Theorem-Proving Distributed Algorithms with Dynamic Analysis

by

Toh Ne Win

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2003, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Theorem provers are notoriously hard to use because of the amount of human interaction they require, but they are important tools that can verify infinite state distributed systems. We present a method to make theorem-proving safety properties of distributed algorithms more productive by reducing human intervention. We model the algorithms as I/O automata, render the automata executable, and analyze the test executions with dynamic invariant detection. The human work in using a theorem prover is reduced because our technique provides two forms of assistance: lemmas generated by the dynamic invariant detection for use in the prover; and prover scripts, or tactics, generated from our experience with the I/O automaton model and the knowledge embedded in the test suite used for execution. We test our technique on three case studies: the Peterson 2-process mutual exclusion algorithm, a strong caching implementation of shared memory, and Lamport's Paxos algorithm for distributed consensus.

In the development and implementation of our method, we also improved the tools for formal verification of I/O automata and for dynamic invariant detection. We describe a new model for specifying I/O automata in the Isabelle theorem prover's logic, and prove the soundness of a technique for verifying invariants in this model in the Isabelle prover. We develop methods for generating proofs of I/O automata for two theorem provers, the Larch Prover and Isabelle/HOL. We show methods for executing I/O automata for testing, by allowing the execution of some automata defined with universal and existential quantifiers that were previously non-executable. Lastly, we present improvements to dynamic invariant detection in order to make it more scalable — in particular, we show how to achieve efficient incremental dynamic invariant detection, where the detection tool is only allowed to make one pass over its input executions.

Thesis Supervisor: Michael D. Ernst
Title: Assistant Professor

Thesis Supervisor: Stephen J. Garland
Title: Principal Research Scientist

Thesis Supervisor: Nancy A. Lynch
Title: Professor

Contents

List of Figures

Chapter 1

Introduction

As we increasingly rely on computers in our lives, the correctness of software systems on these computers becomes crucial. The problem of demonstrating the correctness of software is known as *software verification*. Verifying software is important because it gives us confidence that our systems perform as designed and do not behave harmfully. The most difficult software systems to verify are distributed, or concurrent, ones: these combine the complexity of traditional, centralized software systems with the nondeterminism from processes interleaving their executions.

There are many desirable attributes in software verification techniques. The technique's correctness guarantee should be as mathematically formal as possible, so that its statement of correctness is precise and the system can be discussed using the standard tools of mathematics. Yet, the verification should not require extreme amounts of effort from the programmer. The verification should also provide intuition regarding why the software system is incorrect. Lastly, errors should be caught as early as possible in the design process in order to save programmer time and minimize harm to the users who rely on the system.

Program analysis methods for verification can be classified into two categories: static and dynamic [JR00]. Static methods analyze a program without executing it. A sound, conservative static method produces formal guarantees because its results hold for all executions. However, verifying general correctness properties (i.e., those that go beyond relatively simple properties like type correctness) for all executions of a program is undecidable. A static method that attempts to verify such general properties either restricts itself to particular classes of programs or requires large amounts of human interaction. Thus, verifying correctness with static methods is an expensive operation in terms of programmer time for any complex system.

In contrast, dynamic analysis methods work operationally — that is, by examining executions. For a program with a large or infinite set of executions, a dynamic analysis examines only a subset of the executions. The technique is unsound if it attempts to generalize for all executions. Nevertheless, dynamic analysis is useful because it requires comparatively little programmer time. The programmer merely needs to execute the program on some test suite, and pass the execution data to the dynamic analysis. Because of this low time cost to programmers, dynamic analyses, such as testing, have proven useful in finding errors quickly and early in development.

The primary purpose of this thesis is to present a methodology for verifying the correctness of distributed algorithms that combines the strengths of static and dynamic verification techniques. Our method refines and integrates two formerly disjoint but useful techniques. The first technique formally models distributed algorithms as I/O automata [LT89] and proves them correct using a *theorem prover*, a general-purpose static verification tool. The second technique, called *dynamic invariant detection* [ECGN01b], takes in execution traces of a program and conjectures possible properties about the program in general. Our methodology works as follows:

1. Formally model the algorithm as an I/O automaton. The I/O automaton model is a mathematical way to describe distributed algorithms as state machines whose actions are labeled.

2. Build a test suite, test the I/O automaton by execution, and fix any errors that arise.

3. Analyze the executions over the test suite using dynamic invariant detection to produce a set of candidate invariants that are likely to be true for the automaton.

4. Prove the algorithm correct using a theorem prover. This is where the actual correctness proof happens. Traditionally, programmers have found a theorem prover difficult to use because it is not automatic and requires human input. We make using a theorem prover easier by providing automatically-generated input to the prover that can reduce the human input required. We do this in two ways. First, we develop a set of tactics, or proof strategy programs, that can be used on all structured inductive proofs of I/O automata. Second, some or all lemmas that are needed for proofs are provided by dynamic invariant detection rather than by the programmer.

Steps 1, 2 and 4 require some human interaction, while step 3 is automatic. Part of minimizing programmer effort in verification is reducing the human input in the process. We describe the trade offs we make on this issue when we describe our methodology in detail.

We do not attempt verify all types of program properties, and focus on *safety* properties of distributed systems. Safety properties ensure that incorrect or damaging behavior never occurs. We focus on safety properties because they are the most important ones for ensuring correctness.

In terms of tools to implement our method, we can currently use two theorem provers and one dynamic invariant detector. Our method is not fundamentally limited to these tools, but these are the ones we have spent effort developing in order to support the method. The Isabelle/HOL theorem prover is a supported tool because of the breadth of its higher order logic [Pau93]. The Larch Prover [GG91] is supported because of its already-tight integration with the I/O automaton model [Bog00]. On the dynamic side, our method uses the Daikon tool [ECGN01b] for dynamic invariant detection.

To more clearly introduce our methodology, we first present background on formal modeling of distributed algorithms and the issues of software verification. Thus, Chapter 2 introduces the I/O automaton model; the two major types of general-purpose static verification tools, model checkers and theorem provers, and their capabilities; a basic overview of dynamic invariant detection; and the types of program properties that we shall attempt to verify. With this background, we describe our methodology in Chapter 3. Chapter 4 shows our method in operation on three case studies: Peterson's two-process mutual exclusion algorithm, a strong caching implementation of shared memory; and Lamport's Paxos protocol for distributed consensus.

In order to implement our method, it was necessary to enhance pre-existing tools for verification and to develop ways to describe the formal mathematical models we have in the language of a computer theorem prover. Chapter 5 shows how dynamic invariant detection can be enhanced to make it more scalable for larger and longer-running programs. The primary focus is describing an incremental algorithm for analyzing executions — one that does not use up space as the execution length increases. Chapter 6 shows enhancements to the IOA Simulator (interpreter) in to allow it to execute more programs, so that these programs can be analyzed via dynamic invariant detection. Chapter 7 describes how a new model was developed for I/O automata in the Isabelle theorem prover's language, to increase automation in the verification process, and reduce human involvement.

Finally, Chapter 8 suggests further research and concludes.

1.1 Acknowledgments

I would like to thank Michael Ernst, my primary research supervisor, for guiding every aspect of my work over the last two years, and for providing me encouragement and inspiration during the most difficult times. I would also like to thank Stephen Garland for being *the* source of advice on formal modeling of I/O automata in theorem provers and on using theorem provers in general. I am also grateful to Nancy Lynch for teaching me the fundamentals of distributed algorithms and formal methods, and for suggesting examples and ways to move forward in proofs with her immense experience with the field.

I would also like to thank Laura Dean, Jeremy Nimmer, Josh Tauber, and Michael Tsai, who taught me how to code effectively in a team and corrected and initiated or refined many implementation ideas presented in this thesis. I also express my gratitude to Dilsun KırlıKaynar for helping organize our joint work on this methodology.

1.1.1 Joint work

The improvements to dynamic invariant detection were implemented after extensive discussion with Michael Ernst. Jeremy Nimmer and Michael Ernst developed the idea of hierarchically ordering program points for dynamic invariant detection and implemented the initial incremental version of the Daikon tool, without the optimizations discussed in this thesis. Nii Doodoo initially implemented the handling of conditional invariants in the Daikon tool.

Andrej Bogdanov originally developed the IOA translator to the Larch Prover, while Stephen Garland and Chris Luhrs developed the original Isabelle translator. Bogdanov also wrote the code for the strong caching memory automata in one of the case studies.

Nicole Immorlica was a collaborator in the proof of the Paxos protocol in the Larch Prover. Roberto DePrisco, Nancy Lynch and Alex Shvartsman formulated the original I/O automaton code for Paxos.

Antonio Ramirez-Robredo formulated the original idea of having annotations to IOA programs in order to describe correspondences in execution between two automata.

Chapter 2

Preliminaries and Background

This chapter presents a background on software verification and formal modeling of distributed algorithms in order to better understand our methodology for verification. We discuss the strengths and weaknesses of the two main general static verification techniques, model checking and theorem proving in Sections 2.1 and 2.2. In examining an analysis technique, we are interested in three features: the degree of its automation, the expressivity or limits of its logic, and the variety of programs it can examine. We introduce the concept of dynamic invariant detection in Section 2.3.

In Section 2.4 we introduce the I/O automaton model for distributed systems, the mathematical basis for our work, and formally express our verification goals in this model in Section 2.5. We describe alternative, related approaches to verification in Section 2.7.

2.1 Theorem provers

A theorem prover is a tool that manipulates mathematical facts given to it in order to prove more facts. Provers use logics of varying expressivity. For example, the Larch Prover (LP) uses multi-sorted (i.e., explicitly data typed) first order logic, while the Isabelle/HOL system uses higher order logic with ML syntax and typing. If an algorithm and its computation model can be expressed in the logic of a prover, then a user can use the prover to verify properties of the algorithm.

In order to do this, the user has to define for the prover a model of computation and the behavior of the algorithm. Then the user states the proof goals, or logical predicates that he wishes to show. The prover then soundly manipulates mathematical facts implied by the definitions in order to prove the goal. The manipulations are called *proof methods* and can be chained into higher level manipulations called *proof tactics*. For example, a proof method in Isabelle/LP is proof by assumption/implication. When the proof goal is of the form $A \to B$, the prover can be asked to assume that A holds, and to let B be the new proof goal. Once B is proven, the prover pops its *proof stack* and states that $A \to B$ is true. A simple proof tactic might specify that the implication method is applied whenever A and B have common free variables. The decision to apply proof methods and tactics can be made manually, automatically, or both, depending on the prover.

A sequence of proof methods and tactics that proves a goal is a *proof script* or simply a proof. A prover that supports requires external input at every step is still useful for it can be a tool for developing and checking the correctness of a proof script generated by hand or by another tool.

Since even first order logic is undecidable, provers report failure both when the goal is untrue, and when they cannot prove the goal. If the goal is indeed true at this point, humans have to step in and direct the prover somehow. This is why, traditionally, theorem provers have been considered difficult to use by everyday programmers for software verification: they are not automatic and require humans to provide much of the insight in proofs. Even the most automatic theorem provers only automate small steps using tactics.

In this thesis, we divide the human input into two varieties: lemmas and tactics. Tactics tell the prover to use pre-existing facts and proof methods to make progress on a proof. Examples include: simplification, case analysis, assumption/implication and induction. They are searches on facts known to the prover, accompanied by logical unification mechanisms. Some provers have built-in tactics that automatically attempt to apply themselves on proofs.

Lemmas add new facts to the prover's knowledge base. Before proving a goal, for example, it may be necessary to state and prove a lemma. For example, lemma A may be needed to prove fact B. Or, to prove $A \to B$, it may be necessary to show $A \to C$ and $C \to B$. The search for an appropriate C is a lemma input. In Section 2.5 we show how lemmas are used in proofs of distributed algorithms. Finding the right lemmas is much more difficult for a computer than finding the right tactics, for it is necessary to create facts that do not yet exist, rather than merely search on the prover's knowledge base. Theoretically, a prover could also find these facts by searching on all possible syntactical combinations, but this search space is enormous, and each attempt at lemma would have to be verified by a series of tactics, or worse, more lemmas.

The success of using a theorem prover depends on how much of the above human input can be eliminated. The key to our methodology in Section 3 is the reduction of this human input. The next two sections describe in greater detail the two provers we shall use in our method.

The Larch Prover

The Larch Prover [GG91] (LP) is an interactive theorem proving system for first-order logic. It admits specifications of theories in the Larch Shared Language [GHG+93] (LSL). It is strongly data typed, with declared types, and its type system permits polymorphic types, such as `Set[Int]`, a set of integers. It supports many intuitive proof methods, such as proof by implication and contradiction. However, it does not support the creation of new proof methods, only the creation of new lemmas which can be applied using the built-in proof methods.

As an example, we show a proof of the syllogism rule, $(a \Rightarrow b) \wedge (b \Rightarrow c) \Rightarrow (a \Rightarrow c)$, in LP in Figure 2-1. We direct LP to perform an exhaustive search on the values of a and b, via case analysis, and LP succeeds.

The underlying data structures of the IOA language (described in Section 2.6) are based on LP data libraries, so translating between the two domains is relatively straightforward. Further, this means that any data structures used in IOA already have LSL specifications

```
% Declare variables and types
declare variable a, b, c : Bool

% The goal
prove (a ⇒ b) ∧ (b ⇒ c) ⇒ (a ⇒ c)

% Proof method: by cases
resume by cases a = True, a = False

% LP says:
% Creating subgoals for proof by cases
% New constant: ac
% Case hypotheses:
%   userCaseHyp.1.1: ac
%   userCaseHyp.1.2: ¬ac
% Same subgoal for all cases:
%   (ac ⇒ b) ∧ (b ⇒ c) ⇒ (ac ⇒ c)

% Again follow by cases
resume by cases b = True, b = False

% LP says:
%% Creating subgoals for proof by cases
%% New constant: bc
%% Case hypotheses:
%%   userCaseHyp.2.1: bc
%%   userCaseHyp.2.2: ¬bc
%% Same subgoal for all cases:
%%   bc ∧ (bc ⇒ c) ⇒ c

%% Attempting to prove level 3 subgoal for case 1 (out of 2)

%% Added hypothesis userCaseHyp.2.1 to the system.

%% Level 3 subgoal for case 1 (out of 2)
%% [] Proved by normalization.

%% Attempting to prove level 3 subgoal for case 2 (out of 2)

%% Added hypothesis userCaseHyp.2.2 to the system.

%% Level 3 subgoal for case 2 (out of 2)
%% [] Proved by normalization.

%% Level 2 subgoal for case 1 (out of 2)
%% [] Proved by cases b.

...

%% Conjecture user.2: (a ⇒ b) ∧ (b ⇒ c) ⇒ (a ⇒ c)
%% [] Proved by cases a.
```

Figure 2-1: A proof of syllogism in LP.

```
theorem syllogism: "a ⟹ b & b ⟹ c ⟹ a ⟹ c"
(* goal (theorem (syllogism), 1 subgoal): *)
(* [| a; b & b; c; a |] ⟹ c *)
(*  1. [| a; b & b; c; a |] ⟹ c *)

  apply (cases "a")

(* goal (theorem (syllogism), 2 subgoals): *)
(* [| a; b & b; c; a |] ⟹ c *)
(*  1. [| a; b & b; c; a; a |] ⟹ c *)
(*  2. [| a; b & b; c; a; ¬ a |] ⟹ c *)

  apply (cases "b")

(* goal (theorem (syllogism), 3 subgoals): *)
(* [| a; b & b; c; a |] ⟹ c *)
(*  1. [| a; b & b; c; a; a; b |] ⟹ c *)
(*  2. [| a; b & b; c; a; a; ¬ b |] ⟹ c *)
(*  3. [| a; b & b; c; a; ¬ a |] ⟹ c *)

  apply (simp_all)

(* goal (theorem (syllogism)): *)
(* [| a; b & b; c; a |] ⟹ c *)
(* No subgoals! *)
done
```

Figure 2-2: A proof of syllogism by cases in Isabelle.

generated for the prover.

The Isabelle/HOL system

The Isabelle/HOL system [Pau93, Gor89] is a combination of two parts. The Isabelle system is an interactive theorem prover that verifies logical statements given to it. Isabelle can operate with any given logic, when the logic is specified in its "meta-language". The logic we chose to use for our methodology is HOL, or higher-order logic. Unlike first-order logic, HOL allows functions to be first-class values — i.e., values that can be passed as parameters to functions and returned. HOL also allows quantification on functions. The advantage of using higher-order logic is twofold. First, the prover can be used to prove facts about functions. For example, we can show that a given predicate is an invariant. Second, HOL can be used to prove *meta-theory*, or theory about proof methods. This is useful for verification because it proves that the methods used to verify systems are themselves sound.

Isabelle's syntax is an augmented version of ML. Logical formulae are stated in ML, using the language's type system (and syntax), but the syntax is augmented to allow for the stating of quantifiers. The commands to control the prover are written in very simple syntax. The ML logical formulae are written in quotes as arguments to these commands. A proof of syllogism in Isabelle is shown in Figure 2-2. At the end of the proof, we invoke the `simp_all` proof tactic, which invokes a built-in simplifier.

Isabelle/HOL supports the addition of new proof methods and tactics, both in its language and in ML, the implementation language of the prover. In the latter, Isabelle can be used as a standard programming language, so the user can potentially create powerful tactics.

Isabelle/HOL also has a relatively large user community, and its data libraries are more extensive. However, the Isabelle data structure libraries do not directly match the ones in IOA, so any IOA data structures need to be manually translated for Isabelle.

2.2 Model checkers

Model checkers are a possible alternative verification method. Model checkers examine the entire reachable state space of a program to determine if a property holds. They work in a variety of non temporal and propositional temporal logics [CGP99] and are fully automated provided the user knows the property to verify. Their main limit is that by having to examine all reachable states, their performance can suffer drastically for systems with large state spaces, and they do not work for infinite state systems if soundness is to be preserved. For concurrent systems, models checkers have to examine all interleavings of executions, and their runtime increases exponentially with the number of processes unless clever optimizations are done.

Model checker designers have come up with some ways to overcome the large/infinite state space problem. By using clever data structures such as binary decision diagrams, large state spaces can be checked quickly (log time) for most programs humans would write [CGP99]. However, there are programs for which this technique does not work. For an infinite state system, model checkers are used to check finite state *abstractions* of the system, but human work is involved in formulating and proving the concrete to abstract mapping. Section 2.7.1 discusses techniques related to model checking using this abstraction and how it relates to our work.

2.3 Dynamic invariant detection

The Daikon invariant detector [ECGN01a] proposes properties that are likely to be true throughout a program's execution. Daikon operates dynamically by examining values computed during execution and generalizing over those values. Its output is in the form of invariants over a grammar on the program's variables. Initially, Daikon conjectures that all properties in its grammar are true on the program. Then Daikon examines the execution data and deletes any invariants that are contradicted by the data. Finally, Daikon uses static analysis and statistical tests to reduce the number of false positives by eliminating some of the remaining invariants [ECGN00]. When used in conjunction with the IOA interpreter, Daikon expresses this formal specification in IOA as an invariant of the executed automaton.

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

Daikon produces output in the form of a formal specification that often matches what a human would have written [ECGN00, NE02b]. Even when Daikon was given inadequate test suites in order to artificially degrade its output, it still improved programmer performance (to a statistically significant degree) on a program verification task [NE02c].

Nimmer and Ernst [NE02b] have also used dynamic analysis from Daikon to help static analysis in the ESC/Java tool. Their primary goal was to extract and verify Java program specifications rather than verifying safety properties in first order logic about distributed algorithms. Groce et al. [Gro02] are attempting to use model checkers on programs to generate test output for dynamic invariant detection with the Daikon tool, rather than executing the programs themselves.

2.4 I/O automata

I/O automata [LT89] have been used to model a variety of distributed systems [GL00a]. I/O automata are (possibly infinite, nondeterministic) state machines in which transitions between states are associated with named *actions*. Actions are classified as either *input, output*, or *internal*. The inputs and outputs are external actions used for communication with the automaton's environment; internal actions are visible only to the automaton itself. An automaton controls which output and internal actions it performs, but input actions are not under its control. Actions can be parametrized. An I/O automaton consists of its *signature*, which lists its actions; a set of *states*, some of which are distinguished as start states; a *state-transition relation*, which contains triples of the form (state, action, state); and an optional set of *tasks*.

Action π is *enabled* in state s if there is a state s' such that (s, π, s') is a transition of the automaton. Input actions are enabled in every state. The operation of an I/O automaton is described by an execution $\pi = s_0, a_1, s_1, \ldots$, which is an alternating sequences of states and actions that is valid with respect to the automaton's transitions. The executions of an automaton are the set of all such sequences. A trace τ of an execution π is its projection such that states and internal actions are removed. The traces of an automaton determine, therefore, its external behavior.

Program properties or specifications are described formally in terms of automaton traces. A property is defined by the set of traces it admits. For example, property P might require an automaton to only fire a particular action $a(n)$ where n is an even integer. In this case, P is the set of all traces containing only even invocations of a. An automaton obeys a given property P if its traces are a subset of P.

2.5 Program properties

Every property of a program trace can be written as an intersection of a safety and a liveness property [AS87]. Informally, a safety property is one where "nothing bad happens", while a liveness property is one where "something good eventually happens", ensuring that the program makes progress during its execution.

Formally, a trace property (a set of traces, or alternatively, a predicate on traces) P is a safety property if:

- P contains the empty trace, and
- If β is in P, then any finite prefix $\beta' \leq \beta$ is in P.

If a trace does not satisfy a safety property, then we can find the exact place where the property was violated with the above definition.

Formally, P is a liveness property if:

- For any β there exists an extension β' of β such that β' is in P.

In our verification methodology, we do not attempt verify all types of program properties, but focus on safety properties of distributed algorithms that can be described either by *invariant assertions* or by *a forward simulation relations.* We do this because our tools allow these two demonstrations of safety to be verified, and because liveness properties are not as important for algorithm correctness.

The next two sections show how safety properties apply to I/O automata.

2.5.1 Forward simulation relation

The purpose of a forward simulation relation is to relate two automata, a *specification automaton* and an *implementation automaton*, and to show that the latter implements the behavior of the former. If the specification automaton is known to behave safely, then so will the implementation automaton, and the implementation will be verified.

Formally, an automaton A is defined to implement a specification automaton B if $traces(A) \subseteq traces(B)$. A forward simulation relation [LV95b] is sufficient (but not necessary) to show this implementation. A forward simulation relation f on the states of B and A satisfies the two following two conditions:

- For all $s \in starts(A)$, there exists $u \in starts(B)$ such that $f(s, u)$.
- For all $(s, a, s') \in trans(A)$ and for all u such that $f(s, u)$, there exists an execution $\beta \in execs(B)$ with final state u' such that: $trace(\beta) = trace(a)$ and $f(s', u')$.

For each execution α of A, the above requirement allows us to choose an execution β of B that starts and transitions properly. The *successive refinement* method uses multiple levels of automata are used to show that a concrete implementation implements a specification automaton.

2.5.2 Invariant assertion

An invariant assertion is simply a statement that the reachable states of an I/O automaton obey a given predicate. These assertions apply to states and not to actions. To prove that an automaton obeys an invariant I, it is necessary to show two things:

- For all $s \in starts(A)$, $I(s)$.
- For all $(s, a, s') \in trans(A)$, $I(s) \rightarrow I(s')$.

Invariants are used in three ways. First, it is sometimes convenient to specify program properties using an invariant rather than a specification automaton. Second, invariants may be needed to show a simulation relation — in the simulation proof, the invariant is used as a lemma. Third, invariants may be needed to prove other invariants.

We can formalize and prove many safety properties as implementation relations or invariants. In the next two sections, we examine possible methods to verify other property types.

2.5.3 Liveness

Liveness properties are used to show that a system makes eventual progress towards a goal. A liveness property is either stated explicitly in temporal logic, or relies on the underlying model to provide the temporal framework and is specified in terms of the model. For an example of the latter, the I/O automaton framework allows for a set of *task partitions*, or sets of actions such that a particular set of firings determine a *fair* execution. From these fair executions, which are simple liveness properties, the designer can argue more complex liveness properties.

Live I/O automata [SGSAL98] are a more general way to specify liveness. They allow arbitrary temporal formulae (specified as a set of acceptable traces) for liveness properties, with the only restriction being that the properties remain satisfiable for all inputs. Unfortunately, traditional simulation relations, which show that one automaton implements another, do not work for live I/O automata. This is because a simulation relation shows that each of the implementation automaton's traces are a subset of *all* the traces of the specification automaton, while the liveness requirements restrict the correct traces of the specification automaton to a specific subset.

However, Attie [Att99] has a way of specifying liveness that lends itself to formal verification by simulation relation. Intuitively, the live traces of an implementation will be a subset of the live traces of the specification. In his method, liveness properties must be specified as pairs of predicates $\langle A, B \rangle$ on the automaton state, such that the following temporal holds:

$\Box \Diamond (A \rightarrow \Box \Diamond B)$

This reads "infinitely often, it is true that an occurrence of the property A leads to an infinitely often occurrence of the property B". Thus, the temporal framework is encapsulated entirely in the model, and the designer only has to work with the first order logic of the predicates. With this special form of liveness, Attie has developed a formal theory of "liveness preserving simulation relations", an extension of forward simulation relations for liveness.

2.5.4 Other safety properties

Our method requires safety properties to either be invariant assertions or refinement mappings, so it may leave out a few safety properties, especially the ones relating to adjacent sets of states. For example, neither invariants nor simulation relations can directly say that an action α_1 is never followed by another action α_2. However, there are easy ways to rewrite all such safety definitions into either invariant assertions or refinements, with the addition of history state variables [LV95a]. In the case of this example, a flag could be set whenever α_1 is executed, and the invariant could be "α_2 is never enabled if $flag$ is on".

2.6 The IOA language

The IOA language [GL00a] provides notations for describing I/O automata and for stating their properties; it uses Larch Shared Language [GHG$^+$93] specifications to axiomatize the semantics of I/O automata and the data types used to describe algorithms. In IOA, transition relations are defined in terms of preconditions and effects. These can be written either in an imperative style (as a sequence of assignment, conditional, and loop statements), or in declarative style (as a predicate relating state variables in the pre- and post-states, transition parameters, and other nondeterministically chosen parameters). It is also possible to use a combination of these two styles.

