# Connecting specifications, executions, and proofs: reducing human interaction in theorem-proving

Toh Ne Win, Michael D. Ernst, and Stephen J. Garland

MIT Laboratory for Computer Science
200 Technology Square, Cambridge, MA 02139 USA
{tohn,mernst,garland}@lcs.mit.edu

## Abstract

Verifying with a theorem prover that distributed algorithms meet their formal specifications can be tedious, requiring much human interaction for any interesting property to be proved. By using a combination of runtime analysis and tactics specific to the I/O automaton model, we can eliminate some or all of this human interaction, leading to faster generation of proofs that still provide insight for human readers. We show how a dynamic analysis technique originally made for simple, sequential data structures and properties can be extended and combined with some heuristics to produce proof scripts that only occasionally make the prover stop to ask for human input.

## 1 Introduction

Software is increasingly used in applications where failure has severe consequences to human safety or to property. Formal verification of computer programs has the potential to reduce the risks to users and the public. Despite the obvious benefits and some notable successes, formal verification of computer programs is little used. Several factors contribute to this lack of use. Some techniques (such as model checking) are applicable only to finite-state systems. Others are applicable only to system models, not to actual software. Still others (such as theorem-proving) require significant human interaction. Additionally, users generally must explicitly state the properties to be checked or proved.

This paper describes a generally applicable technique that eases the burden on human users. Our long-range goal is the fully automatic discovery and proof of interesting properties of distributed algorithms: we would like to build a system that, given either a program or specification, outputs a formal, machine-checked proof of the correctness of nontrivial facts about the program or specification. We are still far from this goal. Indeed, we believe that human insight will always be necessary for some programs and specifications. However, this paper describes some promising initial steps toward reducing the human interaction required in machine-verified theorem-proving, including the automated discovery and proof of lemmas necessary for proving distributed algorithm correct.

Our methodology rests on two key tool-assisted techniques. The first technique uses dynamic (runtime) program analysis to propose likely program invariants by generalizing over observed program executions. These likely invariants serve both as goals and as lemmas. The second technique uses automated theorem proving, augmented by automatically generated application-specific tactics, to sift provable invariants from unprovable ones and to structure the proofs of those invariants. Section 2 describes how we use our techniques and tools to produce proofs.

Our methodology combines the complementary strengths of specifications (or programs), execution, and theorem-proving to produce a system with many desirable properties. For example, human intervention is not needed to propose goals, nor to assist in verifying those goals by directing a theorem-prover. However, numerous challenges arise in the process of integrating our tools, largely because they were designed for different domains with different purposes in mind. Specifications are not necessarily executable, and, even when they are, they may be written in a fashion quite different from standard programming style. Specifications often describe complicated relationships, many of which are beyond the scope of our invariant detector, which controls both complexity and its rate of false negatives by limiting the data structures and properties it manipulates. Finally, the detected invariants may be unsound, incomplete, redundant, or unstructured. We describe solutions to each of these problems, which together provide our system with the desired properties.

We have applied our system to proving (using two different automated theorem provers) theorems about distributed algorithms for mutual exclusion, consensus, and consistency. The results (described in Section 4) range from automatically proposing and proving partial correctness properties to automatically providing key lemmas that make up part of a proof, the other parts of which still requires human insight. In each case, machine assistance reduces the human effort required to produce a complete formal verification. Section 5 discusses other work related to reducing the amount of user interaction required in proving properties of distributed algorithms.

[[**Steve: Mention section on future work, if we have one.**]]

## 2 Overview of techniques and tools

We provide a very brief overview of the tools we use and of the steps we take in performing a proof. In the next section, we describe some of the complexities that we had to overcome to use the tools productively in this fashion.

### 2.1 Writing programs and specifications

Our process requires an algorithm to be formulated in the I/O automaton model [Lyn96,LT89], written in the IOA language [GLV97], and then made executable. We discuss the first two of these steps here and the last in Section 3.1.

An I/O automaton is a simple state machine in which transitions between states are associated with named *actions*, which are classified either as externally visible *input* or *output* actions, or as *internal* actions visible only to the automaton itself. An I/O automaton consists of its *signature*, which lists its actions; a set of *states*, some of which are distinguished as start states; and a *state-transition relation*, which contains triples of the form (state, action, state). The operation of an I/O automaton is described by its *executions* $s_0, \pi_1, s_1, \ldots$, which are alternating sequences of states and actions, and by its *traces*, which are the externally visible behavior occurring in executions. A property $I$ is an *invariant* of an automaton if it is true in every state in every trace of that automaton. One automaton *implements* another if all its traces are also traces of the other.

The IOA language provides notations for describing I/O automata and for stating their properties. In IOA, transition relations can be defined in an imperative style (as a sequence of assignment, conditional, and loop statements), in a declarative style (as a predicate relating state variables in the pre- and post-states), or in a combination of these two styles.

### 2.2 Executing programs and specifications

An IOA automaton, once made executable as described in Section 3.1, is run using the IOA Simulator [KCD+02], which is an interpreter that can write, to a text file, information obtained from traces of the automaton that forms the basis for dynamic runtime analysis.

### 2.3 Extracting invariants by dynamic program analysis

Execution data from the Simulator is given to the Daikon invariant detector [ECGN01], which proposes properties likely to be true in all reachable states. Daikon operates dynamically, by examining values computed during execution, postulating and checking properties, and reporting those that pass a battery of statistical and other tests. Daikon's output is potentially unsound, because there is no guarantee that the test suite fully characterizes the execution environment. However, the reported properties are often true and generally helpful in explicating the system under test and/or its test suite.

