Using Simulated Execution in Verifying Distributed Algorithms

Toh Ne Win, Michael D. Ernst, Stephen J. Garland,
Dilsun Kırlı, and Nancy A. Lynch

MIT Laboratory for Computer Science
{tohn,mernst,garland,dilsun,lynch}@lcs.mit.edu

Abstract

This paper presents a methodology for proving properties of distributed systems in which simulated execution assists and enhances formal proofs. It is well known that techniques such as testing can increase confidence in an implementation, but cannot by themselves demonstrate correctness. In addition to detecting simple errors quickly and to providing intuition about behavior, execution-based techniques can also reveal unexpected properties, suggest necessary lemmas, and provide information to structure proofs. This paper also describes the use of these techniques in a machine-checked proof of correctness of the Paxos algorithm for distributed consensus.

1 Introduction

Traditionally, execution serves as a prelude to formal verification. Testing reveals departures from desired behavior that are corrected (either in the code or in the specification of its behavior) before attempting to prove code correct. Testing via simulated execution can do the same even in the absence of a complete implementation. This paper discusses additional ways execution or simulated execution can assist in formal verification, and describes their use in producing a machine-checked proof of a distributed algorithm.

First, execution can serve in a more powerful way as a prelude to formal verification. Tools for dynamic program analysis can extract descriptions of program behavior from executions, and programmers can match the extracted descriptions against their expectations. Unlike the traditional use of execution to test behavior, this use can reveal unexpected behaviors, not just departures from anticipated behaviors.

Second, execution can help produce the lemmas required for successful proofs of correctness. Unlike human proofs, which are peppered with phrases like "it is obvious that," machine-checked proofs often require many explicit lemmas. To avoid the tedium of enumerating these lemmas by hand, verifiers can discover them by using execution and dynamic program analysis.

Third, information that directs simulated execution to examine interesting aspects of a program's behavior can also be used to direct a proof of correctness.

For example, programmers may ensure that executions cover the entire range of expected behaviors by formulating case splits that distinguish between normal and unusual behaviors; these same case splits can also provide helpful ways of organizing a proof.

We illustrate these uses of execution in constructing a formal proof of correctness for Paxos, a distributed algorithm for consensus [Lam98,PLL00]. This paper is concerned primarily with a general methodology for verifying distributed algorithms — and with the role execution and automated tools play in that methodology — and not with the details of the Paxos algorithm itself. Our methodology is based on the input/output (I/O) automaton framework [LT89] for modeling and verifying distributed algorithms, in which each component of a system is represented as an automaton whose external behavior is defined by a simple mathematical object called a trace.

This paper is organized as follows. Section 2 introduces the I/O automaton model, discusses the IOA language and toolkit, which support use of this model, and contrasts the toolkit with related tools that use run-time techniques to aid formal verification. The remainder of the paper presents our execution-based methodology in more detail, using a proof of the Paxos algorithm as a running example. Section 3 formulates specifications and implementations as I/O automata, Section 4 describes how these automata are executed, and Section 5 shows how dynamically detected invariants reveal properties of an automaton. Section 6 describes how two automata, one a specification and one an implementation, can be executed in lock-step, and Section 7 shows how this paired execution can be used to construct a machine-verified proof.

2 Preliminaries

Our methodology uses the I/O automaton model, the IOA language, and three tools in the IOA toolkit [GL98]: the IOA interpreter, the LP theorem-prover, and the Daikon dynamic invariant detector.

2.1 I/O automata and the IOA language

An I/O automaton is a simple state machine in which transitions between states are associated with named *actions*, which are classified as either *input, output*, or *internal*. The inputs and outputs are external actions used for communication with the automaton's environment; internal actions are visible only to the automaton itself. An automaton controls which output and internal actions it performs, but input actions are not under its control. An I/O automaton consists of its *signature*, which lists its actions; a set of *states*, some of which are distinguished as start states; a *state-transition relation*, which contains triples of the form (state, action, state); and an optional set of *tasks* (not considered in this paper).

Action π is *enabled* in state s if there is a state s' such that (s, π, s') is a transition of the automaton. Input actions are enabled in every state. The operation of an I/O automaton is described by its *executions* $s_0, \pi_1, s_1, \ldots$, which are alternating sequences of states and actions, and by its *traces*, which are the externally visible behavior occurring in executions. One automaton *implements* another if all its traces are also traces of the other.

