Verifying Distributed Algorithms via Dynamic Analysis and Theorem Proving

Toh Ne Win and Michael Ernst

Technical report MIT-LCS-TR-841
May 25, 2002

MIT Lab for Computer Science
200 Technology Square
Cambridge, MA 02139 USA
tohn@theory.lcs.mit.edu, mernst@lcs.mit.edu

Abstract

We use output from dynamic analysis to assist theorem-proving of safety properties of distributed algorithms. The algorithms are written in the IOA language, which is based on the mathematical I/O automaton model. Daikon, a dynamic invariant discovery tool, generalizes from test executions, producing assertions about the observed behavior of the algorithm.

We use these relatively simple run-time properties as lemmas in proving program properties. These lemmas are necessary, but easy for humans to overlook. Furthermore, the lemmas decompose complex steps into simple ones that theorem provers can manage mostly unassisted, thus reducing the human effort required to prove interesting algorithm properties.

In several experiments, Daikon produced all or most of the lemmas required for correctness proofs, automating the most difficult part of the process, which usually requires human insight.

This verification technique is a worthwhile alternative to using only static analysis with model checkers or theorem provers, or only dynamic analysis with simulators and runtime analyzers. Our technique combines the advantages of static and dynamic analysis: it is sound and scales to algorithms with unbounded processes and variable sizes. Further, it can suggest and verify new program properties that the designer might not have envisioned.

1 Introduction

Computerized methods for analyzing safety properties of concurrent algorithms fall into two basic categories: static and dynamic. Static analysis reasons about all executions of a program, either by exhaustively checking all reachable states (as in model checkers) or by logical reasoning (as in theorem provers). Dynamic analysis examines some subset of the executions, usually through test cases, and points out violations of safety properties, or generalizes from observed behavior. Dynamic analysis is unsound, as it does not see all executions, while static analysis does not scale well (model checking) or requires much human effort (theorem proving).

Our research combines the complementary strengths of dynamic and static methods. We employ dynamic analysis to discover simple likely program properties and use these as lemmas in the proofs of more complex properties in a theorem prover. These properties are necessary when using a prover, but they can be so numerous and so simple that humans overlook them. Additionally, because theorem-provers can often automatically prove simple properties, proposing these intermediate steps can reduce human effort. The end result is sound, but eliminates some steps that require human interaction and insight. Moreover, our approach is more scalable in some ways than the other major method of static verification, model checking. Finally, it permits runtime exploration and checking, permitting certain problems to be corrected before being discovered during verification.

We prove properties of concurrent programs modeled formally as I/O automata [Lyn96, LT89] and implemented in the IOA language [GLV97] so they can be read and analyzed by computer. The Daikon dynamic specification generator analyzes sample executions of the IOA programs and reports first order logic predicates that appear to be true of the programs. Both these invariants and program properties specified by the designer are then verified using the Larch Prover (LP) [GG91].

1.1 Overview

Section 2 discusses previous research similar to ours. Section 3 overviews the tools we use: the I/O automaton model, the IOA language, the Daikon invariant detector, and the Larch Prover. Section 4 describes how our method was used with these models and tools; in order to make the process concrete, it also presents a case study of the Peterson 2-process mutual exclusion algorithm [Pet81]. Section 5 relates a second case study, of Lamport's Paxos distributed consensus protocol [Lam98], and Section 6 discusses a third case study, of a strong cache for shared memory. Section 7 discusses our method, and Section 8 concludes.

2 Related work

The most closely related work is that of Nimmer and Ernst [NE01, NE02a, NE02b], who also dynamically detected, than statically verified, program properties. Their system fully automatically proved absence of run-time errors in single-threaded Java programs using the Daikon specification generator and the ESC/Java static checker. By contrast, we consider correctness proofs of formally modeled concurrent algorithms. These more sophisticated properties require a more powerful, interactive theorem-prover.

Pnueli et al. [PRZ01] propose use of "invisible invariants," which are automatically detected invariants that can be automatically proved and that never need be shown to a human. The technique works on parametrized, finite-state systems. The automatically detected invariants are generated by the following heuristic: use model-checking to obtain a property characterizing the system; throw away subscripts other than 1 and 2; and replace "1" by one variable and "2" by another. By comparison, our technique uses different generalization techniques, uses executions rather than model-checking, works on infinite-state as well as finite-state systems, and is aimed at general properties rather than only inductively provable ones. A separate publication relates the same ideas, but using model checking rather than deductive verification [APR+01].

Other researchers have used static and dynamic techniques separately to address the same goals as we do.

2.1 Static tools

The two main classes of static verification tools are model checkers and theorem provers. Both are sound.

Model checkers exhaustively analyze the entire state space of concurrent programs. Model checking requires relatively little expertise and provides counterexamples to falsified claims, but supplies no intuition regarding true ones. Despite clever optimization techniques, this search can become computationally infeasible as programs and numbers of processes and possible variable values increase. A more serious problem is that model checking analyzes only a finite state space and so, without potentially sophisticated abstractions, cannot verify algorithms containing an unbounded number of processes or an unbounded variable size.

Theorem provers manipulate logical formulae rather than states. They scale to an unbounded number of processes and variable sizes. However, theorem provers require human interaction of two varieties. *Procedural input* is low-level directives such as trying a particular proof step or method, applying a lemma, or unifying a set of variables. *Substantive input* embodies human insight such as stating a necessary lemma that is not obviously related to the final proof goal.

Some theorem-provers contain strategies or proof tactics that reduce or eliminate the need for procedural inputs. Search in the space of proofs can also assist with this problem. Producing substantive inputs is harder. Our thesis is that dynamic analysis can assist users by providing certain substantive inputs.