Nondeterminism appears in IOA in two ways: *explicitly*, in the form of `choose` constructs in state variable initializations and the effects of the transition definitions, and *implicitly*, in the form of action scheduling uncertainty. Nondeterminism allows systems to be described in their most general forms and to be verified considering all possible behaviors without being tied to a particular implementation of a system design.

Tools in IOA include a checker for well-formedness of automata (e.g., input actions are always enabled), a Simulator (interpreter) [KCD$^+$02], and a translator from IOA to the theorem proving language of the Larch Prover [BGL02].

We use the I/O automaton model and the IOA language for our verification methodology for many reasons. First, the model has proven useful in describing many existing systems and algorithms [GL00a]. Second, we can use the existing tool that translates IOA [Bog00] into the first order language of the theorem prover LP, and adapt it to generate proof tactics. We can also retarget the tool to generate and the other provers. Many proofs of distributed algorithms have already been done with LP [BGL02]. There is also already a model for the semantics of I/O automata in Isabelle [Mül98] and Luhrs has created a design for a tool to translate IOA to Isabelle [Luh02]. Third, an IOA program is executable after some modifications and the executions can be used in dynamic analysis. With various enhancements, the IOA Simulator can produce data the is useful for the Daikon tool to examine [NE02a].

2.7 Related work

In the next two sections, we discuss methods developed to improve the range of systems that can be verified by a model checker. The first is abstraction, a means to map a large state space into a smaller state space for model checking. The second is a verification method for a restricted class of systems called parametrized systems.

2.7.1 Abstraction

Abstraction [CC77a] has been increasingly used by the model checking community [Pod03] in order to deal with both the (finite) state space explosion and the infinite state space problem. In this section, we examine the possible approaches and limitations to using abstraction and model checking for the I/O automaton model. We also describe our work in the terminology of abstract interpretation for comparison.

Abstraction is mapping a concrete state space to an abstract state space, a way to formalize and make sound the approximation of a model. model that may yield useful information. This mapping is most useful when it is from infinite to finite space, and the model checking on the abstract space produced verifies the concrete space. Given a finite property, there exists an abstraction that maps an infinite state space to a finite one that represents the property. However, it is undecidable to actually deduce the right abstraction, and the finite space might still be too large.

A predicate abstraction is a particular abstraction mapping concrete state space to boolean variables. It is equivalent to an invariant. The mapping function is thus predicates on concrete variables, like $x < y$. In theory, for any property, there exists an abstraction such that any model can be mapped to two states — i.e., a predicate abstraction can be sufficient for model checking, where one state is "good" and another is "bad" and we just show that "bad" is not reachable from the start states of an automaton. All transitions from "good" never go to "bad", and this is easy to check because the abstraction has done all the work [Pod03].

However, finding the right predicate abstraction is also undecidable and difficult for humans to do. A predicate I on the concrete state s must have the following three properties to be useful for verification:

- $\forall_{state} I(state) \Rightarrow good(state)$. I implies the property to be proved.
- $\forall_{state \in start} I(state)$. I holds on the start.
- $\forall_{state, state'} (I(state) \wedge transition(state, state') \Rightarrow I(state')$. I is "inductive". That is, if we know that I holds on a pre state, then I is sufficient to prove that I holds under any transition to a post state.

The key to successful verification with abstraction is to discover the inductive invariant I. Note that I is usually a conjunction of many invariants. Methods include "widening" where I is weakened at every step until it is inductive [CC92], or heuristics such as abstractions by "octagons" saying that the values of x and y are within an octagonal region when graphed [Min01]. In this sense, our methodology, which also attempts to find suitable invariants for verification, is not that different from the efforts of researchers of abstraction.

However, our work differs in the way we discover these invariants and how we verify that they imply the "good" property. We discover invariants through execution rather than a static methodology like widening. The grammar of our invariants is also wider. Presently, those who are practicing automatically generating abstractions for verification seem to be restricting themselves to arithmetic on integers [Pod03] and say nothing, for example, about subset inclusion on unbounded sets. In some cases, the useful abstractions must be generated by hand. Since our method uses a theorem prover rather than a model checker, it may require more manual intervention to verify that an invariant I obeys the three properties. However, we can make progress when the invariant is not quite sufficient for verification precisely because of the ability to manually intervene.

2.7.2 Verifying parameterized systems

A parametrized system is an unbounded array of finite state automata that only perform communication with their neighbors. A parametrized system is still an infinite state system, but its infinity is only in the number of participating processes.

One particular model checking technique, called "invisible invariants" by Pnueli, et al. [PRZ01], allows a restricted class of parametrized systems to be model checked for a certain class of properties. By model checking a certain (computable) number of these automata, the result holds for all the automata. For example, they model check 5 automata in a solution to the Dining Philosopher's Algorithm to show that the solution holds for an unbounded system.

Intuitively, the method restricts itself to properties that are local — related to a handful of automata interacting next to each other, rather than the global system. Further, the individual automata not only have to be finite state, but limited in function. The exact limit has not yet been formalized. An example may provide some intuition, however: the individual automata cannot act as cells on a Turing machine tape, whereby they simulate the Turing machine by passing messages back and forth. Otherwise, a verification of such a parametrized system would be able to tell if the Turing machine halts.

Further, many distributed algorithms have state variables that are unbounded, in addition to having an unbounded number of processes. For example, Lamport's Bakery algorithm [Lam74] requires each process to have an unbounded integer counter. For these systems, model checking has no answer so far. Thus we focus our work on theorem provers alone in terms of static verification tools.

Chapter 3

Methodology

This section describes in detail our intended method for verifying distributed algorithms using a combination of formal modeling, dynamic analysis and theorem proving. Recall that properties for I/O automata can be stated either as invariant assertions or as specification automata. We use a slightly different approach for properties stated as invariants versus properties stated via specification automata, but the first three steps are the same. Further, note that each invariant needed for verification by simulation relation can use the version of our method for invariants.

Our method, outlined in Figure 3-1, is summarized by four steps:

1. Model the algorithm as an I/O automaton.

2. Render the nondeterministic and declaratively specified I/O automaton into IOA programs that are imperatively executable. In the case of verification by simulation relation, we only need to rewrite the implementation automaton. Execute the automaton (automata) using a simple test strategy, and fix any errors that may arise, until what the programmer believes are correct executions are produced.

3. Analyze the executions using dynamic invariant detection to produce a set of invariants.

4. Translate all automata into the language of a theorem prover, and verify the safety properties, with assistance from: the Daikon invariants as lemmas; tactics and proof structure from the translation tool.

3.1 Specifying algorithms as I/O automata

The first step in using our method for verifying a system is to define it as an I/O automaton. This means that an algorithm should be converted into a state machine with each atomic step being a transition. Many algorithms have been modeled as I/O automata [PPG$^+$96, SAGG$^+$93a, GL00b] and we do not delve deeper into how the conversion happens. The final result is an I/O automaton that is a mathematical specification for an algorithm.

Now, the definition of safety properties can be done in two ways. Often, it is sufficient to simply state an invariant on behavior, such as with the Peterson case study used in

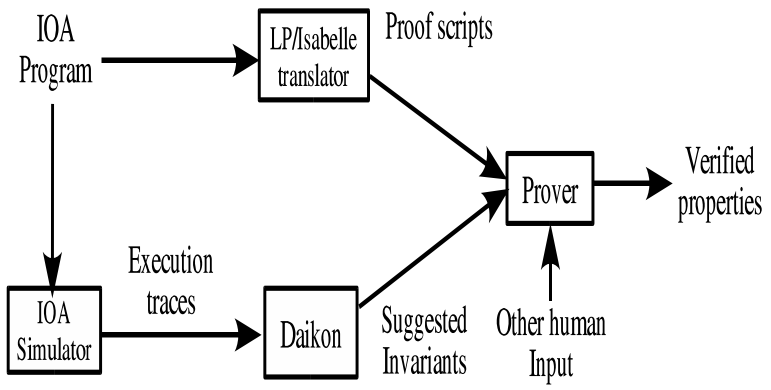

Figure 3-1: An overview of our method. We verify program properties with help from dynamic invariant detection and proof scripts generated by our IOA to prover translators.

Chapter 4.1. Verification by invariants is supported by our method. Otherwise, the user may prefer to state safety properties as another I/O automaton [LV95a]. In this case, the user should specify the correct behavior in terms of a specification automaton and verification shall happen by forward simulation, which is another method we support. The user also has to specify the forward simulation relation.

3.2 Executing automata

The second step in using our method to verify an automaton is to test its behavior through execution. The IOA Simulator simulates (on a single computer) execution of an I/O automaton, allowing the user to help select the executions and to propose invariants for the interpreter to check.

Converting I/O automata into executable IOA code involves two steps: writing the automata in the IOA language, and resolving nondeterminism by scheduling. Afterward, the IOA Simulator executes the program and writes execution data (the concrete representation of interleaved states and actions) to a file.

The former is necessary and relatively trivial: the IOA Toolkit supports a useful but limited set of data type libraries written in LSL, and the specified I/O automaton must be written in terms of these libraries. Other tools like the IOA Composer provide the convenience of having to write only one automaton and being able to form a system of interconnected automata using this automaton as a template.

The latter requires work: we must write *determiners* [Che98, RR00] that choose between nondeterministic steps an automaton can take. When verifying by simulation relation, this is only necessary for implementation automata. The primary place for this is in scheduling which transition fires next among those that are enabled. The IOA language allows us to append a small program, called a `schedule` block, at the end of the automaton definition to perform this scheduling. The `schedule` block not only selects which transition to perform using a language very similar to IOA[1] but also selects which parameters to supply to

[1]It is essentially the same semantics as the `eff` code of IOA, except it only allows imperative code.

the transition. Additionally, where the IOA language allows nondeterministic choice using `choose` terms within `eff` blocks, determiners have to be supplied to choose a specific choice. It is not necessary that a scheduled IOA program is purely deterministic - `schedule` blocks can use explicit calls to randomization functions.

Because transitions have parameters and because their preconditions are in first-order logic, it is generally undecidable to automatically schedule transitions. However, it is still useful to develop methods that either automatically schedule for some types of transitions, or ways to write `schedule` blocks that approximate fair (or other desired) scheduling techniques. A common schedule technique used in our case studies is as follows: pick a node at random, find which actions the node can take and perform one of these actions. Usually, a particular node's choice of actions is limited to one.

It is important to note why we want a schedule that exhibits all the interesting behavior of a program: executions are the source of data for dynamic invariant detection to examine. Although not all possible executions have to be seen (this would amount to a very slow form of model checking), it is desirable to create representative transition steps. The key insight here is that the `schedule` block can be seen as a test suite from the perspective of dynamic analysis.

3.3 Dynamic invariant detection

Verifying safety properties often depends on invariants and on auxiliary lemmas. Machine verification requires that such lemmas be stated and proved explicitly, even if they seem like bookkeeping details to the user. Hence, the third step in our method is to generate candidate invariants and lemmas automatically by using dynamic invariant detection to analyze execution data from the IOA Simulator.

After execution data is generated from the Simulator, it is given to the Daikon tool to perform this dynamic invariant detection. Tool support for this data conversion was initially developed by Dean and Santos [Dea00, NS01] and further extensions are shown in Section 6.3. Daikon generates conjectures (in first-order logic) that are valid IOA syntax in the form of invariants. These invariants are appended to the automaton, to be used in the next step of our method.

If any unusual behavior is seen by Daikon at this point, as with the Peterson case study in Section 4.1, it is an early opportunity to fix the algorithm or its implementation in IOA. This feedback loop of writing of code and dynamic analysis, often including the addition of more test suites, can happen until the user is satisfied that what is reported by the dynamic invariant detector is not unexpected behavior.

Three potential problems with this third step are that the lemmas it produces may be unsound, incomplete, or not very useful. Despite these potential pitfalls, this step tends to perform well in practice as shown by our case studies and others [NE02b, NE02c]. We discuss how to cope with the three potential problems.

Soundness Dynamic invariant detection is unsound: reported properties are true over the test suite, but there is no guarantee that the test suite fully characterizes the execution environment of the program. This does not hinder us, for two reasons. First, we use all of the dynamically detected invariants to help in proposing, understanding, and verifying program properties, but we use a theorem prover to ensure that the lemmas we use in proofs are sound. Second, most of the output in our case studies were correct, and those that were not were easily corrected artifacts of the test suite (execution scheduling). In general, simply covering every interesting aspect of each action seems to be adequate.

Completeness Dynamic invariant detection is incomplete: the proposed invariants may be insufficient for verification, because some true invariants are not reported. Daikon restricts the set of invariants it checks for two reasons: to conserve runtime and, more importantly, to reduce the number of false positives that it reports (the more properties it checks, the larger the number of false or non-useful properties it will report).

Sometimes, as in our Paxos case study, dynamic invariant detection may report an invariant that is not provable in isolation — another invariant may be necessary but not detected. In other words, dynamic invariant detection can postulate a useful property whose proof is complicated. This ability to decompose a proof into parts demonstrates a strength of our technique: it is easy to check properties dynamically, even if they have complicated proofs that are beyond the capabilities of completely automatic static tools.

Usefulness Third, some reported properties may be true but not useful. Daikon uses heuristics to prune useless facts, for instance, by limiting output based on variable types. However, it is impossible for a tool to know what a human will find desirable in a given situation. In practice, we find that it is easy for humans to select the useful invariants and to pass over the uninteresting ones — and examining them helped us solidify our understanding of the algorithm and the implementation. Thus, a moderate amount of extra information does not distract or disable users.

Similarly, the reported properties may be more than are really needed: a proof accepted by a theorem prover may use more invariants than are strictly necessary, thus obscuring the essential argument. We believe it is better to first obtain a working, machine-verified proof, and then to simplify it. Possibly automating this task (following Rintanen [Rin00] by iterating unto a minimal fix-point of invariants) is presented as future work in Section 8.1. We did not have to perform such a reduction in our case study. This finding accords with other user studies of runtime analysis [NE02c], where the output from dynamic invariant detection was able to help verify the absence of certain runtime errors.

3.4 Paired execution

The second step in our method, execution, is also appropriate when attempting to verify a simulation relation. As noted in Section 3.2, the IOA simulator can help users formulate and test the validity of a forward simulation relation, prior to such a verification. In this section, we discuss how a schedule and other information can help in executing paired automata, while the IOA simulator tests the conditions of the relation. This scheduling will later be useful in guiding verification.

A forward simulation relation is a predicate that relates the states of two automata (see the definition in Section 2.5.1). Paired execution requires such a correspondence to resolve nondeterminism. Usually, the designer of an implementation has an idea of the step correspondence. The IOA toolkit allows the designer to annotate the program with this correspondence.

The **proof** block in Figure 3-2 describes a step correspondence for use in testing a simulation relation. The code is from the Paxos case study in Section 4.3.6 but we examine only the syntax here. With a **proof** block, the paired simulator can execute the specification automaton in lockstep with the implementation automaton. The **proof** block contains two sub-blocks, corresponding to the two conditions required for a simulation relation (Section 2.5.1). The first sub-block, started by **initially**, shows how to start the specification automaton.[2] The second sub-block contains an entry for each action of the implementation automaton; this entry provides an algorithm for producing an execution fragment of the specification automaton. Syntactically, each entry uses **fire** statements to tell the simulator to fire specification actions. A **proof** block may also contain a third sub-block that declares auxiliary variables used by the step correspondence.

3.5 Proving properties

The last step in our method is to prove the simulation relation (or invariant) using a theorem prover. This guarantees the soundness of our technique. As described above, theorem provers generally require human input in the form of lemmas and proof tactics. Here we describe how the results of Sections 3.3 and 3.4 can be used to generate this input automatically. First, the invariants suggested by dynamic invariant detection become candidates lemmas, thereby saving the user time in finding auxiliary invariants needed for verification. Second, the annotations for paired execution provide a proof outline and tactics. In this section, we show how to generate the latter.

The outline and tactics are generated from modified versions of the proof translators for LP and Isabelle. The translators use the structure of the program; the annotations written by the user for execution; and our knowledge of commonalities in proofs of I/O automata to generate proof scripts. In practice, we have found that these proof scripts have saved verification time by automating many steps of a proof.

3.5.1 Proving invariants

Proofs of invariants in a theorem prover follow a structured methodology:

- Prove the start condition.
- Prove the transition condition by structural induction on the data type of the action and its parameters.

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

There are two tools that translate IOA to prover languages and somewhat automate providing this kind of assistance. The ioa2lsl tool [BGL02] converts IOA programs and asserted invariants into LSL, an input language for LP. An example of its output is:

```
prove Inv(s) ∧ isStep(s, a, s′) ⇒ Inv(s′)
```

The ioa2Isabelle [Luh02] tool provides similar assistance for reasoning about IOA using Isabelle [NS94, Mül98]. We have used a this tool successfully in case studies. An example of its output is:

```
lemma I_step:
  "∀ state act .
      ((reachable aut state) ∧
       (I state) ∧
       (enablement_of aut state act)
      )→
       I(effects_of aut state act)"
```

Now we describe the tactics that the Isabelle translator outputs for assisting proofs of invariants. The LP translator outputs relatively minimal tactics for invariants because we have less control over the prover than we do for Isabelle. Figure 3-3 shows the lines generated by the Isabelle tool for one invariant of the shared memory case study from Chapter 4.2. Again, the particular algorithm is not important. The general methodology we use is as follows:

1. Prove the start condition by citing specific tactics. This is relatively straightforward and so the tactic for this is omitted from the figure.

2. Split up the proof of the invariant into separate lemmas for each transition. Prove each lemma by citing a specific sequence of theorems about the automaton as lemmas. In the figure, the first paragraph shows the proof for one transition, respond, which assumes sCache is reachable and respond is enabled. The code attempts to prove that the invariant Inv holds for the post-state. The tactics generator cites the three premises, p0, p1, and p2. Then it uses the simp and auto? Isabelle tactics with specific arguments relating to the automaton definition, which were discovered to be effective in proofs.

3. Try to prove the invariant step condition by induction on the action data type and by citing the lemmas for each transition. This is shown in the second paragraph of the figure.

4. Prove the invariance of the predicate itself by citing the step condition and the start condition lemmas. Then prove that as a consequence of the predicate being an invariant, the invariant holds on all reachable states. This last part is useful so that other invariant proofs can take advantage of the current invariant holding should a state be reachable.

For many simple invariants, like in the memory case study, these automatically generated tactics require no human augmentation.

3.5.2 Proving a simulation relation

Recall from Section 2.5.1 that verifying a simulation relation requires verifying both a start condition and a step condition. Translation tools in the IOA toolkit use the **proof** block for a simulation relation to generate proof tactics for each condition. The `proof` block can be seen as a test suite that also contains the programmer's knowledge about how the specification automaton is supposed to behave — this is why it is useful in a formal proof, once properly translated. The `proof` block from the Paxos case study in Figure 3-2 will be used as a running example.

Start condition

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

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

the rules are similar for Isabelle, but wrapped up in a predicate:

```
theorem FCache2Mem_start:
  "isFwdSim_start Cache Mem FCache2Mem"
```

where the predicate is:

```
constdefs isFwdSim_start ::
  "('actionA, 'stateA) ioa ⇒ ('actionB, 'stateB) ioa ⇒
   ('stateA ⇒ 'stateB ⇒ bool) ⇒
    bool"

  "isFwdSim_start autA autB F ==
   (∀ a . a ∈ starts_of autA → (∃ b  . (b  ∈ starts_of autB) ∧
                                        (F a b)
                                 ))
  "
```

The definition of the predicate comes in two pieces. First, we define the data type of `isFwdSim_start`, then we define its meaning. The data type says that `isFwdSim_start` takes in three arguments: two automata (parametrized by their action type, as explained in Section 7) and a relation on their states, and returns a boolean. The definition is the standard start condition for simulation relations.

The proof translator tools extract the witness b from the imperative statements in the **initially** section of a **proof** block, which define initial values for the state variables in the specification automaton in terms of the initial values for the state variables in the implementation automaton. In the Paxos case study, the proof script generator translated the **initially** section of Figure 3-2 into the following commands for LP:

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

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

The commands are similar for Isabelle:

```
  "startRelGlobal12Cons sGlobal1 ==
   Cons_state.make (Global1.initiated sGlobal1)
                   (Global1.proposed sGlobal1)
                   {}
                   (Global1.decided sGlobal1)
                   (Global1.failed sGlobal1)"

theorem FGlobal12Cons_start:
  "isFwdSim_start Global1 Cons FGlobal12Cons"
...
  show "(startRelGlobal12Cons  sGlobal1) ∈ starts_of  Cons ∧
        FGlobal12Cons  sGlobal1 (startRelGlobal12Cons  sGlobal1)"
  apply (simp add: startRelGlobal12Cons_def  Cons_def
                   Cons_start_def  FGlobal12Cons_def
                   Cons_state.make_def)
```

The first line defines `startRel`. The second line proves that `startRel` satisfies the start requirement through the use of the `simp` tactic. The arguments to the tactic are the definitions of parts of the two automata. They were chosen from our experience with I/O automaton proofs and then coded into the Isabelle translator tool to be output for every automaton.

Step condition

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

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

A similar formulation is generated in Isabelle, which can be found in Appendix B. Both tools also generate proof scripts that divide the proof into cases based on the kind of the action `a`. In LP, this script uses the command `resume by induction on a`, which directs LP to perform structural induction on the datatype of `a`, while in Isabelle, the script uses the `induct` tactic. The tools generate lemmas to handle the details of the individual cases. For example, the LP translator generates the following lemma and proof script from the **proof** block for the `init` action in the Paxos case study.

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

The intelligence of this tactic lies in the line that says `resume by specializing....` Which beta to specialize to is derived from the `proof` block's `fire` statements in the corresponding `for` blocks. An analogous script is generated for Isabelle.

When **proof** blocks are more complicated than those in the Paxos case study, the job of the proof script generator is correspondingly more complicated. For example, the generator must expand a **for** entry in the **proof** block that contains a sequence of conditional statements such as

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

into one that contains nested conditionals such as

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

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

Our techniques do not completely eliminate the need for human guidance in proving invariants and simulation relations. Some transitions that have complex semantics may need the citation of specific invariants (that have already been proved). We do not yet have a methodology for choosing which invariants to cite. There may also be other case splits that are not mentioned in the `proof` block. We do not yet consider these splits.

Kırlı, et al. [KCD$^+$02] have also manually performed the above process in LP on Dijkstra's mutual exclusion algorithm. The difference with our method is that our tools now perform the translation automatically from the `proof` block.

3.6 Assessment

The main drawback of our method is that human intervention is still necessary in two important places. First, the right invariants need to be selected to use in verification. Second, when the prover cannot make process, the user has to enter tactics to complete proofs.

Nevertheless, our methodology satisfies five desirable properties for a verification method:

Achieving a formal proof of correctness: Our method uses I/O automata for formal modeling and a theorem prover, whose output is a logically verifiable proof. The method is thus sound.

Catching errors quickly: Our method executes and tests the program before the programmer has to invest time in formal verification. Execution is fast and, compared to formal verification, easy for the programmer to perform. Tests can reveal deviations from expected behavior. Dynamic invariant detection can also reveal such erroneous behavior.

Working on all distributed algorithms: The I/O automaton model can be used for a large variety of asynchronous systems [Gol90a, Lyn96]. By reasoning on mathematical facts, theorem provers can handle infinite state, nondeterministic systems. Dynamic invariant detection can also handle such systems. Thus with our tools, we can verify any system that can be modeled as an I/O automaton.

Reducing the programmer effort in formal verification: As a consequence of their generality, theorem provers have traditionally required significant human interaction (i.e., programmer effort in verification). Our method reduces this human effort in using the prover through assistance with lemmas and tactics.

Providing insight for the programmer: Some of the insight stems from the results of dynamic invariant detection, which may output useful properties that are not necessarily used in a particular correctness proof. More insight is achieved because our method produces a set of facts and their proofs.

```
forward simulation from Global1 to Cons:
    Cons.initiated = Global1.initiated ∧
    Cons.proposed  = Global1.proposed ∧
    Cons.decided   = Global1.decided ∧
    Cons.failed    = Global1.failed  ∧
    ∀ v:Value (v ∈ Cons.chosen ⇔
         ∃ b:Ballot (b ∈ Global1.succeeded ∧ Global1.val[b] = embed(v) ))
proof
initially
  Cons.initiated: Set[Node] := Global1.initiated;
  Cons.proposed: Set[Value] := Global1.proposed;
  Cons.chosen: Set[Value]   := {};
  Cons.decided: Set[Node]   := Global1.decided;
  Cons.failed: Set[Node]    := Global1.failed

for internal start(S: Set[Node], B: Set[Ballot]) ignore
for input init(i: Node, v: Value) do fire input init(i, v) od
for input fail(i: Node) do fire input fail(i) od
for output decide(i: Node, v: Value) do fire output decide(i, v) od
for internal makeBallot(b: Ballot) ignore
for internal abstain(i: Node, B: Set[Ballot]) ignore
for internal vote(i: Node, b: Ballot) ignore
for internal assignVal(b: Ballot, v: Value) do
 if ¬(b ∈ Global1.succeeded) then
   ignore
 elseif ∃ b:Ballot (b ∈ Global1.succeeded ∧ Global1.val[b] ≠ nil)
   then ignore
 else
   fire internal chooseVal(v)
 fi od
for internal internalDecide(b: Ballot) do
 if (b ∈ Global1.succeeded) then
   ignore
 elseif (Global1.val[b] = nil) then
   ignore
 elseif ∃ b:Ballot (b ∈ Global1.succeeded ∧ Global1.val[b] ≠ nil)
   then ignore
 else
   fire internal chooseVal(Global1.val[b].val)
 fi od
```

Figure 3-2: A `proof` block of a forward simulation relation.

```
(* One transition *)
theorem  Inv_trans_respond:
assumes
  p0: "reachable  Cache sCache"
  and p1: "Inv sCache"
  and p2: "enablement_of  Cache sCache((respond n r))"
shows " Inv (effects_of  Cache sCache((respond n r)))"
  apply (insert p0 p1 p2)
  apply (simp_all add:  Inv_def  Cache_def  Cache_enablement_def
         Cache_effect_def  Cache_state.make_def)
  apply (auto ?)
done

(* The invariant step condition itself *)
theorem  Inv_trans:
assumes
  p0: "reachable  Cache sCache"
  and p1: " Inv sCache"
  and p2: "enablement_of  Cache sCache aCache"
shows " Inv (effects_of  Cache sCache aCache)"
  apply (insert p0 p1 p2)
  apply (cases  aCache)
  apply (simp add:  Inv_trans_invoke)
  apply (simp add:  Inv_trans_respond)
  apply (simp add:  Inv_trans_read)
  apply (simp add:  Inv_trans_write)
  apply (simp add:  Inv_trans_copy)
  apply (simp add:  Inv_trans_drop)
done

(* Auxiliary proofs *)
theorem  Inv_invariant:
  "invariant  Cache Inv"
  apply (rule invariantI)
  apply (simp_all add:  Inv_start  Inv_trans)
done

theorem  Inv_reachable:
  "reachable  Cache sCache ⟹  Inv sCache"
  apply (insert  Inv_invariant)
  apply (auto intro: invariantReachable)
done
end
```

Figure 3-3: Sample output of the proof tactics generated for invariant proofs in Isabelle.