Definition 1 (Forward simulation). *A* forward simulation *from automaton A to automaton B is a relation f on $states(A) \times states(B)$ with the following two properties. (1) For every start state a of A, there is a start state b of B such that $f(a, b)$. (2) If a is a reachable state of A, b is a reachable state of B such that $f(a, b)$, and $a \xrightarrow{\pi} a'$, then there is a state b' of B such that $f(a', b')$ and an execution fragment β of B such that $b \xrightarrow{\beta} b'$ and $trace(\pi) = trace(\beta)$.*

Theorem 1. *If there is a forward simulation relation from A to B, then every trace of A is a trace of B [Lyn96].*

The IOA language provides notations for describing I/O automata and for stating their properties; it uses Larch Shared Language [GHG+93] specifications to axiomatize the semantics of I/O automata and the data types used to describe algorithms. In IOA, transition relations are defined in terms of preconditions and effects. These can be written either in an imperative style (as a sequence of assignment, conditional, and loop statements), or in declarative style (as a predicate relating state variables in the pre- and post-states, transition parameters, and other nondeterministically chosen parameters). It is also possible to use a combination of these two styles. Nondeterminism appears in IOA in two ways: *explicitly*, in the form of **choose** constructs in state variable initializations and the effects of the transition definitions, and *implicitly*, in the form of action scheduling uncertainty.

Nondeterminism allows systems to be described in their most general forms and to be verified considering all possible behaviors without being tied to a particular implementation of a system design.

The sample programs in this paper do not exploit the full generality of the language. They all define primitive (i.e., not composite) automata in an imperative style with no explicit nondeterminism.

2.2 Tools used in the IOA toolkit

The IOA interpreter The IOA interpreter [KCD+02a,KCD+02b] assists users in formulating and checking properties of automata. The interpreter can simulate execution either of a single automaton in isolation (checking stated assertions and displaying or logging the automaton's execution) or of two automata running in lockstep. In the latter case, a user presents the interpreter with two automata, a candidate simulation relation, and a mapping, called a step correspondence, from the actions of the lower-level automaton to sequences of actions of the higher-level one. The interpreter simulates execution of the low-level automaton, generates a simulated execution of the high-level automaton induced by the step correspondence, checks that the two executions have the same trace, and checks that the candidate simulation relation holds throughout the executions. The IOA interpreter is also known as the "IOA Simulator," but is called the interpreter in this paper to avoid confusion with the notions of forward and backward simulation.

The Larch Prover The Larch Prover [GG91] (LP) is an interactive theorem proving system for multisorted first-order logic. It admits specifications of theories in the Larch Shared Language (LSL). The IOA toolkit includes a tool called `ioa2lsl` [Bog01], which translates IOA definitions of automata into LSL theories that describe the operation of the automaton. It also generates proof obligations for the invariants and simulation relations of the automaton.

The Daikon invariant detector The Daikon invariant detector [ECGN01] proposes program properties that are likely to be true. It operates dynamically, by examining values computed during execution, postulating and checking properties, and reporting those that pass a battery of statistical and other tests. The technique is unsound, because there is no guarantee that the test suite fully characterizes the execution environment. However, the reported properties are often true and generally helpful in explicating the system under test and/or its test suite. We achieve soundness by using LP to check proofs.

2.3 Related work

Other toolkits, such as AsmL [GSV01], Mocha [AHM+98], SMV [McM], and TLC [LY01], support execution or verification of concurrent and distributed systems. The execution is used mainly for debugging and understanding the behavior of a system. The IOA toolkit uses execution not only for these purposes, but also for automatically discovering program properties that can be used as lemmas in formal proofs. Moreover, the facility for executing of pairs of automata together, matching actions of one against those of the other, helps users in organizing formal proofs of correctness based on simulation relations.

Mocha, SMV and TLC use model checking as the verification method. Model checking is attractive because it requires relatively less expertise than theorem-proving and it provides counter-examples to falsified properties. However, model checkers provide no intuition about true properties and can analyze only a finite state space; theorem-provers apply to finite and infinite systems alike.