2.2 Dynamic tools

Execution works for arbitrary programs. Its disadvantage is that it examines only a specific set of executions, so its results are unsound. Examining every execution would amount to model checking. A test suite represents an attempt to provide a finite but relatively comprehensive set of executions. With concurrent programs, an additional issue arises: not only do test cases have to be representative of a particular program's execution, they also have to exhibit all interesting behavior between interleavings of executions of different processes.

Not all program specifications may be easily executable. In particular, programs that are written declaratively have to have a mechanism for generating executions. When working with programs containing unbounded variable sizes, the generation of an execution is undecidable [Wes01]. IOA gets around this by having the programmer specify next steps [RR00].

Several other tools besides Daikon perform dynamic analysis to recover a specification [CW98, ABL02, RKS02, HL02]. However, none of these tools provides output in first order logic.

3 Background

This section introduces the tools we used in our research.

3.1 I/O automata and IOA

This section introduces the I/O automaton model [Lyn96, LT89] and the IOA language [GLV97]. An I/O automaton A is made up of five parts:

- $states(A)$ is a state space, usually written as a cross product of some variables. Variables of unbounded size induce infinite state spaces.
- $start(A) \subseteq states(A)$ is a set of start states.
- $sig(A)$ is a signature, that lists the actions of the automaton.
- $trans(A) \subseteq states(A) \times actions(A) \times states(A)$ is a transition relation that tells which actions are enabled at which states, and the effects of the actions. Input actions are always enabled as transitions.

- A set of task partitions $tasks(A)$ that group actions into equivalence relations in order to ensure fair execution. This paper ignores $tasks(A)$, which is only used for liveness properties.

An execution of an I/O automaton is a sequence of interleaved actions and states. The set of all possible executions is written as $execs(A)$. A trace of an execution is the sequence of all the external actions in the execution. The set of all traces is written as $traces(A)$.

A safety property P of A defines a set of traces, $traces(P)$ that fulfill the following requirements:

- $traces(P)$ is nonempty.
- $traces(P)$ is prefix-closed: all finite prefixes of a trace in $traces(P)$ are also in $traces(P)$.
- $traces(P)$ is limit-closed: if an infinite sequence of traces $\beta_1, \beta_2, \ldots$ are in $traces(P)$ and each β_i is a prefix of β_{i+1}, then the trace β that is the limit of the sequence is also in $traces(P)$.

Intuitively, a property P is a safety property if it defines executions of the form "nothing bad happens". By adding history variables, safety properties of A can be expressed as predicates on states of A. We refer to the predicate as $P(s)$ where s is a state of an automaton.

For example, the safety property of a leader election algorithm is that at most one process declares itself leader. The "bad" thing is when two processes declare themselves leader. In a good execution of the algorithm, there will be at most one leader declared and all prefixes of the execution will exhibit the same property. We could add a history variable, *leaders* that tracks the set of declared leaders and rewrite the safety property to say that the size of *leaders* is 0 or 1.

There are two major ways to prove that safety properties hold.

The first method is invariant assertion, where we prove that the reachable states s in $states(A)$ satisfy the safety property $P(s)$. Two facts suffice:

- $\forall_{s \in start(A)} P(s)$
- $\forall_{s \in states(A)} [(P(s) \wedge \langle s, a, s' \rangle \in trans(A)) \Rightarrow P(s')]$

The second method is simulation relation. If automaton B satisfies a safety property and $traces(A) \subseteq traces(B)$, then A satisfies the safety property. We can show $traces(A) \subseteq traces(B)$ by showing that there exists a forward simulation relation f from A to B [LV95, LV96]. We say that B is the specification automaton and A the implementation automaton. A forward simulation relation f satisfies:

- $\forall_{s \in start(A)} \exists_{u \in start(B)} f(s, u)$
- $\forall_{s, s' \in states(A), u \in states(B), a \in sig(A)} [\langle s, a, s' \rangle \in trans(A) \wedge f(s, u)] \Rightarrow$
$$\exists_{u' \in states(B), \beta \in execs(B)} u' = exec(u, \beta) \wedge [trace(\beta) = trace(a)] \wedge f(s', u')$$

The second part says that for every enabled transition with action a from state s to s', there exists an execution fragment β of B starting from u that has the same trace as a and maintains the simulation relation. In order to show $f(s', u')$, we usually use invariants of A and the hypothesis that $f(s, u)$. (Technically, we also have to also show that s and u are reachable states. However, for the simulation, we are only interested in the reachable states, where the invariants have been proven to hold.) When multiple levels of simulation relation are used to prove that an algorithm implements a specification, it is called "successive refinement".

Thus, for proving safety properties of I/O automata, invariants can be used as lemmas for invariant assertions (as in Section 4) or as lemmas for simulation relations (as in Sections 5 and 6).

3.2 The IOA language

The IOA language allows I/O automata to be written as programs. Figure 2 is the IOA implementation of the Peterson 2 process mutual exclusion algorithm analyzed in Section 4.1.1. To facilitate implementation, IOA has the following features:

- Each transition contains a (conjoined) set of preconditions. Transition effects may be specified declaratively (as a predicate on pre and post states) or imperatively (using assignments). Figure 2 uses the latter.
- Variables are typed, and new types may be defined.
- The start state is implicit in the variable initializers.
- Safety properties can be expressed as invariants and as simulation relations. These are checked during execution and are written as proof obligations for LP (see Section 3.4).