We produce the goal properties automatically using Daikon. These might or might not be the actual goal desired by a person, for instance, a complete correctness property. However, sometimes it is, and even when it isn't, if it's a provably true property (especially one that is automatically proved), that is still useful.

### 2.4 Proving invariants

Finally, we use two interactive theorem provers, LP [GG91] and Isabelle [NPW02], to prove the invariants proposed by humans or conjectured by Daikon. Although

theorem provers are good at applying routine proof tactics, such as simplification or certain types of case analysis or induction, automatically, they often require procedural or substantive input (as discussed in Sections 3.3 and 5) to establish their goals. Using two provers on the same program property gives us confidence that the substantive steps we provide using Daikon are sufficient to prove the program correct.

**[[Steve: We should move the definition of procedural and substantive input from Section 5 to Section 3.3, which now comes earlier, and shorten the above cross-reference. We should also eliminate or better integrate the following vestigal comment: We ran our tools on each one, producing a partial proof (sometimes just a list of proposed invariants, lemmas, or theorems). We then completed the proofs by hand, resulting in a machine-verifiable proof script for the desired property.]]**

## 3 Obstacles, impedance mismatch

**[[Toh: Is there a formal definition of impedance? Are we just saying tools mismatch?]]**

The idea outlined in the previous section can be relatively simply explained; however, a naive/straightforward integration of the tools would not succeed. The reason is that the tools were not designed to work together; their strengths do not necessarily match up. We can think of this as an impedance mismatch between the various parts of our combined system. In order to obtain a working system, we were forced to make both theoretical and practical advances in the tools. This section lists many of the problems and how we overcame them.

(Here, briefly list the issues/techniques that we then treat in more detail, including giving examples, in each subsection.)

### 3.1 IOA specifications vs. execution

**[[Steve: Someone needs to take a crack at tightening up what I wrote here.]]**

IOA programs are not always convenient for both theorem-proving and execution. There is an inherent tension between providing detailed operational specifications suitable for execution and providing less detailed specifications for essential requirements without overly constraining implementations. IOA programs meant as specifications generally contain considerable non-determinism; those meant as implementations don't have (as much) nondeterminism.

*Impedance mismatch 1:* Hard to execute general specifications.

IOA programs meant as specifications are often formulated with little attention to operational behavior or to efficiency. This makes it hard for Daikon to observe (any or enough of) their behavior.

Solution 1A: Extensions to the IOA language. User resolves scheduling nondeterminism by supplying simulator with an explicit schedule. Simulator resolves other nondeterminism by making random choices.

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.

**[[Steve: Do we want to include a short example of an IOA schedule? Also, the material from here to impedance mismatch 2 needs to be reduced to approximately half its current length.]]**

Solution 1B: Progressive refinement within IOA. For example, suppose we wish to prove that $I$ is an invariant of automaton $A$, but that the specification of $A$ does not lend itself to execution by Daikon. One way to proceed would be to produce an executable specification for another automaton $A'$, use our methodology to produce a proof that $I$ is an invariant of $A'$, and then show that any invariant of $A'$ is also an invariant of $A$. How do we do this? One way is to show that any trace of $A$ is also a trace of $A'$. For this, it suffices to show that there is a simulation relation between $A$ and $A'$. There is a risk here of falling into a vicious circle: since we are trying to reduce the user interaction required to prove that $I$ is an invariant of $A$, it seems counterproductive to require yet another proof (of the existence of a simulation relation) that itself may require user interaction. However, the risk need not be fatal, any more than it it would be fatal to introduce additional assertions that must be proved (i.e., lemmas) in order to reduce the user interaction required to prove some property of interest (i.e., a theorem).

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

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

*Impedance mismatch 2:* Hard to control how users write their specifications.

**[[Steve: Shorten these remarks.]]**

This affects both execution and proving. Specifications in Nancy's book, for example, employ notations from set theory. Generally, this makes those specifications more suitable for proofs than for execution. We could try to train users to rewrite specifications in a way that makes one of our jobs (e.g., execution)

easier. However, as noted above, rewriting specifications may make another of our jobs (e.g., proving) more difficult, or it may make the specifications less suitable for the task for which they were intended.

In any event, it seems desirable and more compelling to work with specifications as they stand. Hence, in this paper, we use specifications that other people wrote and found convenient for some other purpose.

**[[Steve: Do we want to say anything about ugly things in Peterson or other IOA code? How we either worked around them or changed them?]]**

### 3.2 Daikon simplicity vs. specification complexity

The Daikon invariant detector is intentionally simple: it reports simple properties over simple datatypes. There are two reasons for this. First, simpler properties are easier to understand, evaluate, and use: a true property that is incomprehensible is of no use. Second, our intuition and anecdotal evidence suggest that enlarging the grammar of checked invariants increases the number of distracting false positives (properties true in the test suite but not universally true) more than it increases the number of true positives.

However, a general problem in linking specifications and automated tools (especially dynamic ones) is that specifications tend to have greater complexity than implementations or executions. Whereas tools are best at reporting simpler properties, specifications are written at a high level of abstraction, express sophisticated properties, and incorporate domain-specific concepts. This argues for loosening Daikon's stringent restrictions on the complexity of reported concepts and for using a theorem-prover to filter out the additional false positives (along with other unprovable properties). We discuss how we accomplished this in a general-purpose way, generating logically complex properties from a conceptually simple engine. We discuss data structures and then properties.

The Daikon invariant detector operates over basic datatypes: integers (scalars and hashcodes), strings, and sequences. This keeps the implementation simple, fast, and portable to many different programming languages. IOA specifications use more sophisticated datatypes, in particular, tuples and maps from arbitrary types to arbitrary types. Furthermore, the data structures built up from them tend to be more complex and lacking in intermediate abstractions that would help guide understanding. This may be a result of the mathematical bent of the specification writers.

