Computer-Assisted Simulation Proofs

Jørgen F. Søgaard-Andersen[1]*, Stephen J. Garland, John V. Guttag, Nancy A. Lynch, and Anna Pogosyants[2]**

[1] Technical University of Denmark, Building 344, DK-2800 Lyngby, Denmark
[2] MIT Laboratory for Computer Science, Cambridge, MA 02139, USA

Abstract. This paper presents a scalable approach to reasoning formally about distributed algorithms. It uses results about I/O automata to extract a set of proof obligations for showing that the behaviors of one algorithm are among those of another, and it uses the Larch tools for specification and deduction to discharge these obligations in a natural and easy-to-read fashion. The approach is demonstrated by proving the behavior equivalence of two high-level specifications for a communication protocol.

1 Introduction

When showing that two distributed algorithms or protocols are equivalent, or that one is an implementation of another, we are often faced with a choice between intuitive arguments and careful proofs. Intuitive arguments highlight essential ideas, but are often flawed or depend upon unstated assumptions. Careful proofs provide a higher level of confidence, but tend to be long and tedious. At best, they are time consuming to construct; at worst, they too are error prone.

One way to strike a better balance is to use the computer to help move from intuitive arguments to formal proofs. There are two distinct stages to doing this: first, constructing a formal description of the artifacts being reasoned about, and, second, constructing a proof that the desired relationship holds. In performing these steps, it is important to consider at least the following questions:
(1) How confident is one that the formal description of a specification or algorithm corresponds to the artifact (e.g., software or hardware) in which one is actually interested?
(2) Does the formal argument convey the intuition of a good informal argument?
(3) How much can be reused if one changes the specification, algorithm, or proof?

The answers to these questions vary greatly, depending upon the descriptive and proof techniques used. In this paper, we present an approach to reasoning formally about distributed algorithms that we feel addresses these questions in

* Research supported in part by the Danish Research Academy.
** Research supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contracts N00014-92-J-1795 and N00014-92-J-4033, by the National Science Foundation under grants 9115797-CCR and 8915206-CCR, and by the Office of Naval Research under contract N00014-91-J-1046.

a reasonable way. This approach brings together the work that some of us have done using I/O automata to reason about distributed algorithms [8] and the work that others of us have done to provide tools for formalizing specifications [5] and for automating deductions [3].

Our approach uses forward and backward simulation methods, described in [1, 9], to isolate sets of proof obligations that guarantee that the traces of one automaton are included in the traces of another. We formalize the automata using the Larch Shared Language (LSL) [5] and then use LP, the Larch Proof Assistant [3], to construct simulation proofs.

We use simulation proofs because we believe that this method captures formally the natural structure of many informal correctness proofs for both finite and infinite state systems. In particular, it catches the structure of proofs based on successive refinements. Proofs using simulations are generally based on key intuitions about the execution of algorithms. Simulation relations, like invariants, tend to capture central ideas; hence they provide important documentation for algorithms. Simulations also tend to be readily modifiable when implementations are modified or when related algorithms are considered.

Using LSL, we are able to describe automata in a way that corresponds closely to the way they are usually described. LSL's syntactic amenities and facilities for modularizing specifications are particularly useful. Using LP, we are able to construct proofs whose structure is identical to that of the usual careful hand proofs. We supply the same invariants, simulation relations, and lemmas that appear in hand proofs; LP saves us from supplying the tedious details. The process of inventing invariants, simulation relations, and lemmas can involve considerable intellectual effort, but we believe that this effort is worthwhile: it yields considerable insight into why algorithms work. Once an LP proof has been completed, the proof script is easily read by a person, and it contains enough information for the reader to reproduce the elided steps, given access to LP or another sufficiently powerful theorem prover [2, 4].

The remainder of this paper describes our approach in more detail and provides an illustrative example. Section 2 provides background about I/O automata and simulation proofs. Section 3 contains part of a careful hand proof that two example automata simulate each other. Section 4 shows how we formalize the definitions of these automata in LSL, and Section 5 presents the LP proof scripts. Finally, Section 6 draws some conclusions about this approach.

2 Automata

In this paper we consider simplified versions of I/O automata [8]. The major simplification is that we do not deal with fairness or other types of liveness; hence our automata lack a component that defines what it means for an execution to be fair. We also do not distinguish between input and output actions, which we group together into a single set of external actions. In fact, our notion of automata is the same as that of untimed automata in [9], except that we allow multiple internal actions.