3.3 The Daikon dynamic invariant detector

The Daikon invariant detector [ECGN01] performs dynamic analysis of program executions. It reports first-order logical properties that hold over the run-time values of program variables and that statistical tests indicate are likely to hold in general. Dynamic invariant detection is unsound: the properties are likely, but not guaranteed, to be true of the analyzed program. Human examination or static checking can often verify the properties that Daikon produces.

Daikon checks all properties expressible in its grammar (which is described in other papers) and reports those that are never falsified at run time and that satisfy certain other tests. A larger grammar would permit reporting more potentially valuable properties. However, checking a larger set of properties can produce more false positives — properties that are not interesting or are not true in general. A larger set of properties also takes longer to check. Finally, simpler properties are more likely to be useful (and comprehensible) to people and tools. For all of these reasons, Daikon uses a relatively small grammar. Three of its limitations are on atomic boolean formulas, boolean connectives, and quantifiers.

- Daikon postulates and checks atomic boolean formulas (such as "$x \neq y$") containing at most three variables. If there are n program variables in scope at a particular point, there are n^3 such atomic formulas; increasing the number of variables additively increases the exponent. In our experiments, we never needed atomic boolean formulas of arity greater than 2.
- Daikon avoids using the boolean connectives $\vee$, $\Rightarrow$, and $\Leftrightarrow$. Each use of such a connective multiplicatively increases the exponent in the number of formulas to check.
- Daikon avoids existential quantifiers $\exists$. Daikon's current generalization rules work well for universal quantifiers $\forall$ but produce too many false positives for existential quantifiers.

These rules are relaxed in certain cases. For instance, our system presents some compound terms such as `pc[turn]` to Daikon as single (oddly-named) variables. Additionally, some invariants involving boolean connectives are checked [DDLE02]. (Users can request use of all boolean connectives, at the cost of slower performance.)

These restrictions greatly limit the number of properties that Daikon checks and reports. However, we have found that the results, while relatively simple, are effective in our domain. While more complicated invariants might be useful in certain circumstances, they have not proved necessary as of yet.

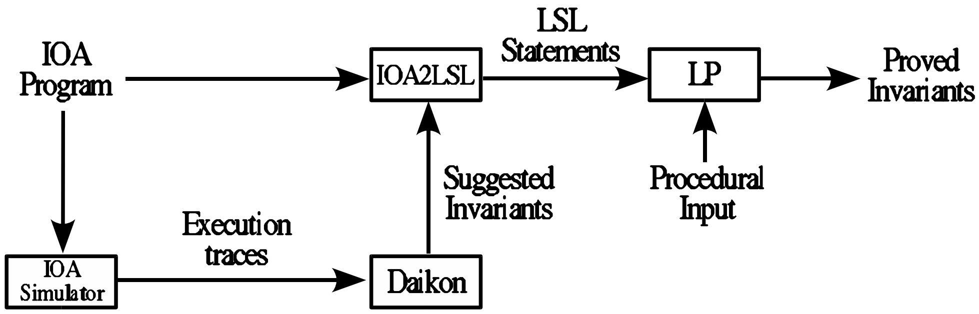

Figure 1: The process of using theorem provers with dynamic analysis on IOA programs. The "IOA Simulator – Execution Traces – Daikon – Suggested Invariants" path is novel to our research.

3.4 The Larch Prover

The Larch Prover (LP) [GG91] is an interactive theorem prover that uses multisorted first-order logic. The `IOA2LSL` tool converts an IOA program and its invariants and simulation relations into first order logic (actually, the Larch Shared Language, or LSL [GHG+93]) for LP, as described in Section 4.5. The I/O automaton's transitions become LP assertions relating pre- and post-states of the automaton.

No theorem prover can determine both truth and falsehood in finite time [Sip97]. LP's solution is to attempt to prove something true by applying some lemmas, and halt if more than some given number of steps are needed. If LP halts on a proof attempt, then either the conjecture is false or something that LP does not yet know is true. As a result, LP requires frequent human input. It can be thought of as a verifier that checks proof scripts for correctness. However, input to LP can be still classified as procedural or substantive, as defined in Section 2.1.

Suppose that we want to prove the conjecture "`A(sc) => B(sc)`", where `sc` is a program variable. A procedural step tells LP to use a particular proof method or to apply known facts. For example, the command "`resume by =>`" tells LP to assume `A(sc)` and to generate the new proof obligation `B(sc)`. A substantive step might begin a proof of `C(sc)`, where $\texttt{C(sc)} \Rightarrow \neg\texttt{A(sc)}$.

The choice of what rules to use and what facts to apply is a search problem. A number of theorem-provers differ from LP in that they require little or no procedural input, but instead guide the search via heuristics called proof tactics. Typically, humans still provide substantive input to direct the proof at a high level.

Because good solutions for relieving uses from providing procedural inputs exist (even though the LP theorem-prover does not yet incorporate them), we focus on the remaining problem of providing substantive inputs. Our thesis is that dynamic analysis can come up with most of the necessary lemmas more efficiently than other strategies such as an exhaustive search through all possible proofs.

4 Proving correctness of IOA programs using Daikon and LP

This section has two purposes. It describes the process of using dynamically detected invariants to assist in theorem-proving, and it also details an example of that process, applied to a mutual exclusion algorithm.

4.1 Overview

Our goal is to prove the correctness of distributed algorithms expressed as IOA programs, using dynamically detected invariants as a theorem-proving aid. Our analysis involves the following steps, as illustrated by Figure 1: write the program in IOA, run it in the IOA Simulator, run Daikon over the executions, and verify Daikon-suggested invariants in the Larch Prover with some human-provided procedural help.