In order to increase the range of data types Daikon can handle, our tool that extracts program executions from the output of the IOA simulator "decomposes" complicated data structures into either scalars or sequences. An example of a data structure decomposition is how our system handles the *voted*, a data structure in the Paxos case study that is a map (IOA array) from *Node* to *Set[Ballot]*. Since nodes are not ordered, it would not make sense to print out the set of ballots voted by each node in any order. Further, the data structure is inherently two-dimensional, since each set is an unordered sequence. To decompose this data structure, we gave Daikon a random sampling of an element

of the *map*. At every transition, the IOA Simulator was made to select two random elements, *anIndex* and *anotherIndex* that are different from each other. We then created synthetic variables *voted[anIndex]* and *voted[anotherIndex]*, which are one-dimensional and examinable by Daikon. Later on, when Daikon reports invariants on these indexed synthetic variables, they are universally quantified on the index variables. We chose to use two index variables because this lets us use Daikon's detection of binary invariants within the same data structure.

**[[Steve: The above paragraph needs to be clear about how the decomposition occurs automatically. Who "prints," "gives Daikon" something, "makes" the IOA Simulator do something, "creates" synthetic variables, "quantifies" them universally? The user is responsible for at least one of these actions (making the simulator do something extra). Say something about this kind of change in a program being safe, because it doesn't affect the program's traces. Say something comforting about the user having to do little else.]]**

**[[Toh: I can also mention slicing of non nil elements if you like, but I think one example may be enough for this]].**

For data structures, we also indicated specific known properties of variables to Daikon. For instance, Daikon reports lack of duplicates for sequences and also tests sortedness and other order-related properties; when a sequence is marked as representing a set, these tests are suppressed. Thus for *voted[anElement]*, Daikon will not report that the variable contains no duplicates, even though this is (obviously) true.

**[[Mike: Toh, what other set-theoretic derived variables are there?. Toh: there's: order, no duplicates, indexed by 0 (so that we don't derive x[0..l-1] type vars, hasSize (maps don't), hasNull. Non set-theoretic annotations: isParam]]**

| Invariants | | Implications | | Simplified |
|---|---|---|---|---|
| $S_1$ | $S_2$ | $a \Rightarrow b$ | $\neg a \Rightarrow \neg b$ | $a \Leftrightarrow b$ |
| $a$ | $\neg a$ | $a \Rightarrow d$ | $\neg a \Rightarrow f$ | $a \Rightarrow d$ |
| $b$ | $\neg b$ | $a \Rightarrow e$ | $\neg b \Rightarrow \neg a$ | $a \Rightarrow e$ |
| $c$ | $c$ | $b \Rightarrow a$ | $\neg b \Rightarrow f$ | $\neg a \Rightarrow f$ |
| $d$ | $f$ | $b \Rightarrow d$ | | |
| $e$ | | $b \Rightarrow e$ | | |

**Fig. 1.** Creation of implications from invariants over subsets of the data. The left portion of the figure shows invariants over two subsets $S_1$ and $S_2$. The middle portion shows all implications that can be generated from the invariants over the two subsets; $c$ appears unconditionally, so does not appear in any implication. The right portion shows the implications after logical simplification.

The Daikon invariant detector reports only simple properties (usually involving no more than three values, some of which may themselves be combinations

of variables). The lemmas in proofs can be much more complicated. One previous (but unpublished) extension that was particularly helpful was reporting implications of the form "$p \Rightarrow q$". To limit false positives and control runtime, Daikon does not test every possible implication that can be generated from its invariant templates. Instead, it heuristically splits the execution data into two parts $S_1$ and $S_2$, then compares the invariants over the two parts. Whenever some property $p$ is true in $S_1$ and false in $S_2$, then $p$ implies all invariants of $S_1$ and $\neg p$ implies all invariants of $S_2$. Figure 1 illustrates this process. As many independent splits as desired can be performed.

**[[Steve: Who determines what's desired?]]**

The execution data can be split into parts based on a variety of factors, including static analysis of the code and dynamic analysis of runtime data. The splitting criterion need not be in Daikon's grammar or even be expressible in the underlying programming language; its purpose is solely to split the runtime data. We found that a simple static analysis sufficed for our generation of useful splitting predicates. We look at the syntax of each precondition, decomposing conjuncts into individual terms. Each term then becomes a splitting predicate at the automaton level. Thus, in addition to Daikon detecting implication invariants for when a precondition holds, Daikon also outputs invariants for when parts of a precondition holds.

There is one complication in this process: preconditions to transitions can have variables of the transition that are not in scope at the automaton level. For example, for the Peterson case study, a precondition to the *checkFlag* transition was:

```
internal checkFlag(p:PCType)
  pre pc[p] = trying2
```

since $p$ is a free variable, using *pc[p]* would be meaningless at the automaton level or as a splitting predicate. We thus use a simple syntax rule: replace every free variable by the generic variable *anIndex*. As mentioned previously, Daikon already sees *pc[anIndex]* as a synthetic variable, so there is enough data for Daikon to use *pc[anIndex] = trying2* as a splitting predicate. Any uses of this variable replacement that does not make sense (e.g. a lone occurance of *anIndex* that is not used to index an array) are ignored by Daikon.

**[[Steve: Can we eliminate everything up to the end of this section?]]**

**[[Toh: We should standardize on using implication or condition as a prefix to "invariant". What I'm trying to say here: we want properties of the entire automaton, which corresponds to the CLASS program point in Daikon. Thus we can't have transition variables.]]**

Thus, our system produced invariants that indicated properties of particular points in the code despite performing invariant detection on the system as a whole.

