Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Using Simulated Execution in Verifying Distributed Algorithms

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

MIT Computer Science and Artificial Intelligence Laboratory
{tohn,mernst,garland,dilsun,lynch}@csail.mit.edu

The date of receipt and acceptance will be inserted by the editor

Abstract

This paper presents a methodology for using simulated execution to assist a theorem prover in verifying safety properties of distributed systems. Execution-based techniques such as testing can increase confidence in an implementation, provide intuition about behavior, and detect simple errors quickly. They cannot by themselves demonstrate correctness. However, they can aid theorem provers by suggesting necessary lemmas and providing tactics to structure proofs. This paper describes the use of these techniques in a machine-checked proof of correctness of the Paxos algorithm for distributed consensus.

1 Introduction

Theorem provers are powerful tools for ensuring that purported proofs are correct, that is, that proofs adhere to the rules of logic. The main hindrance to using theorem provers has been the amount of human input they require. General-purpose theorem provers for sufficiently powerful logics have acted less as automated verification tools than as interactive proof systems or proof assistants. Humans must provide them with two primary types of input: lemmas and tactics. Lemmas provide facts about the programs being verified, which are often necessary for correctness proofs. Tactics guide the prover in making choices during a proof, such as which lemmas to apply or whether to reason by cases or by induction.

The focus of previous work on making provers easier to use has been on analyzing syntactic structures in axioms and conjectures in order to generate potentially useful lemmas and tactics. When these lemmas and tactics do not suffice, humans must provide additional input based on their understanding of the semantic content of the axioms and conjectures. Often this understanding is faulty or incomplete. The focus on the work described here is on making it easier to use theorem provers for verifying distributed algorithms by reducing the need for this kind of human input. To this end, we use a dynamic analysis of the results of executing a program, in addition to a static analysis of the program's text and of its test suite, to increase human insight, to discover semantic content in the program's behavior, and to generate potentially useful lemmas and tactics for correctness proofs.

This is a new use for execution, which has been a traditional part of algorithm and system development, but does not yet play a direct part in formal verification. Because execution requires little human effort, it has traditionally served as a powerful prelude to formal verification, a task that requires much greater human effort. When used for testing, execution can reveal departures from desired behavior that can be corrected before attempting to prove code correct. Execution can serve in additional ways 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. Furthermore, simulated execution can be used to test specifications (expressed, for example, as abstract programs) in the same way that actual execution tests programs, even in the absence of a complete implementation.

These uses of execution to test programs and specifications occur before verification and are largely disjoint from it. This paper describes additional ways to integrate information obtained from execution into the process of formal verification.

First, we use descriptions of program behavior, extracted by dynamic analysis from executions, as lemmas in proofs. Unlike human proofs, which are peppered with phrases like "it is obvious that," machine-checked proofs often require many explicit lemmas. Some lemmas are tedious and obvious, some are not. In either case, using dynamic program analysis to provide these lemmas saves human effort.

Second, information used to construct test suites can also play a role in verification. During testing, such information ensures adequate test coverage by ensuring that all interesting cases—normal, abnormal, or borderline, as determined by the programmer, by the tester, or by static analysis—are tested. During verification, this information can be used to supply proof tactics, for example, to choose helpful case splits. Thus, tactics generated from test suites for simulated execution can complement tactics built into the prover in reducing human input.

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 [GL98], which support its use, and compares the IOA toolkit to 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 a specification and implementations for Paxos as I/O automata. Section 4 describes how these automata are executed, and Section 5 shows how dynamic invariant detection extracts lemmas for proofs from these executions. Section 5 also discusses issues that arise from using this technique. Sections 6 and 7 show how information used to test conformance of an implementation automaton to a specification via simulated execution can also be used to create tactics for use in a correctness proof for the implementation. Section 8 discusses ways to extend this research, and Section 9 concludes.

2 Preliminaries

Our methodology uses I/O automata and the IOA language to model specifications and the programs we wish to verify. Verification involves the use of 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 (possibly nondeterministic, infinite) 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 a *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* for liveness (not considered in this paper).