Definition 1 (Automaton). An automaton A consists of four components:

- $states(A)$ is a (finite or infinite) set of states.
- $start(A)$ is a nonempty set of start states ($start(A) \subseteq states(A)$).
- $sig(A)$ is and action signature $(ext(A), int(A))$, where $ext(A)$ and $int(A)$ are disjoint sets of external and internal actions. The set $acts(A)$ of actions of A is $int(A) \cup ext(A)$.
- $steps(A)$ is a transition relation ($steps(A) \subseteq states(A) \times acts(A) \times states(A)$).

An *execution fragment* $\alpha = s_0, \pi_1, s_1, \pi_2, s_2, \ldots$ of an automaton A is a (finite or infinite) sequence of alternating states and actions starting with a state s_0, ending in a state s_n if the sequence is finite, and such that $(s_i, \pi_{i+1}, s_{i+1}) \in steps(A)$ for all i (less than n if α is finite).

The function *first* gives the first state of an execution fragment, i.e., $first(\alpha) = s_0$. For finite execution fragments, the function *last* gives the final state. An *execution* of A is an execution fragment that begins with a start state, i.e., an α for which $first(\alpha) \in start(A)$.

The *trace* (sometimes also known as the *behavior*) of an execution fragment α, written $trace(\alpha)$, is the sequence of external actions occurring in α. Likewise, the trace of a sequence ω of actions, written $trace(\omega)$, is the restriction of ω to $ext(A)$. A sequence β of actions is a *trace* of an automaton A if there is an execution α of A with $trace(\alpha) = \beta$. The set of traces of A is denoted by $traces(A)$; the set of finite traces is denoted by $finite\text{-}traces(A)$.

Correctness and Trace Inclusion. In this paper, we concentrate on techniques for showing that the traces of one automaton (the implementation) are included among the traces of another (the specification). By itself, trace inclusion is not sufficient to express a notion of correct implementation, because it does not rule out trivial implementations that do nothing.

The definition of I/O automata [8] rules out automata with trivial trace sets by partitioning the external actions into input actions and output actions and by requiring that some step with every input action be enabled in every state. The definition also imposes additional fairness requirements on executions. An I/O automaton A is said to be an *implementation* of another I/O automaton B if the set of *fair* traces of A is a subset of the set of fair traces of B, where fair traces are traces of fair executions, i.e., of executions that satisfy the extra fairness requirements.

This correctness for I/O automata is usually proved in two steps. First, a simulation proof technique is used to prove trace inclusion. Second, other proof techniques, e.g., based on a temporal logic, use the simulation result and fairness requirements to prove fair trace inclusion. Examples like that in [6] show that the simulation step can be complex. Hence the techniques described in this paper for proving trace inclusion can provide significant help in this first step of a correctness proof.

Techniques for Proving Trace Inclusion. Several simulation proof techniques can be used to show trace inclusion. We define two: *forward* and *backward simulations*. Other simulation proof techniques are defined in [9].

Definition 2 (Forward Simulation). Let A and B be automata with the same external actions. A *forward simulation* from A to B is a relation f over $states(A) \times states(B)$ such that:

1. If $s \in start(A)$, then there is a $u \in start(B)$ such that $(s, u) \in f$.
2. If $(s', \pi, s) \in steps(A)$, $u' \in states(B)$, and $(s', u') \in f$, then there is a finite execution fragment α of B such that $first(\alpha) = u'$, $(s, last(\alpha)) \in f$, and $trace(\pi) = trace(\alpha)$.

Definition 3 (Backward Simulation). Let A and B be automata with the same external actions. A *backward simulation* from A to B is a relation b over $states(A) \times states(B)$ such that:

1. If $s \in states(A)$, then there is a $u \in states(B)$ such that $(s, u) \in b$.
2. If $s \in start(A)$ and $(s, u) \in b$, then $u \in start(B)$.
3. If $(s', \pi, s) \in steps(A)$, $u \in states(B)$, and $(s, u) \in b$, then there is a finite execution fragment α of B such that $last(\alpha) = u$, $(s', first(\alpha)) \in b$, and $trace(\pi) = trace(\alpha)$.

To state a soundness result for these simulations, we need the following definition: A relation r over $S_1 \times S_2$ is *image-finite* if for all elements s_1 of S_1 there are only finitely many elements s_2 of S_2 such that $(s_1, s_2) \in r$.

Theorem 4 (Soundness of Simulations [9]). *Let A and B be automata with the same external actions.*

1. *If there is a forward simulation from A to B, then $traces(A) \subseteq traces(B)$.*
2. *If there is a backward simulation from A to B, then $finite\text{-}traces(A) \subseteq finite\text{-}traces(B)$. If there is an image-finite backward simulation from A to B, then $traces(A) \subseteq traces(B)$.* □

Even though both forward and backward simulation techniques are sound with respect to trace inclusion, they are not complete since there exist examples, where the traces of one automaton are included among those of another automaton, but where no forward or backward simulation can be found. Combinations of forward and backward simulations can be shown to be complete. We refer to [9] for details. We note that the forward and backward simulation techniques are more general than refinement mappings of [1]. The example in the next section shows how forward and backward simulations each apply to different situations.