The remainder of this section gives further details about each step of the process and, in parallel, makes the descriptions concrete by way of an extended example, the Peterson 2-process mutual exclusion algorithm [Pet81]. While performing this case study, we discovered a new proof of the algorithm's critical safety property.

4.1.1 Case study: Peterson 2 process mutual exclusion algorithm

The Peterson 2 process mutual exclusion algorithm [Pet81] achieves lockout-free mutual exclusion using multi-writer, multi-reader, read-write shared memory. This is a good subject for a case study because mutual exclusion algorithms can be subtle and testing them is rarely sufficient. In IOA, mutual exclusion is expressed as the invariant that exactly one automaton is in the set of states designated as the critical region.

We proved the Peterson 2-process mutual exclusion algorithm correct in LP using only invariants discovered by Daikon. Daikon's output provided a guide for the proof, so the human user did not need to supply any lemmas, only procedural input (which another theorem-prover could automate). Daikon detected the mutual exclusion property that was the final goal, and the invariants discovered by Daikon were relatively easy to prove.

A previous LP proof of the algorithm, along with IOA-style pseudocode, appears in [Lyn96]. We did not examine the pseudocode or the proof until after completing our own implementation and proof. Our proof ended up quite different from the reference one, but was about the same length in terms of LP commands.

4.2 Writing the program in IOA for execution

Our process requires an algorithm to be translated into the I/O automaton model [Lyn96, LT89], written in the IOA language [GLV97], and then made executable. All of these steps are straightforward.

Making the model executable requires resolution of nondeterminism, because IOA expresses constraints over executions, but any particular execution embodies specific choices. In IOA, a "schedule" specifies which step (chosen among all legal steps) is taken at each point in an execution. The IOA language allows scheduling code to be appended to automaton specifications.

The scheduling code determines what executions are seen by Daikon, so creating representative executions is important. This is analogous to generating test cases for bug detection or any other dynamic analysis.

4.2.1 Case study: Peterson code

Figure 2 gives the IOA code for the Peterson 2-process mutual exclusion algorithm, and Figure 3 shows the state-transition diagram for a single process. The algorithm operates as follows. Every process sets its flag to true, then sets the turn variable to itself. From then on, each process checks the other's flag and the turn variable. If either the other process's flag is off (`checkFlag`), or if the turn variable points to the other process (`checkTurn`), the first process is allowed to go into the

```
% Peterson 2-process mutual exclusoin algorithm, implemented in IOA.

% There are two processes, named p1 and p2.
type ProcType = enumeration of p0, p1
% There are 6 states of the automaton ("PC" stands for "program counter").
type PCType = enumeration of waiting0, trying0, trying1, trying2, critical0, critical1

automaton Peterson
signature
  output trying (p : ProcType)
  internal setFlag (p : ProcType), setTurn (p : ProcType),
           checkFlag (p : ProcType), checkTurn (p : ProcType)
  output critical (p : ProcType), release (p: ProcType)

% Each process has a program counter and a boolean flag, and there are two global variables.
% The array tupe "Array[A,B]" is indexed by keys of type A and contains elements of type B.
states
  pc : Array[ProcType, PCType] := constant(waiting0),
  flag : Array[ProcType, Bool] := constant(false),
  turn : ProcType := if randomBool then p0 else p1,
  critCount : Int := 0         % Number of processes in critical region (0, 1, or 2).

transitions
  output trying(p)
  pre pc[p] = waiting0
  eff pc[p] := trying0

  internal setFlag(p)
  pre pc[p] = trying0
  eff pc[p] := trying1;
      flag[p] := true

  internal setTurn(p)
  pre pc[p] = trying1
  eff pc[p] := trying2;
      turn := p

  internal checkTurn(p)
  pre pc[p] = trying2
  eff if (turn ≠ p) then
        pc[p] := critical0
      fi

  internal checkFlag(p)
  pre pc[p] = trying2
  eff if (flag[if (p = p0) then p1 else p0] = false) then        % Check other process.
        pc[p] := critical0
      fi

  output critical (p)
  pre pc[p] = critical0
  eff pc[p] := critical1;
      critCount := critCount + 1

  output release (p)
  pre
  eff pc[p] := waiting0;
      critCount := critCount - 1;
      flag[p] := false
```

Figure 2: The Peterson 2 process mutual exclusion algorithm in IOA. For brevity, this figure omits the scheduling code that chooses among possible executions at runtime.

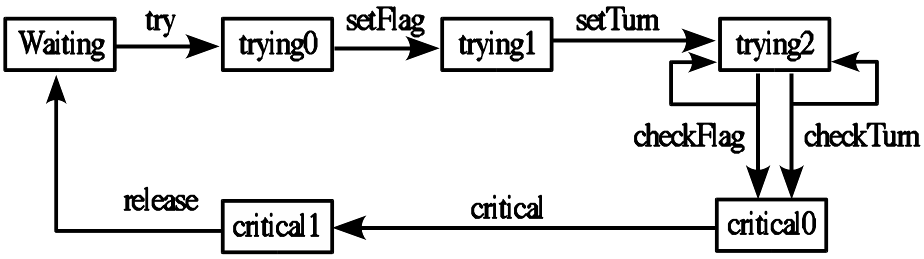

Figure 3: State-transition diagram for one process in the Peterson algorithm.

critical section. The critical region consists of the states in which the program counter has the value `critical0` or `critical1`.

We added the variable `critCount` to let Daikon count how many processes are in the critical region.