An action π is *enabled* in a state s if and only 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 such that (s_i, π, s_{i+1}) is a transition of the automaton for every $i \geq 0$, 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.

2.1.1 Verification methods

Proof methods supported by the I/O automaton model include *invariant assertion techniques* for proving that a particular property is true in all reachable states, *simulation methods* for proving that one automaton implements another [LV95], and *compositional methods* for reasoning about collections of interacting components. We employ the first two of these methods in this paper. A *forward simulation* shows that an automaton A conforms to a specification expressed as another automaton B. Definition 1 and Theorem 1 describe forward simulations formally. Invariants are often needed to assist in verifying forward simulations (see Section 7).

Definition 1 (Forward simulation). A *forward simulation* from an automaton A to an automaton B is a binary relation F on $states(A) \times states(B)$ with the following two properties:

Start condition: For every start state a of A, there is a start state b of B such that $F(a, b)$.

Step condition: If a is a reachable state of A, (a, π, a') is a transition of A, and b is a reachable state of B such that $F(a, b)$, then there is an execution fragment β of B leading from b to a state b' of B such that $F(a', b')$, 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, and automaton A implements specification B [Lyn96].*

2.1.2 The IOA language

The IOA language provides notations for describing I/O automata and for stating their properties; it uses specifications written in the Larch Shared Language [GHG+93] to axiomatize the semantics of I/O automata and the data types used to describe algorithms. In IOA, an automaton's state is defined by a list of state variables, and its transition relation is defined in terms of preconditions and effects, which 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 combine these two styles.

Nondeterminism appears in IOA in two ways: *explicitly*, in the form of nondeterministic choices in state variable initializations and in 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 IOA language. They all define primitive (i.e., not composite) automata in an imperative style with almost no explicit nondeterminism.

2.2 The IOA toolkit

2.2.1 The IOA interpreter

The IOA interpreter [KCD+02a, KCD+02b] simulates execution either of a single automaton running in isolation or of two automata running in lockstep. The interpreter assists users in formulating and checking properties of automata by checking stated assertions (invariants for a single automaton or a simulation relation between two automata) and displaying or logging the automaton's execution. (The IOA interpreter is also known as the "IOA Simulator," but is called an interpreter in this paper to avoid confusion with the notion of a forward simulation.)

For paired execution, a user presents the interpreter with two automata, a candidate forward simulation relation, and a step correspondence mapping actions of the purported implementation automaton to sequences of actions of the specification automaton. The interpreter simulates execution of the implementation automaton, generates the candidate execution of the specification automaton induced by the step correspondence, checks that the two executions have the same trace, and checks that the simulation relation holds at the end of the candidate execution.

2.2.2 The Daikon invariant detector

The Daikon invariant detector [ECGN01] proposes properties that are likely to be true throughout a program's execution. Daikon operates dynamically by examining values computed during execution and generalizing over those values. Its output is in the form of invariants expressible in a grammar on the program's variables. Initially, Daikon conjectures that all properties in its grammar are true on the program. Daikon then examines the execution data and deletes any invariants that are contradicted by the data. Finally, Daikon uses static analysis and statistical tests to reduce the number of false positives by eliminating some of the remaining invariants [ECGN00]. When used in conjunction with the IOA interpreter, Daikon formulates the proposed properties in the IOA language as invariants of the executed automaton.

Dynamic detection of invariants is unsound, because there is no guarantee that the test suite used to generate execution traces fully characterizes the execution environment. In practice, the reported properties are usually true and are generally helpful in explicating the system under test and/or its test suite. Furthermore, the method described in this paper does not rely on unproven lemmas; rather, it uses Daikon to suggest lemmas and a theorem prover both to prove whichever of these lemmas it can and also to use those lemmas in a larger proof.

Experiments indicate that Daikon produces output in the form of a formal specification that often matches what a human would have written [NE02a, NE02b]. In another experiment, Daikon was given inadequate test suites in order to artificially degrade its output, but even the degraded output improved programmer performance (to a statistically significant degree) when verifying the absence of certain runtime errors [NE02b].