3 An Example: The Lossy Message Queue

This section describes two specific automata and presents part of a careful manual proof of the existence of a forward simulation from one automaton to the other, and the existence an image-finite backward simulation in the opposite direction. The soundness result of Theorem 4 allows us to conclude that the two automata have the same traces.

These automata are slight simplifications of the top two levels of the correctness proofs in [6]. The first protocol, S, is the specification of the *at-most-once* *message delivery problem*. It describes a "lossy message queue"—a queue for which special *crash* events can cause the loss of any of the messages in the queue. All proofs in [6] can be done directly via simulations of S; however, doing this requires very complicated combinations of forward and backward simulations. The reason that backward simulations are required is that, while a crash can "cause" loss of messages, the decision as to which messages actually get lost might not be made until long after the time of the crash (depending on certain race conditions in the algorithms).

The method used in [6] to reduce the complexity of the simulations is to split up the mapping into two parts. A new version D of the specification is defined; this is similar to S, except that it *delays* the decision about which messages are lost because of a crash. Thus, in D, a *crash* event merely marks the messages in the queue, and an internal *lose* event is permitted to remove any marked messages from the queue at any time. A backward simulation is shown from D to S, and then simpler forward simulations suffice between the actual algorithms and D.

Both S and D have a queue as their only state variable. An external action $insert(m)$ inserts a message m at the end of the queue, and an external action $remove(m)$ removes the message at the head of the queue, provided this message is m. Both automata also have an external *crash* action. In S this action can remove any number of messages from the queue; in D, it merely marks all messages in the queue. An internal *lose* action in D is allowed, at any time, to remove any number of marked messages from the queue.

Automaton S: A Simple Specification for the Lossy Queue. The only state variable in S is *queue*, which ranges over finite sequences[3] of elements from some arbitrary set *Msg*. Initially, *queue* is empty. We refer to the *queue* in state s by $s.queue$.

We specify the allowable steps of S by giving preconditions and effects for the three different kinds of actions. A triple (s', π, s) is in $steps(S)$ provided s' satisfies the precondition for π and s can be obtained from s' by the changes given in the corresponding effect clause. We omit the precondition if it is "true."

```
$insert(m)$
    Eff:  $queue := queue \,\hat{}\, m$

$crash$
    Eff:  Delete any number of
          elements from $queue$

$remove(m)$
    Pre:  $\neg empty(queue) \wedge$
          $\underline{\mathbf{hd}}\, queue = m$
    Eff:  $queue := \underline{\mathbf{tl}}\, queue$
```

Automaton D: A Delayed Implementation of the Lossy Queue. As in S, the only state variable is *queue*. However, in D, *queue* ranges over finite

[3] We use the following basic operators on sequences: $s \,\hat{}\, e$ and $e \,\hat{}\, s$ denote appending the element e to the end and beginning of the sequence s. For any nonempty sequence s, we let $\underline{\mathbf{hd}}\, s$ and $\underline{\mathbf{last}}\, s$ denote the first and last element of s, and we let $\underline{\mathbf{tl}}\, s$ and $\underline{\mathbf{init}}\, s$ denote the sequences of all but the first and all but the last element of s. Finally, $empty(s)$ is true iff s is the empty sequence.

sequences of pairs of *Msg* and *Mark*, where $Mark = Bool$. Initially, *queue* is empty. To get the components of a pair, we use the normal record notation. Thus, if $e = (m, b)$, then $e.msg = m$ and $e.mark = b$. We say that e is marked if b is true and unmarked otherwise.

```
insert(m)                                   crash
    Eff: $queue := queue \mathbin{\hat{}} (m, false)$        Eff: $queue := mark(queue)$

remove(m)                                   lose
    Pre: $\neg empty(queue) \wedge$                 Eff: Delete any number of marked
         $(\underline{\mathbf{hd}}\, queue).msg = m$             elements from queue
    Eff: $queue := \underline{\mathbf{tl}}\, queue$
```

In the specification of the *crash* action, the function *mark* is intended to change the mark of all the elements of its argument to *true*. In the following proofs, we use *subseq* and *subseqMarked* to denote the relations between queues before and after the *crash* and *lose* actions of S and D, respectively.

Simulation Relations between D and S.

