Mechanical Translation of I/O Automaton Specifications into First-Order Logic

Andrej Bogdanov[1], Stephen J. Garland[2], and Nancy A. Lynch[2]

[1] UC Berkeley, EECS – Computer Science Division, Berkeley, CA 94720
[2] MIT Laboratory for Computer Science, Cambridge, MA 02139

Abstract. We describe a tool that improves the process of verifying relations between descriptions of a distributed algorithm at different levels of abstraction using interactive proof assistants. The tool automatically translates algorithms, written in the IOA language, into first-order logic, expressed in the Larch Shared Language, in a style that facilitates reasoning with a theorem prover. The translation uses a unified strategy to handle the various forms of nondeterminism that appear in abstract system descriptions. Applications of the tool to verify safety properties of three data management algorithms, including a substantial example based on Lamport's logical time algorithm, suggest that the tool can be used to validate complicated, practical designs.

1 Introduction

High-level descriptions of distributed algorithms differ from their low-level counterparts not only in level of detail, but also in style and syntax. The choice of style and syntax is closely related to a description's intended use. High-level descriptions generally serve to specify a range of allowable abstract behaviors. For this purpose, a declarative, nondeterministic style enhances clarity and generality. On the other hand, low-level descriptions generally serve as input to code generators (e.g., compilers). For this purpose, an imperative, deterministic style works best. Program analysis tools, such as model checkers and theorem provers, tolerate nondeterminism better than code generators. In practice, however, nondeterminism can result in search space explosion for model checkers, and it can increase the need for user interaction with theorem provers. For these reasons, designers generally produce different descriptions of an algorithm for different tools—a dangerous practice, given the discrepancies that all too easily appear in multiple descriptions of the same algorithm.

IOA [3], a high-level language based on the I/O automaton formalism [9], intends to bridge the gap between the stages in the design of distributed algorithms through a versatile syntax for describing algorithms, their properties, and their interrelations. The language supports expressing designs at different levels of abstraction, starting with a high-level abstract specification of desired system behavior, appropriate for formal reasoning, and ending with a low-level version which is translatable into real distributed code.

D.A. Peled and M.Y. Vardi (Eds.): FORTE 2002, LNCS 2529, pp. 364–368, 2002.
© Springer-Verlag Berlin Heidelberg 2002

The advantage of IOA is that it enables one to establish a formal connection, within a single language, between specifications written at different levels of abstraction. The mathematical tool used for this purpose is the simulation relation [10]. Since simulation proofs tend to be very stylized, they are amenable to the application of computer-assisted proof tools. Generally, it is harder to write and check formal proofs with mechanized proof assistants than to write unchecked intuitive arguments. However, complicated distributed algorithms, such as data management algorithms with strong coherence guarantees, or fault-tolerant distributed algorithms that are used in practice, can benefit greatly from the careful reasoning style of formal proofs. Our goal is to combine the flexibility of informal arguments with the credibility of formally checked proofs.

We describe `ioa2lsl`, a tool that translates IOA specifications into the Larch Shared Language (LSL), a logical language for describing first-order theories. LSL interfaces with the Larch Prover (LP), an interactive theorem proving assistant for multisorted first-order logic. We have used this tool to verify the correctness of three distributed data management algorithms, Dijkstra's mutual exclusion algorithm and a distributed spanning tree algorithm [7].

A significant feature of our translation procedure is its treatment of nondeterminism. Even though theorem provers are better equipped for handling nondeterminism than other tools, such as model checkers and code generators, the use of nondeterminism still incurs a number of technical difficulties. Generally, these difficulties stem from the representation of nondeterministic choices by existential quantifiers, which require explicit instantiation in a proof session. Our tool uses a simple extension to the mathematical I/O automaton model that captures the essence of nondeterminism and reduces the need for explicit instantiation in proofs.

2 Features of the Translation

Our tool is aimed at producing a translation that appears natural to the user of the theorem proving tool. If the translation is cumbersome and unreadable, interaction with the tool can become extremely difficult. A good translation must preserve the illusion that what appears obvious to the user is also obvious to the mechanized proof assistant. This becomes an especially important issue in the translation of transition effects because the semantics of imperative code may be quite complicated. For example, the proof assistant ought to establish that a state variable was not modified by a transition automatically, without user intervention.

In an IOA program, the conditions under which two states are related by a labeled transition are specified by transition definitions. Transition definitions consist of a precondition predicate specifying the enabling condition for the transition and an effect specifying the relation between the initial and final states, usually written as an imperative-style program. The specification may depend on both explicit transition parameters and nondeterministically selected hidden parameters.

In many formalisms, the precondition and effect of a transition are lumped into a single predicate describing the condition under which a labeled transition is allowed between two states. Our translation produces, for each transition definition, a separate `enabled` clause containing the enabling condition and an `effect` clause specifying the change in the state variable. This allows us to represent `effect` as a function, rather than a predicate, that maps an initial state and a transition label into a final state. This functional representation is useful in the theorem proving session, where `effect` clauses turn into rewrite rules expressing post-state variables in terms of pre-state variables. The utility of these rewrite rules in inductive proofs, including proofs of invariants and simulation relations, is our principal motivation for this design decision.