2.2.3 The Larch Prover

The Larch Prover [GG91] (LP) is an interactive proof assistant for multisorted first-order logic. It admits specifications of theories in the Larch Shared Language (LSL). The IOA toolkit includes `ioa2lsl` [Bog00], a tool that translates IOA definitions of automata into LSL theories that describe the operation of the automata. It also generates proof obligations for asserted invariants and simulation relations.

2.3 Related work

Other toolkits, such as AsmL [GSV01], Mocha [AHM+98], SMV [McM98], and TLC [LY01], support execution or verification of concurrent and distributed systems. They use execution 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 IOA toolkit's facility for executing pairs of automata, 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 finite state spaces; theorem provers apply to finite and infinite systems alike.

The "invisible invariants" method [PRZ01] facilitates automated verification of parameterized, finite-state systems. It uses techniques from model checking to calculate candidate invariants, to check their inductiveness, and to prove the verification conditions generated by the standard invariance rule of deductive verification. For some parametrized systems, model checking for a finite number of processes suffices to verify the system in general. The invisible invariants

```
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

can be proved automatically and need not be shown to a human. By contrast, we regard invariants also as a means to inform users about interesting program properties. Invariants detected by Daikon are intended to be simple and easily readable properties. Additionally, our methodology is not limited to finite-state or parametrized systems.

3 Specifying automata in IOA

The first step in using our method for verifying a system is to define it in IOA. If the goal is to show that an implementation meets a specification, then the user defines both the specification and implementation automata in IOA. For Paxos, we use code from a case study [PLS+] that defines a hierarchy of automata. 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 the main idea of Paxos, that of using ballots and quorums to achieve consensus. A correctness proof involves showing the existence of a series of forward simulation relations between successive levels in the hierarchy. In this paper, we discuss 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)`, which represents the proposal of value `v` by process `i`, an internal action `chooseVal(v)`, which represents the choice of a consensus value `v`, an output action `decide(i,v)`, which represents the report of the consensus value `v` to process `i`, and an input action `fail(i)`, which represents the failure of process `i`. The transitions of the automaton provide 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 using our method to verify an automaton is to test its behavior by simulating its execution. The IOA interpreter simulates (on a single computer) execution of an I/O automaton, allowing the user to help select the executions and to propose invariants for the interpreter to check.

The interpreter requires IOA programs to be transformed into a form suitable for execution. For example, it requires that the initial values for all state variables, such as `quorums` in `Global1`, be specified operationally, rather than declaratively or nondeterministically. We transformed `Global1` into a form suitable for execution by adding an additional internal action

```
internal defineQuorums(theNodes: Set[Node])
  eff quorums := insert(delete([1], theNodes), {})
```

and by invoking it at the beginning of each execution.

Aside from such bookkeeping issues, the crucial problem in this transformation involves resolving nondeterminism. The IOA interpreter solves this problem by requiring the user to supply a nondeterminism resolution (NDR) program, called a *schedule*, to each source of nondeterminism in the definition of an automaton [KCD+02a, KCD+02b]. The schedule contains **fire** statements to tell the interpreter what

```
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 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[Set[Node]],
  dead: Set[Ballot] := {}

transitions
  input init(i, v)
    eff if ¬(i ∈ failed) ∧ ¬(i ∈ initiated) then
             initiated := initiated ∪ {i};
             proposed := proposed ∪ {v}
        fi
  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
        ∧ ∃ q:Set[Node] (q ∈ quorums
              ∧ ∀ j:Node (j ∈ q ⇒ 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

transitions to execute, along with code for control and initializing helper variables. Such schedules are analogous to test suites. The sample schedule

```
  schedule
  states
    theNodes: Set[Node] :=
      insert([0], insert([1], insert([2],
             insert([3], insert([4], insert([5], {})))))))
  do
    fire internal defineQuorums(theNodes);
    fire input init([0],[1]);
    fire input init([1],[2]);
    fire input fail([5]);
    fire internal makeBallot([0]);
```

causes the IOA interpreter to execute five actions and to produce the following output:

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

In the Paxos case study, we wrote schedules to execute `Global1` with different interleavings of actions, some causing nodes to fail or to abstain from a ballot. The suite used 3 nodes, 20 ballots, and fixed schedules that permitted little ordering nondeterminism. We did not use structured test generation methods to produce the schedules, nor did we evaluate them according to specific criteria (e.g., statement coverage). Instead, we simply selected executions that illustrated what we felt was the normal behavior of the automaton (and that exercised every action). In our experience, using simple schedules like these is adequate for the purpose of dynamic invariant detection. Occasionally, as noted in Section 5.2, a preliminary test run reports an unexpected invariant, which indicates a deficiency in the test data that should be corrected. Moreover, as happened in another case study involving the Peterson mutual exclusion algorithm, a preliminary test run can uncover a bug in the IOA transcription of an algorithm.

5 Dynamically detecting likely invariants

A proof of a simulation relation often depends on invariants and on auxiliary lemmas. Machine verification requires that such lemmas be stated and proved explicitly, even if they seem like bookkeeping details to the user. These parts of the proof may not be very interesting or they may be relatively simple; thus, automating them holds promise. Consequently, the third step in our method is to generate candidate invariants and lemmas automatically by using dynamic invariant detection to analyze execution data from the IOA interpreter.

Three potential problems with this third step are that the lemmas it produces may be unsound, incomplete, or not very useful. Despite these potential pitfalls, this step tends to perform well in practice. We discuss the output of dynamic invariant detection for the case study in Section 5.1 and how to cope with the three potential problems in Section 5.2.

5.1 Invariant detection results for the case study

For Paxos, invariant detection with Daikon produced 27 potential invariants. Most were easy for a human to classify. Half (14) of the proposed invariants are clearly false; they are artifacts of the test suite, such as that the size of a quorum is 2 (because there are 3 nodes in the tests) or that, in the tests, nodes do not fail after they have decided. Almost half (10) are clearly true; some of these are alternate formulations of others. (Being obviously true, however, does not imply that

they would be easy for a person to come up with.) Only the last 3 require any substantial effort to classify. Users can ignore them, think hard about them, or expand the test suite to try to falsify them.

Four of the true invariants proved helpful in the simulation relation proof in Section 7. These 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 above invariants. The two missing invariants were:

```
Inv4: ∀ b:Ballot ∀ b′:Ballot (val[b] ≠ nil ∧ b′ < b ⇒
          val[b′] = val[b] ∨ b′ ∈ dead)
Inv6: ∀ b:Ballot (b ∈ succeeded ⇒
          ∃ q:Set[Node] ∀ n:Node
             (q ∈ quorums ∧
              (n ∈ q ⇒ b ∈ voted[n])))
```

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. This is not a fundamental limitation, but a design choice that reduces Daikon's computational requirements and, more importantly, the number of false positives or unhelpful invariants Daikon would otherwise report. Section "Invariant List" of the Daikon user manual, available with the Daikon distribution at `http://pag.lcs.mit.edu/daikon`, lists all the invariants that Daikon checks and reports; the full grammar for an earlier version of Daikon appears in another paper [ECGN01]. Users can easily add new invariants to the Daikon system. Section 8 discusses other ways of improving this grammar.

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 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 all of the dynamically detected invariants to help in proposing, understanding, and verifying program properties, but we use a theorem prover to ensure that the lemmas we use in proofs are sound. Second, most of the output in our case study was correct, and those that were not 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 schedule, thereby improving the quality of the test suite for its use in Section 6. In general, simply covering every interesting aspect of each 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. As noted above, Daikon restricts the set of invariants it checks for two reasons: to conserve runtime and, more importantly, to reduce the number of false positives that it reports (the more properties it checks, the larger the number of false or non-useful properties it will report).

In the Paxos case study, Daikon proposed four of the six required invariants. This reduced the amount of particularly non-imaginative human effort required to construct the the correctness proof, even though it did not eliminate all such human effort.

It is notable that `Inv3`, although true and necessary for the proof, was not provable in isolation: establishing it required use of `Inv6`. In other words, dynamic invariant detection postulated a useful simple property (`Inv3`) whose proof is complicated (because it requires `Inv6`). This ability to decompose a proof into parts demonstrates a strength of our technique: it is easy to check properties dynamically, even if they have complicated proofs that are beyond the capabilities of completely automatic static tools.