Definition 5 (Forward Simulation from S to D). Let s be a state of S and u be a state of D. Define $(s, u) \in f$ iff $messages(u.queue) = s.queue$.

In this definition the function *messages* is intended to take a queue of the automaton D and throw away all the marks, i.e., to return the sequence of message components of the queue.

Definition 6 (Backward Simulation from D to S). Let s be a state of D and u be a state of S. Define $(s, u) \in b$ iff $u.queue$ consists of the message components of a subsequence of $s.queue$ that contains at least all unmarked messages, i.e., iff there is a q such that $u.queue = messages(q) \wedge subseqMarked(q, s.queue)$.

Hand Proofs That Simulations Are Correct. Here we present part of our hand proofs of the forward and backward simulations between S and D. We will not—and cannot—be strictly formal because we have not presented formal definitions for the functions we used to describe the automata and the simulations. Instead, we will rely on our intuitions concerning these functions. For example, we will use facts such as $messages(q \mathbin{\hat{}} (m, b)) = messages(q) \mathbin{\hat{}} m$. Despite this, we have carefully written down all interesting steps in the proofs, and we believe that the level of detail in these proofs is typical of careful hand-written simulation proofs.

Theorem 7. *f is a forward simulation from S to D.*

Proof. We check the two conditions from Definition 2.

1. Initially both queues are empty, and empty queues correspond.

2. Let (s', π, s) be any step of S. Let u' be an arbitrary state of D such that $(s', u') \in f$. We must show that there is a finite execution fragment α of D starting in u' such that $(s, last(\alpha)) \in f$ and $trace(\alpha) = trace(\pi)$. We divide

the proof into cases, one for each action. Here we show the proof for the *crash* action only. The proofs for the *insert* and *remove* actions are similar in style and length.

Define $\alpha = (u', crash, u'', lose, u)$, where u'' is defined to be the state with $u''.queue = mark(u'.queue)$, and u is defined to be the state with $u.queue = addMarks(s.queue)$, where *addMarks* adds a mark of *true* to each message in a sequence of unmarked messages. Obviously, $trace(\alpha) = trace(\pi) = crash$. We must show that α is indeed an execution fragment of D and that $(s, u) \in f$.

It is easy to see that $(u', crash, u'')$ is a step of D. We show that $(u'', lose, u)$ is also a step of D. By the definition of *crash* in S, $subseq(s.queue, s'.queue)$. Because (s', u') is in f, $subseq(s.queue, messages(u'.queue))$. Because changing marks does not affect *messages*, $subseq(s.queue, messages(mark(u'.queue)))$; and $subseqMarked(addMarks(s.queue), mark(u'.queue))$ holds because everything in $mark(u'.queue)$ is marked. Hence, by the definitions of u and u'', we have that $subseqMarked(u.queue, u''.queue)$ so that $(u'', lose, u)$ is a step of D and α is an execution fragment of D.

Also, $messages(u.queue) = messages(addMarks(s.queue)) = s.queue$, so we have $(s, u) \in f$. □

Theorem 8. *b is an image-finite backward simulation from D to S.*

Proof. We first observe that b is image-finite. For any state s of D there are only a finite number of queues q such that $subseqMarked(q, s.queue)$. Hence there are only finitely many states u of S such that $u.queue = messages(q)$. This suffices.

To show that b is a backward simulation, we check the three conditions from Definition 3. We do not include that proof here, but note that it is similar in style to, and about twice as long as, the forward simulation proof. □

4 Formalizing Automata in the Larch Shared Language

In order to formalize our simulation proofs, we must first formalize the definitions and abstractions used in the informal proofs. Here we use the Larch Shared Language (LSL), which provides suitable notational and parametrization facilities, and which is supported by a tool that produces input for LP.

The basic unit of specification in LSL is a *trait*. We begin by defining a generic trait `Automaton` that introduces notations and definitions common to all automata, e.g., an encoding of an automaton's start states as a unary predicate and a definition of what it means to be an execution fragment. Later we use LSL's facilities for combining traits to provide two specializations `AutomatonD` and `AutomatonS` of this trait.

A trait introduces two kinds of symbols, *sorts* and *operators*, and defines their properties. Sort symbols denote disjoint nonempty sets of values. An operator symbol denotes a total mapping from tuples of values (of the same or different sorts) to a value.