The "invisible invariants" method [PRZ01] facilitates automated verification of parameterized, finite-state systems. This method uses model-checking techniques for calculating candidate invariants, for checking their inductiveness, and for proving the verification conditions generated by the standard invariance rule of deductive verification. A key characteristic of this method is that invariants can be proved automatically and they need not be shown to a human. By contrast, we regard invariants as a means to inform users about interesting program properties they might have overlooked. Invariants detected by Daikon are intended to be simple and easily readable properties. Additionally, our methodology is not limited to finite-state systems or inductively provable properties.

```
type Node = tuple of location: Int
type Value = tuple of value: Int

automaton Cons
signature
  input fail(i: Node), init(i: Node, v: Value)
  output decide(i: Node, v: Value)
  internal chooseVal(v: Value)
states
  initiated: Set[Node] := {},     proposed: Set[Value] := {},
  chosen: Set[Value]   := {},     decided: Set[Node]   := {},
  failed: Set[Node]    := {}
transitions
  input init(i, v)
    eff if ¬(i ∈ failed) ∧ ¬(i ∈ initiated) then
            initiated := initiated ∪ {i};
            proposed := proposed ∪ {v}
        fi
  internal chooseVal(v)
    pre v ∈ proposed ∧ chosen = {}
    eff chosen := {v};
  output decide(i, v)
    pre i ∈ initiated ∧ ¬(i ∈ decided) ∧
          ¬(i ∈ failed) ∧ v ∈ chosen
    eff decided := decided ∪ {i}
  input fail(i)
    eff failed := failed ∪ {i}
```

Fig. 1. Specification of consensus in IOA

3 Specifying automata in IOA

The first step in verifying that an implementation is correct with respect to a specification is to define the specification and implementation automata in IOA. The I/O automaton version of Paxos defines a hierarchy of four automata for achieving consensus. The highest-level automaton, `Cons`, provides a specification for consensus. The lowest-level automaton, `Paxos`, provides a distributed implementation. An intermediate-level automaton, `Global1`, although non-distributed, captures how Paxos uses ballots and quorums to achieve consensus. The correctness proof involves showing the existence of a series of forward simulations, between each pair of successive levels in the hierarchy. Our case study examines the forward simulation between `Cons` and `Global1`.

3.1 Specification automaton

Paxos implements distributed consensus in an asynchronous system in which individual processes can fail. Suppose that I is a finite set of nodes representing the processes in the system and V is the set of possible consensus values. Processes in I may propose values in V. The consensus service is allowed to return decisions to processes that have proposed values. It must satisfy two conditions: all nodes must receive the same value ("agreement") and that value must have been proposed by some process ("validity").

The signature of the specification automaton `Cons` (Figure 1) contains an input action `init(i,v)`, representing the proposal of value `v` by process `i`, an internal action `chooseVal(v)`, representing the choice of a consensus value `v`, an output action `decide(i,v)`, representing the report of the consensus value to process `i`, and an input action `fail(i)`, representing the failure of process `i`. The automaton provides the required agreement and validity guarantees: only a single consensus value can be chosen, and that value must have been previously proposed.

3.2 Implementation automaton

The automaton `Global1` (Figure 2) specifies an algorithm that implements consensus in a non-distributed setting. This automaton uses a totally ordered set of ballots for values, one of which may eventually be chosen as the consensus value if sufficient approval is collected from the processes in the system.

In addition to the external actions of the automaton `Cons`, the signature of `Global1` includes internal actions for making ballots, assigning them values, and voting for or abstaining from ballots. The automaton `Global1` determines the fate of a ballot by considering the actions of quorums, which are finite subsets of I, on that ballot. `Global1` allows a ballot to succeed only if every node in a quorum has voted for it.

4 Simulating execution of an automaton with the IOA toolkit

The second step in verifying the correctness of an implementation using the IOA toolkit is to test its behavior by simulating its execution. The IOA interpreter simulates execution of an I/O automaton on a single machine, allowing the user to help select the executions and to propose invariants for the interpreter to check.

The interpreter requires that IOA programs be transformed into a form suitable for execution. For example, quorums in Paxos have to be initialized operationally, whereas they were specified declaratively in the original I/O automaton model. Aside from such bookkeeping issues, the crucial problem in this transformation is resolving nondeterminism. The IOA interpreter solves this problem by requiring the user to supply a program, called an *NDR program*, to each source of nondeterminism in an automaton [KCD+02a,KCD+02b].

In our case study, we wrote several NDR programs to execute `Global1` with different interleavings of actions, causing some nodes to fail and some to abstain from a ballot. For example, the NDR program statement

```
fire output decide([4], [1]);
```