Third, some reported properties may be true but not useful. As an example, Daikon reported a number of properties, such as `decided` $\subseteq$ `initiated`, which we did not use 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.

The reported properties may be more than are strictly necessary. A machine-checked proof may not require all the reported invariants. Furthermore, as for proofs in general, there may be an alternative proof that requires fewer invariants. In any event, it is reasonable to first obtain a complete machine-verified proof and then to simplify it. Automating this task (possibly following Rintanen [Rin00] by iterating to a minimal fix-point of invariants) is future work. We did not have to perform such a reduction in our case study.

6 Paired execution

The fourth step in our method is appropriate when attempting to verify a simulation relation. As noted in Section 2.2.1, the IOA interpreter can help users formulate and test the validity of a forward simulation relation, prior to such a verification. In this section, we discuss how a schedule and other information can help in executing paired automata, during which the IOA interpreter tests the conditions of the relation. This scheduling will later be useful in guiding verification.

A forward simulation relation is a predicate that relates the states of two automata (see Definition 1). Figure 3 defines

```
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
start
  Cons.initiated: Set[Node] := Global1.initiated;
  Cons.proposed: Set[Value] := Global1.proposed;
  Cons.chosen: Set[Value]   := {};
  Cons.decided: Set[Node]   := Global1.decided;
  Cons.failed: Set[Node]    := Global1.failed

for internal defineQuorums(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 the `Global1` specification (Figure 2) to the `Cons` implementation (Figure 1).

a forward simulation relation from `Global1` to `Cons`. Paired execution requires such a correspondence to resolve nondeterminism. Usually, the designer of an implementation has an idea of the step correspondence. The IOA toolkit allows the designer to annotate the program with this correspondence. Hence, Figure 3 also specifies how each step in the implementation `Global1` corresponds to a sequence of steps in the specification `Cons`.

The **proof** block in Figure 3 describes a step correspondence for use in testing 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 conditions required for a simulation relation (Definition 1). The first sub-block, which begins with **start**, shows how to start the specification automaton.[1] The second sub-block contains an entry for each action of the implementation automaton; this entry provides an algorithm for producing an execution fragment of the specification automaton. Syntactically, each entry uses a **fire** statement to tell the interpreter to fire the corresponding specification action. A **proof** block may also contain a third sub-block that declares auxiliary variables used by the step correspondence.

In Figure 3, the 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 implementation automaton is matched by the action with the same name in the specification automaton. The internal actions **start**, `makeBallot`, `abstain`, and `vote` are matched by an empty execution sequence of the specification automaton.

In the Paxos case study, the IOA interpreter was able to reveal two problems with the following more naive treatment in the **proof** block for the internal actions `assignVal` and `internalDecide`.

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

First, given a schedule that executes the `internalDecide` action 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 this 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 general, but it complicates the correctness proof. Hence it was helpful to use paired simulation to debug the details of the step correspondence and arrive at the formulation shown in Figure 3.

7 Verifying a simulation relation in LP

The last step in our method is to prove the simulation relation (or invariant) using a theorem prover. This guarantees the soundness of our technique. As described above, theorem provers generally require human input in the form of lemmas and proof tactics. Here we describe how the results of Sections 5 and 6 can be used to generate this input automatically. First, the invariants suggested by dynamic invariant detection become candidates lemmas, thereby saving the user time in finding auxiliary invariants needed for verification. Second, the annotations for paired execution provide a proof outline.

Recall from Definition 1 that verifying a simulation relation requires verifying both a start condition and a step condition. Translation tools in the IOA toolkit use the **proof** block for a simulation relation to generate proof tactics for each condition.

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

7.1 Start condition

The start condition requires finding a witness start state b of the specification automaton. In LP, the proof obligation is

```
start(a:States[A]) ⇒
   ∃ b:States[B] (start(b) ∧ F(a, b))
```

The IOA tools extract the witness b from the imperative statements in the **start** section of a **proof** block, which define initial values for the state variables in the specification automaton in terms of the initial values for the state variables in the implementation automaton. In the Paxos case study, the proof script generator translated the **start** section of Figure 3 into the following commands for LP:

```
declare operator
  StartRel: States[Global1] → States[Cons]
  ..
assert
  StartRel(a:States[Global1]) = [{},{},{},{},{}]
  ..
prove
  start(a:States[Global1])
    ⇒ ∃ b:States[Cons] (start(b) ∧ F(a, b))
  ..
  resume by ⇒
  resume by specializing b to StartRel(ac)
  qed
```

Here, the two **resume by** commands direct LP to begin the proof by using its built-in implication tactic, which assumes the hypothesis and replaces the universally quantified variable `a` by a fixed constant `ac`, and then using `StartRel(ac)` as the witness for the existential quantifier ∃ `b`. In the case study, these commands are sufficient to complete the proof of the start condition.

7.2 Step condition

The step condition requires finding a witness execution β of the specification automaton for each transition of the implementation automaton. The proof script generator formulates this proof obligation for LP as follows:

```
prove
  F(a:States[A], b:States[B])
      ∧ step(a, alpha: Actions[A], a′:States[A]) ⇒
  ∃ beta:ActionSeq[B]
      (execFrag(b, beta) ∧ F(a′, last(b, beta))
       ∧ trace(beta) = trace(alpha))
  ..
```

It also generates a proof script that divides the proof into cases based on the kind of the action `a` (using the command **resume by induction on** `a`, which directs LP to proceed by structural induction on the datatype of `a`) and generating lemmas to handle the details of the individual cases. For example, it generates the following lemma and proof script from the **proof** block for the `init` action in the Paxos case study.

```
prove
  F(a:States[Global1], b:States[Cons])
      ∧ step(a, init(i, v), a′) ⇒
  ∃ beta:ActionSeq[Cons]
      (execFrag(b, beta) ∧ F(a′, last(b, beta))
       ∧ trace(beta) = trace(alpha))
  ..
  resume by ⇒
  resume by specializing beta to init(ic, vc) * {}
  qed
```

LP finishes the proof of this lemma automatically, as it also does for the `fail`, `makeBallot`, `abstain`, and `vote` actions.

The proof scripts for the lemmas for the `assignVal` and `internalDecide` actions are themselves divided into cases, in accordance with the **for** statements for those actions in the **proof** block. For example, the proof script generator produces the following lemma and script for the `internalDecide` action.

```
prove
  F(a:States[Global1], b:States[Cons])
     ∧ step(a, internalDecide(b:Ballot, a′) ⇒
  ∃ beta:ActionSeq[Cons]
      (execFrag(b, beta) ∧ F(a′, last(b, beta))
       ∧ trace(beta) = trace(alpha))
  ..
resume by ⇒
resume by cases bc ∈ ac.succeeded
  % True case
  resume by specializing beta to {}
  % Elseif case
  resume by cases ac.val[bc] = nil
    % True case
    resume by specializing beta to {}
    % Elseif case
    resume by cases
      ∃ b:Ballot (b ∈ ac.succeeded
                  ∧ ac.val[bc] ≠ nil)
      % True case
      resume by specializing beta to {}
      % False case
      resume by specializing beta to
        chooseVal(ac.val[bc].val) * {}
```

LP needs further assistance, in the form of invariants `Inv1` through `Inv5`, to finish the proof of this lemma. Invariant `Inv2` is used when the action `chooseVal` is the witness execution for `InternalDecide`; there it shows that the value being chosen belongs to `Cons.proposed`. The other four invariants, which show that all ballots not in `Global1.dead` have identical or nil values, help establish that changes to `Global1.succeeded` and `Global1.val` preserve the simulation relation.

When **proof** blocks are more complicated than those in the Paxos case study, the job of the proof script generator is correspondingly more complicated. For example, in **for** entries in the **proof** block, the generator must expand sequences of conditional statements into nested conditionals. The proof script generator expands a **for** entry containing

```
if P1 then fire a1 else fire a2 fi
if P2 then fire a3 else fire a4 fi
```

into one that contains nested conditionals such as

