lemma1.5	154	2	1
lemma1.6	155	2	1
lemma1.7	156	2	1
lemma1.8	157	2	1
lemma1.9	158	2	1
lemma1.10	159	9	3
lemma1.11	160	9	2
lemma1.12	161	6	3
lemma1.13	162	6	5
lemma1.14	150	5	3
lemma1.15	164	6	5+5
lemma1.16	165	2	2
lemma1.17	166	8	7
lemma2.1	167	9	3+4
lemma2.2	168	4	2
lemma2.3	169	8	13
lemma3.1	172	3	2
lemma3.2	173	11	17
lemma3.3	175	9	13
sync	176	31	3+12+5+15+20

*: Includes rewrite rules and deduction rules.

**: A proof can be divided into several stages (e.g., by case analysis or induction steps). n+m+... expreases the maximum numbers of rules produced by LP during each stage.

Table 4: Numbers of Rules and Commands Used in Proofs

Proofs	*Number of successful applications of rules, time*	*Number of attempted applications of rules, time*	*Heap size (bytes)*	*Total time (min:sec)*
Invariant 1	505, 1:20.62	8811, 38.94	351,312	2:49.77
Invariant 2	280, 17.60	5878, 18.79	347,528	1:16.06
Invariant 3	88, 5.53	2197, 6.01	341,240	24.77
lemma.1	20, 0.34	210, 0.53	320,460	6.96
lemma.2	24, 0.74	212, 0.61	333,816	6.26 1
lemma.3	19, 0.49	372, 1.11	337,276	3.85
lemma1.1	6, 0.04	640, 1.72	560,096	9.98
lemma1.2	101, 9.93	1338, 4.16	522,084	22.99
lemma1.3	19, 0.53	548, 1.73	526,022	5.71
lemma1.4	13, 0.15	556, 1.59	529,004	5.61
lemma1.5	13, 0.16	560, 1.63	531,804	5.67
lemma1.6	16, 1.02	990, 2.99	535,352	9.06
lemma1.7	21, 0.24	668, 1.74	538,396	5.90
lemma1.8	21, 0.24	672, 1.79	541,444	6.18
lemma1.9	14, 0.16	676, 1.83	545,340	6.16
lemma1.10	162, 9.34	2836, 11.38	552,412	42.01
lemma1.11	162, 10.35	2850, 14.10	558,616	46.47
lemma1.12	66, 3.80	2152, 7.76	563,484	21.88
lemma1.13	116, 11.77	2912, 16.69	899,696	50.68
lemma1.14	216, 15.87	4444, 20.06	573,348	1:38.07
lemma1.15	40, 0.77	3580, 12.22	578,244	30.67
lemma1.16	11, 0.24	1048, 2.50	579,336	7.22
lemma1.17	239, 21.98	3327, 12.57	587,612	57.25
lemma2.1	119, 4.59	3331, 14.19	566,048	46.44
lemma2.2	48, 2.13	1474, 9.40	570,752	20.22
lemma2.3	81, 12.56	6381, 32.50	586,236	1:35.90
lemma3.1	32, 4.93	1088, 3.79	579,592	14.42
lemma3.2	139, 39.83	7712, 59.31	597,828	3:08.25
lemma3.3	101, 1:31.66	6923, 4:02.78	619,916	6:45.95
sync	387, 3:22.64	30672, 2:15.45	623,804	14:06.52

Table 5: Time and Space Statistics on Proofs