Chapter 4

Case studies

This chapter describes three case studies that were used to test our methodology of using dynamic invariant detection and model-specific insights to more efficiently prove safety properties in a theorem prover.

- Peterson's mutual exclusion algorithm. This is a lockout-free implementation of mutual exclusion for two processes using shared memory, proven by invariant assertions.

- An algorithm that implements atomic shared memory through distributed caching, drawn from Bogdanov's M.Eng. thesis [Bog00]. The implementation of shared memory is proven correct by simulation relation.

- Lamport's Paxos protocol, as written in IOA by De Prisco, et al. [DPLS$^+$02]. Paxos implements consensus using a ballot and quorum system, and is proved correct by successive refinement.

4.1 Peterson mutual exclusion algorithm

The Peterson two-process mutual exclusion algorithm [Pet81] achieves lockout-free mutual exclusion using multi-writer, multi-reader, read-write shared memory. The algorithm we present here is designed for two processes, but the algorithm can be staged in a (single elimination) tournament to support more processes.

There are three variables in the algorithm: $\texttt{flag}_0$, $\texttt{flag}_1$ and `turn`. Figure 4-1 shows the state-transition diagram for a single process. The algorithm operates as follows. Every process `p` sets `flag[p]` to true, then sets `turn` to itself. From then on, each process checks the other's `flag` and `turn`. 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`.

Figure 4-2 gives the IOA code for this algorithm, which we proved correct with both LP and Isabelle using only invariants discovered by Daikon. Daikon detected the mutual exclusion property that was the final goal, along with two invariants required for its proof.

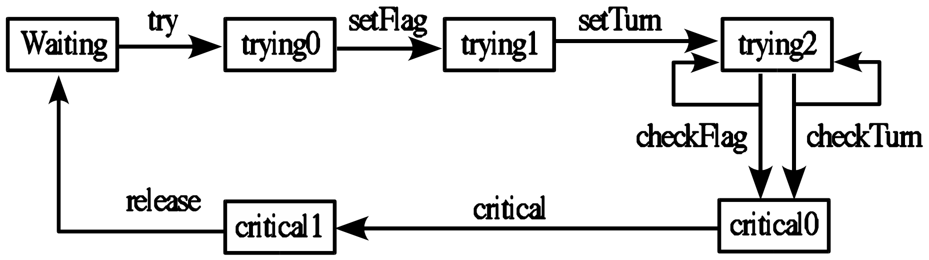

Figure 4-1: State-transition diagram for one process in the Peterson algorithm.

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 5000 transitions.

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.

4.1.1 Peterson invariants detected by Daikon

We now describe Daikon's output when given the Peterson executions.

On an initial run over the Peterson executions, Daikon reported 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.

After correcting the problem, we reran the IOA Simulator and Daikon. Daikon reported the mutual exclusion property, namely, that no two processes may simultaneously be in the critical region. Expressed in LP syntax, this invariant is

```
InvMutex:
pc[anIndex]) = critical0 ∧ anIndex ≠ anotherIndex ⇒
  pc[anotherIndex] ≠ critical1

pc[anIndex]) = critical0 ∧ anIndex ≠ anotherIndex ⇒
  pc[anIndex] ≠ pc[anotherIndex]

pc[anIndex]) = critical1 ∧ anIndex ≠ anotherIndex ⇒
  pc[anotherIndex] ≠ critical0

pc[anIndex]) = critical1 ∧ anIndex ≠ anotherIndex ⇒
  pc[anIndex] ≠ pc[anotherIndex]
```

Daikon also reported two lemmas needed to prove this property:

```
InvA:
∀ p (pc[p] = trying1 ∨ pc[p] = trying2
        ∨ pc[p] = critical0 ∨ pc[p] = critical1
```

```
% 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
```

Figure 4-2: The Peterson two-process mutual exclusion algorithm in IOA. For brevity, this figure omits the scheduling code that chooses among possible executions at runtime.

```
              ⇒ flag[p])

InvB:
∀ p (pc[p] = trying2 ⇒ pc[turn] = trying2)
```

Daikon expressed these lemmas as a set of separate terms based on enablement of the corresponding actions:

```
enabled(setFlag(p)) ⇒   flag[p]
enabled(setTurn(p)) ⇒   flag[p]
enabled(checkFlag(p)) ⇒ flag[p]
enabled(checkTurn(p)) ⇒ flag[p]
enabled(critical(p)) ⇒  flag[p]
enabled(release(p)) ⇒   flag[p]

enabled(checkFlag(p)) ⇒ pc[turn] = trying2
enabled(checkTurn(p)) ⇒ pc[turn] = trying2
```

The first clause says that whenever the `checkFlag` transition is enabled in a process, its `flag` is true. These simple conditions are not stated in the code, but some of them are apparent from static inspection. Daikon also reported some invariants that were not used for our proof, but might be useful for other proofs. One example is:

```
enabled(release(p)) ⇒ flag[turn]
```

Section 8.1 discusses possible future methods to find such true but not useful invariants.

4.1.2 Proving Peterson invariants

We used `InvB` and `InvA` to prove the mutual exclusion property `InvMutex` with both LP and Isabelle.

With Isabelle, our prototype automatically generated tactics that were sufficient to prove `InvA`. The tactics proved `InvB` for all but one transition (`checkFlag`). For this transition, we had to supply some procedural steps by hand, reasoning by case analysis depending on whether or not `anIndex` = `p`. For `InvMutex`, the same was true for two transitions, `checkFlag` and `checkTurn`.

Because LP lacks programmable tactics, we had to supply all procedural input manually. That proof, too, went through without difficulty.

An earlier proof of the algorithm, along with IOA-style pseudocode, appears in Lynch's book [Lyn96]. We did not examine the pseudocode or the proof until after completing our own implementation and proof. This proof also used two invariants. The first is our `InvA`. The second is like `InvB` but explicitly mentions the other process and is written in terms of program counters:

```
InvC:
∀ p ∀ p′ (p ≠ p′ ∧
            (pc[p] = critical0 ∨ pc[p] = critical1) ∧
            (pc[p′] = trying2 ∨ pc[p′] = critical0
                 ∨ pc[p′] = critical1)
               ⇒ turn = p′))
```

In order to compare the two pairs of invariants, we performed a second LP proof, using `InvC` instead of `InvB`. This proof was quite different, but was about the same length in terms of LP commands. Both LP proofs took about the same effort, but development of the proof based on `InvC` had the advantage of our additional experience.

The invariant proposed by Daikon has several advantages. First, it was provided automatically, requiring no user effort beyond providing test cases. Second, it was simpler and easier to understand than the expert-provided invariant. Third, it unexpectedly provided insight, revealing interesting information about the relationship between `p` and `turn`, information that we put to good use in our proof.

4.2 Atomic memory for distributed caches

We report on another case study, a strong cache for atomic shared memory, where every processor has a cache and there is a central store. Each cache can have a value or be empty. The code for this algorithm is based on Bogdanov [BGL02], but we analyze the algorithm differently from his proof.

The proof for this algorithm uses a forward simulation relation. The specification automaton, `Mem` (Figure 4-3), provides the definition for atomic shared memory. The `Null[T]` data type creates a pointed domain for a given data type `T` so that state variables can hold `nil` values. The `embed` operator takes an element of type `T` and creates an element of type `Null[T]` while the `.val` operator does the reverse.

Each node can take in an invocation via its `invoke` action, and apply the invocation to the central memory in the `update` action. The central memory `memVar` is updated according to some function `perform` while a response is calculated by the function `result`. The node responds to its user with the response. The automaton implements atomic shared memory because: 1) the invocations are all serialized on the shared variable `Mem` and 2) each invocation's time interval contains the exact point of serialization on the shared memory variable.

The automaton `Cache` implements the caching algorithm. The `invoke` and `respond` actions are the same. 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.

Our goal is to prove the existence of a forward simulation relation between `Cache` and `Mem` using our methodology.

4.2.1 Executing the cache automaton

A schedule was written for the cache algorithm in order to execute it to perform dynamic invariant detection. As with the Peterson case study, the schedule selects a node at random and attempts to move it forward in the algorithm at whatever state the node is. This randomized schedule was run for 500 steps. No significant changes needed to be made to the

```
automaton Mem
signature
  output invoke(n: Node, a: Action), respond(n: Node, r: Response)
  internal update(n: Node)

 states
   memVar: Value := v0, % an arbitrary default value
   act: Array[Node, Null[Action]] := constant(nil),
   rsp: Array[Node, Null[Response]] := constant(nil)

transitions

  output invoke(n, a)
    pre
      act[n] = nil
    eff
      act[n] := embed(a)

  internal update(n; local a:Action)
    pre
      rsp[n] = nil ∧ act[n] = embed(a)
    eff
      rsp[n] := embed(result(a, memVar));
      memVar := perform(a, memVar);

  output respond(n, r)
    pre
      rsp[n] = embed(r)
    eff
      rsp[n] := nil;
      act[n] := nil
```

Figure 4-3: The specification automaton, `Mem`.

automaton code itself to render it executable — however, for practical reasons we chose to implement each node id as an integer rather than an abstract data type.

4.2.2 Invariants required and detected

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:

```
    ∀ n : Node (cache[n] ≠ nil ⇒ cache[n].value = memVar)
```

As written in Bogdanov [BGL02], this invariant is:

```
    ∀ n : Node (cache[n] = nil ∨ cache[n] = embed(memVar))
```

The results are logically equivalent but syntactically different because Daikon does not detect invariants over disjunctions, but rather picked out the values in cache that were not `nil` and saw that the values were equal to `memVar`. That is, Daikon arrived at the same result as a human did, but via a different method.

```
automaton Cache
signature
  output invoke(n: Node, a: Action), respond(n: Node, r: Response)
  internal read(n: Node), write(n: Node), copy(n: Node), drop(n: Node)
states
  memVar: Value := v0,
  act: Array[Node, Null[Action]] := constant(nil),
  rsp: Array[Node, Null[Response]] := constant(nil),
  cache: Array[Node, Null[Value]] := constant(nil)

transitions
  output invoke(n, a)
  pre
    act[n] = nil
  eff
    act[n] := embed(a)

  internal read(n; local a : Action)
  pre
    embed(a) = act[n];
    isRead(a);
    rsp[n] = nil;
    cache[n] ≠ nil
  eff
    rsp[n] := embed(result(a, cache[n].val))

  internal write(n; local a : Action)
  pre
    embed(a) = act[n];
    isWrite(a);
    rsp[n] = nil
  eff
    rsp[n] := embed(result(a, memVar));
    memVar := perform(a, memVar);
    cache := constant(nil)

  internal copy(n)
  eff
    cache[n] := embed(memVar)

  internal drop(n)
  eff
    cache[n] := nil

  output respond(n, r)
  pre
    rsp[n] = embed(r)
  eff
    rsp[n] := nil;
    act[n] := nil

invariant Inv of Cache:
  ∀ n : Node (cache[n] = nil ∨ cache[n] = embed(memVar))
```

Figure 4-4: Strongly caching implementation, Cache.

```
forward simulation from Cache to Mem:
  Mem.memVar = Cache.memVar ∧ Mem.act = Cache.act ∧ Mem.rsp = Cache.rsp

proof
start
  Mem.act := Cache.act;
  Mem.rsp := Cache.rsp;
  Mem.memVar := Cache.memVar

for internal copy(n : Node) ignore

for internal drop(n : Node) ignore

for internal write(n : Node, a : Action) do
  fire internal update (n, a)
od

for internal read(n : Node, a : Action) do
  fire internal update (n, a)
od

for output respond(n : Node, r : Response) do
  fire output respond (n, r)
od

for output invoke (n : Node, a : Action) do
  fire output invoke (n, a)
od
```

Figure 4-5: The simulation relation and execution annotations between `Mem` and `Cache`.

4.2.3 The simulation relation

The simulation relation between the two automata is shown in Figure 4-5. The simulation relation says that the all the state variables map via the identity relation, and says nothing about the `cache` variable in the `Cache` automaton (this is why the above invariant is necessary). Notice the execution annotations for each transition, which will be used a template for generating the proof by the LP and Isabelle translation tools.

4.2.4 Proving the simulation relation

Using the proof translator tools, we translated the two automata and the simulation relation into input for both LP and Isabelle. The first step we chose to do before proving the simulation relation was to prove the crucial invariant.

The proof scripts we generated were quite successful. In LP, the invariant had to be done by hand, as we do not generate proof scripts. In Isabelle, the crucial invariant, was automatically proved just from the proof script. Thus nearly zero human work was involved in proving this invariant. Chapter 7 describes the Isabelle model for the automata in detail and the methodology for generating proof scripts for invariants.

As for the simulation relation, both prover translators generated proof scripts for the automata following the methodology presented in Chapter 3. In both provers, the proof scripts proved adequate to automatically prove all transitions except for one, the `read` transition. Here, a few lines of human-generated commands were necessary to leverage the crucial invariant: when reading from cache, the cache's value contains the value of `memVar`. Our computer-generated proof was nearly identical to Bogdanov's proof [BGL02] for LP.

4.2.5 Assessment

For the memory case study, little human intervention was required. The steps involving the most human help were: 1) selecting the invariant that might be useful for the simulation relation proof; 2) 2) giving a few commands to the prover when it halted. Thus, our methodology was quite successful here.

4.3 Distributed Consensus with Paxos

Lamport's Paxos protocol [Lam98] implements distributed consensus in an asynchronous system in which individual processes can fail. We define distributed consensus as follows. Suppose that I is a finite set of nodes representing the processes in the system and V is the set of possible consensus values. Processes in I may propose values in V. The consensus service is allowed to return decisions to processes that have proposed values. It must satisfy two conditions: all nodes must receive the same value ("agreement") and that value must have been proposed by some process ("validity").

Paxos implements consensus through the use of quorums and ballot voting and we prove the implementation through successive refinement of multiple simulation relations. We use code from a case study [DPLS+02] that defines a hierarchy of automata. The highest-level automaton, `Cons`, provides a specification for consensus. The lowest-level automaton, `Paxos`, provides a distributed implementation. An intermediate-level automaton, `Global1`, although non-distributed, captures the main idea of Paxos, that of using ballots and quorums to achieve consensus. A correctness proof involves showing the existence of a series of forward simulation relations between successive levels in the hierarchy. In this section, we discuss the forward simulation between `Cons` and `Global1`. This is based on the work by Ne Win, et. al. in [NEG+03].

4.3.1 Specification automaton

The signature of the specification automaton `Cons` (Figure 4.3.1) contains an input action `init(i,v)`, representing the proposal of value `v` by process `i`, an internal action `chooseVal(v)`, representing the choice of a consensus value `v`, an output action `decide(i,v)`, representing the report of the consensus value to process `i`, and an input action `fail(i)`, representing the failure of process `i`. The automaton provides the required agreement and validity guarantees: only a single consensus value can be chosen, and that value must have been previously proposed.

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

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

Figure 4-6: Specification of consensus in IOA

4.3.2 Implementation automaton

The automaton `Global1` (Figure 4.3.1) specifies an algorithm that implements consensus in a non-distributed setting. This automaton uses a totally ordered set of ballots for values, one of which may eventually be chosen as the consensus value if sufficient approval is collected from the processes in the system.

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

4.3.3 Executing the Global1 automaton

The second step in using our method to verify an automaton is to test its behavior by simulating execution. The simulator requires that IOA programs be transformed into a form suitable for execution. For example, the simulator requires quorums in Paxos to be initialized operationally, whereas they were specified declaratively in the original I/O automaton model.

```
type Ballot = tuple of ordering: Int

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

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

Figure 4-7: A ballot-based implementation of consensus in IOA

Aside from such bookkeeping issues, the crucial problem is resolving nondeterminism by specifying a schedule. An example schedule is:

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

causing the IOA simulator to execute five actions. The output is:

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

For our case study, we wrote schedules to execute `Global1` with different interleavings of actions, some of which cause nodes to fail or to abstain from a ballot. We did not use structured test generation methods to produce the schedules, nor did we evaluate them according to specific criteria (e.g., code coverage). Instead, we simply selected executions that illustrates what we felt was the normal behavior of the automaton (and that exercised every action). In our experience, using simple schedules like these is adequate for the purpose of dynamic invariant detection.

4.3.4 Dynamic invariant detection results

For Paxos, invariant detection with Daikon produced 23 invariants, four of which were helpful in the simulation relation proof in Section 4.3.6. The four were:

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

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

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

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

These two invariants are outside Daikon's grammar, so it neither checked nor reported them. Daikon does not report invariants with existential quantifiers, nor does it report those with more than a given number of subterms. This is not a fundamental limitation, but a design choice that reduces Daikon's computational requirements and, more importantly, the number of false positives or unhelpful invariants that Daikon would otherwise report. Section 5 discusses potential ways of improving this grammar.

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

It is notable that `Inv3`, although true and necessary for the proof, was not provable in isolation: establishing it required use of `Inv6`. In other words, dynamic invariant detection postulated a useful simple property (`inv3`) whose proof is complicated (because it requires `Inv6`). This ability to decompose a proof into parts demonstrates a strength of our technique: it is easy to check properties dynamically, even if they have complicated proofs that are beyond the capabilities of completely automatic static tools.

4.3.5 Paired execution

The fourth step in our method is appropriate when attempting to verify a simulation relation. As noted in Section 4.3.3, the IOA simulator can help users formulate and test the validity of a forward simulation relation, prior to such a verification. In this section, we discuss how a schedule and other information can help in executing paired automata, while the IOA simulator tests the conditions of the relation. This scheduling will later be useful in guiding verification.

In Figure 4-8, the simulation relation is the identity on all state variables of `Cons` except `chosen`, which is not a state variable of `Global1`. The simulation relation defines `chosen` in `Cons` to contain a value `v` if and only if there is a successful ballot in `Global1` with value `v`. The **proof** block is straightforward for the start state and for the external actions: each external action in the implementation automaton is matched by the action with the same name in the specification automaton. The internal actions `start`, `makeBallot`, `abstain`, and `vote` are matched by an empty execution sequence of the specification automaton.

In our case study, the IOA simulator can reveal two problems with a more naive treatment

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

in the **proof** block for the internal actions `assignVal` and `internalDecide`. First, given a schedule that executes `internalDecide` twice in `Global1`, the simulator discovers that the precondition for `chooseVal` fails the second time it is executed in the lockstep execution of `Cons`. Second, `assignVal` needs to fire `chooseVal` if a ballot has been decided internally but does not yet have a value assigned; hence we must fire `chooseVal` when firing `assignVal`, but only if no other ballot in `Global1.succeeded` has a non-nil value. Most of this case analysis is necessary because `Global1` allows ballots to be voted on (and to succeed) before they are assigned values.

```
forward simulation from Global1 to Cons:
    Cons.initiated = Global1.initiated ∧
    Cons.proposed  = Global1.proposed ∧
    Cons.decided   = Global1.decided ∧
    Cons.failed    = Global1.failed  ∧
    ∀ v:Value (v ∈ Cons.chosen ⇔
         ∃ b:Ballot (b ∈ Global1.succeeded ∧ Global1.val[b] = embed(v) ))
proof
initially
  Cons.initiated: Set[Node] := Global1.initiated;
  Cons.proposed: Set[Value] := Global1.proposed;
  Cons.chosen: Set[Value]   := {};
  Cons.decided: Set[Node]   := Global1.decided;
  Cons.failed: Set[Node]    := Global1.failed

