# Experience with the Larch Prover

Jeannette M. Wing and Chun Gong
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

March 8, 1990

## 1. Experimental Goals and Summary

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. People often miss boundary cases or forget 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.

This position paper summarizes our experience [18] using the Larch Prover (LP) [6] as a mechanical aid for proving properties of Avalon/C++ programs [5]. Avalon/C++ is a programming language 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*, the basic correctness condition that must be shown of each Avalon/C++ object. 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 [8], LCF [10], HOL [9], and Clarke's model checker [2].

We began our 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 [11];
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.

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 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++

[1] We have implemented Avalon/C++ at CMU as an extension of C++ [17], using the Camelot transaction facility [16] for its runtime system. The Larch Prover was implemented at MIT.



Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development. Napa, California, May 9-11, 1990.

with our use of LP, we were forced to be extremely explicit about Avalon/C++'s computational model 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.

## 2. The Experiment and Concrete Results

*The Experiment.* We encoded in the Larch Shared Language (LSL) [11] an implementation of a FIFO queue written in Avalon/C++. In particular, we wrote LSL specifications to describe the underlying history-based computational model for Avalon/C++, the queue representation (a pair of a partially ordered heap and a stack), the queue operations (enqueue and dequeue), constraints on an abstraction function mapping a representation state to an abstract (queue) "state" (see below), and a queue-specific correctness condition. We performed a trivial transformation from LSL specifications to LP input.[2] Finally, we used LP to prove the correctness condition, which essentially states that all histories (which may have interleaving queue operations performed by different transactions) preserve the first-in first-out property of queues.

*Results.* One concrete result from this exercise is a three-page LSL specification of a one-page Avalon/C++ implementation of a FIFO queue. This specification includes an encoding of Avalon/C++'s computational model specialized for the queue. Since an abstraction function for Avalon/C++ objects maps a single representation state to a set of sequences of abstract operations, rather than to a single abstract state, this encoding is non-trivial.

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. The proof transcripts of the queue's correctness condition plus all helping lemmas came to 168 pages, though the proof outline (user commands only) of the correctness condition is only one page long.

## 3. What We Learned and Where to Go From Here

*The specificand domain is complex.* We knew this from the start. Going through the exercise of formally specifying Avalon/C++'s computational model 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 did, the user needs to understand concepts from rewrite-rule theory (e.g., confluence, termination, convergence, and termination orderings), and 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.[3]

*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. The specification changes because not enough has been stated for the proof to go through. The proof design changes because the current proof path leads nowhere or because the specification has changed.

[2] There was no direct interface between the LSL syntax checker and LP at the time we did the experiment.

[3] Informally, computing critical pairs produces equational consequences from incomplete rewriting systems.

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

We believe that current mechanical theorem proving tools can be used today for medium-sized, well-defined, domain-specific problems, e.g., hardware circuits [7,3], microprocessors [13,4], operating systems kernels [1], and secure systems [15]. 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 [14]). 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 [12], 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.

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, in particular David Detlefs, for realizing Avalon/C++, and Maurice Herlihy for his work with us on the hand-proof of the queue example.

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.

**References**


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

[2] 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.

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

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

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

[6] S.J. Garland and J.V. Guttag. An overview of LP, the Larch Prover. In *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, pages 137–151, Chapel Hill, NC, April 1989. Lecture Notes in Computer Science 355.

[7] 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.

[8] 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.

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

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

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

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

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

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

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

[16] 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.

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

[18] J.M. Wing and C. Gong. *Machine-Assisted Proofs of Properties of Avalon Programs.* Technical Report CMU-89-171, Carnegie Mellon University, August 1989.