The trait `ExternalActions` in Figure 1 defines a sort consisting of the external actions for the lossy queue. This trait is similar to specifications in many "algebraic" specification languages. The part following the keyword `introduces`

```
ExternalActions: trait
  introduces
    insert : Msg → ExternalActions
    remove : Msg → ExternalActions
    crash  :     → ExternalActions
  asserts
    ExternalActions generated by
     insert, remove, crash
```

```
CommonActions (A): trait
  includes ExternalActions
  introduces
    insert     : Msg → A$Actions
    remove     : Msg → A$Actions
    crash      :     → A$Actions
    external   : A$Actions →
                       ExternalActions
    isExternal : A$Actions → Bool
  asserts ∀ m: Msg
    external(insert(m)) == insert(m);
    external(remove(m)) == remove(m);
    external(crash) == crash;
    isExternal(insert(m));
    isExternal(remove(m));
    isExternal(crash)
```

Fig. 1. LSL traits defining external actions for lossy queue

declares a set of operators and provides each with its signature (the sorts of its domain and range). Sorts are declared implicitly by their appearance in signatures, and their names need have no relation to the name of the trait. The part of the trait following the keyword `asserts` constrains the operators, in this case by an assertion that all values of sort `ExternalActions` can be obtained as values of one of the three listed functions. In general, a `generated by` assertion (such as `Nat generated by 0, succ`) corresponds to a principle of induction.

Two technical problems arise when we try to extend the `ExternalActions` trait to a general LSL definition for an automaton. Because LSL requires sorts to represent disjoint nonempty sets, we cannot represent (possibly empty) sets of internal actions as sorts, and we cannot have the sorts `D$Actions` and `S$Actions` for two automata D and S overlap in a common set of external actions. Instead, we encode the sets of all actions of D and S as sorts `D$Actions` and `S$Actions`, we encode a copy of their external actions as another sort `ExternalActions`, and we define predicates to recognize their external actions and to map them onto this third sort.

The trait `CommonActions` in Figure 1 shows how we do this for the lossy queue by defining a single trait. This trait extends the trait `ExternalActions` (which it `includes`) by introducing additional sorts and operators, and also by constraining the values of these operators. The new constraints are expressed by equations[4] and by Boolean-valued predicates. The parameter `A` in the trait definition can be specialized whenever the trait is used, for example, by including `CommonActions(S)` to define the common actions for the automaton S and by including `CommonActions(D)` to define the common actions for the automaton D.

Note that each of the operators `insert`, `remove`, and `crash` has two *over-*

[4] An equation consists of two terms of the same sort, separated by `=` or `==`. The operators `=` and `==` are semantically equivalent, but have a different precedence: `==` is the main connective in an equation.

loadings in the trait `CommonActions`, one with range sort `ExternalActions` and one with range sort `A$Actions`. The ability to use overloaded operators in LSL, together with LSL's ability to disambiguate them in most contexts, contributes substantially to the readability of specifications.

The trait `Automaton` in Figure 2 provides a generic LSL definition for an automaton A. The states of A are encoded as a sort `A$States`. When we encode a specific automaton in LSL, we will define an appropriate structure for the state space `A$States`, usually as a tuple of finitely many state components. The transition relation of A is encoded, quite naturally, as a ternary predicate `isStep`. Again, the actual definition of this predicate is given when a specific automaton is encoded.

```
Automaton (A): trait
  includes CommonActions(A)
  introduces
    start       : A$States                      → Bool
    isStep      : A$States, A$Actions, A$States  → Bool
    null        : A$States                      → A$StepSeq
    __≪__,__≫ : A$StepSeq, A$Actions, A$States → A$StepSeq
    execFrag    : A$StepSeq                     → Bool
    first, last : A$StepSeq                     → A$States
    empty       :                               → Trace
    __ ^ __     : Trace, ExternalActions        → Trace
    trace       : A$Actions                     → Trace
    trace       : A$StepSeq                     → Trace
  asserts ∀ s, s': A$States, a, a': A$Actions, ss: A$StepSeq
    execFrag(null(s));
    execFrag(null(s')≪a,s≫) == isStep(s', a, s);
    execFrag((ss≪a',s'≫)≪a,s≫) ==
      execFrag(ss≪a',s'≫) ∧ isStep(s', a, s);
    first(null(s)) == s;
    last(null(s)) == s;
    first(ss≪a,s≫) == first(ss);
    last(ss≪a,s≫) == s;
    trace(a) == if isExternal(a) then empty ^ external(a) else empty;
    trace(null(s)) == empty;
    trace(ss≪a,s≫) ==
      if isExternal(a) then trace(ss) ^ external(a) else trace(ss)
```

Fig. 2. LSL trait defining the notion of an automaton

The execution fragments of A are defined by its transition relation. The `Automaton` trait introduces a sort `A$StepSeq` (for Step Sequences of A) that contains finite sequences of alternating states and actions of A. The `null` function produces a step sequence consisting of a single state and no actions; the ternary operator `__≪__,__≫` extends a step sequence by appending an action and a state. Double underscores (`__`) in an operator declaration indicate that the operator will be used in *mixfix* terms. Infix, prefix, postfix, and mixfix operators (such as `__+__`,

-__, __!, {__}, and __[__]) are integral parts of many familiar notations, and their availability in LSL enables us to write readable specifications.