for internal start(S: Set[Node], B: Set[Ballot]) ignore
for input init(i: Node, v: Value) do fire input init(i, v) od
for input fail(i: Node) do fire input fail(i) od
for output decide(i: Node, v: Value) do fire output decide(i, v) od
for internal makeBallot(b: Ballot) ignore
for internal abstain(i: Node, B: Set[Ballot]) ignore
for internal vote(i: Node, b: Ballot) ignore
for internal assignVal(b: Ballot, v: Value) do
 if ¬(b ∈ Global1.succeeded) then
   ignore
 elseif ∃ b:Ballot (b ∈ Global1.succeeded ∧ Global1.val[b] ≠ nil)
   then ignore
 else
   fire internal chooseVal(v)
 fi od
for internal internalDecide(b: Ballot) do
 if (b ∈ Global1.succeeded) then
   ignore
 elseif (Global1.val[b] = nil) then
   ignore
 elseif ∃ b:Ballot (b ∈ Global1.succeeded ∧ Global1.val[b] ≠ nil)
   then ignore
 else
   fire internal chooseVal(Global1.val[b].val)
 fi od
```

Figure 4-8: Forward simulation relation and step correspondence (`proof` block) from the `Global1` specification (Figure 4.3.1) to the `Cons` implementation (Figure 4.3.1).

This nondeterminism makes the algorithm more general, but it complicates the correctness proof. Hence it was helpful to use paired simulation to debug the details of the step correspondence and arrive at the formulation shown in Figure 4-8.

4.3.6 Verifying a simulation relation in LP

The last step in our method is to prove the simulation relation (or invariant) using a theorem prover. Here we describe how the results of Sections 4.3.4 and 4.3.5 can be used to generate this input automatically. First, the invariants suggested by dynamic invariant detection become candidates lemmas, thereby saving the user time in finding auxiliary invariants needed for verification. Second, the annotations for paired execution provide a proof outline.

Recall from Section 2.5.1 that verifying a simulation relation requires verifying both a start condition and a step condition. Translation tools in the IOA toolkit use the **proof** block for a simulation relation to generate proof tactics for each condition.

Start condition

The start condition requires finding a witness start state b of the specification automaton. In the Paxos case study, the proof script generator translated the **initially** section of Figure 4-8 into the following commands for LP:

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

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

Step condition

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

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

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

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

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

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

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

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

When **proof** blocks are more complicated than those in the Paxos case study, the job of the proof script generator is correspondingly more complicated. For example, the generator must expand a **for** entry in the **proof** block contains a sequence of conditional statements such as

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

into one that contains nested conditionals such as

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

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

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

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

Chapter 5

Achieving fast, scalable and correct dynamic invariant detection

5.1 Introduction

A dynamic invariant detection tool observes program executions and outputs properties (logical formulae) that are true of the executions and are likely to be true of the program in general. For example, an invariant detector might report that at entry to procedure 2`foo`, $\mathsf{a} \neq \mathsf{null} \Rightarrow \mathsf{x} = \mathsf{a.length}$; that at exit from procedure `bar`, $\mathsf{y}' = \mathsf{y} + 1$ (where y is the variable's value at entry and y' is the value at exit); or that at from the point of view of any client of class `C`, $\mathsf{C.x} \geq \mathsf{C.y}$. The reported properties, which are syntactically identical to formal specifications, are also known as likely invariants.

Invariant detection has applications to program evolution [ECGN01b], static verification [NE01, Nim02a, NE02b, NE02c, ?], program refactoring [KEGN01], automated theorem proving [NE02a, NEG$^+$03, ?], testing [?, Har02, ?, ?], software upgrades [?], anomaly detection [RKS02], fault detection [DDLE02, Dod02, HL02, ?], error isolation [?, ?], and error prevention [?], among others.

A naive implementation of dynamic invariant detection is not difficult. We present an abstract algorithm in Section 5.1.2. Doing it correctly and efficiently, however, requires some thought. The two correctness properties we wish to preserve while optimizing an invariant inference algorithm are *soundness* and *completeness*. We formally define these as in Section 5.1.3, within the context of a *grammar*, or the set of formulae an algorithm can output.

An efficient inferencing algorithm is desirable, in terms of time and space needed. The primary efficiency concern, either when adding other optimizations or expanding the grammar, is to keep the algorithm *incremental*. An incremental algorithm exhibits two properties that are dual to each other. First, it consumes no extra space with respect to the length of the executions that are given, and it runs linearly in time. Second, it can run *on-line*, alongside the program to be examined, receiving values as they are generated and discarding them afterward. For example, an incremental algorithm can monitor a web server over days or weeks, without having to store the record of each transaction. This is important as a week's transactions may not even fit on disk. An incremental algorithm is also a one-pass algorithm.

Once incrementality is achieved, there are secondary efficiency concerns. It is also important to save space and time as a function of the size of the program, measured in the number of variables. We optimize along this line by taking advantage of one general idea: invariants often imply each other. The space and/or time resources used to check such implied invariants can often be avoided. We present three optimizations based on the idea of implied invariants in this chapter.

This chapter is organized as follows. Section 5.1.2 describes a naive invariant inferencing algorithm as a benchmark to compare against. Section 5.1.4 describes the different parameters that affect resource consumption and the data structure space over which an algorithm would operate. Section 5.2.1 classifies the different methods by which optimizations could be performed. Sections 5.3.1(largely borrowed from Nimmer [Nim02b], but repeated here for clarity of exposition), 5.4, and 5.5 show major optimizations that lead to better use of space and time by employing the fact that some invariants imply each other in some way. These optimizations are performed on the Daikon invariant detector [ECGN01b], a general purpose tool with a wide and useful grammar. Section 5.7 shows extensions to Daikon's grammar to make it more useful and allow it to extract more information from any given execution. Section 5.8 presents future work and Section 5.9 concludes.

5.1.1 Terminology

Here we define terminology, to give a foundation for the rest of this chapter.

A *program point* is slightly more general than just a specific location in the program. Instead, it represents a specific scope (set of variables) and its associated semantics. For example, consider a program point associated with the pre-state of a method. Its scope is all fields of the class and any arguments to the method. Its semantics are that every time the method is called, a snapshot of all pre-state within scope is taken. For a program point associated with the object invariants of a class, its scope is all fields of the class, and its semantics are that every time the any public method is called, snapshots of all pre-state and post-state within scope is taken.

A *sample* is the snapshot of program state taken for a specific program point.

An *execution* is a sequence of samples taken over the run of a program over time.

A *variable* is an expression associated with a given scope (a program point). It may be a simple field reference (such as `this.x`), may involve array indexing or slicing (such as `this.myArray[this.x..this.y]`), or may involve other compound expressions. A *primitive variable* is one whose value is provided in a sample. The `a.[x]` operator denotes array subscripting as via C or Java syntax.

A *derived variable* is a variable whose value is not provided in a sample, but is instead computed as a function of other variables after the fact. For example, array slices like `this.myArray[this.x..this.y]` are derived from the full array `this.myArray[]` given in the sample.

An *invariant object* is a data structure that relates variables at a program point. It represents a boolean formula that will be checked against data from samples in dynamic invariant detection. For brevity, we will often refer to an invariant object simply as an invariant, but we do not mean that it is true over all runs of a program.

A *grammar* is the set of boolean formulae over which dynamic invariant detection occurs. It is also thus the set different invariant object types. For example, a grammar might contain the equality and less than operators on integers.

There are also some terms specific to the Daikon dynamic invariant detection system.

A *frontend* is a tool that executes a program, interprets runtime values and outputs the values into a format understandable by the Daikon *back end*, which analyzes the data samples in terms of invariants. For example, the IOA Simulator is the IOA front end for Daikon, and the Daikon Frontend for Java (dfej) instruments Java programs to output values to a file readable by Daikon. The Daikon back end is responsible for computing derived variables from primitive variables given in data samples.

5.1.2 Naive algorithm

Recall that an invariant detector reports all properties in its grammar that are not violated at runtime. This section gives a naive algorithm that achieves that goal. As noted later in Section 5.1.4, this algorithm runs too slowly to be practical.

1. Initially, assume all properties in the grammar to be true. Instantiate an invariant for each property and combination of variables. For example, if the available properties are odd and =, and the variables are a, b, and c, instantiate odd(a), odd(b), odd(c), $a = b$, $a = c$, and $b = c$.

2. For each sample, check each candidate invariant associated with the same program point as the sample and eliminate (falsify) any that are contradicted by the sample. For example, the sample $\langle 3, 4, 3\rangle$ eliminates the invariants odd(b), $a = b$, and $b = c$ from the above list.

3. Report the invariants that remain after all samples have been processed.

5.1.3 Correctness properties

This algorithm has two desirable properties, soundness and completeness.

Soundness Any invariant reported by a sound invariant detection algorithm holds for all samples in the input (that is, in the test executions).

Like other dynamic techniques such as testing, invariant detection is unsound over all possible executions of a program. The observed executions are not guaranteed to characterize all possible execution environments of the program. Our definition is for soundness with respect to observed executions.

Completeness A complete algorithm reports any property that: 1) is expressible in its grammar, and 2) holds for all samples in the input.

Completeness with respect to all possible properties is possible in theory [CC77b] but unattainable in practice. An infinite number of formulae are true of any set of executions,

Per-program-point variables:	
V	number of primitive variables at a program point
L	number of samples for a program point
I	number of instantiated invariants at a program point $= O(V^6)$
RI	number of reported invariants at a program point $= O(I)$; in common case, $= O(1)$
FI	number of falsified invariants at a program point $= O(I)$
Total (sum of all program points) variables:	
P	number of program points
I_t	total number of invariants $= O(P \cdot I)$
L_t	total execution length ("exploded") $= O(P \cdot L)$
Other variables:	
G	number of invariant templates (grammar)

Figure 5-1: Variables used in the running time and space analyses (see Section 5.1.4 for details). All variables except for RI, FI, L, and L_t are known statically.

and many of them are beyond the scope of any static or dynamic analysis. Our definition is completeness with respect to a grammar of properties.

Given these definitions, the naive algorithm is sound because every invariant is tested against every relevant sample in Step 2. It is complete because all possible invariants are instantiated in Step 1. It is incremental because it runs in one pass.

However, the algorithm achieves these three goals at a large time and space cost, as shown in Section 5.1.4. By instantiating all invariants, many redundant invariants are instantiated and tested. For example, if $a = c$ and $odd(a)$ holds, then $odd(c)$ holds.

5.1.4 Performance analysis

This section analyzes the time and space performance of the naive detection algorithm.

Analysis variables

Figure 5-1 lists the variables that influence performance:

V denotes the number of primitive variables at a program point. Conceptually, this is the number of variables in scope. In practice, it is the number of variables that are measured by a frontend. The practical value may be less if the frontend omits certain variables, or may be more if it breaks out of scope (for example, by sampling private fields of classes in object oriented programs).

L denotes the number of samples at a program point.

I denotes the number of instantiated invariants at a program point. V will have a positive, superlinear effect on I, for two reasons. First, invariants often involve more than one variable. For example, there are $\Theta(V^2)$ pairwise equality invariants. Second, the grammar of a tool like Daikon will create *derived* variables that represent a computation on the original variables. For example, if A is a sequence, and i is an integer, the derived variable $A[i]$ may be interesting, even if it is not mentioned in the program text. In general, if the grammar supports j-way derived variables, k-way invariants, and G different invariant types, then there could be as many as $I = \Theta(GV^{jk})$ invariants. In the case of Daikon, $j = 2$ and $k = 3$, yielding the definition of I given in the figure.

RI and FI, the reported and falsified invariant counts, respectively, are used to better elucidate common case analyses where the reported invariants are a minuscule subset of the grammar's candidate set.

P, the number of program points, scales with program size. Also, the "total" values I_t and L_t denote the product of P and their subscript-less counterpart.

Naive performance

To study the performance of the naive algorithm, we compute its space and time performance both in the worst and common case. We measure space S by the maximum number of instantiated invariants, and time T by the number of checks on whether an invariant holds .

The naive algorithm performs quite poorly. Since the algorithm instantiates every possible invariant (in step 1), its maximum space usage is $O(I_t)$. This is the same for both the common and worst cases, and makes the algorithm incremental.

$S = O(I_t) = O(P \cdot I)$

Worst-case time occurs when all invariants are checked against all samples for their program point; this happens when no invariants are falsified.

$T = O(I \cdot L_t) = O(I_t \cdot L) = O(P \cdot I \cdot L)$

Common-case time occurs when falsified invariants are falsified quickly, so that there are vanishingly few checks against falsified invariants.

$T = O(P \cdot (RI \cdot L + FI \cdot O(1))) = O(P \cdot RI \cdot L) + O(P \cdot FI \cdot O(1))$

The first term is for the useful invariants that are actually true on the program, which are always checked, while the second term is for invariants that are untrue and, in the common case, quickly falsified.

Within the constraints of soundness, completeness and the incremental requirement, a major way to save space and/or time, as well as to reduce clutter for the user, is to eliminate some derived variables and invariants. There is no way to avoid examining all program points and every execution sample if soundness is to be preserved.

Lastly, it is important to note that the asymptotic notation hides one important detail: it takes some amount of time to check if a sample contradicts an invariant. If Daikon could discern that no sample may ever falsify a certain invariant, it could omit checking the invariant, leading to significant improvement in the asymptotic constants.

5.2 Goal: eliminate invariants and derived variables

As mentioned in Section 5.1.4, the most promising way to optimize invariant inference is to eliminate some derived variables and invariants that are uninteresting or logically implied. This also helps to reduce clutter for the user.

When examining which invariants and derived variables to eliminate, there are two orthogonal concerns. The first is whether the elimination is static or dynamic, and the second is how much it saves in resources.

5.2.1 Elimination type: static, dynamic negative or dynamic positive

The first concern for eliminating invariants and derived variables is whether the process can be done statically or must be done dynamically. Dynamic elimination can be further split into two cases: negative and positive.

A static elimination is one that can be done without execution trace data. An example is $A[i] \in A[]$, which is always true. A static elimination is advantageous, as it can be done even before data is seen. In contrast, a dynamic elimination needs execution data. The example where $(x = y) \wedge odd(x)$ eliminates $odd(y)$ is a dynamic elimination, for we cannot be sure that $x = y$ until some data is seen. Within dynamic elimination, there are two cases:

Nonsensical/Negative elimination is where one sample can forever remove an invariant or derived variable. For example, if ever $x \geq A.length$, then $A[x]$ is nonsensical. This is called negative elimination because examples for elimination are contradictory to the invariant or derived variable.

Implied/Positive suppression is where an invariant or derived variable is eliminated only through being implied by some other invariant. For example, $(x = y) \wedge odd(x) \Rightarrow odd(y)$. Here, a positive statement is being made about the eliminated invariant by the two invariants on the left. No one sample is enough: we cannot be sure of the elimination until all data has been seen, since $x = y$ or $odd(x)$ could be falsified. Thus we use the term "suppression" because invariants are never completely eliminated.

5.2.2 Resources saved in elimination: time, space or printing

The second concern for elimination is what resources it saves. At best, it will save space, save time and let the inferencing algorithm avoid printing out extra information that is obvious or uninteresting to the user. At worst, it will only reduce printing, but have the same time and space costs.

To avoid printing extra invariants is relatively simple. Invariant inference can be run as before. When invariants are to be output, an invariant that is logically implied by the rest can be eliminated, and this process can be repeated.

To save space, it is necessary to actually delete invariants or never instantiate them. However, soudness and completeness need to be preserved. For example, if $x = y \wedge odd(x) \Rightarrow odd(y)$, then deleting $odd(y)$ might be a good idea. However, this naive deletion produces an incomplete algorithm: should later $x \neq y$, yet $odd(y)$ remain true, the user will not be able to deduce $odd(y)$ from $odd(x)$. The correct solution, if $odd(y)$ is to be eliminated, is to preserve some mechanism for regenerating $odd(y)$ should $x \neq y$ — i.e., should the dynamic suppression no longer apply.

To save time, invariants can be removed as with for space. Alternatively, invariants can be marked so they are not checked. These invariants would take up space, but they would not need to be checked against data. For complex invariants like $\forall_i A[i] > 5$ checking can take up a significant amount of time.

In the following sections, we do not discuss any static elimination techniques for there is already adequate literature on standard static analysis [CC77b, GC96]. Similarly, dynamic negative elimination is also relatively straightforward: simply stop keeping track of the invariant when it is no longer interesting. Instead, we describe three techniques for dynamic positive suppression.

5.3 Previous optimizations

Before this work, three optimizations were already performed on the Daikon system.

First, the concept of invariant *weakening* was added. For a series of invariants such as $x \geq 2, x \geq 1, x \geq 0$ that imply each other in a total order, it is only necessary to keep track of one of them, and *weaken* that invariant as facts come in. With the $\geq$ invariant, we start with $x \geq$ `MAX_INT`. The correct amount of weakening has to be done: if $[5, 8, 1, 2]$ are seen, then the final invariant must be $x \geq 1$. This ensures soundness and completeness.

Second, a version (V2) of Daikon attempted to eliminate some invariants, but had to make multiple passes over the data to do so. For example, Daikon V2 did a pass to collect which variables were equal to each other before any other invariants were generated. If $x = y$ then only $f(x)$ would be instantiated on the second pass. Thus V2 eliminated many obvious invariants and avoided examining extraneous data. This saved time at the expense of space: it took up space in memory that was linear on E and was not incremental.

Third, the concept of a data hierarchy was introduced, to relate variables across program points. We describe this hierarchy in the next section.

5.3.1 Hierarchy and flow

5.3.2 Nimmer's thesis paste, to be edited

This section introduces a technique to perform dynamic positive suppression for invariant detection by taking advantage of implication relationships between different program points, and organizing program points into a lattice.

5.3.3 Staged inference

In a previous version(Section 5.3), the Daikon system operated in multiple passes over the samples to infer invariants. The multiple passes permited optimizations because certain invariants are always true or false, or certain derived variables are undefined. By testing the strongest invariants in earlier passes, the weaker invariants or certain derived variables may not need to be processed at all. For example, if $\mathsf{x} = 0$ always holds over an earlier pass, then $\mathsf{x} \geq 0$ is necessarily true and need not be instantiated, tested, or reported on a later pass. Similarly, unless the invariant $\mathsf{i} < \mathsf{theArray.length}$ holds, the derived variable $\mathsf{theArray[i]}$ may be non-sensical.