causes the IOA interpreter to execute the `decide` action with the given arguments. We did not use structured test generation methods (e.g., code coverage) to produce the NDR programs; instead, we simply selected executions that exhibited what we felt was the normal behavior of the automaton (and that exercised every action). In our experience, such an intuitive scheduling is adequate for the purpose of dynamic invariant detection. However, as noted in Section 5.2, a

```
type Ballot = tuple of ordering: Int

automaton Global1
signature
  input fail(i: Node), init(i: Node, v: Value)
  output decide(i: Node, v: Value)
  internal start(theNodes: Set[Node]), makeBallot(b: Ballot),
           abstain(i: Node, B: Set[Ballot]), assignVal(b: Ballot, v:Value),
           vote(i: Node, b: Ballot), internalDecide(b: Ballot)
states
  initiated: Set[Node] := {},     proposed: Set[Value]   := {},
  decided: Set[Node]   := {},     failed: Set[Node]      := {},
  ballots: Set[Ballot] := {},     succeeded: Set[Ballot] := {},
  val: Array[Ballot, Null[Value]]      := constant(nil),
  voted: Array[Node, Set[Ballot]]      := constant({}),
  abstained: Array[Node, Set[Ballot]] := constant({})
  quorums: Set[Node],
  dead: Set[Ballot] := {}

transitions
  internal start(theNodes)
    eff quorums := delete([1], theNodes);
        for i: Node in theNodes do voted[i] := {};
                                   abstained[i] := {} od;
  input init(i, v)
    eff % As in Cons (Figure 1)
  input fail (i)
    eff failed := failed ∪ {i}
  internal makeBallot(b)
    pre ¬ (b ∈ ballots);
    eff ballots := ballots ∪ {b};
  internal assignVal(b, v)
     pre b ∈ ballots ∧ val[b] = nil ∧ v ∈ proposed
         ∧ ∀ b′:Ballot (b′.ordering < b.ordering ⇒
                            val[b′] = embed(v) ∨ b′ ∈ dead)
     eff val[b] := embed(v)
  internal vote(i, b)
    pre i ∈ initiated ∧ ¬(i ∈ failed) ∧ b ∈ ballots ∧ ¬(b ∈ abstained[i])
    eff voted[i] := voted[i] ∪ {b}
  internal abstain(i, B)
    pre i ∈ initiated ∧ ¬(i ∈ failed) ∧ voted[i] ∩ B = {}
    eff abstained[i] := abstained[i] ∪ B;
        for aBallot:Ballot in B do
            if ∀ aNode:Node (aNode ∈ quorums ⇒ aBallot ∈  abstained[aNode])
               then dead := insert (aBallot, dead);
            fi;
        od;
  internal internalDecide(b)
    pre b ∈ ballots ∧ ∀ j:Node (j ∈ quorums ⇒ b ∈ voted[j])
    eff succeeded := succeeded ∪ {b}
  output decide(i, v)
    pre i ∈ initiated ∧ ¬(i ∈ decided) ∧  ¬(i ∈ failed)
        ∧ ∃ b:Ballot (b ∈ succeeded ∧ embed(v) = val[b])
    eff decided := decided ∪ {i}
```

Fig. 2. A ballot-based implementation of consensus in IOA

preliminary test run reported an unexpected invariant, which indicated a (subsequently corrected) deficiency in the test data. In another case study, involving the Peterson mutual exclusion algorithm, use of the IOA simulator uncovered a bug in the IOA transcription of the implementation.

5 Dynamically detecting likely invariants

A proof of a simulation relation often depends on invariants and on auxiliary lemmas; machine verification requires that such bookkeeping details be made explicit. These parts of the proof are usually not the most interesting parts and also tend to be relatively simple; thus, automating them holds promise. We attempt to automatically generate invariants and lemmas by use of dynamic invariant detection.

The Daikon invariant detector is a run-time tool that proposes invariants based on program executions [ECGN01]. It examines the values that a program computes, generalizes over them, and reports the generalizations in the form of IOA invariants. Daikon's heuristics and analyses result in output in the form of a formal specification that often matches what a human would have written [NE02]. Three potential problems with the technique are that it is unsound, that it is incomplete, and that the reported properties are not guaranteed to be useful. We discuss Daikon's output and how to cope with the potential problems.

5.1 Daikon results for the case study

For Paxos, Daikon analysis produced 23 invariants, four of which were helpful in the simulation relation proof in Section 7. The four were:

```
Inv1: ∀ anIndex:Node (size(voted[anIndex] ∩ abstained[anIndex]) = 0)
Inv2: val.values.val(nonNull) ⊆ proposed
Inv3: size(succeeded ∩ dead) = 0
Inv5: succeeded ⊆ ballots
```

We have added the names `Inv`i for convenience in this presentation.