A fundamental feature of the IOA language is its support for nondeterministic choices. There are two sources of nondeterminism in IOA programs: (1) at a given state, more than one action may be enabled, and (2) a transition definition may contain explicit nondeterministic choices, indicated by **choose** statements. Nondeterminism of the first kind causes no difficulty for theorem proving (although code generators must provide some way to schedule actions when more than one is enabled). Nondeterminism of the second kind can complicate theorem proving, particularly because standard techniques for translating **choose** statements (cf. [5], page 312) introduce explicit universal or existential quantifiers.

We avoid the introduction of additional quantifiers by transforming explicit nondeterministic choices into transition parameters. For example, consider the following transition of an automaton with a single state variable `x` of type `Int`:

```
input chooseroot
  eff x := choose t: Int where t * t = 1
```

This transition definition is nondeterministic because it specifies two possible post-states: $\mathtt{x}' = -1$ and $\mathtt{x}' = 1$. We can interpret this nondeterministic choice using a *transition parameter* `t` by rewriting the transition definition in the following form:

```
input chooseroot(t: Int)
  pre t * t = 1
  eff x := t
```

The two specifications are almost the same: the only difference is that we have replaced the action `chooseroot` with a family of actions `chooseroot(t)`, where `t` ranges over the sort `Int`.

We note that our translation scheme avoids the use of quantifiers to specify either bounded or unbounded nondeterministic choices, as in earlier approaches [5,2,4]. This translation benefits theorem proving, since establishing statements involving existential quantifiers often requires explicit user instantiation.

3 Applications of the Tool

We have used our translation tool to verify three distributed data management algorithms of successively increasing complexity: a simple caching algorithm, a

majority voting algorithm, and a replicated storage algorithm based on Lamport's logical time transformation. In each case, we verified that the implementations of read/write operations were atomic. In the spirit of successive refinement, we wrote an abstract specification for an atomic variable and showed that each of the algorithms implements the atomic variable. The proofs use forward simulation relations.

In the majority voting algorithm (cf. [8], page 573), each concurrent process employs a local copy of the data, and the algorithm preserves the invariant that the most recent value is always present in a majority of the processes. To verify this invariant, it is necessary to argue that any majority of processes intersects the set of processes containing the most recent value. Establishing set theoretic properties such as this requires a number of lemmas, which constitute the bulk of our simulation proof script. The sections of the proof owing to our choice of representation for automata and their properties were straightforward, with one exception: the prover required a great deal of assistance to establish intuitively clear properties of a **for** loop used to store values at a majority of processes during the write operation.

Our most challenging example was the replicated storage algorithm. The atomicity assumptions in this algorithm are much less strict, and closer to actual practice, than in the other two. In particular, no action involving more than one process is required to be atomic. The algorithm ensures consistency with the help of Lamport's logical time algorithm [6].

Originally, the correctness of this algorithm was established by demonstrating indistinguishability with respect to certain orderings of events [6], an approach quite different from the techniques supported by our tool. We developed a new correctness proof based on successive refinement and forward simulation relations. Our proof introduces an auxiliary data management model that preserves the data replication features of the replicated storage algorithm, but pretends that the interactions are synchronous. We first establish that this auxiliary model implements an atomic variable. The mathematics of the proof are quite intricate, and this is reflected in the proof script ([1], Appendix B), which is 800 lines in length. As in the proof of the majority voting algorithm, most of the proof script is concerned with establishing lemmas about data structures; relatively little work is required because of our translation process. Another proof, dealing with synchronization properties, establishes that the replicated storage algorithm implements the auxiliary algorithm. This proof is complicated mathematically, and we have not yet attempted to verify it mechanically.

References

1. Andrej Bogdanov. Formal verification of simulations between I/O automata. Master of engineering thesis, Massachusetts Institute of Technology, 2001. `http://theory.lcs.mit.edu/~adib/thesis`.
2. Marco Devillers. Translating I/O automata to PVS. Preliminary report, Computing Science Institute, University of Nijmengen, 1999.

3. Stephen J. Garland and Nancy A. Lynch. Using I/O automata for developing distributed systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 285–312. Cambridge University Press, 2000.
4. Stephen J. Garland, Nancy A. Lynch, and Mandana Vaziri. *IOA: a language for specifying, programming, and validating distributed systems.* MIT Laboratory for Computer Science, 1997 (revised January, 2001).
5. David Gries. *The Science of Programming.* Springer-Verlag, 1981.
6. Leslie A. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
7. Chris Luhrs. Distributed spanning tree algorithms coded in IOA: Challenge problems for software analysis and synthesis methods. Technical Note, 2001.
8. Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufman Pubishers Inc., 1996.
9. Nancy A. Lynch and Mark Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical Report MIT/LCS/TR-387, MIT Laboratory for Computer Science, 1987.
10. Nancy A. Lynch and Frits Vaandrager. Forward and backward simulations – part I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.