4.3 Running the program

The IOA program and its schedule are run using the IOA Simulator, which is an interpreter that can write, to a text file, runtime information needed by Daikon, namely the states of the I/O automaton and what actions were fired. We call this information "executions" rather than "traces" because the term "trace" already has a meaning in the I/O automaton model (Section 3.1).

4.3.1 Case study: Peterson run

The Peterson IOA program was scheduled for execution using random scheduling. That is, the scheduler selected one of the two processes at random and advanced its state, if it was possible. The IOA program was run for 2000 transitions.

4.4 Running Daikon

Execution data from the Simulator is given to Daikon for analysis, and Daikon outputs a set of invariants in IOA syntax. Thus, the results from Daikon can be easily recombined with the original IOA program.

Some of Daikon's invariants are beyond the IOA syntax. In particular, Daikon can detect transition-specific invariants. These become global invariants when turned into an implication, where the left side is the precondition of the transition and the right side is the program property. An example is:

```
enabled(t1) => x > 5
```

The IOA language does not allow transition invariants to be expressed succinctly — the precondition `enabled(t1)` must be fully expanded to the text of the precondition. Since the Larch Prover can parse the above term, we write transition invariants directly into the Larch Shared Language (LSL) described in the next section.

4.4.1 Case study: Reported invariants

We now describe Daikon's output when given the Peterson executions. We first describe an implementation error that Daikon quickly and conveniently exposed. After correcting the problem, we reran the IOA Simulator and Daikon. The remainder of the section describes Daikon's output given the correct implementation.

On an initial run over the Peterson executions, Daikon reported (among other properties)

```
(critCount = 0) \/ (critCount = 1) \/ (critCount = 2)
```

The initial implementation did not achieve mutual exclusion: `critCount` was sometimes greater than 1. We had made an error in writing the IOA code: in our initial implementation, each process set its `turn` variable first, then its `flag` variable.

Running Daikon immediately revealed this error. We would have been unlikely to notice this property in the raw executions. We could have detected the error by having the IOA Simulator check that `critCount < 2` during execution, but the user would have needed to decide a priori which of potentially many properties to check during execution. In particular, we did not think to make the Simulator check this particular property. We could also have detected the error during theorem-proving, but again, it was faster, easier, and more convenient to notice it in Daikon's summarization of the runtime properties.

After correcting the error, we reran the Simulator and Daikon; Daikon reported 82 invariants, all of which we believe to be true. Of these, 54 were redundant (and could be easily removed by a simple filtering process) and 28 were non-redundant. Our proof used 8 of the 28 non-redundant invariants; different proofs might have used different sets.

The 54 redundant invariants are implied by the semantics of IOA. For instance, 6 invariants state action preconditions and 21 invariants state action effects. Daikon is provided only with the runtime values, not with the IOA program, and some of Daikon's output may be syntactically present in the original program. Other redundant invariants stated that transition parameters did not change, but IOA transition parameters are immutable. Finally, some redundant invariants related variables that were generated from one another, such as stating that `pc[turn]` is a member of `pc` (when both variables were presented to Daikon without any indication of their relationship). A tool that interpreted both IOA programs and Daikon's output could easily filter out all of these redundant invariants, but we have not yet built such a tool.

The 28 non-redundant invariants were potentially useful. We used 8 of these in our proof. These 8 invariants fall into two groups, both of which were necessary for the correctness proof.

- Simple conditions. These gave basic information about the local state of an automaton. The invariants were of the form:

  ```
  enabled(checkFlag(p)) => flag[p] = true
  ```

  This says that whenever the checkFlag transition is enabled in a process, its flag variable is on. These simple conditions are not stated in the code, but some of them are apparent from static inspection.
- Global conditions that relate more than one process. There were two such invariants:

  ```
  enabled(checkFlag(p)) => flag[turn] = flag[p]

  enabled(checkTurn(p)) => pc[turn] = pc[p]
  ```

  Together, they produced the fundamental lemma that was needed for the proof.

For convenience, we combined the invariants in each class into one by grouping invariants with identical right-hand sides. We also rewrote the `enabled` conditions back to the preconditions of the transitions. Thus we ended up with two invariants (in LP format), which we named `InvA` and `InvB`.

```
InvA(s) <=> \A p ((s.pc[p] = trying1
                  \/ s.pc[p] = trying2
                  \/ s.pc[p] = critical0
                  \/ s.pc[p] = critical1)
                 => s.flag[p])

InvB(s) <=> \A p ((s.pc[p] = trying2) => (s.pc[s.turn] = trying2))
```

We did not use the other 20 non-redundant invariants, though at least some of them would be useful for alternate proofs. One example of an invariant we did not use is

```
enabled(release(p)) => flag[turn] = true
```

In the future, we hope to use methods outlined in Section 7 to automatically use or eliminate these invariants.

4.5 Proving invariants

Theorem-proving with the Larch Prover (LP) requires input in Larch Shared Language (LSL) [GHG+93] syntax, plus human assistance to guide LP. We used the `IOA2LSL` tool to converts the IOA programs, invariants, and simulation relations into LSL. Thus, the proof obligations are automatically generated along with the assertions about the program's code.

Given the lemmas proposed by Daikon, completing a proof requires two steps, which can be done in any order.

1. Prove the lemmas proposed by Daikon, working directly from the program about which they are asserted. Since the proposed lemmas are relatively simple, this step should be easy.
2. Assume the lemmas and prove the final goal (a safety property, simulation relation, or other property). The lemmas take the place of substantive human input, which greatly eases proving program properties.

Together, these steps lead to a sound proof of safety properties. Now, we describe how invariants and simulation relations are proved in LP, as developed in [Bog00].