The `Automaton` trait also defines a unary predicate `execFrag` that identifies which elements of the sort `A$StepSeq` are legal execution fragments of A, as well as functions `first` and `last` that extract the first and the last states from execution fragments.

LSL Definitions for the Lossy Queue Automata. We now encode the two lossy queue automata in LSL by writing two specializations of the `Automaton` trait. The trait `AutomatonS` in Figure 3 defines the simple automaton S for the lossy queue. This automaton has no internal actions, and its state consists of a queue of messages.

```
AutomatonS: trait
  includes Sequence(Msg), Automaton(S)
  S$States tuple of queue: Msg$Seq
  asserts
    S$Actions generated by insert, remove, crash
    ∀ s, s': S$States, m: Msg
      start(s) == isEmpty(s.queue);
      isStep(s', insert(m), s) == s.queue = s'.queue ^ m;
      isStep(s', remove(m), s) ==
        ¬ isEmpty(s'.queue) ∧ hd(s'.queue) = m ∧
        s.queue = tl(s'.queue);
      isStep(s', crash, s) == subseq(s.queue, s'.queue)
```

Fig. 3. LSL trait defining automaton S

The trait defines the properties of queues of messages by including a library trait `Sequence` (not shown here), which defines the properties of operators (such as ^, `isEmpty`, `hd`, `tl`, and `subseq`) on finite sequences (of sort `E$Seq`) of elements of some sort `E`. By instantiating `E`, we can talk about messages of sort `Msg` and sequences of messages (of sort `Msg$Seq`) in the specification of `AutomatonS`. Elements of sort `S$State` are one-tuples whose only component is a queue of messages.

The trait also defines the `start` and `isStep` predicates. In the definition of the `insert` action, which we characterized earlier in less formal and more operational terms as `queue := queue ^ m`, we now make explicit the fact that the first occurrence of `queue` refers to the prestate `s'` and the second to the poststate `s`.

Figure 4 defines the delayed-action automaton D for the lossy queue. This automaton has a single internal action `lose`, and its state consists of a queue of marked messages. The trait `AutomatonD` defines the properties of queues of marked messages by including a trait `MarkedMessages` (also not shown here). This trait defines the sort `Mmsg` of marked messages, introduces the notation `[m, b]` to construct an element of this sort from a value `m` of sort `Msg` and a value `b` of sort `Bool`, defines the sort `Mmsg$Seq` of sequences of marked messages (by

```
AutomatonD: trait
  includes Automaton(D), MarkedMessages
  D$States tuple of queue: Mmsg$Seq
  introduces lose: → D$Actions
  asserts
    D$Actions generated by insert, remove, crash, lose
    ∀ s, s': D$States, m: Msg
      ¬ isExternal(lose);
      start(s) == isEmpty(s.queue);
      isStep(s', insert(m), s) == s.queue = s'.queue ^ [m, false];
      isStep(s', remove(m), s) ==
        ¬ isEmpty(s'.queue) ∧ hd(s'.queue).msg = m ∧
        s.queue = tl(s'.queue);
      isStep(s', crash, s) == s.queue = mark(s'.queue);
      isStep(s', lose, s) == subseqMarked(s.queue, s'.queue)
```

Fig. 4. LSL trait defining automaton D

reusing the `Sequence` trait), and provides precise definitions for the operators `mark` and `subseqMarked` used in our informal proofs.

5 Automating Simulation Proofs Using LP

LP is a theorem prover for first-order logic. It differs from many other provers in that its design is based on the assumption that initial attempts to state conjectures correctly, and then prove them, usually fail. As a result, LP is designed to carry out routine (and possibly lengthy) steps in a proof automatically and to provide useful information about why proofs fail, if and when they do. LP is not designed to find difficult proofs automatically. Instead, it is designed to assist users who employ standard techniques such as proofs by cases, induction, and contradiction.

In this section we use LP to prove both Theorem 7, which shows that there is a forward simulation from S to D, and Theorem 8, which shows that there is a backward simulation from D to S. From Theorem 4 and the fact that the backward simulation is image-finite, it follows that D and S have the same traces. We do not use LP to prove Theorem 4 or the fact that the backward simulation is image-finite. Proofs of these theorems do not involve the kind of detail that demands machine assistance or that benefits from it. In particular, Theorem 4 only needs to be proved once (not once for each simulation).