A full proof of the Paxos simulation relation required six invariants: five for the simulation relation proper, and one more for one of the invariants. The two missing invariants were:

```
Inv4: ∀ b:Ballot ∀ b′:Ballot
      (val[b] ≠ nil ∧ b′ < b ⇒ val[b′] = val[b] ∨ b′ ∈ dead(abstained))
Inv6: ∀ b_Inv6:Ballot
      (b_Inv6 ∈ succeeded ⇒ ∃ q_Inv6:Set[Node] ∀ n_Inv6:Node
        (q_Inv6 ∈ wquorums ∧ (n_Inv6 ∈ q_Inv6 ⇒ b_Inv6 ∈ voted[n_Inv6])))
```

These two invariants are outside Daikon's grammar, so it neither checked nor reported them. (Daikon does not report invariants with existential quantifiers, nor does it report those with more than a given number of subterms.)

5.2 Discussion of dynamically detected invariants

We now discuss how to cope with potential problems in the invariant detector output.

First, dynamic invariant detection is unsound: reported properties are true over the test suite, but, as with all execution-based techniques, there is no guarantee that the test suite fully characterizes the execution environment of the program. This does not hinder us for two reasons. First, we use Daikon's output to help in proposing, understanding, and verifying program properties, but soundness is provided by the theorem prover. Second, most of the output in our case study was correct. Most false facts Daikon produced were easily-corrected artifacts of the test suite (execution scheduling). For example, in one set of executions, Daikon reported that the size of the `failed` variable was a constant. We corrected this by randomizing failures in our NDR program, thereby improving the quality of the test suite for its use in Section 6. In the general, however, simply covering every action seems to be adequate.

Second, dynamic invariant detection is incomplete: the proposed invariants may be insufficient for verification, because some true invariants are not reported. Daikon restricts the set of invariants it checks for two reasons: to conserve runtime and to reduce the number of false positives that it reports (the more properties it checks, the larger the number of false properties it will report). In our case study, we had to add `Inv4` and `Inv6` to the set proposed by Daikon. We did not find this a hindrance because our methodology does not aim for completely automatic verification. Rather, we aim to reduce human effort — particularly non-imaginative effort. Qualitatively, we believe the output did so, by providing four of the six required invariants. Some assistance was better than none, even though work remained.

It is notable that `Inv3`, while true and necessary for the proof, was not provable in isolation: establishing it required use of `Inv6`. In other words, Daikon was able to postulate a simple property with a complicated proof, prompting a user to find that proof. In addition to nicely decomposing the proof into parts, this demonstrates a strength of our technique: it is easy to dynamically check properties that may have quite complicated static proofs and thus are likely to be beyond the capabilities of static tools.

Third, some reported properties may be true but not useful. As an example, Daikon reported `decided` $\subseteq$ `initiated` (and a number of other properties), but we did not use that fact in the proof. Daikon uses heuristics to prune useless facts, for instance, by limiting output based on variable types. However, it is impossible for a tool to know what a human will find desirable in a given situation. We found that although there were over a dozen true but irrelevant invariants, it was easy to pass over the uninteresting ones — and examining them helped us solidify our understanding of the algorithm and the implementation. Thus, a moderate amount of extra information does not distract or disable users.

Finally, the reported properties may be more than are needed for a proof: a proof accepted by a theorem-prover may use more invariants than are strictly necessary, thus obscuring the essential argument. We believe it is better to first obtain a working, machine-verified proof, and then to reduce it after the fact. Automating this task (possibly following Rintanen [Rin00]) is future work. We did not have to perform such a reduction in our case study.

```
forward simulation from Global1 to Cons:
    Cons.initiated = Global1.initiated ∧
    Cons.proposed  = Global1.proposed ∧
    Cons.decided   = Global1.decided ∧
    Cons.failed    = Global1.failed  ∧
    ∀ v:Value (v ∈ Cons.chosen ⇔
         ∃ b:Ballot (b ∈ Global1.succeeded ∧ Global1.val[b] = embed(v) ))
proof
initially Cons = [{}, {}, {}, {}, {}]
for internal start(S: Set[Node], B: Set[Ballot]) ignore
for input init(i: Node, v: Value) do fire input init(i, v) od
for input fail(i: Node) do fire input fail(i) od
for output decide(i: Node, v: Value) do fire output decide(i, v) od
for internal makeBallot(b: Ballot) ignore
for internal abstain(i: Node, B: Set[Ballot]) ignore
for internal vote(i: Node, b: Ballot) ignore
for internal assignVal(b: Ballot, v: Value) do
 if ¬(b ∈ Global1.succeeded) then ignore
 elseif ∃ b:Ballot (b ∈ Global1.succeeded ∧ Global1.val[b] ≠ nil) then ignore
 else fire internal chooseVal(v)
 fi od
for internal internalDecide(b: Ballot) do
 if (b ∈ Global1.succeeded) then ignore
 elseif (Global1.val[b] = nil) then ignore
 elseif ∃ b:Ballot (b ∈ Global1.succeeded ∧ Global1.val[b] ≠ nil) then ignore
 else fire internal chooseVal(Global1.val[b].val)
 fi od
```

Fig. 3. Forward simulation relation and step correspondence (**proof** block) from `Global1` to `Cons`

6 Paired execution

As noted in Section 2.2, users can also exploit the IOA interpreter in formulating and checking the validity of a forward simulation relation, as they work toward the goal of proving the correctness of an implementation with respect to a specification.