Invariants The goal is to prove an invariant `Inv` on all reachable states. We prove first that `Inv` holds on the start state. Then we prove that if `Inv` holds on state s, and if a is a valid action from s, `Inv` also holds on the post state s'. This is written in LP as:

```
prove Start(s) => Inv(s)
prove Inv(s) /\ isStep(s, a, s') => Inv(s')
```

which is nearly identical to the mathematical formulas in Section 3.1.

If other invariants are required, we write them on the left side of the implication:

```
prove Inv(s) /\ AuxInv(s) /\ isStep(s, a, s') => Inv(s')
```

Later, we should also prove `AuxInv(s)` for all reachable s.

Simulation relations To prove a forward simulation relation $f(s, u)$ between the states of the lower level automaton s and the states of the upper level automaton u in the IOA program, we write:

```
prove Start(s) => \E u : States[UpperLevel] (f(s, u) /\ Start(u))
prove isStep(s, a, s') /\ f (s, u) =>
  \E beta : Execs[UpperLevel]
        (trace(beta) = trace(a)
     /\ f(s', last(u, beta)))
     /\ execFrag(u, beta)
```

where *beta* is an execution fragment of the upper level automaton, $last(u, beta)$ is the last state of the fragment, and $execFrag(u, beta)$ is a predicate indicating that *beta* is a valid execution from u. With an auxiliary invariant, the second line above becomes:

```
prove isStep(s, a, s') /\ f (s, u) /\ AuxInv(s) =>
  \E beta : Execs[UpperLevel]
        (trace(beta) = trace(a)
     /\ f(s', last(u, beta)))
     /\ execFrag(u, beta)
```

Both invariant and simulation relation proofs are completed using induction in LP:

```
resume by induction on a : Actions[LowerLevel]
```

LP then produces a proof subgoal for each possible action the lower level automaton can take. This is an intuitive way of reasoning, as hand proofs of simulation relations and invariants would have to go through each enabled transition. Each subgoal is proven relatively easily using procedural steps, as long as the assumed lemmas (i.e., substantive steps) are sufficient.

4.5.1 Case study: Proving Peterson invariants

We used `InvB` and `InvA` described in Section 4.4.1 to prove the mutual exclusion property. The LP script was 68 lines (without comments). All of the input into LP was procedural — telling the prover to apply one of the lemmas that were in its knowledge base, or to attempt to continue by case analysis.

The two invariants were also proved correct, with fewer lines of commands (36 and 24 versus 68). All invariants were proved using the methods described in Section 4.5.

Lynch [Lyn96] also proves the algorithm correct, using two invariants. The first is our `InvA`:

```
InvA(s) <=> \A p ((s.pc[p] = trying1
                 \/ s.pc[p] = trying2
                 \/ s.pc[p] = critical0
                 \/ s.pc[p] = critical1)
                => s.flag[p])
```

The second invariant is like `InvB` but explicitly mentions the other process and is written in terms of program counters:

```
InvC(s) <=> (\A p \A p' ((p ~= p'
                         /\ (s.pc[p] = critical0 \/ s.pc[p] = critical1)
                         /\ (   s.pc[p'] = trying2
                             \/ s.pc[p'] = critical0
                             \/ s.pc[p'] = critical1))
                        => s.turn = p'))
```

Lynch sketched out a manual proof but did not verify the invariants mechanically. We proved them using LP. This was not difficult, and the proof of mutual exclusion was about the same length in LP commands as our original proof. However, `InvB` is simpler than `InvC`, because Daikon can only detect relatively simple invariants.

While simple invariants and a short proof are desirable, the main benefit of using Daikon to produce lemmas is relieving the user of the burden of doing so. Coming up with `InvC` or even `InvB` requires time and careful reasoning, whereas Daikon produced `InvB` with little human effort. Later, users can re-examine the reasoning that leads to the correctness proof using Daikon invariants and thereby achieve a deeper understanding of the algorithm.

5 Case study 2: Lamport's Paxos algorithm

To see how well our method would work on a larger example, we applied it to the Paxos distributed consensus protocol [Lam98]. Paxos implements consensus in the presence of stopping node failures. It works for an unbounded number of processes, and our implementation uses sets of unbounded size.[1] A companion paper [LS] gives three simulation relations to prove the algorithm correct using successive refinement, and supplies three automata, `Global1`, `Global2`, and `Paxos`, that implement the `Cons` specification. The `Cons` specification is at the highest level of abstraction, `Global1` and `Global2` are next, and `Paxos` is the implementation itself, at the lowest level of abstraction.

Whereas the Peterson case study proved an algorithm correct from scratch, for this cases study we wished to determine how much of an existing proof we could semi-automatically reconstruct — that is, how many of the invariants Daikon could discover. We tested our method on the simulation relation between `Global1` and `Cons` because it contained the greatest number of invariants and was relatively easy to execute.

The goal of the Paxos algorithm is for a quorum of nodes to agree on a value for a variable. Each round of the algorithm consists of a ballot proposing a value, and each node either abstains or casts a vote (accepts the value). If a quorum of the nodes accept a value, then it is selected as the consensus value.