**[[Give a concrete example here. Toh: just did]]**

**[[free variables are converted into sample variables. * universals Universals over non-numerically-indexed arrays: this gives essentially**

**all the interesting properties of Peterson. Toh: just did in the data structures section ]]**

**[[Toh needs to say something here or in the next section about how some Daikon detected properties don't at first look like useful properties, but are when properly formatted and quantified. Maybe I'll use the banking no duplicates example here, or the inequality technique and quantification]].**

### 3.3 Daikon output vs. proofs (and theorem provers)

The goal of our system is to produce proofs that both demonstrate a property and provide intuition to a human. The proposed invariants are far from a proof, even if they capture the key elements of one. In particular, the invariants may be unsound (untrue), hard to prove (even if true), irrelevant, or redundant. Furthermore, interesting relationships between invariants (for example, the order in which they are most easily proved) may not be apparent, and theorem provers may require procedural input to complete their proof. Proper integration of Daikon's output with a theorem-prover overcomes these difficulties.

Theorem provers automatically apply routine proof tactics, such as certain types of simplification, case analysis, and induction; we can think of this as a local search over the space of logical formulae that permits provers to move from one fact to a nearby one. For example, a prover may be able to establish automatically that a transition such as

```
output transfer(amt: Int, i: Index, j: Index)
  eff Accounts[i] := Accounts[i] - amt;
      Accounts[j] := Accounts[j] + amt
```

preserves an invariant such as $total(Accounts) = initialTotal$ by simplifying an expression for the value of that invariant in the post-state of the transition to the invariant assumed to be true in the pre-state. However, a prover may need assistance if either the transition definition or the invariant is more complicated (for example, if the transition takes some complicated action when deposits are placed on hold). Daikon can help a theorem prover overcome this problem by providing substantive input in the form of additional likely invariants (for example, that characterize the effects of placing deposits on hold). The theorem prover may be able to prove some of these additional invariants automatically, and the invariants so proved may enable the theorem prover to perform the remaining steps of the previously difficult proof automatically.

Other situations in which Daikon can produce similar help arise in proofs by induction, where it is often necessary to formulate and prove a somewhat stronger property than was originally conjectured. In both kinds of situations, the required additional invariants can be tedious and time consuming for a human to discover and enumerate. Dynamic program analysis provides a way of overcoming this problem.

Theorem provers may also require procedural assistance to show, for example, that an invariant is preserved by all transitions of an automaton, not just by those

associated with a single named action. For simple automata, this assistance is easy to supply, for example by a command

```
prove Inv(s) /\ isStep(s, a, s') => Inv(s') by induction on a
```

in LP and by a proof script

```
lemma A_inv_ind:
  "!! s s' a. ((reachable A s) & (Inv s)) ==> ((s, a, s'): trans_of A --> (Inv s'))"
  apply(simp add : A_def A_trans1_def A_trans2_def ... trans_of_def Inv_def)
  apply(case_tac a, auto)
done
```

in Isabelle. We have automated providing this kind of assistance for LP by a tool `ioa2lsl` tool [Bog00] *Cite paper instead* that converts IOA programs and asserted invariants into LSL, an input language for LP, and that automatically generates proof obligations and proof tactics for establishing these invariants, and we are now developing a tool `ioa2Isabelle` that provides similar assistance for Isabelle.

In general, we believe that tools such as `ioa2lsl` and `ioa2Isabelle`, which perform static analysis for specific programs, can suggest better proof tactics for those programs than can logical frameworks such as `Isabelle/IOA` [?] or heuristically driven theorem provers, which have less knowledge about program structure. Tools based on static analysis, for example, can propose case splits that correspond to preconditions for transitions or to conditional expressions used to specify the effects of a transition.

Even with good tactics, proofs may still be hard. For example, in human-directed theorem proving, it can be difficult to figure out the order in which to prove a series of lemmas, particularly because some of those lemmas may require additional but unstated lemmas for their proof. Here, dynamic invariant detection can provide stepping-stones for a proof that would have been impossible if a prover had to take bigger steps. With many candidate lemmas at its disposal, an automated theorem prover can bootstrap a proof by first establishing whatever lemmas it can with completely automatic proofs, and then by automatically using those lemmas as substantive input for the proofs of further lemmas and eventually the main theorem of interest.

[[**Steve: Turn the rest of this section into short, but decent prose.**]]

Finally, good integration can help produce humanly digestible proofs.

* humans want short, direct proofs without redundancy; machines don't mind lots of extraneous properties or long proofs

```
  * proof search vs. proof presentation (eg, Isabelle or LP vs. Isar)
  * Talk about the typical structure of the proof.
    Claim:  the proofs are so stylized that we can generate the proof
      scripts automatically.
```

Most notably, the system can arrange the invariants in the form of a proof, indicating which ones were relied on in the proofs of others. Doing so requires

The system can report either only those invariants that are provably true or, alternately, those that are not provably false. Furthermore, it can present the invariants to the user in the order that they would appear in a proof.

## 4 Case studies

This section illustrates the difficulties described in the last section, the capabilities of our techniques for overcoming them, and the salient features of the resulting proofs.

[[**It also compares the proofs created with the assistance of our tools with other proofs of the same algorithms.**]]

We consider, in turn, four distributed algorithms: the Peterson mutual exclusion algorithm; an algorithm for ensuring memory coherence in the presence of distributed caches; the Paxos algorithm for achieving distributed consensus; and an algorithm that guarantees the consistency of accounts in a distributed banking system. We followed the process described in Section 2, resulting in a machine-verifiable proof script for a partial correctness property of the IOA program.