A forward simulation relation is a predicate that relates the states of two automata (see Definition 1). Figure 3 contains a candidate forward simulation relation from `Global1` to `Cons`. The simulation relation is just a predicate relating the states of the two automata. It does not specify how each step in the implementation `Global1` corresponds to a sequence of steps in the specification `Cons`. In general, there might be multiple step correspondences that preserve the simulation relation; even if there is only one, it can be difficult to find it. Hence Figure 3 also contains a "proof block," which describes a step correspondence for use as an "attempted proof" of the simulation relation. With this proof block, the paired interpreter can execute the specification automaton in lockstep with the implementation automaton.

The proof block contains two sub-blocks, corresponding to the two properties needed to show a simulation relation (Definition 1). The first sub-block, started by **initially**, shows how to start the specification automaton[1]. The second sub-

[1] The set of legal start states of the specification automaton is determined by the **states** block in its code as usual; the **initially** block picks a particular start state, which may depend on the start state of the implementing automaton.

block contains an entry for each action of the low-level automaton; this entry provides an algorithm for producing a high-level execution fragment. A **proof** section may also contain a third sub-block that declares auxiliary variables used by the step correspondence.

In Figure 3, the proposed simulation relation is the identity on all state variables of `Cons` except `chosen`, which is not a state variable of `Global1`. The simulation relation defines `chosen` in `Cons` to contain a value `v` if and only if there is a successful ballot in `Global1` with value `v`. The proof block is straightforward for the start state and for the external actions: each external action in the low-level execution is matched by the action with the same name in the high-level automaton. The internal actions `start`, `makeBallot`, `abstain`, and `vote` are matched by an empty execution sequence of the automaton `Cons`.

The IOA interpreter reveals the need for the careful treatment of the internal actions `assignVal` and `internalDecide` in Figure 3. Given a naive treatment

```
for internal assignVal(b: Ballot, v: Value) ignore
for internal internalDecide(b: Ballot)
    do fire internal chooseVal(Global1.val[b].val) od
```

for these actions in the proof block, the interpreter catches two problems with the purported step correspondence. First, given a (legal) schedule that executes `internalDecide` twice in `Global1`, the interpreter discovers that the precondition for `chooseVal` fails the second time it is executed in the lockstep execution of `Cons`. Second, `assignVal` needs to fire `chooseVal` if a ballot has been decided internally but does not yet have a value assigned; hence we must fire `chooseVal` when firing `assignVal`, but only if no other ballot in `Global1.succeeded` has a non-nil value.

Most of the above case analysis is necessary because `Global1` allows ballots to be voted on (and to succeed) before they are assigned values. This nondeterminism makes the algorithm more flexible, but the proof a bit longer.

7 Verifying a simulation relation in LP

Since a paired execution provides only empirical evidence for the correctness of a simulation relation, it is desirable to supplement this evidence with a proof—ideally, a proof checked by an automated tool such as LP. The uses of simulated executions described in Sections 5 and 6 assist the LP user in constructing such a proof that the purported forward simulation relation in Figure 3 has the required properties. First, the proof block of the paired execution provides an outline for the proof. Second, invariants suggested by Daikon provide insight and can save the user time in finding auxiliary invariants needed for verification.

The LP proof that the purported simulation relation satisfies property (1) of Definition 1 is straightforward. The only interaction required from the user is to supply the start state of `Cons` specified in the **initially** section of Figure 3 as a "witness" for an existential quantifier:

```
prove start(a:States[Global1]) ⇒ ∃ b:States[Cons] (start(b) ∧ F(a, b)) by ⇒
  resume by specializing b to [{}, {}, {}, {}, {}]
```

Given this witness, LP automatically rewrites the conjecture and finds that `start(b)` and `F(a, b)` are both true, thereby completing the proof.