```
public class A {                    public class B {
  public static int n;                private int x;
  private B b;                        public int m2();
  public int m(B arg);              }
}
```

Figure 5-2: Example declarations for two simple Java classes.

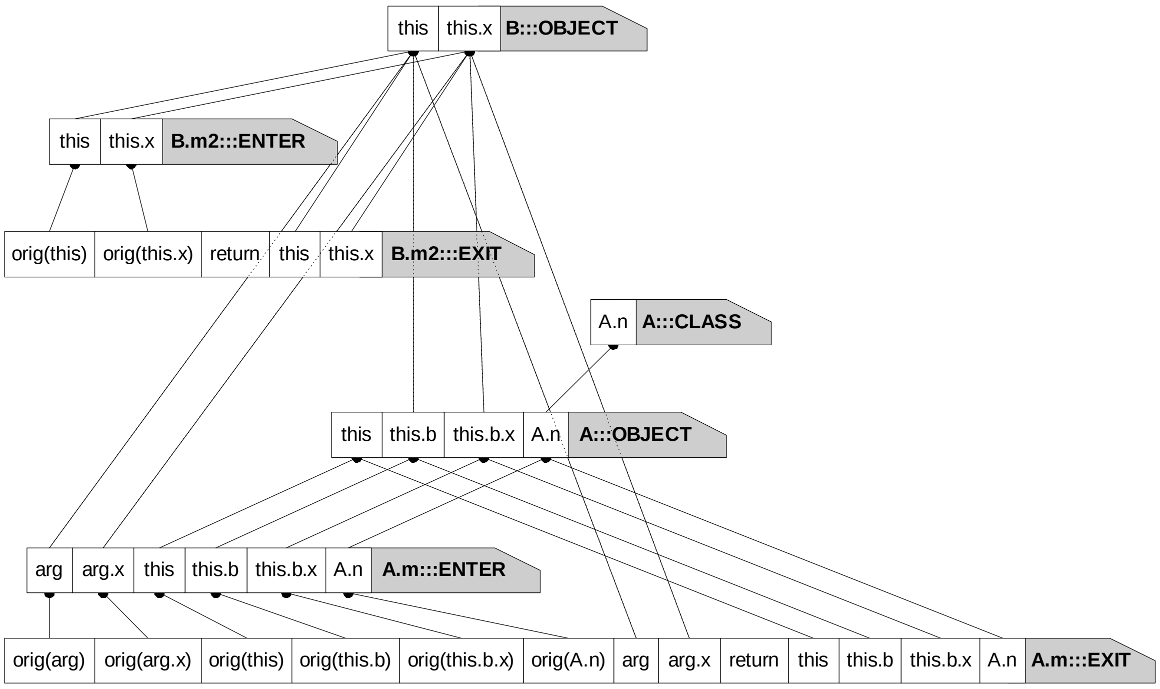

Figure 5-3: Flow relationship between variables for the code shown in Figure 5-2. Shaded areas name the program point, while unshaded boxes represent variables at that program point. Lines show the partial ordering $\sqsubseteq_D$ described in Section 5.3.4, with a nub at the lesser end of the relation. (For instance, `arg` $\sqsubseteq_D$ `orig(arg)` in the lower left corner.) Relations implied by transitivity of the partial order are not explicitly drawn.

While operating in multiple passes, Daikon also treated each program point independently. Therefore, data from one program point may be discarded before the other points' data is processed.

5.3.4 Variable ordering

One major way to improve performance for dynamic invariant detection is for program points to no longer be processed independently. Instead, the dynamic invariant detection algorithm relates variables from all program points in a partial order.

The relationship that defines the partial order $\sqsubseteq_D$ is "sees as much data as". If variables `X` and `Y` satisfy `X` $\sqsubseteq_D$ `Y`, then all data seen at `Y` must also be seen at `X` — `X` sees as much data as Y. As a consequence, the invariants that hold over X are a subset of those that hold over Y, since any data that contradicts an invariant over Y must also contradict the same invariant over X.

Figure 5-3 shows the partial order formed by $\sqsubseteq_D$ for the example classes of Figure 5-2. Consider the relationship between B:::OBJECT and B.m2:::ENTER. First, recall that all data from method entries must also apply to the object invariants. (In other words, object invariants must always hold upon method entry.) Therefore, we have $\texttt{this}_{\texttt{B:::OBJECT}} \sqsubseteq_D \texttt{this}_{\texttt{B.m2:::ENTER}}$ and $\texttt{this.x}_{\texttt{B:::OBJECT}} \sqsubseteq_D \texttt{this.x}_{\texttt{B.m2:::ENTER}}$. The same holds true for method exits: $\texttt{this}_{\texttt{B:::OBJECT}} \sqsubseteq_D \texttt{this}_{\texttt{B.m2:::EXIT}}$. Finally, note that $\texttt{this}_{\texttt{B.m2:::ENTER}} \sqsubseteq_D \texttt{orig(this)}_{\texttt{B:::EXIT}}$, since any pre-state data associated with a method exit must have been seen on entry.

For reasons similar to ones that relate B's variables across program points, the relationships that contribute to the partial order are as follows.

Definition of orig().

Variables on ENTER points are $\sqsubseteq_D$ the corresponding orig() variables at all EXIT and EXCEPTION program points.

Object invariants hold at method boundaries.

Instance variables from the OBJECT program point are $\sqsubseteq_D$ the corresponding instance variables on all ENTER, EXIT, and EXCEPTION program points.

Object invariants hold for all instances of a type.

Instance variables from the T:::OBJECT program point are $\sqsubseteq_D$ the corresponding instance variables on instrumented arguments and fields of type T. (For example, see $\texttt{arg}_{\texttt{A.m:::ENTER}}$ and $\texttt{this.b.}_{\texttt{A:::OBJECT}}$ in Figure 5-3.)

Subclassing preserves object invariants.

Instance variables from the T:::OBJECT program point are $\sqsubseteq_D$ the same instance variables on subclasses or non-static inner classes of T.

Overriding methods may only weaken the specification.

Argument(s) to a method m are $\sqsubseteq_D$ argument(s) of methods that override or implement m, by the behavioral subtyping principle.

5.3.5 Consequences of variable ordering

As shown in the previous section, the partial ordering of variables implies that when invariants hold true over variables at certain program points, those invariants also must hold true at lower (as drawn in Figure 5-3) program points. For instance, if we have $\texttt{this.x}_{\texttt{B:::OBJECT}} \geq 0$, then we also know that $\texttt{arg.x}_{\texttt{A.m:::ENTER}} \geq 0$. We call this the *hierarchy* property of program points.

Dynamic invariant detection can be optimized by taking advantage of this hierarchy property. One way to do this, which we have implemented in the Daikon system, is by instantiating, checking, and reporting invariants at the most general place they could be stated. For instance, if an invariant always holds over an object's fields, it would only exist at the `OBJECT` program point (instead of each method's `ENTER` and `EXIT` points), and would only need to be tested once per sample.

For printing invariants for the user, the algorithm could locate all invariants over a set of variables V at a program point P by forming the closure of V at P using the partial ordering, and taking the union (conjunction) of the invariants present at each point in the closure.

However, for this technique of minimal invariant instantiation to work, both the samples and the invariants must flow through the partial order in a specific way, as explained in the next two sections.

5.3.6 Invariant flow

The naive algorithm instantiates all possible invariants (modulo type constraints) at all program points. Now the hierarchy property described above leads to new rules for instantiating invariants and for what to do when checking invariants against data.

Instantiation During the setup phase prior to seeing data, instantiate invariants only where they would not be inferrable from a corresponding version higher in the hierarchy. For example, the algorithm would not instantiate $\texttt{arg.x}_{\texttt{A.m:::ENTER}} \geq 0$ since it would have instantiated $\texttt{arg.x}_{\texttt{A.m:::OBJECT}} \geq 0$ (or the equivalent invariant at a higher program point). Formally, instantiation is only allowed when one or more of the variables of an invariant has no predecessor in the $\sqsubseteq_D$ partial order. That is, a set V of n variables should be used to instantiate an n-ary invariant only if $\exists v \in V \forall x : x \not\sqsubseteq_D v$.

Checking When an invariant is falsified or weakened at a program point during inference, copy it "down" to the immediate child program points. By child program point, we mean program point(s) where every variable used by the invariant is less in the partial ordering. By nearest we mean that there must be no intermediate program points. For example, when $\texttt{arg.x}_{\texttt{A.m:::OBJECT}} \geq 0$ is falsified, it is copied down to `A.m:::ENTER` as $\texttt{arg.x}_{\texttt{A.m:::ENTER}} \geq 0$. Formally, a falsified invariant over a set of source variables A should be copied to a destination set B when all variables in B are at the same program point and when $\forall a \in A : \exists b \in B : (a \sqsubseteq_D b) \wedge (\neg\exists x : a \sqsubset_D x \sqsubset_D b)$.

One positive consequence of this approach is that methods defined in interfaces will have invariants reported over their arguments, even though no samples can ever be taken on interfaces directly. For example, if every implementation of an interface's method is called with a non-null argument, Daikon will report this property as a requirement of the interface, instead of as a requirement of each implementation.

5.3.7 Sample flow

In the invariant flow algorithm, invariants flow down as they are falsified. This property suggests a corresponding flow algorithm to process samples.

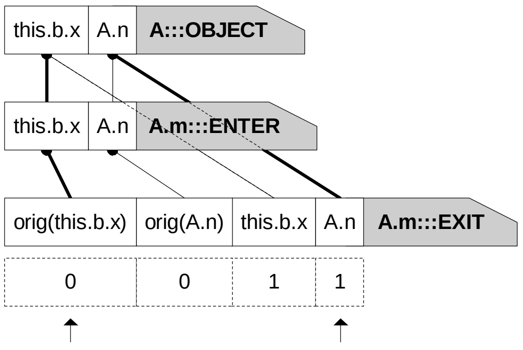

Figure 5-4: Example indicating the need for path information in sample and invariant flow, as described in Section 5.3.8. A portion of Figure 5-3 is reproduced, along with a potential sample (0,0,1,1). Given only that sample, the invariant this.b.x = A.n at A:::OBJECT should hold. However, if the sample flows as indicated by the bold links of the partial order, the invariant would be incorrectly falsified. Therefore, the path taken is important.

1. Identify the exact program point where the sample was drawn from.

2. Form the closure of program points that have any variable filled in by following the relations upward in the partial order.

3. Feed the sample to each of these program points in a topological order. A sample is fed to a point after it has been fed to all points where a variable is greater. Therefore, any falsified invariants are always copied to lower program points before the sample is fed there to falsify them.

5.3.8 Paths

For both invariant and sample flow, the path taken through the partial order is important. For example, consider Figure 5-4. Given only this data, Daikon should report that this.b.x = A.n at A:::OBJECT. However, since we have $\texttt{this.b.x}_{\texttt{A:::OBJECT}} \sqsubseteq_D \texttt{orig(this.b.x)}_{\texttt{A.m:::EXIT}}$ and $\texttt{A.n}_{\texttt{A:::OBJECT}} \sqsubseteq_D \texttt{A.n}_{\texttt{A.m:::EXIT}}$, the values for orig(this.b.x) and A.n would falsify the invariant. The problem is that the two paths through the partial order are different — they traverse different program points.

To address this problem, we annotate each edge in the partial order with some nonce. A pair of variables <A1,A2> is together related to <B1,B2> by the partial order if the path from A1 to B1 follows the same nonces as the path from A2 to B2. The nonces must be chosen so that sets of variables from two program points that are related due to the same item from the list starting on page 63 must share the same nonce. In terms of Figure 5-3, parallel or nearly-parallel lines from one program point to another will have the same nonce.

5.3.9 Hierarchy shape

An important property of the hierarchy property presented so far is that the variables and program points form a tree under the $\sqsubseteq_D$ relationship for any one reason that variables

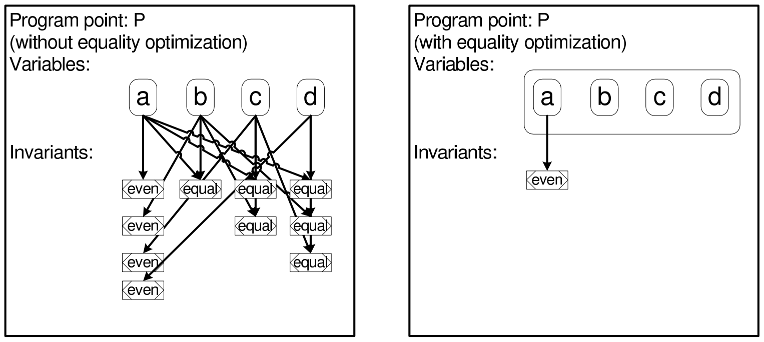

Figure 5-5: The advantage of handling equality specially. Without equality handling (above) there are 4 invariants saying a, b, c and d are even. There are 6 invariants saying the variables are equal. With equality handling all the equality invariants collapse into one equality set, while even invariants are represented by $even(a)$ on the leader of the set.

relate, such as method instance variables to object instance variables. The variables of a program would thus form a forest under the partial order. However, when multiple reasons are combined, variables are no longer always related a tree — they can become a general directed acyclic graph.

For example, with multiple inheritance (due to interfaces), a method's specification could be governed by multiple interfaces, so its arguments would have multiple parents in the partial order.

Thus we can no longer say that "an invariant only appears at the *one place* where it may be most generally stated". Instead, we reword "one place" to *minimal number of places.* An implementation would have to take into account the non-tree nature of the partial order when flowing samples and variables.

5.4 Handling equality

One dynamic positive elimination technique is to handle equality specially. This saves space and time in two ways. Given v variables that are always equal — i.e., belong to the same equality set: 1) We have one equality invariant for the set rather than $\Theta(v^2)$ two-way equality invariants. 2) We have invariants only on one member of each equality set rather than v invariants for each member. With invariants of more than one variable, the savings for the latter can be enormous.

As seen in Figure5-5, let us say a program point has four variables, a, b, c, d and they are all equal. There is an invariant f that can apply over any one of them. Normally, we would need 6 equality invariants to express the equality:

$a = b, a = c, a = d, b = c, b = d, c = d$

and 4 invariants to show f held on each:

$f(a), f(b), f(c), f(d)$

With the use of equality sets, we have one data structure that keeps track of the equality relationship, and only create one copy of f for a canonical member, or *leader* of the set, $f(a)$. The leader is chosen arbitrarily from the set. This saves space because we do not need to keep track of other instances of f or equality invariants. Further, this saves time, especially when checking f is computationally expensive.

This is a dynamic, positive elimination. $f(b)$ is suppressed by $f(a)$ and $a = b$, and should the latter be falsified, $f(b)$ will have to be created as an invariant to be checked. We do not have to specially handle $f(a)$ being falsified, since as long as $a = b$ held, $f(b)$ would also be falsified.

The equality optimization to the Daikon tool is summarized below, as an augmentation to the native algorithm.

Initialization Before running the rest of the algorithm, all comparable variables at a program point are placed into the same equality set (a data structure). Each equality set is given an arbitrary leader from within the set. When the normal algorithm instantiates (non-equality) invariants, we allow instantiation only for invariants whose variables are leaders of their equality sets.

During checking At every data sample during checking, each equality set is first checked to see if its members are still equal. If not, the equality set is split into new sets, so that variables with same values are placed into the same set. The invariants of the old set's leader are copied to each of the new leaders. After this, all other invariants (some of which may be newly copied) are checked as usual.

Post processing The equality sets can, at the end of the run through the execution data, be "devolved" back into regular two-way equality invariants. How this is done depends on the user's preferences. By default, this post processing stage creates two-way equality invariants between the leader of each equality set and its members, rather than between all members of an equality set.

Notice that the dynamically suppressed invariants, such as $f(b)$, are kept implicitly by the equality mechanism: when b no longer equals a, we simply copy f from a. The space and time savings depend on how many invariants are equal during checking. The more variables that are equal for a longer amount of time, the more the savings.

The above summary gives an overview of how a naive invariant inference algorithm can be altered for saving time and space using equality. It is one-pass because it simply adds some steps to the algorithm before initialization and before each sample is processed. However, ensuring soundness and completeness requires handling five issues:

- Copying and instantiating non-equality invariants between members of the same equality set.
- Counting the number of data samples seen
- Counting the number of missing data samples seen
- When invariants are not instantiated with respect to data flow
- Other post processing issues

We discuss each of these in the sections that follow.

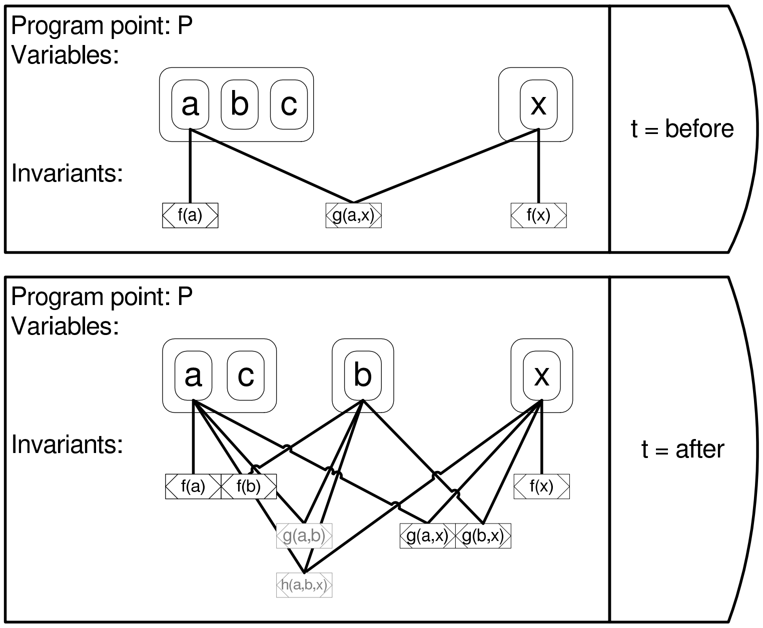

Figure 5-6: When b splits off from $\{a, c\}$ it is not adequate to simply copy the invariants from the old leader to the new. The two grayed-out invariants, $g(a, b)$ and $h(a, b, x)$ show what would be missing.

5.4.1 Copying and instantiation of invariants

When an equality set is split during inference, any invariants from the leader of the old set are copied onto the leaders of the new sets. This is necessary to represent the fact that invariants may remain true on the new sets but falsified on the old sets. However, mere copying is not adequate.

Each of the new leaders and the old leader might have other invariants between themselves. The example in Figure 5-6 demonstrates this. The top part shows the state of invariant inference before a certain sample is seen, while the bottom shows the state after the sample. Before, a is the leader of the equality set of $\{a, b, c\}$. After, b splits off into its own equality set. x is a variable that was not in the original equality set a, b, c. The invariant templates are f (unary), g (binary) and h (ternary). Assume the invariants are commutative for their variables (i.e., $g(a, b) \Leftrightarrow g(b, a)$). Originally, the invariants are:

$f(a), f(x), g(a, x)$.

If we simply copied over the invariants of the old leader onto those of the new leaders, we would additionally have:

$f(b), g(b, x)$

This does not cover all possibilities, for we are missing:

$g(a, b), h(a, b, x)$

as shown grayed in the figure. A possible g that could hold at this point, for example, is $g(a, b) : a \geq b$.

One way to have these invariants is to instantiate them at the time of the equality split. Another is to instantiate them during initialization. We first explain why the first method is unsound, and then describe the second method.

Method 1: Instantiating invariants at the time of split

An algorithm could instantiate new invariants at the time of splitting as follows: Let N be the set of new leaders, o be the old leader, and X be the leaders outside of the old equality set. The invariants to instantiate are all the invariants on $\{o\} + N + X$ minus the invariants that were copied, minus the invariants that hold on X alone. This precisely covers the set

$g(a, b), h(a, b, x)$ shown grayed in Figure 5-6.

The problem with this implementation is that some of the instantiated invariants may have been falsified by data at this point. Clearly, for an invariant like $g(a, b) : a < b$ this is easy to determine. However, there are some invariants for which equality implies nothing. For example, consider the invariant $g(a, b) : a + b = 4$. Knowing that, until now, $a = b$ gives no information on whether $a + b = 4$ has been true. Short of actually storing all samples of a and b (which would make a non-incremental algorithm), there is no way to determine if $g(a, b)$ held for a general formula. Thus instantiating invariants at the time of split is unsound, because we do not know which invariants to instantiate.

Method 2: Instantiating invariants during initialization

A way to know if $g(a, b)$ held previously is to actually instantiate $g(a, b)$ at the beginning. This way, the invariant itself is checked against the samples before the time of the equality split. However, we do not want to instantiate $g(a, b)$ since this would obviate the benefits of equality optimization. However, we know that $a = b$ before the split, so we *can* instantiate $g(a, a)$ instead for both $g(a, b)$ and $g(a, c)$. We call these new types of invariants *reflexive* invariants. When b splits off from a, both the reflexive and non reflexive invariants are copied onto b, using the same mechanism. This approach can be seen more generally in two new rules, one for initialization and one for during copying:

Initialization When instantiating (non-equality) invariants during initialization, let there be r equality sets initially. In the summary description of equality optimization, for every invariant on k variables, the algorithm would instantiate ${}_rC_k$ invariants. This is because we wanted all combinations of k variables on r equality leaders. Now the algorithm instantiates all k-way combinations *with repetition*. This allows cases like $g(a, a)$ and $h(a, a, d)$.

In the example above, the invariants that would exist before the equality set split are shown in Figure5-7:

$f(a), f(x), g(a, x), g(a, a), h(a, a, a), h(a, a, x), h(a, x, x), h(x, x, x)$

Copying during a split When an equality set is split off, invariants are only copied (no instantiation), but this copying is no longer a direct substitution of new leader for old leader. We again use the combinations with repetition technique. Let r be the number of new equality sets from one equality set. For each invariant containing k instances of the old leader, we copy the invariant for each k-way combination with repetition on the r leaders. Figure 5-8 shows how each of the invariants in the example would be copied when the set with a as leader splits off.

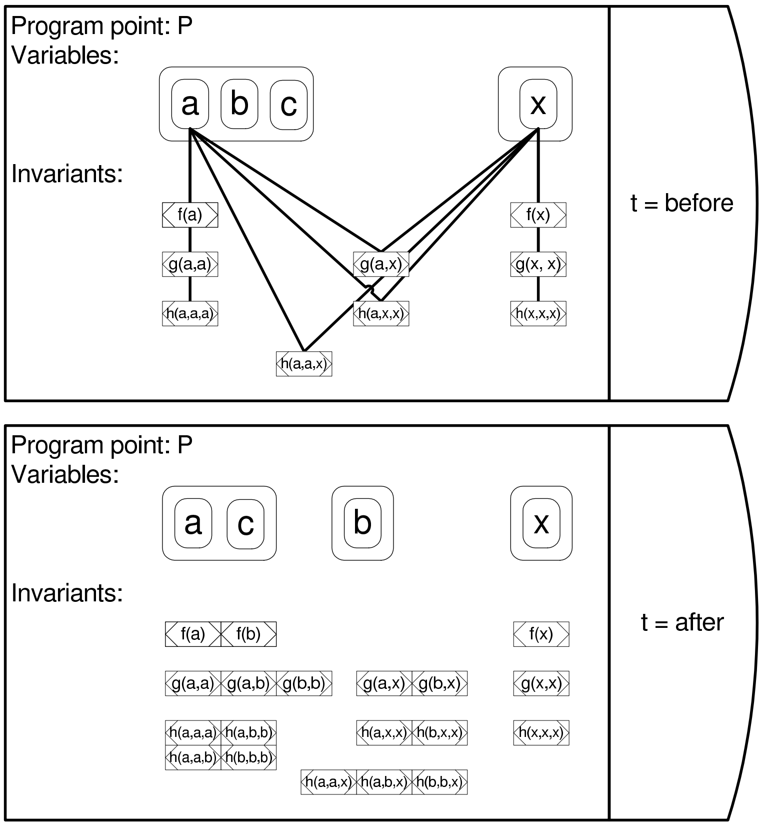

Figure 5-7: Solving the copying problem by keeping invariants on the same variable. Lines connecting invariants to variables have been removed for clarity.

Original invariant	Instances of leader	Copied invariants
$f(a)$	1	$f(b)$
$f(x)$	0	
$g(a, x)$	1	$g(b, x)$
$g(a, a)$	2	$g(a, b),\ g(b, b)$
$h(a, a, a)$	3	$h(a, a, b),\ h(a, b, b),\ h(b, b, b)$
$h(a, a, x)$	2	$h(a, b, x),\ h(b, b, x)$
$h(a, x, x)$	1	$h(b, x, x)$
$h(x, x, x)$	0	

Figure 5-8: Copying of invariants using combinations with repetition, where a is the old leader and b is the new leader.

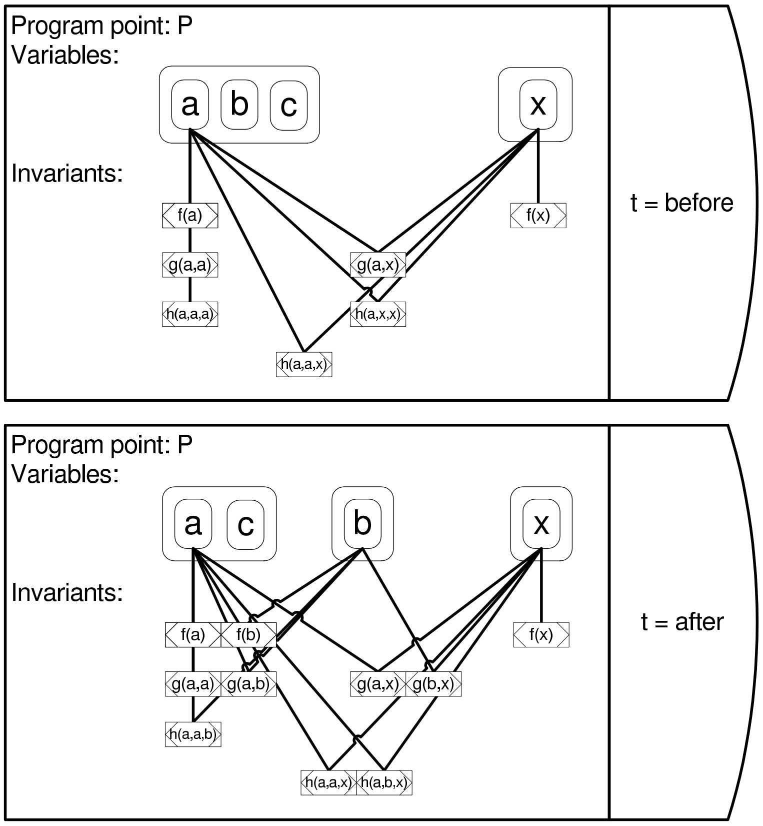

Figure 5-9: Optimizing the copying and reflexive process. Some invariants with more than one instance of a variable do not have to be instantiated.

This method enumerates at least the invariants that would be instantiated in method 1, and has adequate information from keeping around invariants within the same equality set. We use combinations with repetition rather than those without because otherwise $g(a, a)$ would not appear. We use combinations rather than permutations because we assume the data structures for invariants are designed so that g will work regardless of the order of the variables (i.e., g tracks both $g(a, b)$ and $g(b, a)$ if needed, as the Daikon system does).

It might appear at first that this method instantiates an unnecessary number of invariants. However, we perform the following optimization: if a variable occurs k times within an invariant, the variable's equality set must have at least k members. Otherwise, the invariant is no longer interesting and is destroyed (during run). This is shown in Figure 5-9. In the example, if b is the only member of its equality set, then the invariants $g(b, b)$, $h(b, b, b)$, $h(b, b, c)$, $h(b, b, d)$ are destroyed. Thus the number of invariants is bounded by the number of invariants without equality optimization.

5.4.2 Sample counts

For statistical tests, invariant inference systems like Daikon track the number of sample values seen for a variable. For each variable, at each sample data point, there is one of three pieces of information: that the data is 1) present and was assigned to since it was last seen;

2) present but unmodified; 3) missing. Data can be missing for reasons discussed in Section 5.4.3. The count of the number of modified and unmodified samples is used to later classify invariants as justified or not (there is no need to track missing samples).

Modified/unmodified/missing data is also counted for each group of k variables that has invariants generated. For a k-tuple, if any variable is missing, then the tuple is considered missing. Otherwise, if any variable is modified, then the tuple is considered modified. Thus each invariant can perform a more accurate statistical test.

For statistical tests to continue to yield the same results under equality optimization, we must provide the right sample data to variables and invariants, even though the variables are combined into equality sets so not all invariants are instantiated to keep track of data. We use the following conservative mechanism: if any of the values in an equality set is modified, then the equality set's leader is considered modified. Further, when invariants are copied (due to equality sets splitting), sample counts are also copied.

Missing values have to be handled specially, as explained in the next section.

5.4.3 Missing data samples

Sometimes, execution data for invariant inference contains *missing* samples. This can happen, for example, for variables that are nonsensical in some cases, such as $s.a$ when s is null. In the naive implementation of inference, invariants ignore missing data samples. This is sound for the naive implementation because a missing sample means there is inadequate information to contradict an invariant. However, with equality optimization, either ignoring missing samples or handling missing samples like modified/unmodified values is unsound.

Figures 5-10 and 5-11 explain this. Consider a program point with two variables, a and b, and a unary invariant f. Initially, a and b are equal, a is the leader, and $f(a)$ holds. Now a sample arrives that has a missing, but contains a value for b that contradicts $f(b)$. We cannot falsify $f(a)$ because f might hold for a (and when a is missing, this is still true). We cannot maintain $f(a)$ and say this implies $f(b)$ because we have seen a falsifying sample.