The correctness of the algorithms is not a novel result — they have been previously proven correct. In fact, some of them are finite-state, so model checking (which requires much less human interaction than theorem-proving) could have proven their correctness. We used these algorithms as examples for three reasons. First, while the correctness is not a new result, these particular proofs are. The proofs were easier to generate than the corresponding proofs performed without the assistance of our tool. In some cases the proofs were largely or completely automatically generated, and in other cases they indicated properties that humans had been unaware of, thus leading to new insight. Second, the IOA specifications were created by several different people in several different styles to solve several different problems. Thus, the proofs provide both preliminary evidence of the generality of our techniques. They also illustrate the techniques and give a feel for their range. Third, these examples are fairly simple. This helped us to more easily understand the difficulties that arose during our case studies, and it made these examples appropriate starting points. (And appropriate points for a first report of results.)

(Especially for the finite-state examples, de-emphasize how interesting the proof is; these can be completely solved with model checking.)

[[**Maybe this goes at the end; it doesn't really feel like it should go before the discussion.**]] (The point of using two theorem-provers is to show that the properties we get are really the interesting ones, that the substantive output is sufficient, and that the procedural output really is incidental, as shown by how different the proofs are despite the use of the identical lemmas. Also, we can perhaps say a few words about the structure of the proofs, and in the future we can automate the tactics.)

Isabelle and LP needed the same lemmas for the proof. This is evidence that the lemmas really are substantive, and that Daikon is detecting things that

are really interesting. The procedural input is remarkably different for the two theorem-provers, which is more evidence of the substantiveness of the lemmas.

(In each subsection, start out talking about our work, then bring in the others as a comparison.)

### 4.1 Peterson mutual exclusion algorithm

**[[Steve: Shorten the section on Peterson to two pages.]]**

```
Peterson: 3 proofs
 * Nancy's book.
   Nancy wrote lemmas by hand.
 * Toh LP proof, done without reference to either previous version.
   Lemmas automatically generated using Daikon.
   Lemmas different than Nancy's.  Are they the same as Steve's?
 * Toh Isabelle proof, using same invariants as his LP proof.
   How do Toh's two proofs compare?
```

**[[Steve: Delete the following comments about a related proof of Dijkstra mutual exclusion, which I did with an NTT visitor, but which has not yet been pushed through Daikon? Attempted to push Nancy's proof through LP. Started off with invariant from Nancy's book. Had to simplify it (by hand) to get it to go through the theorem-prover; found a simpler invariant that was clearer and easier to understand, but just as adequate for the proof (speculate on why that might be; for instance, a hand proof is easy, but a machine proof is hard, so optimization was necessary). Or, the way in which the prover got stuck without the invariant suggested what the invariant should be.]]**

Side note: As in computer chess, thorough shallow search by computer may in some cases be better than search by a human.

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 when one automaton is in the set of states designated as the critical region, the other automaton is not. This is written by Daikon as:

**[[Steve: Move later, to where we discuss Daikon output?]]**

```
((pc[anIndex:ProcType]) = (critical0:PCType)) /\ anIndex ~= anotherIndex =>
  pc[anotherIndex:ProcType] ~= critical1:PCType
((pc[anIndex:ProcType]) = (critical0:PCType)) /\ anIndex ~= anotherIndex =>
  (pc[anIndex:ProcType] ~= pc[anotherIndex:ProcType])
((pc[anIndex:ProcType]) = (critical1:PCType)) /\ anIndex ~= anotherIndex =>
  pc[anotherIndex:ProcType] ~= critical0:PCType
((pc[anIndex:ProcType]) = (critical1:PCType)) /\ anIndex ~= anotherIndex =>
  (pc[anIndex:ProcType] ~= pc[anotherIndex:ProcType])
```

We proved the Peterson 2-process mutual exclusion algorithm correct using both LP and Isabelle 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.

```
% There are two processes, named p0 and p1.
type ProcType = enumeration of p0, p1
% 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)

  states
    % A program counter and boolean flag for each process are kept
    % in arrays of type Array[A,B], which are indexed by keys of type
    % A and contains elements of type B.
    pc : Array[ProcType, PCType] := constant(waiting0),
    flag : Array[ProcType, Bool] := constant(false),
    turn : ProcType

  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] then pc[p] := critical0 fi
    output critical (p)
      pre pc[p] = critical0
      eff pc[p] := critical1

    output release (p)
      pre pc[p] = critical1
      eff pc[p] := waiting0; flag[p] := false
```

**Fig. 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 2 gives the IOA code for the Peterson 2-process mutual exclusion algorithm. 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 critical section. The critical region consists of the states in which the program counter has the value `critical0` or `critical1`.

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.

**[[Need to update the description of Daikon's output for Peterson.]]**

**Detecting Peterson 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) that the mutual exclusion property did not actually hold, for 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 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.

**[[Steve: Eliminate numbers for now. Put them in only when we get interesting ones (or if the program committee insists).]]**

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 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.

**Proving Peterson invariants** We used `InvB` and `InvA` described in Section 4.1 to prove the mutual exclusion property in LP and Isabelle. All of the input into the provers was procedural—telling them 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. All invariants were proved using the methods described in Section 6.3.

With Isabelle, the tactics automatically generated by the IOA to Isabelle translator were enough to entirely prove correct `InvA`. For `InvE` the steps were automatic enough to follow the pattern for all but one transition (*checkFlag*), where we had to add some procedural steps (again on reasoning by case analysis of whether $anIndex = p$). For `InvM`, the same was true for two transitions, *checkFlag* and *checkTurn*.

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. Lynch's proof also uses two invariants. The first is our `InvA`.

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.

### 4.2 Atomic memory for distributed caches

```
Memory: 3 proofs
 * Daikon-assisted LP proof
   Daikon-assisted Isabelle proof
   Andrej LP proof
```

Andrej has the same lemma, but quite different proof steps. Perhaps the tools helped lead Toh to a cleaner, less monolithic invariant? (Probably not arguable except as a very brief side note.)