The LP proof that the purported simulation relation satisfies property (2) of Definition 1, being lengthier, benefits to a greater extent from the results in earlier sections. This proof proceeds by cases, one for each action of the implementation automaton `Global1`. In each case, the user must supply an execution fragment β of `Cons`, which is readily available from the **for** statements in the proof block in Figure 3: each action referred to in a **fire** statement is just an element of the witness execution, while the `ignore` statement represents the null execution. For the `init`, `fail`, `makeBallot`, `abstain`, and `vote` actions, the user need supply nothing more: LP finishes the proof automatically. For example, to guide the proof for the `init` action, it suffices to type

```
resume by specializing beta to init(n, v) * {}
```

Only a trivial amount of additional guidance (telling LP to work harder) is needed for the `decide` action.

The cases for the `assignVal` and `internalDecide` actions are themselves further divided into subcases, in accordance with the **for** statements for those actions in the proof block. In addition, the proof in these cases uses invariants `Inv1` through `Inv5`. Invariant `Inv2` is used when `ChooseVal` is the witness execution for `InternalDecide` to show that the value being chosen belongs to `Cons.proposed`. The other four, which show that all ballots not in `Global1.dead` have identical or nil values, help show that changes to `Global1.succeeded` and `Global1.val` preserve the simulation relation.

Of course, the invariants used to establish the simulation relation must be verified themselves. Here too, the interpreter and Daikon provide help. First, invariants sometimes require other invariants in their proofs. In the case study, only `Inv3` required auxiliary invariants: `Inv1` and `Inv6`. Daikon detected one of these. Second, the statement of complicated invariants such as `Inv6` can be tested via simulated execution; once stated properly, the proof of this invariant was rather simple.

Our techniques do not completely eliminate the need for human guidance in proving invariants and simulation relations. They can automatically discover, and prove with little human assistance, invariants such as `Inv1`, `Inv2`, and `Inv5`. They cannot yet discover invariants such as `Inv4` and `Inv6`, even though their proofs are simple. And although they discover invariant `Inv3`, which is simple, the proof of this invariant using LP requires moderate human guidance.

8 Conclusion

Theorem provers are the only tools that can soundly reason about general infinite state systems, leading to guarantees of correctness or other properties. A machine-checked proof provides more assurance than a hand proof, but it also carries a cost in terms of human interaction. We propose a methodology that reduces but does not eliminate the human effort required for formally proving properties of programs. In particular, the methodology partially automates some of the tedious, low-level aspects of using a theorem prover, freeing the user to focus on the proof itself.

The methodology integrates simulated execution — running a distributed algorithm over a test suite on a uniprocessor — with theorem proving. Exploratory analysis by experimenting with a system is a well-known technique for building intuition and performing inexpensive sanity checks. We extend the use of runtime techniques in two ways.

First, we use a dynamic invariant detector to generalize over observed executions, reporting logical properties that are likely to be true of the implementation. This technique reifies properties that would otherwise have to be synthesized by a person. These properties can reveal unexpected properties of the implementation, can buttress understanding more effectively than merely examining execution traces, and can provide invariants and lemmas that simplify proofs and reduce theorem-proving effort.

Second, we observe that the effort to build good test suites can be re-used in theorem-prover scripts: the proof scripts often mirror the form of the scripts for driving paired executions, and it pays to get these scripts right before investing effort in attempting a formal proof.

We have illustrated the use of the methodology, and of a toolset that supports the methodology, by means of a case study that formally proves the correctness of an implementation of consensus based on Lamport's Paxos protocol.

References

[AHM+98] Rajeev Alur, Thomas A. Henzinger, F.Y.C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. Mocha: Exploiting modularity in model checking. In *Proceedings of the Tenth International Conference on Computer-aided Verification*, volume 1427 of *Lecture Notes in Computer Science 1427*, pages 521–525, 1998.

[Bog01] Andrej Bogdanov. Formal verification of simulations between I/O automata. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 2001.

[ECGN01] Michael Ernst, Jake Cokrell, William G. Grisworld, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, 2001.

[GG91] Stephen Garland and John Guttag. *A guide to LP, the Larch Prover*. Technical report, DEC Systems Research Center, 1991. Updated version avaliable at URL `http://nms.lcs.mit.edu/Larch/LP`.

[GHG+93] John V. Guttag, James J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, New York, 1993.

[GL98] Stephen J. Garland and Nancy A. Lynch. The IOA language and toolset: Support for designing, analyzing, and building distributed systems. Technical Report MIT/LCS/TR-762, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, August 1998. URL `http://groups.csail.mit.edu/tds/papers/Lynch/IOA-TR-762.ps`.

[GSV01] Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Toward industrial strength abstract state machines. Technical Report MSR-TR-2001-98, Microsoft Research, 2001. URL for software `http://www.research.microsoft.com/foundations/asml/`.