5.1 Invariants required

In order to prove the simulation relation between `Global1` and `Cons` correct, we required seven invariants. A hand proof [LS] mentioned four (numbered 1 through 4 here) but in the LP proofs, #3 needed #6, #4 needed #7, and the simulation relation needed #5. The invariants in IOA syntax were:

```
1. \A n : Node \A b : Ballot (b \in voted[n] => ~(b \in abstained[n]))
2. \A b : Ballot (val[b] ~= nil => val[b].val \in proposed)
3. \A b : Ballot (b \in succeeded => ~(b \in dead))
4. \A b : Ballot \A b' : Ballot
         ((val[b] ~= nil /\ b' < b) => (val[b'] = val[b] \/ b' \in dead))
5. \A b : Ballot (b \in succeeded => b \in ballots)
6. \A b : Ballot (b \in succeeded =>
                    \E b' : Ballot \A n : Node (n \in writequorums(b') => b \in voted[n]))
7. \A b : Ballot (~(b \in ballots) => val[b] = nil)
```

[1] An improvement suggested by Lamport does not require unbounded sets, but the unique IDs of the ballots are still unbounded.

A succeeded ballot has been (affirmatively) voted upon by a quorum. A dead ballot has been abstained from by a quorum. A ballot may be succeeded, dead, or neither (but not both). Variable `voted[n]` is the list of all ballots for which node `n` has affirmatively voted.

5.2 Invariants detected

Daikon was able to detect invariants 1–5, but not 6 and 7. Invariant 6 was not detected because it contains an existential quantifier on quorums. Section 3.3 noted Daikon's weakness with respect to existential quantifiers. Invariant 7 was not detected because the left hand side of the implication featured a negation, and the set of ballots not in the `ballots` variable did not appear in a variable supplied to Daikon. (It would be easy to supply to Daikon the difference of all sets used in a program, but we are not sure how generally useful such an enhancement would be.)

The Daikon output for #1 and #3 differed slightly from that shown above. For instance, Daikon produced

```
\A n : Node size(voted[n] \I abstained[n]) = 0
```

(where `\I` stands for intersection). We rewrote that to the logically identical #1 for convenience in theorem-proving: LP has more powerful axioms for dealing with elements of sets than with operations over entire sets, such as intersection.

5.3 Proving the invariants and assessing results

One of the authors proved all 7 invariants and the simulation relation between `Global1` and `Cons` [IN02]. Each of the invariant proofs was relatively short (24 lines on average), and the simulation relation proof was a bit longer (116 lines), mainly due to the greater number of cases to consider. All steps were procedural.

The full proof [IN02] also addresses the lower levels of the Paxos protocol, ultimately showing that the distributed version of Paxos, with communications channels, implements consensus. We have not yet applied our technique to the other parts of the proof. However, in the part we examined, results from dynamic invariant detection would have been useful.

6 Case study 3: Strong cache shared memory

We report on one more case study, a strong cache for shared memory, where every processor has a cache and there is a central store. Each cache can have a value or be empty. A separate paper proves this implementation correct via simulation relations and successive refinement [Bog00].

When performing a write to shared memory, a processor updates the central memory location and clears the caches of other processors (in one synchronous step). Processors can arbitrarily copy from the central memory location to their caches and can delete their cached values. When performing a read, a processor either reads from its cache or waits for the cache to be filled.

One invariant is enough to show that this strong caching algorithm implements shared memory: when a processor's cache is not empty, its value is equal to the central memory's value. This invariant was detected by Daikon as:

```
\A n : Node (cache[n] ~= nil => cache[n].value = mem)
```

We proved the simulation relation between the strong caching algorithm and shared memory specification in LP, and our proof was nearly identical to Bogdanov's [Bog00].

7 Discussion

We have demonstrated that dynamic analysis is capable of detecting some lemmas needed for safety proofs of the examples we studied. While these results are promising, it is uncertain whether our method and tools will generalize more broadly.

7.1 Automation

This section lists a number of tasks that could be automated, further reducing human effort to use our technique and tools. Even without any of these automation techniques, using the dynamically proposed invariants made theorem-proving easier. Since theorem-provers can be notoriously difficult to use, this is a worthwhile accomplishment. However, we would like to ease the task further.

One disadvantage of our system is that several automatable steps still require manual intervention. The first of these is that LP is a proof verifier or human-directed theorem-prover, rather than an automatic theorem-prover. It requires guidance that other systems incorporate as "tactics," heuristics indicating how to solve particular problems. We expect that our system would translate with little change to such a system, which would nearly eliminate human effort. We used LP for our experiments because LP is already integrated with the IOA language, which in turn is executable and is integrated with the Daikon invariant detector.

A second variety of missing automation is elimination of invariants that are obvious from the structure of the IOA program, as discussed in Section 4.4.1. These, too, should not be difficult to remove. Given the LSL translation of the IOA program that is supplied to LP, these invariants are vacuously true, they will not hinder tools, but they do clutter the output for humans.

A third variety of automation is determining which invariants proposed by the unsound dynamic analysis are correct. We believe that the technique proposed by Rintanen [Rin00] and implemented in the Houdini annotation assistant for ESC/Java [FL01, FJL01] will be effective. The technique starts with a set of potential invariants and iterates until fixpoint, at each stage weakening some invariant that cannot be determined to be correct. If even one required invariant is missing, then this technique eliminates all other invariants that depend on it. If the desired property is not automatically proved, then the eliminated invariants could be examined by a human. This technique can be augmented by trying to verify the negation of unprovable invariants. Together, these techniques can split the proposed invariants into those known to be true, those known to be false, and those whose correctness is unknown.

A final variety of automation is determining which true invariants are useful for a proof. If (a subset of) the proposed invariants lead to a proof, then our work is nearly done. However, it may be useful to reduce the size of the proof. A number of approaches might be successful here, including removing invariants one by one to see which are not useful, or performing a search over all proofs using the set of proposed invariants. The latter search is more computationally tractable than one that attempts a search over all possible proofs. Even an invariant that is not useful for a particular proof may be valuable in other contexts.