Did Daikon generate the main lemma? How is this related to the hand proof?

We report on another 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 Isabelle, and our proof was nearly identical to Bogdanov's [Bog00]. The invariant shown above was proved from our automatically generated tactics without human intervention. The simulation relation proof also followed a semi-automated tactic, where the user only had to enter a witness execution and some procedural steps to state the properties implied by the simulation relation[1].

### 4.3 Paxos algorithm for distributed consensus

**[[Steve: Make sure that what we say here addresses different issues than we did in the VMCAI paper. Perhaps move some of the earlier material about synthesized invariants here, because that material used Paxos as an illustration.]]**

The Paxos algorithm implements distributed consensus in an asynchronous system in which individual processes can fail. Processes propose values and the service returns decisions to processes such that the decision is a proposed value and is the same for all proposing processes. We applied our technique to prove that an implementation of the algorithm matches a specification; both are written in IOA, but at different levels of abstraction. The correctness proof involves showing the existence of a forward simulation between the two automata. Paxos is an infinite-state system, so model checking cannot prove it correct.

The invariant detector produced 23 invariants, of which we used 4 in our proof:

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

The existence of 19 unused invariants was not a hindrance. It was easy to select ones that were useful, and the others took little time to consider. They were all true (for instance, `decided` ⊆ `initiated`); some might have led to different proofs, and others would not have been useful for this task. On an earlier run, an obviously incorrect invariant (the size of the `failed` set was a constant) indicated a problem with our test suite.

In order to complete the proof, we first added two more invariants that were outside Daikon's grammar (they contained too many clauses): a dead ballot has been abstained from by a quorum, and a succeeded ballot has been (affirmatively) voted upon by a quorum. One was required to prove the third invariant above; in other words, our system proposed a true invariant that could not be immediately proved, but that nonetheless indicated how to proceed with the proof. The other invariant was required to prove the simulation relation.

Given these six invariants, using LP to prove them and the simulation relation was mechanical and straightforward. We have not yet applied Isabelle to this

[1] We suspect that when we become more adept at using Isabelle and start to control its actions at a finer level, this latter step will no longer be necessary

algorithm, but based on our other experience, we believe that Isabelle extended by relatively simple tactics like those described in Section 3.3 would be able to complete this proof automatically.

The proposed invariants were not complete: they required additions by hand. However, 4 of the 6 necessary invariants were automatically proposed. Even though human effort was required (both to select the 4 used invariants and to add 2 more), it was less than would have been required to perform the entire proof by hand. Thus, we consider the use of our tools and methodology a success in this case.

A hand proof of the simula5tion relation [LS02] was also incomplete: it mentioned four of the invariants but omitted the other two. Our use of a theorem-prover enabled us to avoid this problem — and we even detected one[[**is this right?**]] of the two missing invariants automatically!

## 5 Related work

[[**Cite the work of the members of the program committee. Compress to under a page in length.**]]

Many researchers have tried to reduce the difficulty of formal verification. Model checking is exhaustive search of all reachable configurations, combined with a check of each configuration; model-checking implementations incorporate sophisticated optimizations. No human guidance is required during the search, and counterexamples to failed properties are provided. In order to control the number of states, users may need to specify a model that abstracts the actual system. Model checking is most applicable to finite-state systems that can be exhaustively checked[[**but see citations for automatic abstraction**]]. By contrast, our approach works for systems with unbounded variables or number of processes, does not require abstraction, does not require users to provide the goal properties to be verified, and provides intuition regarding true claims.

Theorem provers manipulate logical formulae rather than states. Fully automated theorem-provers require no human guidance and can be used by systems that must determine the consistency of a set of logical constraints. Such theorem-provers use a small set of pre-defined techniques for proving their goals, and they fail when those techniques do not work. Additionally, they tend to use fairly restricted logics, in order to restrict the complexity of the search.

Our work follows the tradition of interactive theorem provers, or proof assistants. A human selects a logical rule (such as simplification, case analysis, or induction) and the proof assistant applies the rule. The proof assistant ensures that the rules together form a valid proof. The scope of the proofs is greater than that of automatic theorem provers or model checkers. Proof assistants require a high level of skilled user interaction, both to select lemmas that determine the outline of the proof and to prove each of those lemmas in turn. (Some theorem-provers already contain tactics that let users ignore some of the latter bookkeeping details.) We aim to reduce both varieties of interaction in order to make proof assistants more practical.

For over a decade, researchers have investigated integrating model checkers with theorem provers in order to obtain the benefits of both exhaustive search and deductive techniques [MAB+94,ORS92,RSS95,BBC+96,Hun93,Sha00]. Our research suggests a new way to combine the techniques: generate an exhaustive test suite and use the properties that it produces, which are presumably less unsound and more comprehensive, as the theorem-prover inputs.

Several other researchers have proposed generalizing from program executions to properties true on those executions [CW98,ABL02,RKS02,HL02]. None of these tools produces output in first order logic suitable for verification by a theorem prover. It is also possible to generalize (either soundly or heuristically) from model checking; two recent examples are [PRZ01,APR+01]; by contrast, our technique generalizes to infinite-state systems and is aimed at more general properties.

Nimmer [NE01,NE02] verified dynamically detected program properties with the assistance of a specification checking tool, ¡¡ESC/Java [FLL+02], that internally uses an automated theorem prover. By contrast, our research addresses arbitrary properties including program correctness, not just absence of run-time errors such as array bounds overruns and null pointer dereferences; we address distributed algorithms in IOA rather than single-threaded Java programs; we use a more expressive logic, handle more sophisticated data structures, integrate with a more powerful theorem prover, concern ourselves with the structure of the proof, and automate routine theorem-prover interaction.