[KCD⁺02a] Dilsun Kırlı, Anna Chefter, Laura Dean, Stephen J. Garland, Nancy A. Lynch, Toh Ne Win, and Antonio Ramirez-Robredo. The IOA simulator. Technical Report MIT-LCS-TR-843, MIT Laboratory for Computer Science, July 2002.

[KCD⁺02b] Dilsun Kırlı, Anna Chefter, Laura Dean, Stephen J. Garland, Nancy A. Lynch, Toh Ne Win, and Antonio Ramirez-Robredo. Simulating nondeterministic systems at multiple levels of abstraction. In *Proceedings of Tools Day 2002*, pages 44–59, Brno, Czech Republic, August 2002. Also available as Masaryk University Technical Report FI MU-RS-2002-05.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, September 1989.

[LY01] Leslie Lamport and Yuan Yu. *TLC – The TLA+ Model Checker*. Compaq Systems Research Center, Palo Alto, California, 2001. URL `http://research.microsoft.com/users/lamport/tla/tlc.html`.

[Lyn96] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, March 1996.

[McM] Kenneth L. McMillan. *The SMV Language*. Cadence Berkeley Labs, 2001 Addison Street, Berkeley, CA 94 704, USA. URL `http://www.cis.ksu.edu/santos/smv-doc/`.

[NE02] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *Proceedings of the 2002 International Symposium on Software Testing and Analysis (ISSTA)*, pages 232–242, Rome, Italy, July 22–24, 2002.

[PLL00] Roberto De Prisco, Butler Lampson, and Nancy Lynch. Fundamental study: Revisiting the Paxos algorithm. *Theoretical Computer Science*, 243:35–91, 2000.

[PRZ01] Amir Pnueli, Sitvanit Ruah, and Lenore Zuck. Automatic deductive verification with invisible invariants. In *Tools and Algorithms for the Analysis and Construction of Systems (TACAS)*, volume 2031 of *LNCS*, pages 82–97, Genova, Italy, April 2–6, 2001.

[Rin00] Jussi Rintanen. An iterative algorithm for synthesizing invariants. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 806–811, Austin, TX, July 30–August 3, 2000.

A A schedule block for executing `Global1`

Following is a sample schedule block for `Global1`, which produces the output in Appendix B. The full test suite, used for our runtime analysis with Daikon, employs more sophisticated constructs, such as loops and conditionals, along with randomized ballot creation. We omit it here to conserve space.

```
schedule
states
  theNodes: Set[Node] := insert([0], insert([1], insert([2], {}))) ∪
                         insert([3], insert([4], insert([5], {})))

do
  fire internal start(theNodes);
  fire input init([0], [1]);
  fire input init([1], [2]);
  fire input fail([5]);
  fire internal makeBallot([0]);
  fire input init([2], [1]);
  fire input init([4], [3]);
  fire internal assignVal([0], [1]);
  fire internal vote([0], [0]);
  fire internal vote([1], [0]);
  fire internal vote([2], [0]);
  fire internal vote([4], [0]);
  fire input init([3], [2]);
  fire internal makeBallot([1]);
  fire internal abstain([3], {[0]});
  fire internal assignVal([1], [1]);
  fire internal makeBallot([2]);
  fire internal abstain([0], {[1]});
  fire internal abstain([1], {[1]});
  fire internal abstain([2], {[1]});
  fire internal abstain([3], {[1]});
  fire internal assignVal([2], [1]);
  fire internal vote([0], [2]);
  fire internal vote([1], [2]);
  fire internal vote([2], [2]);
  fire internal vote([3], [2]);
  fire input fail([0]);
  fire internal internalDecide([2]);
  fire output decide([1], [1]);
  fire output decide([4], [1]);
od
```

B Paired interpreter output for `Global1`

Following is the beginning of the output of a paired execution of `Global1` and `Cons`, in which execution of `Global1` is driven by the schedule block shown in Exhibit A and execution of `Cons` is driven by the **proof** block of the forward simulation relation.

```
1: internal start(([0] [1] [2] [3] [4] [5])) in automaton Global1
2: input init([0], [1]) in automaton Global1
2: input init([0], [1]) in automaton Cons
3: input init([1], [2]) in automaton Global1
3: input init([1], [2]) in automaton Cons
4: input fail([5]) in automaton Global1
4: input fail([5]) in automaton Cons
5: internal makeBallot([0]) in automaton Global1
6: input init([2], [1]) in automaton Global1
6: input init([2], [1]) in automaton Cons
```