Lemmas for Simulation Proofs. In order to prove the simulation theorems, we need two lemmas that relate queues of marked messages to queues of unmarked messages. These lemmas are supplied by the trait `Mark` in Figure 5. This trait defines the operators `messages` and `addMarks` by using the library trait `SequenceMap` (not shown here) to extend the operations `.msg` and `addMark` on messages to operations on sequences of messages. It lists the two lemmas following the keyword `implies`.

We illustrate LP by showing how it is used to prove the first lemma in the trait `Mark`. If the user types

```
Mark: trait
  includes MarkedMessages, Sequence(Msg)
  includes SequenceMap(Mmsg, Msg, .msg, messages)
  includes SequenceMap(Msg, Mmsg, addMark, addMarks)
  implies ∀ m: Msg, ms: Msg$Seq, mms, mms': Mmsg$Seq
   messages(addMarks(ms)) == ms;
   subseqMarked(mms, mark(mms')) == subseq(messages(mms), messages(mms'))
```

Fig. 5. LSL trait relating marked messages to unmarked messages

```
     prove messages(addMarks(ms)) == ms by induction
```

LP generates and automatically discharges the appropriate subgoals for a proof by induction based on the assertion that all sequences are generated by `empty` and `^:Msg$Seq,Msg→Msg$Seq`. First, it uses the axioms `addMarks(empty) == empty` and `messages(empty) == empty`, which come from the trait `SequenceMap`, to establish the basis case `messages(addMarks(empty)) == empty`. Then it introduces a new constant `msc`, assumes `messages(addMarks(msc)) == msc` as an induction hypothesis, and uses the facts in the subsidiary traits of `Mark` to prove `messages(addMarks(msc^m)) == msc^m`. This completes the proof by induction.

Forward Simulation from *S* to *D*. We use the LSL Checker to create an input file for LP from the files containing the LSL traits `AutomatonD`, `AutomatonS`, and `Mark`. This input file contains LP commands that declare appropriate sorts, operators, and variables, and that assert facts known to be in the theories of the traits (i.e., facts that are either asserted or implied in these traits).

The following LP commands declare variables for use in the simulation proof and define the forward simulation relation `f`. (The line containing `..` terminates a multiple-line command.)

```
    declare variables
      s, s' : S$States
      u, u' : D$States
      pi    : S$Actions
      alpha : D$StepSeq
      ..
    declare operator f: S$States, D$States → Bool
    assert f(s, u) == messages(u.queue) = s.queue
```

We prove first that for every start state of S there is a corresponding start state of D by typing the following LP commands:

```
    prove start(s) ⇒ ∃ u (start(u) ∧ f(s, u))
      resume by specializing u to [empty]
      qed
```

The first line introduces the conjecture we wish to prove, the second guides LP in instantiating the existential quantifier in the conjecture, and the third requests that LP confirm that the proof is indeed complete. The guidance in the second line is the formal counterpart of the statement "initially both queues are empty" in the hand proof.

We prove now that each action of S can be simulated by a sequence of actions of D. Three forms of user guidance are required for the proof. The first concerns general proof strategy. The `set proof-methods` command directs LP to attempt to prove conjectures by rewriting them to normal form after assuming the hypotheses of any conjecture that is an implication (and after replacing variables such as `s` and `u'` in the hypotheses by constants `sc` and `u'c`). The `prove` command itself directs LP to proceed by dividing the proof into cases based on the action `pi` of automaton S being simulated (expressed here as a proof "by induction" because the sort `S$Actions` is generated by `insert`, `remove`, and `crash`). The second form of guidance is to supply the simulating execution fragment `alpha` of D in each case in the proof; this guidance is the same as that contained in three sentences starting with "Define $\alpha =$ " in the hand proof. The third form is to suggest that LP perform additional forward inferences (by the `critical-pairs` operation, which derives equational consequences from rewrite rules) involving the hypotheses of the conjecture (named by `*Hyp`) and all other known facts (named by `*`).

```
set proof-methods ⇒, normalization
prove
  (isStep(s', pi, s) ∧ f(s', u')) ⇒
    ∃ alpha (execFrag(alpha) ∧ first(alpha) = u' ∧
             f(s, last(alpha)) ∧ trace(pi) = trace(alpha))
  by induction on pi
  ..
  % Simulate "insert" action
  resume by specializing alpha to
    null(u'c) ≪ insert(mc), [u'c.queue ^ [mc, false]] ≫
    ..
  % Simulate "remove" action
  resume by specializing alpha to
    null(u'c) ≪ remove(mc), [tl(u'c.queue)] ≫
    ..
  critical-pairs *Hyp with *
  % Simulate "crash" action
  resume by specializing alpha to
    (null(u'c) ≪ crash, [mark(u'c.queue)] ≫)
              ≪ lose, [addMarks(sc.queue)] ≫
    ..
  qed
```

We emphasize that what appears above is the entire interaction between the user and the prover. In particular, we note that the LP proof is considerably shorter than the hand proof.