We previously advocated the use of simulated execution in order to assist theorem proving [NWEG+03]. The current paper gives details about the tools, which are also improved over the previous research; presents several new case studies; uses two different theorem provers; presents automatically generated tactics; and compares the automatically-assisted proofs to hand proofs.

# 6 Conclusion

## 6.1 Contributions

Even though IOA specifies distributed algorithms non deterministically, often with complex data structures, we were able to present execution data to Daikon's simple structures through the use of scheduling and data transforms. We also used static analysis of the IOA program to find predicates as hints to Daikon to find conditional invariants. When we took Daikon's simple output invariants and expressed them back in terms of IOA's data structures, it was sufficient to make significant progress in formally proving program specifications in a prover. We helped this substantive output of lemmas through tactics and a proof structure tuned for the I/O automaton model.

## 6.2 Future work

**[[Downplay this, or intersperse it through the paper.]]**

The methodology we have shown here is a promising first step for producing computer-checkable proofs that require fewer human interventions. We hope to improve upon this in two major ways. First, we are starting to use the Isabelle system, which allows for more precise control over its prover than LP, so we can have more control over the mechanical generation of proofs scripts. Our present Isabelle tactic of three lines suffices for simple properties, but we hope to provide more proof support. Second, we are developing methods using shrinking fixpoints [Rin00] both to prune false invariants and to show which ones were essential to a proof.

## Acknowledgments

Nancy Lynch provided feedback on our ideas and presentation and suggested the Paxos case study. 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.

[BBC+96] Nikolaj Bjorner, Anca Browne, Eddie Chang, Michael Colon, Arjun Kapur, Zohar Manna, Henny Sipma, and Tomas Uribe. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *8th International Conference on Computer-Aided Verification*, pages 415–418, jul 1996.

[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 *FSE '98, Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 35–45, Lake Buena Vista, FL, USA, 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.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245, Berlin, Germany, 2002.

[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.

[Hun93] Hardi Hungar. Combining model checking and theorem proving to verify parallel processes. In *Computer Aided Verification, CAV' 93*, pages 154–165, 1993.

[KCD+02] Dilsun Kırlı Kaynar, 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 *CONCUR 2002 Tools Day*, Brno, Czech Republic, August 24, 2002.

[LS02] Nancy Lynch and Alex Shvartsman. Paxos made even simpler (and formal). In preparation, 2002.

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

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

[MAB+94] Zohar Manna, Anuchit Anuchitanukul, Nikolaj Bjorner, Anca Browne, Edward Chang, Michael Colon, Luca de Alfaro, Harish Devarajan, Henny Sipma, and Tomas Uribe. STeP: The stanford temporal prover. Technical Report STAN-CS-TR-94-1518, Computer Science Department, Stanford University, June 1994.

[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.

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

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[NWEG+03] Toh Ne Win, Michael D. Ernst, Stephen J. Garland, Dilsun Kırlı, and Nancy Lynch. Using simulated execution in verifying distributed algorithms. In *VMCAI'03, Fourth International Conference on Verification, Model Checking and Abstract Interpretation*, New York, New York, January 9–11, 2003.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction (CADE-11)*, volume 607, pages 748–752, Saratoga Springs, NY, June 1992.

[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 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.

[RSS95] S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking with automated proof checking. In *Proceedings of the 7th International Conference On Computer Aided Verification*, pages 84–97, Liege, Belgium, 1995.

[Sha00] Natarajan Shankar. Combining theorem proving and model checking through symbolic analysis. In *CONCUR 2000: Concurrency Theory*, State College, PA, August 2000.

**[[Steve: Shorten the bibliography. Do not reference technical memoranda when there is a subsequent paper. Add a reference to Isabelle/IOA. Use a format that prints initials rather than first names.]]**

## Toh's notes on theorem proving with Daikon

Maybe it's better if I present this part as a problem-solution approach.

**Complexity**

Problem: proofs can become complex and hard to manage, both for the interactive user and for the theorem prover, which has to search through a larger space of lemmas to instantiate/unify.

Solution: divide and conquer. But: how much of this is just good theorem proving practice and how much of it is related to dynamic analysis? On the other hand, maybe the paper's about the whole system, where the Toolkit helps structure proofs in the way written here.

The best way both to mechanize and to automate theorem proving is to structure one's proofs so they can be sub-divided. This works at many levels:

- Invariants and simulation relations should be separated into as many pieces as possible. For example, if we have $A \rightarrow F \wedge G$ then this should be written as two invariants, $A \rightarrow G$ and $A \rightarrow F$. Example: in Paxos case study, we split a biconditional in the simulation relation into two implications. This is essentially a kind of proof tactic (done by the user), but it's justified for two reasons: a) it helps the prover by directing it towards steps that can make progress b) simple rules can be used to perform this tactic by simply analyzing the syntax of program properties.
- Proofs of each automaton transition should be separated as much as possible. In particular, there are two ways to provide witness executions for simulation relations: a) create a monolithic function that predicates on each state and transition involved; b) do case analysis for each transition, and for each transition provide a separate execution. The latter is preferable because it keeps each witness execution small and manageable in the prover. Essentially, this is using the syntax of the prover language to structure the witness execution rather than the syntax of the logic language. This is always a good idea because we've translated the meaning of the proof logic into the control language of the prover. So although the proof of the strong cache we did was similar to the one by Andrej, we used the second method. Another benefit of the second method is that paired simulation code directly corresponds to these structured witness executions (see Paxos).
-

### 6.3 Additional, probably obsolete stuff

Question: Theorem-proving is easiest on a declarative system, not one that is procedural. How can we make the expression of the algorithm (the implementation; the automaton) more operational, which is necessary in order to use our new tools, without harming theorem-proving?