7.2 Enhancements to dynamic invariant detection

Our current tools are limited by the properties discovered by the dynamic analysis. Ideally, Daikon would detect every interesting property, and in particular a complete chain of invariants that lead to a proof. As briefly mentioned in Section 3.3, Daikon's grammar is relatively limited, in order to control both the runtime of the system (checking more properties takes more time) and the number of false positives reported (if more properties are checked, then more properties that are not actually true will slip through and be reported). A more extensive grammar would improve completeness and result in more good properties being reported. We are examining the impact of using a larger grammar, such as increasing the number of variables in each atomic predicate, increasing the number of conjuncts that it considers, and using existential quantifiers.

Another enhancement would be to feed back information from theorem-proving as hints to the dynamic analysis. For example, Daikon would not need to check properties that have been proved true, or it could expand the number of properties checked over variables appearing in properties that could not be proved, or over variables that appear in many provable invariants.

The dynamic analysis is limited by the executions it analyzes. Ideally, it would be provided all executions (and all possible interleavings) or all interesting ones. This approach scales badly with process count. Model checkers either suffer from this problem or rely on the user to create an abstract finite-state version of the multiprocess algorithm [CGP99]. However, real program behavior can often, but not always, be seen with just a few processes and key example executions — that is, good test cases. Daikon works with these sample executions, suggesting general properties. These general properties can be proved in LP for all executions, since process count is irrelevant to theorem provers.

The same argument applies to variables of unbounded (or very large) size. Thus, Daikon and LP can reason about sets rather than fixed size arrays, and integers rather than bytes.

8 Conclusion

We have presented a new method for verifying safety properties of distributed algorithms where a theorem prover is assisted by output from dynamic analysis. The main advantage of this method is that it reduces the amount of human insight required in a proof, leading to faster progress with the prover. At the same time, the logic behind is proof is visible to the user in the form of invariants, so human understanding is not diminished. Moreover, by suggesting properties of algorithms, our method might lead to better understanding of, or different ways of looking at an algorithm. Our method seems to scale well on unbounded variable sizes and number of processes. Finally, the method can suggest and verify new program properties that the designer might not have envisioned.

Acknowledgments

Nancy Lynch provided feedback on our ideas and presentation and suggested the Paxos case study. Stephen Garland helped us use the Larch Prover. Dilsun Kirli Kaynar was the first to execute mutual exclusion algorithms in the IOA Simulator, including Dijkstra's and Peterson's mutual exclusion algorithms. Nicole Immorlica did many of the LP safety proofs for Paxos.

References

[ABL02] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, Portland, Oregon, January 16–18, 2002.

[APR+01] T. Arons, A. Pnueli, S. Ruah, J. Xu, , and L. Zuck. Parameterized verification with automatically computed inductive assertions. In *13th International Conference on Computer Aided Verification*, volume 2102, pages 221–234, Paris, France, July 2001.

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

[CGP99] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[CW98] Jonathan E. Cook and Alexander L. Wolf. Event-based detection of concurrency. In *Proceedings of the ACM SIGSOFT '98 Symposium on the Foundations of Software Engineering*, pages 35–45, Orlando, FL, November 1998.

[DDLE02] Nii Dodoo, Alan Donovan, Lee Lin, and Michael D. Ernst. Selecting predicates for implications in program analysis, March 16, 2002.

[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.

[FJL01] Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino. Annotation inference for modular checkers. *Information Processing Letters*, 2(4):97–108, February 2001.

[FL01] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *International Symposium of Formal Methods Europe 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517, Berlin, Germany, March 2001.

[GG91] Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Technical Report 82, Digital Equipment Corporation, Systems Research Center, 31 December 1991.

[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, NY, 1993.

[GLV97] Stephen J. Garland, Nancy A. Lynch, and Mandana Vaziri. IOA: A language for specifying, programming, and validating distributed systems. Technical report, MIT Laboratory for Computer Science, 1997.

[HL02] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE'02, Proceedings of the 24th International Conference on Software Engineering*, Orlando, Florida, May 22–24, 2002.

[IN02] Nicole Immorlica and Toh Ne Win. Verifying Lamport's Paxos protocol using computer-assisted methods, May 24, 2002.

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

[LS] Nancy Lynch and Alex Shvartsman. Paxos made even simpler (and formal). In preparation.

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

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

[LV96] Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations, II: Timing-based systems. *Information and Computation*, 128(1):1–25, 1996.

[Lyn96] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, CA, 1996.

[NE01] Jeremy W. Nimmer and Michael D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Proceedings of RV'01, First Workshop on Runtime Verification*, Paris, France, July 23, 2001.

[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)*, 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-00)*, Charleston, SC, November 20–22, 2002.

[Pet81] Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.

[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 *Lecture Notes in Computer Science*, 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 on Innovative Applications of Artificial Intelligence*, pages 806–811, Austin, TX, July 30–August 3, 2000.

[RKS02] Orna Raz, Philip Koopman, and Mary Shaw. Semantic anomaly detection in online data sources. In *ICSE'02, Proceedings of the 24th International Conference on Software Engineering*, Orlando, Florida, May 22–24, 2002.

[RR00] J. Antonio Ramírez-Robredo. Paired simulation of I/O automata. Master of engineering, Massachusetts Institute of Technology, Cambridge, MA, September 2000.

[Sip97] Michael Sipser. *Introduction to the Theory of Computation*. Brooks/Cole Publishing, 1997.

[Wes01] Michael Wessel. Obstacles on the way to qualitative spatial reasoning with description logics: Some undecidability results. In *International Workshop on Description Logics (DL2001)*, Palo Alto, CA, August 1–3, 2001.