```
if P1 then
  if P2 then fire a1 fire a3 else fire a1 fire a4 fi
else
  if P2 then fire a2 fire a3 else fire a2 fire a4 fi
fi
```

in order to generate the appropriate case splits in the proof script.

Of course, invariants used to establish a simulation relation must be verified themselves. Here, too, the interpreter and invariant detector provide help. First, invariants sometimes require other invariants in their proofs. In the Paxos case study, only `Inv3` required auxiliary invariants (`Inv1` and `Inv6`), one of which Daikon detected. 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 Future work

There are at least three ways to extend this research. First, the dynamic invariant detector could be improved, in order to find more lemmas for proofs and to increase human insight regarding program behavior. For example, boolean expressions appearing in IOA program code could be used as templates in the grammar of invariants. Since these templates come from the semantics of the program, they are likelier to be useful invariants. For example, in Paxos, `Inv4` closely resembles the precondition for the `assignVal` transition.

Second, improved static analysis of I/O automata could generate more detailed proof scripts. For example, in performing case splits, we currently examine **if** statements in the annotations for paired execution, but we could also look at **if** statements within the effects code of the automaton itself.

Third, we are extending our tools to use the Isabelle/HOL logic system and theorem prover [Pau93, Gor89]. Since Isabelle has a larger user community and a more extensive set of libraries, this may make our methodology accessible to more people. Further, we can state theorems about invariants and forward simulation relations in Isabelle/HOL's higher order logic. This allows us to prove the soundness of our method for verifying invariants and simulation relations.

9 Conclusion

Theorem provers can be used to reason soundly about the correctness of general infinite state systems. Machine-checked proofs provide more assurance than hand proofs, but incur a cost in terms of human interaction. Our methodology reduces, but does not eliminate, the human effort required for formally proving properties of programs. In particular, our methodology partially automates some of the tedious, low-level aspects of using a theorem prover, freeing the user to focus on the proof itself.

Our methodology integrates simulated execution, which runs a distributed algorithm with a test suite on a uniprocessor, with theorem proving. Exploratory analysis based on such executions is a well-known technique for building intuition and performing inexpensive sanity checks. Our methodology extends the use of run-time techniques in two ways.

First, we use a dynamic invariant detector to generalize over observed executions and report logical properties that are likely to be true of the program. This technique proposes properties that would otherwise have to be synthesized by a person. Such properties can reveal unexpected properties of a program, and they can buttress understanding more effectively than can be done merely examining execution traces. Most importantly for our methodology, such properties can provide invariants and lemmas that simplify proofs and reduce theorem proving effort.

Second, we leverage the effort used to build good test suites to produce scripts for theorem provers, which mirror the form of the scripts for driving paired executions.

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.

[Bog00] Andrej Bogdanov. Formal verification of simulations between I/O automata. Master of engineering thesis, Massachusetts Institute of Technology, Cambridge, MA, September 2000.

[ECGN00] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *ICSE 2000, Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, Limerick, Ireland, June 7–9, 2000.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001. A previous version appeared in *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

[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://theory.lcs.mit.edu/tds/papers/Lynch/IOA-TR-762.ps`.

[Gor89] M. J. C. Gordon. HOL: A proof generating system for higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 73–128. Springer-Verlag, 1989.

[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.

[LV95] Nancy Lynch and Frits Vaandrager. Forward and backward simulations — Part I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

[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.

[McM98] Kenneth L. McMillan. *The SMV Language*. Cadence Berkeley Labs, 2001 Addison Street, Berkeley, CA 94 704, USA, 1998. URL `http://www.cis.ksu.edu/santos/smv-doc/`.

[NE02a] 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.

[NE02b] Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking: An empirical evaluation. In *Proceedings of the ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE 2002)*, pages 11–20, Charleston, SC, November 20–22, 2002.

[Pau93] Lawrence C. Paulson. The Isabelle reference manual. Technical Report 283, University of Cambridge, Computer Laboratory, 1993.

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

[PLS+] Roberto De Prisco, Nancy Lynch, Alex Shvartsman, Nicole Immorlica, and Toh Ne Win. A formal treatment of Lamport's Paxos algorithm. Manuscript.

[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.