One focus of our current work is an attempt to reduce the amount of user interaction required in proofs about I/O automata. We hope to do this (1) by combining theorem proving with dynamic program analysis and (2) by developing application-specific proof tactics for a theorem prover like Isabelle.

For (1), we use Daikon, an dynamic analysis tool maintained by Mike Ernst and his students here at LCS, to examine simulated executions of I/O automata

and to propose likely invariants that hold over those executions. The proposed invariants come in several flavors. Some are simply not true (although Daikon is remarkably good at filtering out bad proposals). Some indicate unexpected behavior, perhaps indicative of a bug in the automaton. And some express facts that are obvious to humans, but are nonetheless useful as lemmas in proving less obvious properties. This last category is of greatest interest.

For (2), we use a theorem prover to verify whichever of the proposed invariants we can, thereby filtering out the invalid proposals. These proofs are very stylized; hence some fairly standard tactics may suffice for their proof. Then we use the verified invariants as lemmas for proofs of humanly asserted invariants or simulation relations. Our hope is that by factoring a proof into two components, one for some relatively simple automatically suggested invariants and one for a theorem that depends on simple invariants as lemmas, we can substantially reduce (or even eliminate) the need for human guidance in both components of the proof.

**[[Even if you can't get all the way there, you learn something that is guaranteed to be true.]]**

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

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 **??**.

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.

### Ideas and syntax for overcoming Daikon impedance mismatch

Problem: It might not at first appear that Daikon detected the properties desired.
(this is what Mike and Toh talked about on Monday)

Solution: even though Daikon outputs in IOA syntax, its way of processing data is different from the way people would succinctly write algorithm properties in proofs and in IOA. We developed ways of exchanging data between the three systems (simulator, Daikon, LP) that let us get the invariants we need. The point: we can use Daikon's simple and efficient engine to analyze complex data structures and properties.

Daikon looks at simple mathematical and numerical properties like less than, equality, inequality, element membership, subset. Further, Daikon works only on scalars and sequences, not more advanced data structures like maps or tuples. The properties in IOA often are not that simple (though many are, as in Paxos's invariants).

Maybe explain why Daikon doesn't do maps? The search space on them would be really bad.

First, they are usually written in a universally quantified expression - Daikon detects properties about particular variables, and its language doesn't naturally have quantified expressions as standard invariants. For example: $\forall_{i,j} ops[i].location = ops[j].location$. But really, what this means here is that the location fields of the *ops* array are identical. So to get this invariant, the simulator gives Daikon a vector that's the location fields, and Daikon detects that all elements are the same. We then reconvert the syntax into IOA format so it represents a property of the whole *ops* array of tuples.

Second, the expressions within the quantifiers can be implications, usually about the same data structure on both sides of the implication. For example: $\forall_{i,j} ops[i].location = ops[j].location \rightarrow ops[i].seqno \neq ops[j].seqno$. The right hand side by itself is not true, but were it true, Daikon would be able to find it simply by detecting that the vector has no duplicates. But the left hand side of the implication at first looks like it will require a complex checking of each $i$ and $j$, with Daikon having to explicitly realize that it's a quantifier. However, we get around this problem by creating "derived" variables: syntactically, we know $ops[].location$ and $ops[].seqno$ come from the same structure (that's too complex for Daikon to want to look at). So we create a derived variable called the "join" of the two arrays that is an array: $join(ops.location, ops.seqno)[i] =< ops.location[i], ops.seqno[i] >$. But remember, Daikon doesn't do tuples, and we don't need their complexity, so we simply use hashcodes. Now, Daikon can detect that this joint data structure contains no duplicates.

Third, the *ops* data structure shown above is actually not even an array indexed by integers. Many arrays in IOA are actually data mappings with non-integral (and non enumerable) domain types. *ops* is actually a mapping from nodes to bank records. We get around this problem by giving what we know to Daikon: we split each map into tranches of arrays. Daikon is given a hint by the simulator to avoid numerical analysis (such as indexing or sorting) on these fake arrays.

Fourth, some properties specifically index a map data structure using a variable of the automaton. For example, in Peterson, $pc[p] = trying2 => pc[turn] = trying2$ ($pc$ is a map from nodes to program counters). Daikon cannot simply look at the varible $p$ and the $pc$ fake array. So the simulator gives Daikon derived variables like $pc[p]$ using a simple rule: index a map with all variables in context so Daikon can examine them.

Fifth, some data types in IOA are pointed, and are often used as the range to a map. A property in Paxos, for example, is $\forall_n val[n] \neq \perp \rightarrow val[n] \subseteq proposed$. We get around the quantifier problem using the method mentioned in the first point above. But now, the right hand side of the implication is not true in general (because there are no nil values in *proposed*). To get around this, the simulator gives Daikon two views of *val* - the range with nil values, and the range without.

Lastly, note that all the above methods are orthogonal and recursive. Thus, if the non-nil elements of *val* were tuples, we'd do a join operation on them.

### 6.4 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.

## 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.

**[[Steve: If we still want something like the following paragraph, we need to revise it in light of our latest work with Isabelle.]]**

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.

**[[Steve: The next comments are mildly interesting, if we have space to talk about future work.]]**

A second variety of missing automation is elimination of invariants that are obvious from the structure of the IOA program, as discussed in Section 4.1. 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 6.4, 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.

**[[Steve: The next comment is also interesting, if we have space to talk about future work.]]**

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.

**[[Steve: Does this last paragraph suggest any enhancement?]]**

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.