Backward Simulation from *D* to *S*. The soundness proof for the backward simulation is more complicated than that for the forward simulation because the simulation relation `b` is defined using an existential quantifier. However, the general style of user interaction with the prover is the same. Once again, the LP proof follows the hand proof, but is considerably shorter. Because of lack of space, we omit the proof from this abstract.

6 Conclusions

In this paper, we described, largely by way of an example, a semi-automated approach to constructing formal proofs of the equivalence of two protocols. The proofs shown involve a forward simulation in one direction and a backward simulation in the other direction. We used LSL to represent the protocols and the simulations, and LP to show that the simulations work.

In some places, the LP proofs involved more work than the hand proofs; in other places, it involved less. More work was required to define the underlying data types axiomatically rather than informally in set theory. Axiomatizations for standard data types, such a finite sequences, can be found in data type libraries and need not be redone for each application. But axiomatizing customized data types in the stylized way required by the prover can take a significant amount of extra time. We believe that the burden of this extra work will decrease as the size of data type libraries increases. Additional work was also required in producing formal definitions for basic concepts related to automata. However, this is a one-time cost, and the LSL definitions in this paper can be reused in other simulation proofs.

Once the basic data types had been defined, the automated proof using LP required *considerably less* work on the part of the user than did the hand proof. LP was able to fill in many of the details that had previously been done by hand. The little guidance LP required took the following forms:

(1) a way of instantiating each existential quantifier, and

(2) a small amount of guidance during the proof procedure, to suggest which facts might be relevant to apply.

The first type of guidance contains key insights about the proof, and we think that it is both reasonable and desirable for the user to supply these. The second type of guidance generally takes a very stylized form (e.g., "use the hypotheses") that is easy to learn.

Although the example presented here is fairly small, it is typical of the kinds of proofs that are usually done in the distributed algorithms and verification community. Our proofs involve many of the complexities of "practical" proofs, including multivalued relations, both forward and backward simulations, and extensive nondeterminism. Because of these complexities, we believe that our methods will scale to larger examples. Even though larger examples typically involve a larger number of state components and actions, so that a larger number of cases must be considered, each case appears to be no more complicated than the cases of the proofs in this paper, and the number of cases appears to remain manageable.

Larger examples also tend to utilize *invariants*, i.e., state predicates that are true for all reachable states, to restrict the states that need to be considered in a simulation proof. Proofs of such invariants involve, like proofs of simulations, checking of cases based on the actions of the automaton. Such proofs can easily be incorporated into our approach and add no further complexity. We are currently working on incorporating also proofs of *timing-based* systems into our approach. This involves reasoning about reals but seems, at this point, to be feasible with minor extensions to the work presented in this paper.

Other works in the area of automating simulation proofs exist. In [7] the equivalent of forward simulations for state based automata are considered, however, using a higher-order logic approach (HOL). In [10] forward simulations are considered for I/O automata using the Isabelle theorem prover.

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 2(82):253–284, 1992.
2. R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.
3. S. J. Garland and J. V. Guttag. A guide to LP, the Larch Prover. Technical Report 82, DEC Systems Research Center, December 1991.
4. M. J. C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer, 1988.
5. J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.
6. B. Lampson, N. Lynch, and J. F. Søgaard-Andersen. Reliable at-most-once message delivery protocols. Tech. report under preparation, Laboratory for Computer Science, Massachusetts Institute Technology, 1993.
7. P. Loewenstein and D. L. Dill. Verification of a multiprocessor cache protocol using simulation relations and higher-order logic. In E. M. Clarke and R. P. Kurshan, editors, *Computer-Aided Verification '90*, number 531 in LNCS, pages 302–311. Springer-Verlag, 1990.
8. N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, September 1989.
9. N. Lynch and F. Vaandrager. Forward and backward simulations for timing-based systems. In J. W. de Bakker, C. Huizing, and G. Rozenberg, editors, *Proceedings of REX Workshop "Real-Time: Theory in Practice"*, number 600 in LNCS, pages 397–446. Springer-Verlag, 1992.
10. T. Nipkow. Formal verification of data type refinement. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, number 430 in LNCS, pages 561–589. Springer-Verlag, 1990.