The sound solution is to split the equality set $\{a, b\}$ and instantiate a two-way equality invariant between a and b, as shown in Figure 5-12 In fact, when two-way equality invariants are used during initialization, the instantiation is unnecessary: the copying mechanism produces the right results. The general rule, then, is to split off all variables with missing values from the old equality set into a new equality set and to use the copying mechanism as usual, while including two-way equality invariants in the ones instantiated during initialization. Not including two-way equality invariants would be unsound, as the invariant $a = b$ would still hold in the example since a was missing.

5.4.4 No more non-instantiation of invariants

One drawback of equality optimization is that a performance optimization done in data flow is no longer correct (it is incomplete). Data flow does not instantiate invariants in a program point if a higher program point subsumes the invariants. However, the example in figure 5-13 shows how this is now incomplete. The upper program point contains variable a_{upper} and the lower contains variables a_{lower} and x. The two as are connected by flow, while initially

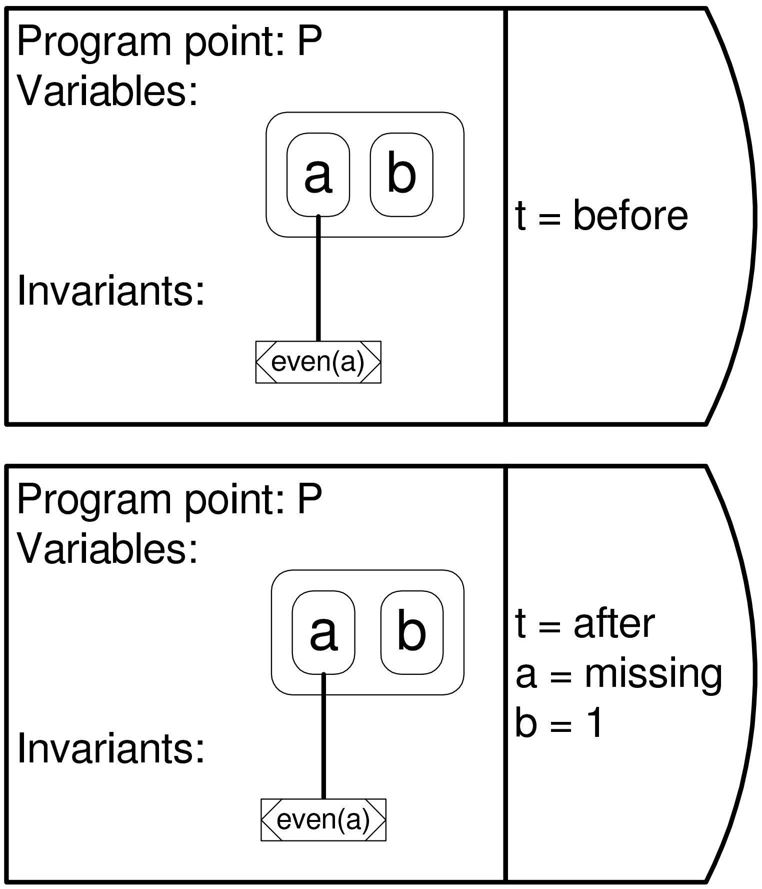

Figure 5-10: It is wrong, as shown here, to assume $even(b)$ holds even though a is missing. It could be the case that b is not even.

$x == a_{lower}$ with a_{lower} as leader. If the standard data flow non-instantiation is used, then we initially only instantiate invariant $f(a_{upper})$. $f(a_{lower})$ is not instantiated due to dataflow optimization, while $f(x)$ is not instantiated because x is not the leader.

Say a sample first arrives that shows x is not equal to a_{lower}, followed by a sample that falsifies f on the as but not on x. An example of this could be if f means "even" and the samples seen are $(2, 2)$, $(2, 4)$ and $(3, 4)$. Since a_{lower} has no invariants, when the inequality sample is seen, x is split off but no invariants are copied. Then when a_{upper} has f falsified, f flows to a_{lower}, where it is also falsified. However, x never gets f, so now the output from Daikon is wrong (i.e., incomplete). Note that if f is falsified before the inequality is seen, then correctness is preserved, since $f(a_{lower})$ is first created, and then copied as $f(x)$ when inequality happens.

One way to fix this dataflow problem is to make the non-instantiation rule no longer apply. That is, all leaders at all program points now instantiate invariants, even if another program point would eventually flow the invariant down. In the above counterexample, this approach is sound because $f(a_{upper})$ and $f(a_{lower})$ both exist, so when the unequal sample is seen, $f(x)$ can be copied.

This fix increases the space used for invariants, but does not add much to the time, since

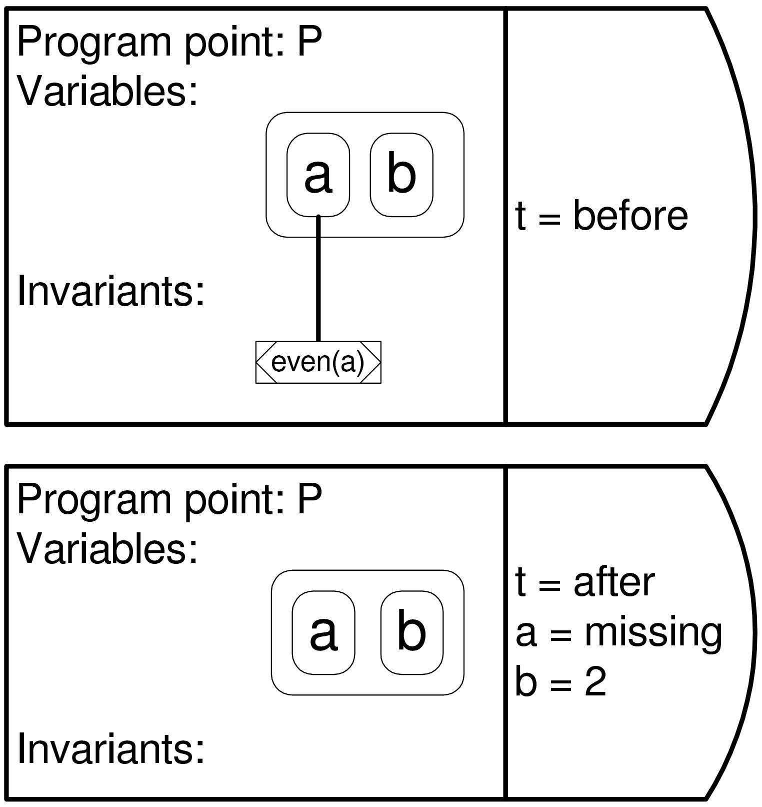

Figure 5-11: It is wrong, as shown here, to assume $even(b)$ does not hold when a is missing. It could be the case that f is even.

suppression optimization (Section 5.5) prevents them from being checked. We also claim that the number of invariants eliminated by the equality optimization dominates the number that would be eliminated by data flow, especially since any invariant that weakens cannot use the data flow mechanism.

An alternative way to fix this problem is to search for all invariants that are on the old leader in all the parent program points, and to copy the invariants onto the new leaders, In the example in the figure, x would have f copied to it when x is split from a. This is shown in Figure 5-15.

5.4.5 Post processing

The basic step in post processing was described in the overview of equality optimization in the start of Section 5.4 — equality sets are converted into two-way equality invariants. However, it is often the case that some variables are not interesting for printing, and these variables may be the leaders of equality sets, in which case none of the invariants on any of the variables will be printed. For example, $a == b$, and a is the leader, but a is not interesting. In that case, $odd(a)$ will not be printed (this is correct) but neither will $odd(b)$ (this is wrong). Thus, before the post processing conversion stage, some equality sets are *pivoted*. If the leader of an equality set would not be printed, the leader is switched to one

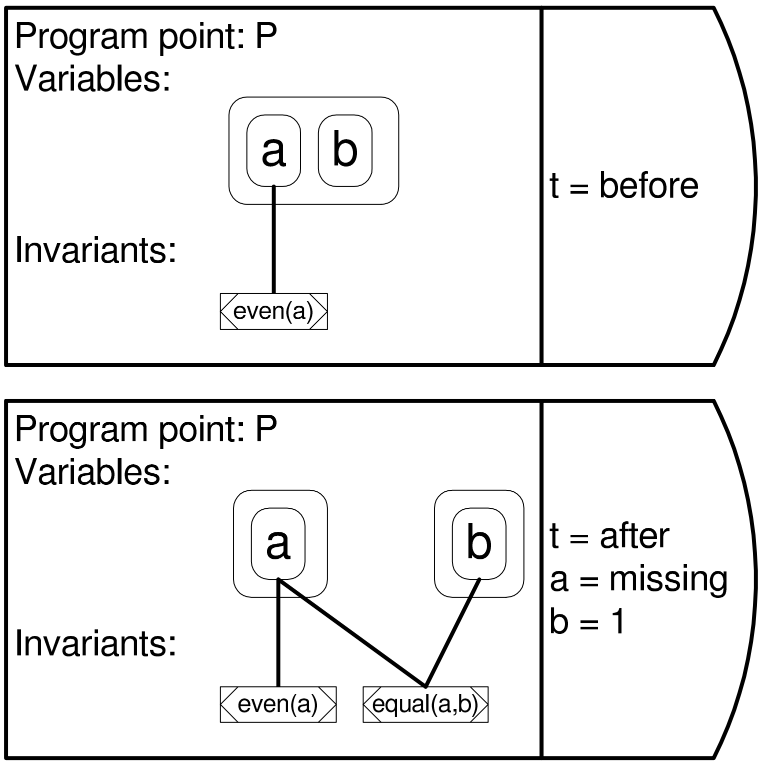

Figure 5-12: The sound way to handle missing values: keep a two-way equality invariant between a and b and split off the equality sets.

that would be (if any exists). The invariants attached to the former leader are switched over to the new leader.

5.5 Suppression optimization

An invariant is *suppressed* if it is implied by some set of other invariants. Suppression optimization attempts to save time by not checking suppressed invariants. It performs dynamic, positive elimination only, but is general enough to work for all logical implications. Suppression and equality optimization are the only forms of dynamic suppression used in the Daikon tool. Suppression saves times and reduces clutter from printing, but it does not save space: recall that for positive elimination, it is not possible to entirely eliminate an implied invariant. With equality optimization, we keep track of invariants implicitly, since the copying mechanism is adequate for re-creating invariants. With suppression optimization, since it works for all implications in general, it is necessary to actually keep the suppressed invariant.

A *suppression link* connects an invariant and its suppressors. Using suppression links entails two major changes to the inferencing algorithm (we will assume we are changing it after equality optimization has been added). First, each invariant is given *suppression factories* that describe what kind of logical conditions can suppress it. Second, the invariant checking mechanism is modified to save time by using suppression links.

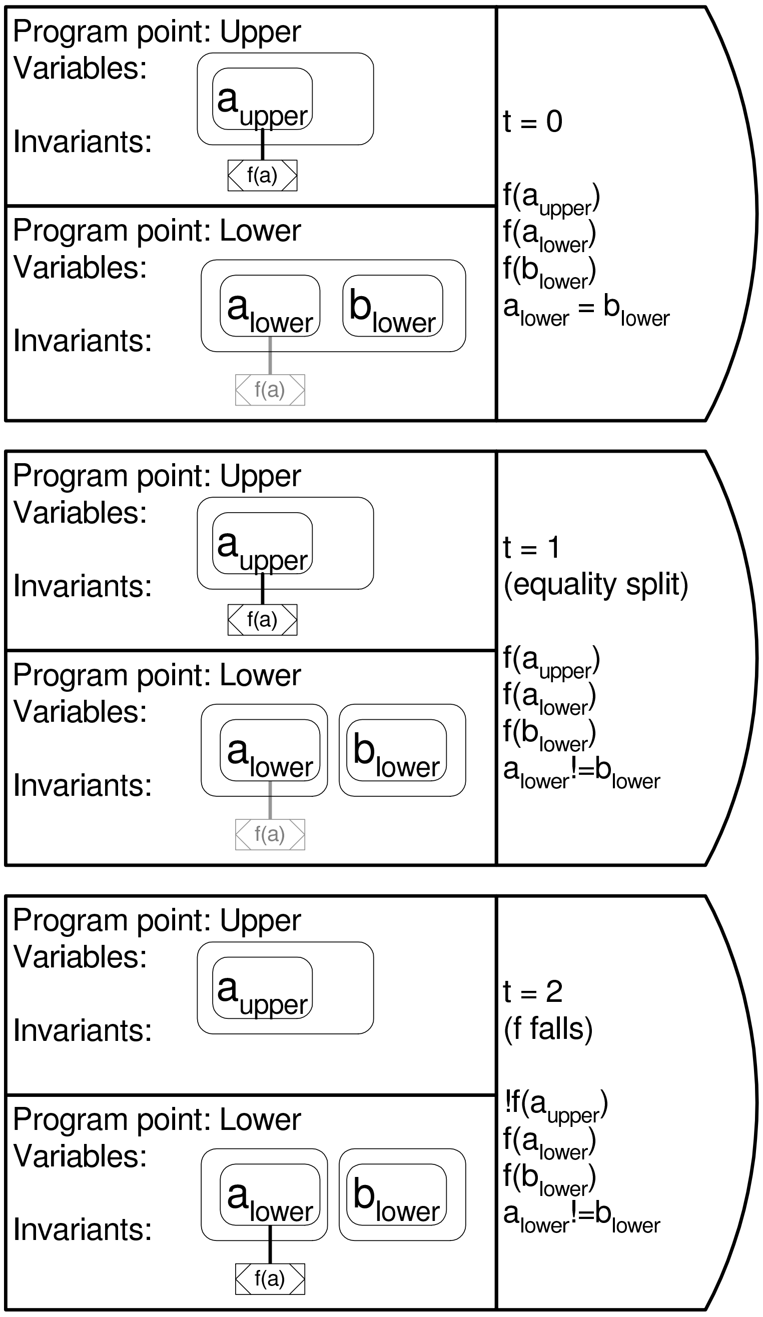

Figure 5-13: The problem with not instantiating invariants and relying on the data flow hierarchy. Problem with non-instantiation of invariants: since $f(b)$ is never instantiated, if x splits off from a in LOWER before f drops from UPPER, then x will never have f as an invariant.

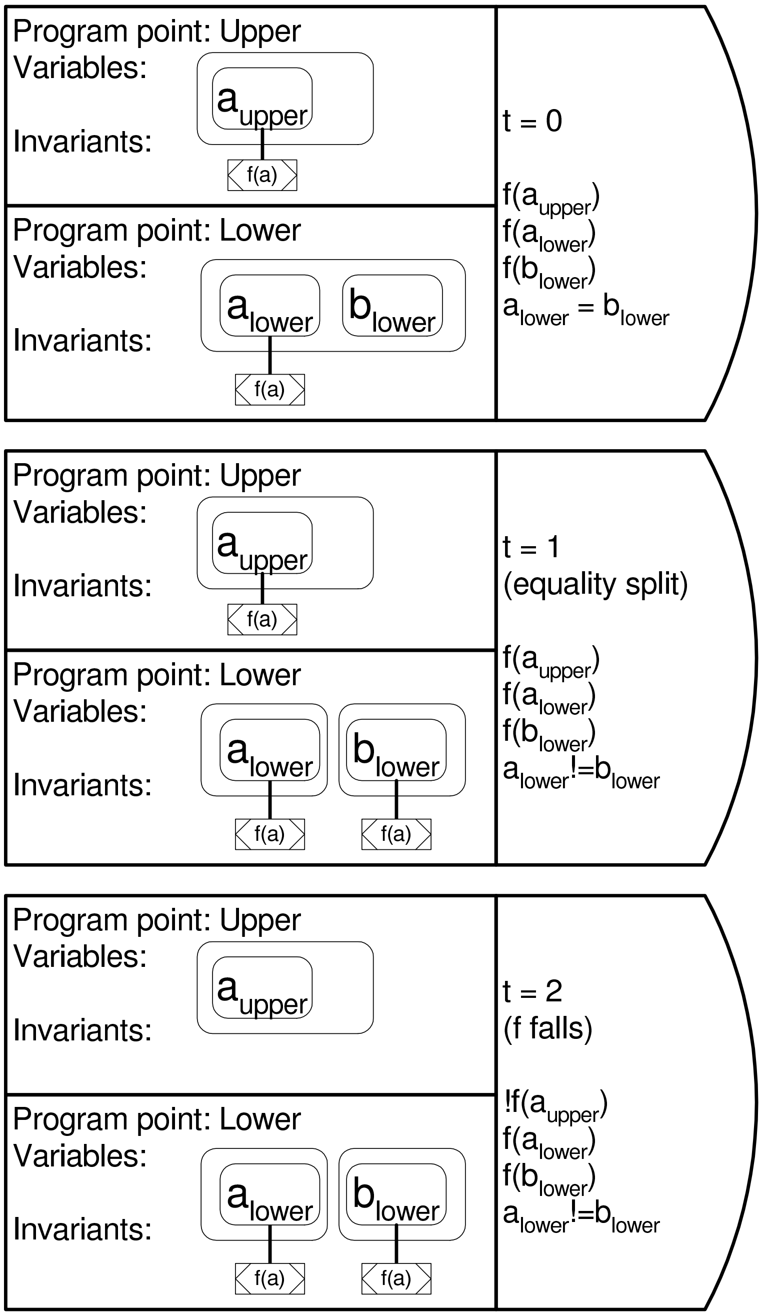

Figure 5-14: One way to correct the problem of non-instantiation: instantiate invariants that would hold on equality leaders at every program point.

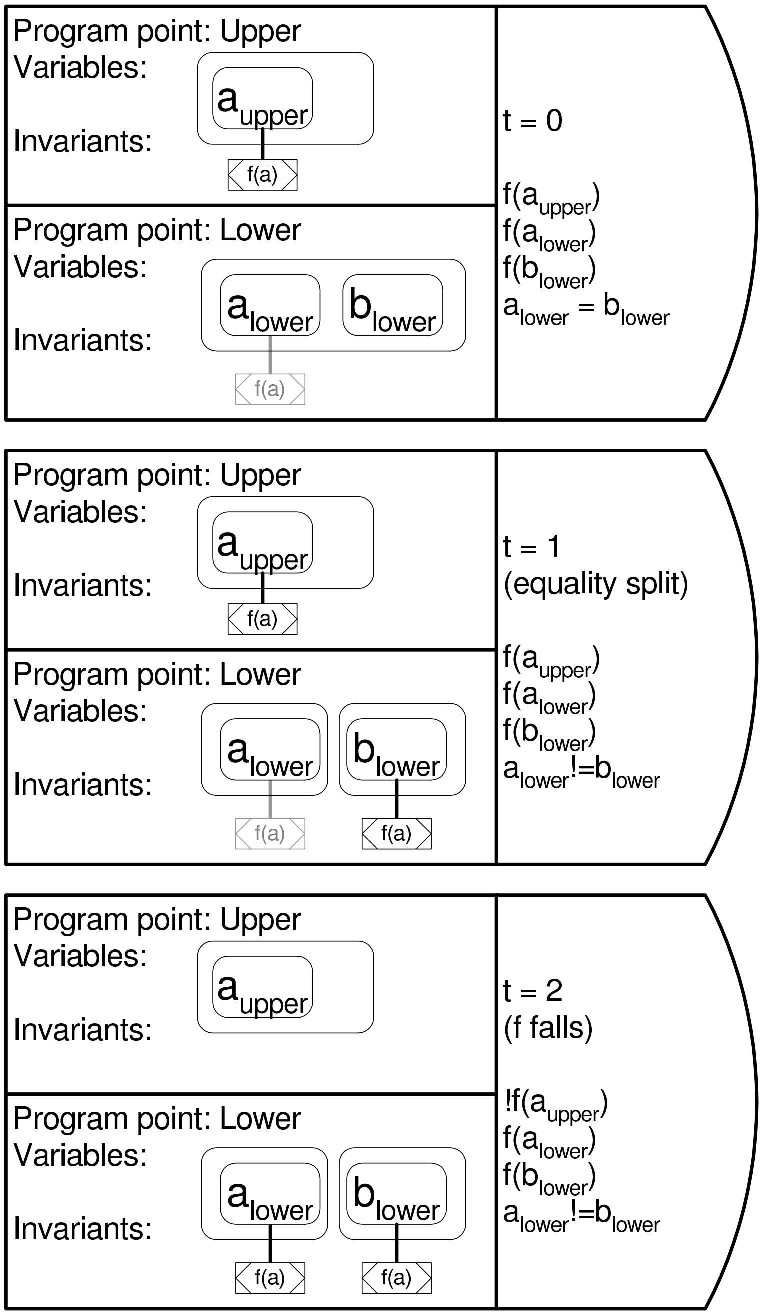

Figure 5-15: Another way to correct the problem of non-instantiation: copy relevant invariants from upper program points upon splitting the equality set.

When an invariant is copied over to a new equality set, we do not copy over suppression links. Instead, we re-attempt suppression on the copied invariant, as shown in Section 5.5.2.

5.5.1 Suppression factories

The conditions for suppression are dependent on the type of invariant and the structure of the variables suppression works on. For example, $A[i] \in B$ is implied by $A \subseteq B$ — this suppression depends both on the fact that the suppressed invariant is an "element of" invariant and that one of the variables is a location in an array.

Suppression factories are attached to the invariant they might suppress, since each suppression link can connect to multiple suppressors but only one suppressee. When asked to attempt suppression on an invariant, a suppression factory generates a *suppression template.* A suppression template consists of a list of invariants and variables on these invariants. If there exists these invariants on the variables, then the suppression template is *filled*, and the suppression factory is allowed to suppress the suppressee. The suppression template for the above example would be $[\langle \subset, [A, B] \rangle]$, consisting of one invariant and variable group. The conditions for suppression are other invariants, including equality sets.

The most important complication to filling a suppression template is that suppression can happen across program points via their data flow connections. Recall that every invariant that holds on a parent program point holds on the child. Thus, we also want to scan all the ancestor program points of an invariant, in addition to the invariant's program point, when looking for suppressors, while remapping the variables via their flow connections. An invariant does not have to have all its suppressors in the same program point.

Lastly, we also use suppression to hide invariants implied by data flow that the data flow mechanism cannot catch. With equality, we can no longer leave invariants uninstantiated if parent program points have them. However, we can use suppression to save time on checking these invariants, since when equality sets split, invariants are cloned but not their suppression factories.

5.5.2 Handling suppression during Daikon's run

Below is a description of how suppression is integrated into Daikon's inferencing loop. Even though suppression links may connect across program points, suppression itself is examined in the context of one program point. For a program point:

1. When invariants are instantiated, attempt to suppress them. An invariant is suppressed when any suppression link can be created for it, but we only keep track of one suppression link.

2. During inferencing, check only the invariants that are not suppressed with values.

3. When an invariant is falsified, collect all its suppressees, and remove their suppression links. If the falsified invariant was suppressing an invariant with another invariant, the suppression link is still removed since it is the conjunction that implies the suppression. Take all the formerly suppressed invariants, and attempt to re-suppress them (using

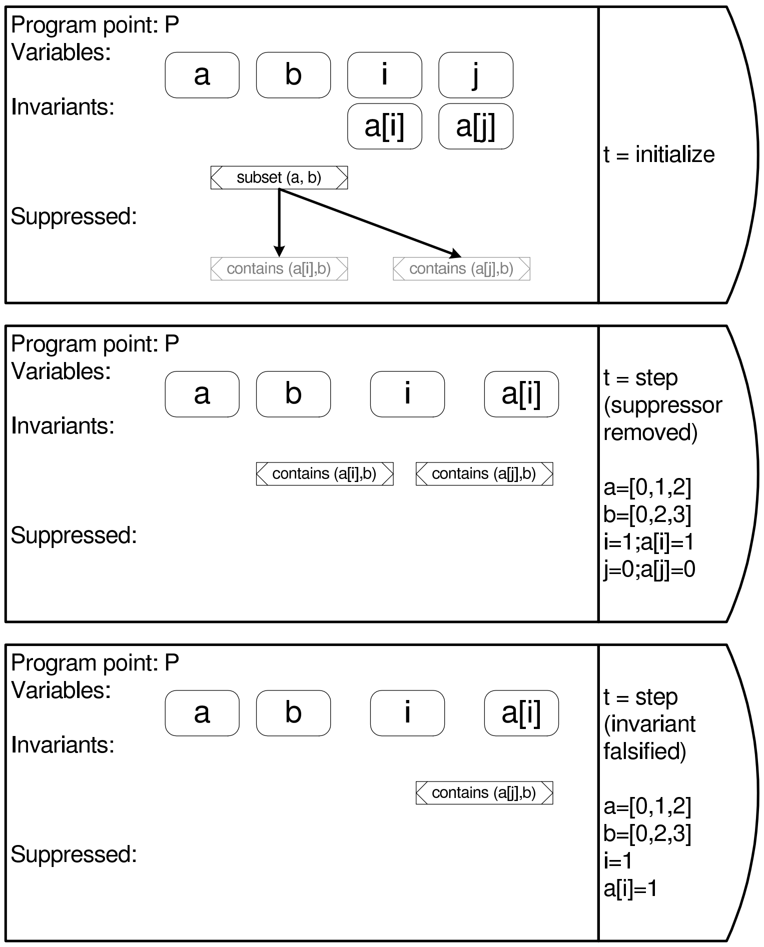

Figure 5-16: Suppression in action for the logical implications $A \subset B \Rightarrow A[i] \in B$ and $A \subset B \Rightarrow A[j] \in B$

unfalsified invariants). For any unsuppressed invariants that remain, check them with values if the invariant was in the same program point as the former suppressor. We do not check invariants in different program points because they have to be in child program points and will be checked when values arrive.

4. Before invariants are printed (i.e., during post processing) attempt to suppress each invariant again.

The above algorithm ensures the following property:

Theorem 5.5.2.1 (Suppression never falls): A suppressed invariant can never flow to a lower program point.

Figure 5-16 shows suppression in action. The variables are the arrays a and b, integers i and j and derived variables $a[i]$ and $a[j]$. Initially, $A \subset B$, so this implies that $A[i] \in B$ and $A[j] \in B$ have to hold. A suppression is established connects $A \subseteq B$ to the two suppressed invariants. Should any samples arrive that preserve $A \subset B$, the two suppressed invariants will not be checked.

At a later time, a sample is seen that falsifies $A \subseteq B$ (and $A[i] \in B$). The two invariants that were suppressed now become regular invariants. They are checked against the data sample, and $A[i] \in B$ is falsified, while $A[j] \in B$ is preserved.

5.5.3 Suppressors that do not have their values set

Another issue that arises with suppression is that some suppressors may be invariants that set their parameters only after a few samples have been read. For example, in invariants of the form: $y = ax + b$, a and b are not set until at least 2 samples of (x, y) have been observed. We know that $y = x + 4$ can suppress $y > x$ but it is incorrect to follow either of the following approaches:

- Connect $y = x + 4$ to $y > x$ via suppression links at the beginning. This is incorrect, as we could encounter just one value for (x, y), $(1, 1)$ that does not set a and b but contradicts $y > x$. This would be unsound.
- Not connect $y = x + 4$ to $y > x$ via suppression links. In this case, the connection will never form, even when $a = 1$ and $b = 4$. Since it is the suppressor that is changing, the potential suppressee will not know when to look for the change, as the two invariants are not connected.

The only way to have the suppression link form at the right time is to check occasionally whether $x > y$ is suppressed. One such way to do this is to exactly look for when an invariant of the form $y = ax + b$ changes, but this would require adding special case handlers to every suppressor that changes. Instead, it is easier to run a general suppression check (as done during initialization) periodically. If the suppression check is done using exponential back off (every 10, 100, 1000, etc. samples) then the time cost is minimal, as most changing invariants have their parameters set early.

5.5.4 Suppression cycles

Lastly, there is a question of whether suppressed invariants can suppress other invariants. The advantage of allowing this is greater suppression, the danger the presence of cycles. If there is a cycle, then a set of invariants may never be falsified. We allow suppressed invariants to be suppressors, but check (at design time) that all the suppression factories we have contain no logical cycles.

5.6 Conditional invariant detection

Recall that the grammar of any dynamic invariant detection system must be limited because there are an infinite number of invariants that are true of a program. For example, the Daikon system limits itself to invariants that contain at most 3 variables as the atoms of its grammar. This limitation is a tradeoff between the general usefulness of the invariants and the extra time (both computer and user) involved in processing them.

However, setting a hard limit like 3 variables is not sufficient for some applications. Thus, the Daikon system supports a limited extension of its grammar by allowing for boolean combinations of atomic invariant units. For any two boolean values, there can be $2^4 = 16$ different binary functions, but it is not useful to have all these functions. Daikon allows the checking of two specific forms [DDLE02]: $A \Rightarrow B$ and $\neg A \Rightarrow B$. These are useful because implications are commonly used to describe program behavior, especially in exceptional cases.

In terms of terminology, we shall call the left hand side the *condition* and the whole invariant a *conditional invariant.*

However, even with potentially useful implications, adding just one type of boolean expansion could expand the grammar by squaring the number of invariants. Not all of these invariants would be useful. Daikon thus decides on which invariant atoms to choose as left hand sides. One mechanism for deciding this is to let the user specify. Other mechanisms, such as examining boolean predicates in the program syntax and data clustering analysis have been attempted [DDLE02]. This section shows how to efficiently run dynamic invariant detection, despite the seeming explosion in the number of invariants that implications introduce.

The main issue in detecting $A \Rightarrow B$ is the waste of computation if A or B are always true. Using a non-incremental, multiple pass approach to this problem is easy: after dynamic invariant detection is run the first time for atomic invariants, we only instantiate conditional invariant $A \Rightarrow B$ if A and B are not invariants themselves. In Daikon V2, this is implemented via *conditional program points*, virtual program points that have been created to represent all the possible invariants on the right hand side of the implication assuming A is true. Conditional program points are passed in the right subset of the data for their conditions (i.e., all the samples where A is true) and then each invariant B in the conditional program point is used as before.

5.6.1 Incremental detection of conditional invariants

In an incremental approach to dynamic invariant detection, creating conditional invariants or conditional program points cannot be done after data samples have been read. A naive incremental approach would be to simply instantiate all conditional invariants, but this would result in the resource consumption presented above. An optimizing approach is to take advantage of the optimizations that are already used in the Daikon system. The key insight here is that the hierarchy established between program points also works for conditional program points.

At the start of invariant detection, when the program point hierarchy is created, conditional program points are created just like regular program points. Each conditional program point $PptCond_i$ is connected to two groups of program points via the $\sqsubseteq_D$ relationship: 1) the regular program point without the condition Ppt_i; 2) the conditional program points $PptCond_j$ for which the corresponding regular program points are in the relation $Ppt_i \sqsubseteq_D Ppt_j$.

This connection is precisely in accordance with the $\sqsubseteq_D$ relation. $PptCond_i \sqsubseteq_D Ppt_i$ because any any data samples seen under a condition should be seen by the program point that subsumes all samples. $Ppt_i \sqsubseteq_D Ppt_j \Rightarrow PptCond_i \sqsubseteq_D PptCond_j$ is true because the upper program point Ppt_j claims to see all samples at the lower program point Ppt_i. This new rule is also necessary because it is not the case that $Ppt_i \sqsubseteq_D PptCond_j$: the condition may not be true at Ppt_i. Since the $\sqsubseteq_D$ ordering takes the transitive closure of the rules, we are sure that $PptCond_i \sqsubseteq_D Ppt_j$. These two new rules together establish a lattice (rather than a tree) for data flowing up from a particular sample under a condition, but this is already handled by the hierarchy.

Once the hierarchy is established, invariants are instantiated and checked under the same rules as before. During checking, on corresponding variables between related program points, the upper invariant will shadow the lower one. Thus if B is always true, the invariant $A \Rightarrow B$ will not be checked. $A \Rightarrow B$ will be checked precisely when it needs to be: when B is destroyed at the upper program point.

There is one minor change to the operation of the system during checking: samples are no longer always inserted at exactly one program point: for each regular program point, many conditions could hold. However, performing multiple insertions is not a problem since the system is already capable of handling multiple sample flows to one program point. At each insertion, memoization can be used to make sure that a sample is not processed again at a program point.

One more optimization can be done to the system to efficiently handle the case where A is always true. While A remains true, it is not necessary to keep the conditional program points for Ppt. Thus, the creation of the hierarchy could be done dynamically, once A is falsified. The rules would be:

- For $PptCondA$, the conditional program point where A holds, copy no invariants, since the invariants in the two program points must be identical at the moment of A's falsification and $PptCondA \sqsubseteq_D Ppt$. The condition must be checked first however, because we want to copy the invariants that exist *before* the falsifying data sample is seen.

- For $PptCondA'$, the condition program point where A is false, instantiate exactly the invariants not in Ppt. Since until now, A has never been false, everything must be true at $PptCondA'$. However, we do not need to copy the invariants in Ppt since $PptCondA' \sqsubseteq_D Ppt$.

These rules soundly and incrementally performs conditional invariant detection with the same optimizations as in Daikon V2. If A is always true, then the conditional program points are never created. In fact, the rules are advantageous over a multiple pass approach: rather than instantiating the conditional program point for A and $\neg A$ early on, when there will be many other invariants there are true, the incremental optimizations create the conditional program point only when needed.

5.7 Function parameters

During the implementation of dynamic invariant detection for the IOA language, it was discovered that many invariants were obvious because the variables involved were parameters to transitions. To reduce clutter for the user, we developed a mechanism to eliminate certain invariants on parameter variables, and ported this technique to the semantics of Java and C function parameters.

A transition parameter in IOA is immutable. This means that the values of the variable before and after the transition (called the pre- and post- values) will be equal and any invariant on the post variable is uninteresting because it will be present on the pre variable. The same is true for any derived variable of the post variable.

In Java and C, function parameters are mutable, but some ways of changing them render them uninteresting. Consider a data structure s with field $s.a$, passed into function $f()$. If the value (of the pointer) s is changed, there are two cases: 1) s is set to a new structure; 2) s is set to an existing data structure. Case (1) cannot be seen by the caller, unless s is also in the return of f, in which case the return value will have the relevant information. Case (2) may be seen by the caller, but the information is useless unless the new value of $post(s)$ is in a data structure visible to the caller. If this is the case, then the data structure will contain all the necessary information. So for both cases, $post(s)$ is not interesting.

If the value of s is not changed, but the value of $s.a$ is changed, this effect may be visible to the caller. In this case, $post(s.a)$ is interesting, even though $post(s)$ is not.

In implementation, language front ends are responsible for labeling variables as parameters. Any parameter and derivation done by Daikon of a parameter is treated like $post(s)$ above and is not interesting. Any variable issued by front ends that is related to a parameter variable is labeled as a secondary (or front-end derived) parameter. In the example above, this is true for $s.a$ — the front end does not label it as a parameter, but Daikon labels it as a secondary parameter. Post states of secondary parameters are interesting if the parameter variable they are related to have not changed. Since parameters in IOA are deeply immutable, the IOA front end labels all secondary parameters as primary parameters.

Since determining whether secondary parameters are interesting depends on the equality between the pre- and post- variables of the corresponding primary parameters, suppressing invariants that contain secondary parameters is a dynamic suppression. In contrast, suppressing invariants with primary parameters can be done statically. However, the equality set between the pre- and post- variables cannot be eliminated, since it will be used for determining if secondary parameters are interesting.

5.8 Future work

automatically generate (or check) suppressors

(do more ambitious experiments using V3's capabilities)

implement online

5.9 Conclusion

Chapter 6

Improving the IOA Simulator

This chapter covers improvements that were made to the IOA Simulator so that it would be better suited for executing IOA programs, especially in relation to dynamic invariant detection. We describe three major improvements:

Simulating paired automata How the semantics of paired simulation was changed to make it identical to the semantics in the proof tactics that are generated by the translator tools.

Handling quantifiers How the IOA simulator handles quantifiers in the language.

Connecting to the Daikon tool How the IOA Simulator passes data to the Daikon tool such that data structures in IOA become data structures understandable by Daikon.

6.1 Simulating paired automata

The `proof` block of the IOA language syntax specifies how the IOA Simulator executes a specification automaton given an implementation automaton's transition. In a previous version of the IOA Simulator, simulation of paired automata was implemented so that only the post-state of the implementation automaton was visible to the `proof` block. Further, with each `fire` statement in the execution, the specification automaton would change state, so that any `if` statements that examined the state of the specification automaton afterwards would be doing so on the state after the `fire`. This made the semantics of the IOA Simulator different from the execution given by the proof translator tools. The proof translator tools assume that any examination of the state of either automaton would be the pre-state of both automata. For example, in the Paxos case study, one of the `proof` actions was:

```
for internal assignVal(b: Ballot, v: Value) do
  if ¬(b ∈ Global1.succeeded) then
    ignore
  elseif ∃ b:Ballot (b ∈ Global1.succeeded ∧ Global1.val[b] ≠ nil) then
    ignore
  else
    fire internal chooseVal(v)
```

```
fi od
```

The `if` statements would test the post-state of `Global1` rather than the pre-state, as desired in the proof. And after the `fire` statement, `Cons` itself would be changed, so any subsequent `if` statements would check a different state. With this semantic mismatch, paired executions cannot not match simulation relation proofs, so our verification methodology cannot be used. Further, users of the IOA tools should be able to expect a consistent handling of `proof` blocks.

Other than the semantic mismatch, another limitation of this design is the lack of information for the `proof` block code. There is no way the code can tell what the pre-state is, since there could be multiple pre-states that map to the same post-state. Choosing the correct β execution might depend on the pre-state.

Thus, the IOA Simulator was altered such that its paired execution semantics match what the proof translator tools would output. All references to automaton state variables now refer to the pre-state of the automaton. The drawback is that we cannot directly observe the post-state. However, it can be deduced from the action that has been fired and its parameters. If there is any explicit nondeterminism in the post-state, it is captured in the local variables. Ideally, we would want to be able to refer to both the pre and post-states in the IOA Simulator, but the IOA language itself does not permit this dual reference.

6.2 Handling quantifiers in the Simulator

It is generally undecidable for the IOA Simulator to determine certain predicates, since the IOA language allows quantifiers on infinite data types to be used (though they are usually used in preconditions or `choose` blocks). For example, the IOA Toolkit allows the use of multiplication and addition in quantified expressions over the integers. Even without quantifiers, if a transition contains a local variable, the local variable is implicitly (existentially) quantified. nevertheless, we still want to be able to simulate most IOA programs. We therefore wish to develop two ways of handling quantifiers. One is sound and the other unsound.

6.2.1 Sound execution

Any expression without a quantifier (or with a quantifier over a finite data type) is defined as *executable*. The sound method relies on the fact that certain quantified expressions are decidable through iteration on the possible values that the quantified variable can take. If S is a finite set and P is executable, the following two expressions are executable:

$\forall_e(e \in S \rightarrow P(e))$

$\exists_e(e \in S \wedge P(e))$

The expressions are executable because iterating through the set S is sufficient to determine the truth of the quantified expression. If a program contains only such limited syntactical forms, then the IOA Simulator can execute them. In practice, these forms are used quite often, because the programmer often means to say that every object in a set obeys a given property, or that there exists an object in a set that obeys a given property.

Note that the two formulae above are actually instances of a more general set of two-way boolean expressions of the form:

$(\neg)?A$ `op` $(\neg)?B$ where `op` is either $\vee$ or $\wedge$, A is $e \in S$ and B is executable. There are 4 forms of each expression for each operator, and thus 8 total. Only the two presented above are executable.

There may be more such soundly executable expressions that are worth discovering. It may be necessary to syntactically transform some executable expressions so that they can be recognized by the IOA Simulator as executable.

6.2.2 Unsound execution

The unsound method iterates through some large values of the infinite datatype and hopes that seeing these is enough to determine the truth value of the quantifier. The method is sound when a positive result is found for an existential quantifier or a negative result for a universal quantifier, but unsound in general. For instance, it would be difficult to determine the values of x, y and z to check for the following expression:

$\exists_{x,y,z:\texttt{Int}} x^3 + y^3 = z^3$

The disadvantage of using the unsound method is that it is possible to have an execution in the IOA Simulator that would not be a valid execution according to the code. However, we still permit unsound execution in order to allow for program testing.

In practice with IOA programs, we have found that the quantifiers used are on a specific data set that is finite *within the context of a particular execution*. For example, in the Paxos case study, we speak of quantifiers on all ballots. However, we know exactly how many ballots are currently in existence. Thus it is sufficient to simply check ballot quantifiers on these ballots. The sound method used above is not sufficient for the Paxos case study, for there is no data structure that keeps track of all the existing ballots, including future, yet-to-be-created ones.

6.3 Connecting to the Daikon invariant detector

This section discusses what changes were made to the IOA Simulator in order to have it output useful data in the input format of the Daikon tool. The key problem here is that the data structures in IOA do not match the data structures in the Daikon tool. Since we use the Daikon tool to analyze IOA executions, we need to better connect the two data representations.

A general problem in linking specifications and automated tools is that specifications tend to have greater notational complexity than implementations or executions. Whereas tools are best at reporting simple properties of simple data structures, specifications may express sophisticated properties of complicated domain-specific data structures. We therefore extend Daikon's grammar for checked invariants to include properties and data structures relevant to IOA specifications.

The Daikon invariant detector operates over basic datatypes: integers (scalars and hashcodes), strings, and sequences. This keeps its implementation simple, fast, and portable to many different programming languages. IOA specifications use more sophisticated datatypes, such as sets, tuples, and maps between arbitrary types.

In our translation between the two tools, we alter the IOA Simulator so that complicated data structures are represented using scalars and sequences. For example, sets become sequences. Since Daikon ordinarily tests sequences for duplicates and order-related properties, which are irrelevant for sets, we extend Daikon to recognize when a sequence is marked as representing a set and Daikon avoids these tests.

The next two subsections present the most important data translations done by the IOA Simulator. First, we look at how IOA maps, or functions from a domain data type to a range data type, are translated into Daikon data structures. Second, we look at how conditional invariant detection, as described in Section 5.6, can be implemented for IOA programs.

6.3.1 Translation of map data structures

Map data types, such as Array and Map are used in many IOA programs. A map is a function that connects a domain data type to a range data type. Often, the domain data type is not totally ordered, so the map cannot be presented as a simple array to Daikon.

The IOA Simulator map data types in two ways. First, it linearizes the range of the map, and reports the values that are present. For an array `m`, the Simulator thus reports `m.values[]` as an array to Daikon. Second, the Simulator samples key, value pairs of a map m using two distinct random keys named *anIndex* and *anotherIndex*. The output to Daikon is two derived variables (see Section 5.1.1) *m[anIndex]* and *m[anotherIndex]*. We extend Daikon to report invariants it detects involving such synthetic variables as being universally quantified on the variables. For any map that has a range with a null element (i.e., a pointed range data type), we introduce another synthetic variable to represent the elements of the map that are not null.

We used mechanisms such as these in our case study involving the Paxos algorithm, which uses a map `voted` from participating nodes to sets of ballots. Without enhancement, Daikon reports few interesting properties of this data structure, which is inherently two-dimensional and involves sets of nodes and ballots represented as sequences. With information about the type of this data structure, and with synthetic variables `voted[anIndex]` and `voted[anotherIndex]`, Daikon is able to detect and report properties useful for proofs, such as:

```
∀ anIndex:Node (size(voted[anIndex] ∩ abstained[anIndex]) = 0)
```

6.3.2 Conditional splitting in IOA

As mentioned in Section 5.6, Daikon is capable of detecting conditional invariants, or invariants with implications. However, choosing which conditional invariant to search for is difficult to do without overly expanding Daikon's search space. Daikon leaves it up to other tools to choose the condition, or left hand side, of implications. This section shows how conditions are chosen in IOA.

A simple syntactic analysis suffices. The Simulator uses each clause (conjunct) in a precondition as a condition predicate. We replace transition parameters, which appear in preconditions but are not in scope at the automaton level, by special the variable `anIndex`. The quantification technique discussed above ensure that the resulting expressions are sensible. For example, in the Peterson case study (Figure 4-2), the `checkFlag(p)` precondition becomes the condition `pc[anIndex] = trying2`. This condition is used in the Peterson invariant:

```
∀ p (pc[p] = trying2 ⇒ pc[turn] = trying2)
```

Thus, Daikon is able to report invariants that indicate properties of particular points in the code despite performing invariant detection on the system as a whole.

Chapter 7

IO automata in Isabelle: enhancing the representation of I/O automata in the prover

Every theorem prover requires a *prover model* in its language for the system model it is to verify. For example, Garland and others developed the model for I/O automata in the Larch Prover [SAGG$^+$93b]. A prover model specifies the semantics of the system model in the logic of the prover.

The original prover model for I/O automata in Isabelle was designed by Mueller [Mül98]. Its main purpose was to prove the meta theory about IO automata, such as the soundness of forward simulation relation proofs. It also permitted proofs of properties of specific automata. Luhrs and Garland [Luh02] used this model to design and partially implement a translator from IOA to Isabelle. However, the model and translation were, in practice, inconvenient for designing automatable proofs of specific automata. We describe why below.

For our work, we chose to change the Isabelle system model to be more similar to the one created by Bogdanov for his work with the Larch Prover [BGL02]. This made tactic writing easier. Here, we describe the modifications and formally specify the prover model. We use the memory case study from Section 4.2 to show the differences between the two prover models.

7.1 The Mueller model

In the Mueller prover model, an IO automaton is a triple consisting of an action signature, a set of start states, and a set of transitions. This is the same as the intuition for the actual system model of an I/O automaton described in Section 2.4. However, the Mueller model also includes ancillary data definitions for the prover to understand the model. Thus, the whole prove model is represented by the following:

- Data type declaration describing the state space of the automaton.
- Data type declaration describing the action type of the automaton.

- Classification of the actions as input, output or internal. These predicates are combined into the action signature.
- Definition of the set of start states.
- Definition of the valid transitions relating a pre-state, action and post-state of the automaton.

The next sections describe each component in more detail.

State spaces

State spaces are declared as records of the automaton state variables:

```
record Mem_state =
  memVar :: "Value"
  act :: "(Node, Action Null) Array"
  rsp :: "(Node, Response Null) Array"
```

The command creates an Isabelle data type, not a data value. Having a record represent the data type is the correct approach, because state spaces are cartesian products of the underlying variables.

Action data type

Actions are declared as ML type constructors:

```
datatype mem_action =
  invoke node action |
  respond node result |
  update node
```

This creates a mutually exclusive set of constructors because automaton actions are mutually exclusive. The parameters to the type constructor correspond to the parameters of the action. We use a different means of constructing the action data type from the state data type because the automaton state is a cartesian product of each state variable, while the actions are mutually exclusive.

Classification of actions

We classify the actions into three sets, input, output and internal:

```
Mem_input :: "Mem_action set"
Mem_input == {a. case a of
    (invoke n a) => False |
    (respond n r) => False |
    (update n) => False
  }"

Mem_output :: "Mem_action set"
Mem_output_def:
  " Mem_output == {a. case a of
    (invoke n a) => True |
    (respond n r) => True |
    (update n) => False
  }"

Mem_internal :: "Mem_action set"
Mem_internal_def:
  " Mem_internal == {a. case a of
    (invoke n a) ⇒ False |
    (respond n r) ⇒ False |
    (update n) ⇒ True
  }"
```

By convention, we use the collect operator (".") to define the sets. The collect operator fills a set with all values that satisfy its predicate. However, they could be defined by any other set operators or by explicitly enumerating the values.

The action signature does not list the type of actions (for this is already done in action the data type) but simply groups the above three sets. It is an ML triple:

```
Mem_asig_def : "Mem_asig == (Mem_input,  Mem_output,  Mem_internal)"
```

The start state

The start state is a subset of the states, also defined using the collect operator:

```
defs
 Mem_start_def:
  " Mem_start == { sMem.
   (rsp  sMem = (constant nil)) &
   (act  sMem = (constant nil)) &
   (memVar  sMem = v0)
  }"
```

Transitions

A transition of an automaton is an ML triple of a pre-state, an action, and a post-state. The set of transitions is also defined via the collect operator:

```
types
   ('a,'s)transition =   "('s * 'a * 's)"

Mem_trans :: "( Mem_action,  Mem_state) transition set"

Mem_transitions = "{tr .
Let   s = fst tr
    act = snd tr
     s' = lst tr
in
case act of
 (invoke n a) ⇒
   (memVar s') = (memVar s)                      ∧
   (act s') = (assign (act s) n (embed a))   ∧
   (rsp s') = (rsp s)
 (respond n r) ⇒
   (memVar s') = (memVar s)                      ∧
   (act s') = (assign (act s) n nil)          ∧
```

```
    (rsp s') = (assign (rsp s) n nil)
  (update n) ⇒
   ∃ a . (sub act n) = (embed a)              ∧
   Let a = SOME a. (sub act n) = (embed a) in
    (memVar s') = (perform a (memVar s))       ∧
    (act s') = (act s)                         ∧
    (rsp s') = (assign (rsp s) n (embed (result a (memVar s))))
}"
```

The ticked variables (e.g., `'s`) are data type variables in ML, The star operator (`*`) makes a tuple. We use these operators here to abbreviate the meaning of a transition as a triple. The transition definition permits any number of post-states to match a particular pre-state, action pair — this is a form of nondeterminism. The existential quantifier and `Let` syntax handle the local variable `a`, for `a` is not part of the action's formal parameters but is used in the transition for `update` (see the automaton code in Section 4.2). The precondition for `update` is written using an existential quantifier, and then `a` is referred to in the effect section using a `Let` and a `SOME` operator. The `sub` operator corresponds to array subscripting.

Defining the automaton

Lastly, the automaton itself is defined:

```
Mem :: "(Mem_action,  Mem_state) ioa"

Mem_def:
  Mem == (Mem_asig, Mem_start, Mem_transitions)
```

The datatype of `Mem` is an `ioa`, which is a pre-defined data triple. The following definitions are true for all automata, since they are built into a helper file in Isabelle:

```
    ('a,'s)transition =   "('s * 'a * 's)"
    ('a,'s)ioa        =   "'a signature * 's set * ('a,'s)transition set"
```

The automaton has a few other helper procedures:

```
asig_of_def:   "asig_of aut  == (first aut)"
starts_of_def: "starts_of    == (second aut)"
trans_of_def:  "trans_of aut == (third aut)"
```

Issues with the Mueller model for verification

There are two problems with this model for use in proofs of specific automata, either for humans or machines:

- The enablement of a particular transition is hidden in the transition definition, and requires an existential quantifier to specify. Thus, to say that a is enabled in state s, it is necessary to say:
  ```
  (enabled aut s a) == ∃ s' . (s, a, s') : (trans_of aut)
  ```
  This is unfortunate, as existential quantifiers can only be proved in the prover by actually providing witness variables. Thus to say that an action is enabled, the user must provide a witness, even though the IOA code itself made the enablement obvious in the transition's precondition.

- An execution fragment is not determined just by a given sequence of actions but requires the post-states of each of the actions. To show that an action sequence is valid, the user must provide multiple existential witnesses to describe the post-states.

The first shortcoming is less of a problem because the standard invariant proof tends to be a theorem in the form:

```
theorem Mem_Inv_step:
  (s, a, s') : (trans_of Mem) ∧ (Inv s) ⟹ (Inv s')
```

Thus, the post-state is explicitly provided. However, the second shortcoming is a problem because when we want to show that there exists a witness execution β, we have to provide all the intermediate states also. This is tedious for the user, and clutters computer-generated proofs.

7.2 The new Isabelle model

The fundamental problem for practical use of the Mueller model is that it allows nondeterminism in the post-state for each action, so that a proof is required to choose a particular post-state. This arises from a direct translation of I/O automata, which are by nature nondeterministic for each state, in both the next action and the post-state given an action.

In practice, when we write I/O automata, the main source of nondeterminism is in the *action* that is taken, not in the post-state. Further, when there is nondeterminism in the post-state, in an overwhelming majority of cases, the nondeterminism is made explicit by the use of `local` variables, which encapsulate the nondeterminism. Handling nondeterminism in the action is easier, since we can perform structural induction on the action data type and cover all possibilities. It is not possible to do the same on the post-state since the nondeterminism is mixed in with the actual state variables. There is thus an advantage in pushing all nondeterminism into the action and making the post-state deterministic once the pre-state and action are known. This was Bogdanov's insight [Bog00] for the Larch theory of I/O automata, an approach we closely follow in the new model.

The general rule is that every `choose` variable becomes a `local` parameter to the action. The action datatype itself is now modified to include the local variable as a parameter to each action. Any constraints on `local` and `choose` variables become preconditions to transitions [1]. The post-state is a deterministic function of the pre-state and the new action. Since we change the action datatype, we need to also change how actions between two automata correspond. We explain this later in Section 7.2.2.

An automaton is now a 4-tuple, consisting of an action signature, start states, an enablement function and an effect function. We preserve the method of defining the automaton signature and action and state datatypes. By separating the enablement from the effect, we can say when an action is enabled without having to quantify the post-state. For implementation convenience, an automaton is no longer a tuple, but instead constructed by the ML type constructor `ioa`:

[1]Bogdanov did not resolve what happens when `choose` variables appear inside `for` loops. We do not attempt to remedy this either.

```
datatype ('action, 'state) ioa =
  IOA
     "'action signature"            (* Signature *)
     "'state set"                   (* Starts *)
     "('action,'state) enablement"  (* Enablement *)
     "('action,'state) effect"      (* Effects *)
```

Isabelle automatically defines accessor functions when we use this means of definition. We now define the memory automaton in this prover model:

```
Mem == (IOA  Mem_asig  Mem_start  Mem_enablement  Mem_effect)"
```

where the data types `enablement` and `effect` are defined by:

```
types
   ('action,'state) enablement =  "'state ⇒ 'action ⇒ bool"
   ('action,'state) effect = "'state ⇒ 'action ⇒ 'state"
```

Thus an enablement function tells whether a particular action is enabled from a state, while the effect function determines the new state. The effect function is unspecified when the action is not enabled. Now it is possible to have a quantifier-free predicate to determine whether an action is enabled, as long as the precondition itself does not contain a quantifier. For the shared memory case study in Isabelle, the new translation results in the specifications shown in Figure 7-1 for the `Mem` automaton.

With determinism in actions, an execution fragment is just a list of actions:

```
type
   ('action) execution = "'action list"
```

the head of the list contains the *last* action, so that we can recursively (or inductively) define the execution as shown below.

Using this execution format requires some helper functions. The `lastOf` function returns the state that is the result of executing a particular execution fragment upon a state of an automaton, from a given state:

```
consts lastOf :: "('action, 'state) ioa ⇒
                  'state ⇒
                  ('action) execution ⇒
                  'state"


primrec
  lastOf_def:
  "lastOf aut s [] = s"
  lastOf_def2:
  "lastOf aut s (Cons act rest) =
   effects_of aut (lastOf aut s rest) act
  "
```

Note: the `primrec` allows the definition of a partially specified function in the Isabelle language using ML pattern matching. The `isExecution` function determines whether an entire execution fragment is valid:

```
consts
  isExecution :: "('action, 'state) ioa ⇒
                  'state ⇒
                  ('action) execution ⇒
                  bool"
primrec
  isExecution_def:
  "isExecution aut s [] = (
record Mem_state =
  memVar :: "Value"
  act :: "(Node, (Action) Null) Array"
  rsp :: "(Node, (Response) Null) Array"

datatype Mem_action =
  invoke "Node" "Action"  |  respond "Node" "Response"  |  update "Node" "Action"

defs
 Mem_start_def:
  " Mem_start == { sMem.
   (rsp  sMem = (constant nil)) ∧
   (act  sMem = (constant nil)) ∧
   (memVar  sMem = v0)
  }"

defs
 Mem_enablement_def:
  " Mem_enablement  sMem  aMem == case  aMem of
  (invoke n a) ⇒
    ((sub (act sMem) n) = nil)       % sMem.act[n] = nil
  |
  (respond n r) ⇒
    ((sub (rsp sMem) n) = (embed r)) % sMem.rsp[n] = embed(r)
  |
  (update n a) ⇒   % sMem.rsp[n] = nul ∧ sMem.act[n] = embed(a)
    (((sub (rsp sMem) n) = nil) ∧ ((sub (act sMem) n) = (embed a)))
  "
defs
 Mem_effect_def:
  " Mem_effect  sMem  aMem == case  aMem of
  (invoke n a) ⇒
    Mem_state.make
     (memVar sMem)
     (assign (act sMem) n (embed a))
     (rsp sMem) |
  (respond n r) ⇒
    Mem_state.make
     (memVar sMem)
     (assign (act sMem) n nil)
     (assign (rsp sMem) n nil) |
  (update n a) ⇒
    Mem_state.make
     (perform a (memVar sMem))
     (act sMem)
     (assign (rsp sMem) n (embed (result a (memVar sMem))))
  "
defs
 Mem_def:
  " Mem == (IOA  Mem_asig  Mem_start  Mem_enablement  Mem_effect)"
```

Figure 7-1: The specification automaton, Mem translated to Isabelle. Notice how the local variable a in the update transition is now a full parameter. This requires a change in how we match traces for simulation relations.

```
   True
  )"
  isExecution_def2:
  "isExecution aut s (act#rest) = (
   (enablement_of aut (lastOf aut s rest) act) ∧
   (isExecution aut s rest)
  )"
```

Now that we have the basic infrastructure, there are three useful aspects of this model to discuss. First is the practical view of using this model for proofs. This has already been presented in Section 3.5.2. The main benefit we get is that, like the LP translation, the transition semantics is free of existential quantifiers. Second is the use of this model for the theory of I/O automata in general. This is a useful discussion because we want to ensure that the model can do meta-theoretic proofs, and that it is sound. Third is how this model relates to simulation relations.

7.2.1 Meta theory for the new I/O automaton model

In this section, we describe how the infrastructure we have developed is still adequate for meta-theoretic proofs as done by Mueller. First we define two functions on automata: reachability and invariance. We then prove that the standard method of proving IOA invariance (start condition; step condition) is a sound way to show invariance as defined here.

In order to define reachability, we first define a helper, `reachableWith` which says that a particular state `s` is reachable from a particular state `s0` using a particular execution `alpha`:

```
constdefs
  reachableWith :: "('action,'state)ioa ⇒
                    'state ⇒
                    'state ⇒
                    'action execution ⇒
                    bool"
  "reachableWith aut s s0 alpha == (
   (s0 : starts_of aut) ∧
   (isExecution aut s0 alpha) ∧
   (lastOf aut s0 alpha = s)
)"
```

Now this helper predicate is used by the definition of `reachable` in saying that there exists a start state `s0` and an execution `alpha` that satisfies `reachableWith`, for a given state `s`:

```
constdefs
  reachable   :: "('action,'state)ioa ⇒ 'state ⇒ bool"
  "reachable aut s ==
  (∃ s0 . (∃ alpha . reachableWith aut s s0 alpha))
  "
```

Given this definition of reachability, we can now define what it means for an invariant to hold on an automaton. An invariant is a predicate on the states of an automaton. Thus:

```
constdefs
  invariant     :: "[('action,'state)ioa, 'state⇒bool] ⇒ bool"
 "invariant aut P == (∀ s. reachable aut s → P(s))"
```

An invariant `P` is an invariant on automaton `aut` if for all reachable states `s` the invariant holds. We now show that the IOA method for proving invariants is sound. We define the start and step conditions holding as follows:

```
constdefs
  invariant_start :: "[('action,'state)ioa, 'state ⇒ bool] ⇒ bool"
  "invariant_start aut I ==
   ∀ state . (state : (starts_of aut) → I state)"
constdefs
  invariant_trans :: "[('action,'state)ioa, 'state ⇒ bool] ⇒ bool"
  "invariant_trans aut I ==
   ∀ state act .
       ((reachable aut state) ∧
        (I state) ∧
        (enablement_of aut state act)
       )→
        I(effects_of aut state act)"
```

These simply are the standard way we prove invariants in IOA. We then prove the following main theorem:

```
theorem invariantI:
  assumes p0: "invariant_start aut I"
      and p1: "invariant_trans aut I"
     shows "invariant aut I"
```

This is a 200 line proof shown in Appendix A. The reason why the two conditions are sufficient to prove invariance is the expected one: a state is reachable if it is a start state or can be reached from a valid execution. If the state is a start state, `invariant_start` handles this case. If a state is reachable due to a valid execution, `invariant_trans` handles this case. The second case is a bit intricate, as we have to show that the invariant step condition holding is sufficient to show that the invariant holds for the entire valid execution. We do this by structural induction on the valid execution sequence (case `nil`, case `Cons`).

Thus, we have proven, using only our Isabelle theory of I/O automata, that our methodology for proving invariants is sound. This gives a concrete mathematical foundation for the theorem prover tactics we use on our model.

7.2.2 Simulation relations

The model is also adequate for handling simulation relations. We have not yet performed the meta-theoretic proofs showing that the IOA method for showing simulation relations implies trace inclusion (as is proven by hand in Lynch's book [Lyn96] and by Mueller in his model). We simply present here the functions for defining that a predicate is indeed a simulation relation.

The actual mapping of the states between the implementation and specification automata for a simulation relation is trivial: we simply use any `F` that takes in the right automaton states. For the mapping from the `Cache` automaton to the `Mem` automaton, this is simply:

```
consts
  FCache2Mem :: "Cache_state ⇒ Mem_state ⇒ bool"
defs
FCache2Mem_def:
  "FCache2Mem sCache sMem ==
    ((((Mem.memVar sMem) = (Cache.memVar sCache)) ∧
      ((Mem.act sMem) = (Cache.act sCache))) ∧
      ((Mem.rsp sMem) = (Cache.rsp sCache)))"
```

The difficulty is showing that the corresponding executions exhibit the same traces. This is because of two reasons. First, the action data types of the two automata are different, according to the model. Thus, we cannot simply equate the two actions in a strongly typed language like Isabelle or LP. Second, we chose to resolve the nondeterminism of post-states by including extra parameters in the action. Now these parameters may not match between the automata (e.g. one automaton has nondeterminism, the other does not).

We resolve both of these problems using a technique almost identical to the one used by Bogdanov in the LP translator. We define a new action type that represents the actions common to both automata, and two action mapping functions that map from each automaton's action type to the common type. With the memory example, this is:

```
datatype Cache2Mem_action =
  invoke Node Action |
  respond Node Response |
  dummy

consts
  mapCache :: "Cache_action ⇒ Cache2Mem_action"
  mapMem :: "Mem_action ⇒ Cache2Mem_action"

defs
mapCache_def:
  "mapCache aCache == case aCache of
   (invoke n a) ⇒ (invoke n a) |
   (respond n r) ⇒ (respond n r) |
   (read n a) ⇒ dummy |
   (write n a) ⇒ dummy |
   (copy n) ⇒ dummy |
   (drop n) ⇒ dummy"

defs
mapMem_def:
  "mapMem aMem == case aMem of
   (invoke n a) ⇒ (invoke n a) |
   (respond n r) ⇒ (respond n r) |
   (update n a) ⇒ dummy"
```

Notice that the internal transitions are mapped to `dummy` to indicate that there is no need to compare traces between them. Unfortunately, the memory example does not have any external actions with local parameters. Had `update` been an external action, the mapping would have been:

```
datatype Cache2Mem_action =
  invoke Node Action |
  respond Node Response |
  update  Node
...

defs
mapMem_def:
  "mapMem aMem == case aMem of
   (invoke n a) ⇒ (invoke n a) |
   (respond n r) ⇒ (respond n r) |
   (update n a) ⇒ (update n)"
```

Given this mapping function, we can now fully specify what it means in Isabelle for two automata to have a forward simulation relation.

A forward simulation between two automata is defined as follows:

```
constdefs isFwdSim :: "('actionL, 'stateL) ioa ⇒ ('actionU, 'stateU) ioa ⇒
                       ('stateL ⇒ 'stateU ⇒ bool) ⇒
                       ('actionL ⇒ 'action) ⇒
                       ('actionU ⇒ 'action) ⇒
                       bool"
  "isFwdSim autL autU relation mapL mapU ==
      (isFwdSim_start autL autU relation)
    ∧ (isFwdSim_trans autL autU relation mapL mapU)
  "
```

The `isFwdSim` predicate takes in two automata, the actual simulation relation, and the two mapping functions and says that two conditions have to hold: the start and step conditions. The start condition is defined as expected:

```
constdefs isFwdSim_start ::
  "('actionL, 'stateL) ioa ⇒ ('actionU, 'stateU) ioa ⇒
   ('stateL ⇒ 'stateU ⇒ bool) ⇒
    bool"

  "isFwdSim_start autL autU relation ==
   (∀ s0. s0: starts_of autL → (∃ u0 . (u0 : starts_of autU) ∧
                                         (relation s0 u0)
                                 ))
  "
```

This is precisely the definition we use in LP or on paper: for all start states of the implementation automaton, there exists a start state of the specification automaton that satisfies the relation.

The step condition is also as expected:

```
constdefs isFwdSim_trans ::
  "('actionL, 'stateL) ioa ⇒ ('actionU, 'stateU) ioa ⇒
   ('stateL ⇒ 'stateU ⇒ bool) ⇒
   ('actionL, 'action) actionMap ⇒
   ('actionU, 'action) actionMap ⇒
    bool"
  "isFwdSim_trans autL autU relation mapL mapU ==
   (∀ s act u. (enablement_of autL s act) ∧
               (relation s u) ∧
               (reachable autL s) →
   (∃ beta . correspExec autL autU relation mapL mapU s u [act] beta
   )

   )
  "
```

For all implementation actions `act`, there exists an execution `beta` of the specification automaton such that `act` and `beta` are corresponding executions. The correspondence relation is given by:

```
constdefs correspExec ::
  "('actionL, 'stateL) ioa ⇒ ('actionU, 'stateU) ioa ⇒
   ('stateL ⇒ 'stateU ⇒ bool) ⇒
   ('actionL, 'action) actionMap ⇒
```

```
    ('actionU, 'action) actionMap ⇒
    'stateL ⇒ 'stateU ⇒
    ('actionL execution) ⇒ ('actionU execution) ⇒
     bool"
  "correspExec autL autU relation mapL mapU s u alpha beta ==
       (map mapU (traceOf autU beta)) = (map mapL (traceOf autL alpha))
     ∧ (isExecution autU u beta)
     ∧ (relation (lastOf autL s alpha) (lastOf autU u beta))
"
```

This gives the three requirements of the step condition: trace equality, valid execution, and that the last states relate. The `map` function is the standard mapping function that applies its first argument onto every element of its second argument. In this case, we map the action mapping function onto the automaton trace, which is a list of actions.

7.3 Assessment

From out case studies, we have seen that the new prover model for I/O automata, based on Bogdanov's work, is adequate infrastructure for enacting practical proofs of simulation relations in Isabelle. We have also seen in Chapter 3 that the definitions have been useful for generating proof tactics. The usefulness of this model for proofs stems from restricting nondeterminism to the actions of a transition, rather than allowing it also in the post-state.

In the future, it would be helpful to prove the soundness of proving simulation relations in Isabelle: showing that the simulation relation start and step conditions imply trace inclusion. This is a meta-theoretic proof. It is likely that the mapping function will need to be used in the definition of trace inclusion.

Chapter 8

Discussion

8.1 Further research

There are at least four ways to extend this research: improving dynamic invariant detection, generating better proof tactics, programming in the prover, and selecting invariants for use in proofs.

8.1.1 Improved dynamic invariant detection

The Daikon dynamic invariant detector could be improved, in order to find more lemmas for proofs and increase human insight regarding program behavior. A major problem in dynamic invariant detection is choosing a grammar of invariants that is useful for programmers but does not generate an excessive amount of output. One way to choose a grammar would be to take boolean expressions appearing in IOA program code as templates in the grammar. Since these templates come from the semantics of the program, they may be likelier to be useful invariants. For example, in the Paxos case study (Section 4.3), `Inv4` closely resembles the precondition for the `assignVal` transition.

8.1.2 Improved proof scripts from automaton code

Static analysis of I/O automata could generate more detailed proof scripts so that prover can do more work without human intervention. For example, in performing case splits, we currently examine `if` statements in the annotations for paired execution, but we could also look at `if` statements within the effects code of the automaton itself.

8.1.3 Programming in the prover language

We can further extend our tools to use the Isabelle/HOL logic system and theorem prover [Pau93, Gor89]. Isabelle is a programmable prover, so we could use its programming language, ML, to derive better tactics. Presently, we have the IOA translator to Isabelle generate the proof script — using ML to drive the proof may be more productive. Since Isabelle has a larger user community and a more extensive set of libraries, this may make our methodology accessible to more people.

8.1.4 Filtering invariants with automated heuristics for proofs

One of the stages in our method that involves human intervention is having to manually select invariants to use in a proof. We are in search of true and useful invariants when we perform this manual selection. Here we suggest a way to automate this process. Invariants output by dynamic invariant detection can be classified as follows:

- True and useful. These are later used in verification in theorem provers. There may be more than one useful set of invariants that are used in verification. We want just one such set.
- True and not useful. These may be useful for others, but are not used as lemmas in our verification proof.
- False.

However, this classification is generally undecidable. We expect to rely on a combination of human classification and automatic invariant filtering algorithms to compensate. Human classification is still human intervention, but the intervention is greatly reduced compared to using the prover alone, because the programmer no longer has to come up with the invariants.

In proving invariants true with the prover, the main difficulty is the dependencies between different invariants. It could be that invariant I_A is true, but cannot be proved without another invariant I_B in the inductive assumption. It could also be that I_A is false, but holds true I_B is true. We suggest an algorithm based on one by Rintanen [Rin00] that relies on arriving at fix points:

> Start with some set `Inv` of invariants to filter and check for truth. Check if invariants in `Inv` hold on the start state of the automaton. Remove any invariants in `Inv` that fail. Assume the invariants in `Inv` hold for some state `a` in the automaton. For all enabled actions from `a` to `a'`, prove that they hold for `a'`. Remove any invariants in `Inv` that fail. Repeat this process with the new `Inv` until the members of `Inv` no longer change. At this fix point, all the invariants in `Inv` are true.

Once we have a set of true invariants, they can be easily checked to see which are needed to prove a simulation relation (or any other property):

> Assume the simulation relation `F(a, b)` holds. Also assume `I` holds for `a`. Attempt to prove the simulation relation holds for a reachable state `a'` from `a` by showing the standard witness β from `b` to `b'`. Temporarily remove one invariant `i` in `Inv` and test if the simulation relation can still be proved somehow. If so, permanently remove `i` from `Inv`. Attempt this removal process on each invariant.

The set `Inv` that remains is true, since we started with a true set. It is "minimal" in the sense that no subset of it will result in a proof, using a prover with the same capabilities. There may be an alternative set of invariants that is of a smaller size, but we are interested in only finding some minimal set. Further, some provers may be able to identify which invariants were used in a proof, so the second algorithm may not be necessary.

We could implement both algorithms presented here using more automated proving methods built into Isabelle.

8.2 Conclusion

The purpose of software verification is to ensure programmers and users that the systems they develop and employ behave correctly. In this thesis, we tackled the problem of verifying distributed or concurrent systems, which are often infinite state and nondeterministic. Theorem provers can be used to reason soundly about the correctness of such systems. Such machine-checked proofs provide more assurance of correctness than hand proofs, but incur a cost in terms of human interaction. The methodology presented in this thesis reduces the human effort required in the theorem prover for verifying safety properties of distributed algorithms modeled formally as I/O automata.

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

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

Second, we leverage the effort used to build good test suites to produce scripts for theorem provers, which mirror the form of the scripts for driving paired executions. These tactics combine with our knowledge of proofs of all I/O automata to provide the proof outline in a prover.

We have illustrated the use of the methodology, and of the toolset that supports the methodology, by means of three case studies: Lamport's Paxos protocol, distributed strong caching memory, and Peterson's 2-process mutual exclusion algorithm.

In order to efficiently implement our methodology, it was necessary to extend the three tools employed. We modified the IOA Simulator to allow for simulation of some quantified expressions, and so that its semantics matched that of the tools that translate IOA into prover languages. We formalized the correctness properties of dynamic invariant detection, and modified the Daikon tool to make it more scalable via two optimizations. Lastly, we developed a new prover model for I/O automata in the Isabelle/HOL system based on Bogdanov's work with LP and proved the soundness of our invariant proving methodology. The prover model facilitates generation of proof tactics in our methodology.

Appendix A

Soundness proof of invariant method in Isabelle

The following is Isabelle code showing that in the Isabelle prover model, our method for proving invariants is sound. We show that proving the start condition and the step condition for invariants, `invariant_start` and `invariant_trans` for an automaton is sufficient to prove invariance (as defined in Isabelle). Invariance is defined for a predicate by saying that the predicate holds on all reachable states. Along the way, we define a few helper lemmas.

```
constdefs
  invariant_start :: "[('action,'state)ioa, 'state⇒bool] ⇒ bool"
  "invariant_start aut I ≡
   ∀ state . (state : (starts_of aut) −→ I state)"

constdefs
  invariant_trans :: "[('action,'state)ioa, 'state⇒bool] ⇒ bool"
  "invariant_trans aut I ≡
   ∀ state act .
      ((reachable aut state) ∧
       (I state) ∧
       (enablement_of aut state act)
      )−→
       I(effects_of aut state act)"

theorem executionStep:
 assumes a0: "isExecution automaton s alpha"
     and a1: "   enablement_of automaton (lastOf automaton s alpha) act"
    shows "isExecution automaton s (act#alpha)"
  apply (simp add: isExecution_def2 prems)
done

theorem lastOfStep:
 "lastOf automaton s (act#alpha) =
  effects_of automaton (lastOf automaton s alpha) act"
  apply (simp add: lastOf_def)
done

theorem reachableStep:
  "[|
   reachable automaton s;
   enablement_of automaton s act
   |] ⟹
   reachable automaton (effects_of automaton s act)"
  apply (simp add: reachable_def)
  proof -
    assume
      a0: "∃ s0. Ex (reachableWith automaton s s0)"
      and a1: "enablement_of automaton s act"
  def s0 == "SOME s0 . Ex (reachableWith automaton s s0)"
  have t0: "Ex (reachableWith automaton s s0)"
    apply (simp add: s0_def a0 existenceSome)
    apply (rule existenceSome)
    apply (simp add: a0)
  done
  have t1: "∃ u. reachableWith automaton s s0 u"
    apply (simp add: t0)
  done
  def alpha == "SOME u . reachableWith automaton s s0 u"
  have t2: "reachableWith automaton s s0 alpha"
    apply (simp add: alpha_def t1)
  done
  have a2: "s0 : starts_of automaton"
   and a3: "isExecution automaton s0 alpha"
   and a4: "lastOf automaton s0 alpha = s"
    apply (insert t2)
    apply (simp_all add: reachableWith_def)
  done
  show "∃ s0. Ex (reachableWith automaton (effects_of automaton s act) s0)"
    proof (rule exI)+
  show "reachableWith automaton (effects_of automaton s act) s0 (act#alpha)"
    apply (insert t2 a1)
    apply (simp add: reachableWith_def)
  done
  qed
qed

theorem executionStep_back:
  "isExecution aut s0 (act#alpha) --> isExecution aut s0 alpha"
  apply (auto)
done

theorem invariantI2:
  assumes p0: "invariant_start aut I"
      and p1: "invariant_trans aut I"
      and p2: "s0 : (starts_of aut)"
     shows
     "isExecution aut s0 alpha --> I (lastOf aut s0 alpha)"
proof (induct alpha)
have f1: "I s0"
proof -
  have f1_1: "∀ state. (state : starts_of aut --> I state)"
    by (insert p0, simp add: invariant_start_def)
  show "?thesis"
    by (insert f1_1, simp add: p2)
qed
{case Nil
  show "?case"
  by (simp add: lastOf_def f1)
}
{case Cons
  show "?case"
  proof (safe)
  assume a0: "isExecution aut s0 (a_ # list_)"
  show "I (lastOf aut s0 (a_ # list_))"
  proof -
  have
      a1: "enablement_of aut (lastOf aut s0 list_) a_"
  and a2: "isExecution aut s0 list_"
    apply (insert a0)
    apply (auto)
  done
  have f2: "I (lastOf aut s0 list_)"
    apply (simp add: Cons a2)
  done
  have f3: "reachable aut (lastOf aut s0 list_)"
    apply (simp add: reachable_def)
    apply (rule exI, rule exI)
    proof -
    show "reachableWith aut (lastOf aut s0 list_) s0 list_"
      apply (simp add: reachableWith_def)
      apply (simp add: p2 a1 a2)
    done
  qed
  have f4: "reachable aut (lastOf aut s0 (a_ # list_))"
    apply (simp add: reachable_def)
    apply (rule exI, rule exI)
    proof -
    show "reachableWith aut
                     (effects_of aut (lastOf aut s0 list_) a_)
                     s0 (a_ # list_)"
      apply (simp add: reachableWith_def)
      apply (simp add: p2 a1 a2)
    done
  qed
  have f5: "!! state act.
           [|reachable aut state ∧
            I state ∧ enablement_of aut state act|]
        ⟹ I (effects_of aut state act)"
    apply (insert p1)
    apply (simp add: invariant_trans_def)
  done
  show "?thesis"
    apply (auto)
    apply (insert f5 f3 f2 a1)
    apply (auto) (* Booyeah, let's see some unification! *)
  done
  qed
  qed
}
qed

theorem invariantI1:
  assumes p0: "invariant_start aut I"
      and p1: "invariant_trans aut I"
      and p2: "s0 : (starts_of aut)"
      and p3: "isExecution aut s0 alpha"
     shows
     "I (lastOf aut s0 alpha)"
  apply (simp add: invariantI2 p0 p1 p2 p3)
done


theorem invariantI:
  assumes p0: "invariant_start aut I"
      and p1: "invariant_trans aut I"
     shows "invariant aut I"
  apply (simp add: invariant_def)
  apply (auto)
  proof -
  fix s
  show "reachable aut s ⟹ I s  "
    apply (simp add: reachable_def)
  proof -
  assume a0: "∃ s0. Ex (reachableWith aut s s0)"
  show "?thesis"
  proof -
  def s0 == "SOME s0 . Ex (reachableWith aut s s0)"
  have t0: "Ex (reachableWith aut s s0)"
    apply (simp add: s0_def a0 existenceSome)
    apply (rule existenceSome)
    apply (simp add: a0)
  done
  have t1: "∃ u. reachableWith aut s s0 u"
    apply (simp add: t0)
  done
  def alpha == "SOME u . reachableWith aut s s0 u"
  have f1: "reachableWith aut s s0 alpha"
    apply (simp add: alpha_def t1)
  done
  have f2: "s0 : starts_of aut"
   and f3: "isExecution aut s0 alpha"
   and f4: "lastOf aut s0 alpha = s"
    apply (insert f1)
    apply (simp_all add: reachableWith_def)
  done
  show "I s"
    apply (insert f4)
    apply (auto)
    apply (rule invariantI1)
    apply (simp_all add: f2 f3 f4 p0 p1)
  done
  qed
  qed
qed
```

Appendix B

The Paxos simulation relation proof in Isabelle

The following is the automatically generated proof outline and tactics for the simulation relation proof from the `Global1` to `Cons` automaton in the Paxos case study.

```
(* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% *)
(* %% Simulation from Global1 to Cons *)
(* %%  *)

theory Global12Cons = Global1 + Cons:
datatype Global12Cons_action =
  init Node Value |
  fail Node |
  decide Node Value |
  dummy

consts
  mapGlobal1 :: "Global1_action ⇒ Global12Cons_action"
  mapCons :: "Cons_action ⇒ Global12Cons_action"
  FGlobal12Cons :: "Global1_state ⇒ Cons_state ⇒ bool"

defs
mapGlobal1_def:
  "mapGlobal1 aGlobal1 == case aGlobal1 of
   (init i v) ⇒ (init i v) |
   (fail i) ⇒ (fail i) |
   (decide i v) ⇒ (decide i v) |
   (makeBallot b) ⇒ dummy |
   (abstain i B) ⇒ dummy |
   (assignVal b v) ⇒ dummy |
   (vote i b) ⇒ dummy |
   (internalDecide b) ⇒ dummy"

defs
mapCons_def:
  "mapCons aCons == case aCons of
   (init i v) ⇒ (init i v) |
   (fail i) ⇒ (fail i) |
   (decide i v) ⇒ (decide i v) |
   (chooseVal v) ⇒ dummy"

defs
FGlobal12Cons_def:
  "FGlobal12Cons sGlobal1 sCons == (((((((Cons.initiated sCons) =
  (Global1.initiated sGlobal1)) & ((Cons.proposed sCons) =
  (Global1.proposed sGlobal1))) & ((Cons.decided sCons) =
  (Global1.decided sGlobal1))) & ((Cons.failed sCons) =
  (Global1.failed sGlobal1))) & (∀ v:: Value . ((∃ b:: Ballot . ((b
  ∈ (Global1.succeeded sGlobal1)) & ((sub (Global1.val sGlobal1) b) =
  (embed v)))) --> (v ∈ (Cons.chosen sCons))))) & (∀ v:: Value . ((v
  ∈ (Cons.chosen sCons)) --> (∃ b:: Ballot . ((b ∈ (Global1.succeeded
  sGlobal1)) & ((sub (Global1.val sGlobal1) b) = (embed v)))))))"

constdefs
startRelGlobal12Cons :: " Global1_state ⇒  Cons_state"
  "startRelGlobal12Cons sGlobal1 ==  Cons_state.make (Global1.initiated sGlobal1)
   (Global1.proposed sGlobal1) {} (Global1.decided sGlobal1) (Global1.failed sGlobal1)

theorem FGlobal12Cons_start:
  "isFwdSim_start Global1 Cons FGlobal12Cons"
  apply (rule isFwdSim_startRule)
  proof (- )
  fix  sGlobal1
  assume a0: " sGlobal1: starts_of  Global1"
  show "∃  sCons .  sCons: starts_of  Cons & FGlobal12Cons sGlobal1 sCons"
  proof (rule  exI)
  show "(startRelGlobal12Cons  sGlobal1) ∈ starts_of  Cons &
         FGlobal12Cons  sGlobal1 (startRelGlobal12Cons  sGlobal1)"
  apply (simp add: startRelGlobal12Cons_def  Cons_def
           Cons_start_def  FGlobal12Cons_def  Cons_state.make_def)
  apply (insert a0)
  apply (auto )
  apply (simp_all add:  Global1_def  Global1_start_def)
done
qed
qed


(* For enabled  ( Global1.init i v) *)
theorem FGlobal12Cons_trans_init:
assumes
  p0: "enablement_of Global1 sGlobal1 ( Global1.init i v)"
  and p1: " FGlobal12Cons sGlobal1 sCons"
  and p2: "reachable Global1 sGlobal1"
shows "∃  betaCons . correspExec  Global1 Cons FGlobal12Cons
  mapGlobal1 mapCons sGlobal1 sCons [ ( Global1.init i v)] betaCons"
proof (- )
(* Proof entry available *)
def  betaCons == "[( Cons.init i v)] ::  Cons_action list"
show "?thesis"
  proof (rule exI)
  show "correspExec  Global1 Cons FGlobal12Cons mapGlobal1
        mapCons sGlobal1 sCons [ ( Global1.init i v)] betaCons"
  apply (simp add: correspExec_def)
  apply (insert p0 p1)
  apply (auto )

  (* For trace equality:  *)
  apply (simp add: traceOf_def p0 p1  betaCons_def
       Global1_def Cons_def FGlobal12Cons_def prems Let_def
       asig_internals_def Global1_asig_def Cons_asig_def
       Global1_internal_def Cons_internal_def mapGlobal1_def mapCons_def)

  (* For enablement: *) apply (simp add: traceOf_def p0 p1
  betaCons_def Global1_def Cons_def FGlobal12Cons_def prems
  Global1_enablement_def Cons_enablement_def Global1_effect_def
  Cons_effect_def Global1_state.make_def Cons_state.make_def)

  (* For posteffect relation: *) apply (simp add: traceOf_def p0 p1
  betaCons_def Global1_def Cons_def FGlobal12Cons_def prems
  Global1_effect_def Cons_effect_def Global1_state.make_def
  Cons_state.make_def) done qed qed


(* For enabled  ( Global1.fail i) *)
theorem FGlobal12Cons_trans_fail:
assumes
  p0: "enablement_of Global1 sGlobal1 ( Global1.fail i)"
  and p1: " FGlobal12Cons sGlobal1 sCons"
  and p2: "reachable Global1 sGlobal1"
shows "∃  betaCons . correspExec  Global1 Cons FGlobal12Cons mapGlobal1
  mapCons sGlobal1 sCons [ ( Global1.fail i)] betaCons"
proof (- )
(* Proof entry available *)
def  betaCons == "[( Cons.fail i)] ::  Cons_action list"
show "?thesis"
  proof (rule exI)
  show "correspExec  Global1 Cons FGlobal12Cons mapGlobal1 mapCons
        sGlobal1 sCons [ ( Global1.fail i)] betaCons"
  apply (simp add: correspExec_def)
  apply (insert p0 p1)
  apply (auto )

  (* For trace equality:  *)
  apply (simp add: traceOf_def p0 p1  betaCons_def
       Global1_def Cons_def FGlobal12Cons_def prems Let_def
       asig_internals_def Global1_asig_def Cons_asig_def
       Global1_internal_def Cons_internal_def mapGlobal1_def mapCons_def)

  (* For enablement:  *)
  apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def
         Cons_def FGlobal12Cons_def prems Global1_enablement_def
         Cons_enablement_def Global1_effect_def Cons_effect_def
         Global1_state.make_def Cons_state.make_def)
  (* For posteffect relation:  *)
  apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def Cons_def
         FGlobal12Cons_def prems Global1_effect_def Cons_effect_def
         Global1_state.make_def Cons_state.make_def)
done
qed
qed


(* For enabled  ( Global1.decide i v b) *)
theorem FGlobal12Cons_trans_decide:
assumes
  p0: "enablement_of Global1 sGlobal1 ( Global1.decide i v b)"
  and p1: " FGlobal12Cons sGlobal1 sCons"
  and p2: "reachable Global1 sGlobal1"
shows "∃  betaCons . correspExec  Global1 Cons FGlobal12Cons mapGlobal1
          mapCons sGlobal1 sCons [ ( Global1.decide i v b)] betaCons"
proof (- )
(* Proof entry available *)
def  betaCons == "[( Cons.decide i v)] ::  Cons_action list"
show "?thesis"
  proof (rule exI)
  show "correspExec  Global1 Cons FGlobal12Cons mapGlobal1 mapCons
        sGlobal1 sCons [ ( Global1.decide i v b)] betaCons"
  apply (simp add: correspExec_def)
  apply (insert p0 p1)
  apply (auto )

  (* For trace equality:  *)
  apply (simp add: traceOf_def p0 p1  betaCons_def
       Global1_def Cons_def FGlobal12Cons_def prems Let_def
       asig_internals_def Global1_asig_def Cons_asig_def
       Global1_internal_def Cons_internal_def mapGlobal1_def mapCons_def)

  (* For enablement:  *)
  apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def
         Cons_def FGlobal12Cons_def prems Global1_enablement_def
         Cons_enablement_def Global1_effect_def Cons_effect_def
         Global1_state.make_def Cons_state.make_def)

  (* For posteffect relation:  *)
  apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def Cons_def
         FGlobal12Cons_def prems Global1_effect_def Cons_effect_def
         Global1_state.make_def Cons_state.make_def)
done
qed
qed


(* For enabled  ( Global1.makeBallot b) *)
theorem FGlobal12Cons_trans_makeBallot:
assumes
```

```
  p0: "enablement_of Global1 sGlobal1 ( Global1.makeBallot b)"
  and p1: " FGlobal12Cons sGlobal1 sCons"
  and p2: "reachable Global1 sGlobal1"
shows "∃  betaCons . correspExec  Global1 Cons FGlobal12Cons mapGlobal1
           mapCons sGlobal1 sCons [ ( Global1.makeBallot b)] betaCons"
proof (- )
(* Proof entry available *)
def  betaCons == "[] ::  Cons_action list"
show "?thesis"
  proof (rule exI)
  show "correspExec  Global1 Cons FGlobal12Cons mapGlobal1 mapCons sGlobal1
        sCons [ ( Global1.makeBallot b)] betaCons"
  apply (simp add: correspExec_def)
  apply (insert p0 p1)
  apply (auto )

  (* For trace equality:  *)
  apply (simp add: traceOf_def p0 p1  betaCons_def
       Global1_def Cons_def FGlobal12Cons_def prems Let_def
       asig_internals_def Global1_asig_def Cons_asig_def
       Global1_internal_def Cons_internal_def mapGlobal1_def mapCons_def)

  (* For enablement:  *)
  apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def
         Cons_def FGlobal12Cons_def prems Global1_enablement_def
         Cons_enablement_def Global1_effect_def Cons_effect_def
         Global1_state.make_def Cons_state.make_def)

  (* For posteffect relation:  *)
  apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def Cons_def
         FGlobal12Cons_def prems Global1_effect_def Cons_effect_def
         Global1_state.make_def Cons_state.make_def)
done
qed
qed


(* For enabled  ( Global1.abstain i B) *)
theorem FGlobal12Cons_trans_abstain:
assumes
  p0: "enablement_of Global1 sGlobal1 ( Global1.abstain i B)"
  and p1: " FGlobal12Cons sGlobal1 sCons"
  and p2: "reachable Global1 sGlobal1"
shows "∃  betaCons . correspExec  Global1 Cons FGlobal12Cons mapGlobal1
           mapCons sGlobal1 sCons [ ( Global1.abstain i B)] betaCons"
proof (- )
(* Proof entry available *)
def  betaCons == "[] ::  Cons_action list"
show "?thesis"
  proof (rule exI)
  show "correspExec  Global1 Cons FGlobal12Cons mapGlobal1 mapCons
        sGlobal1 sCons [ ( Global1.abstain i B)] betaCons"
  apply (simp add: correspExec_def)
  apply (insert p0 p1)
  apply (auto )

  (* For trace equality:  *)
  apply (simp add: traceOf_def p0 p1  betaCons_def
       Global1_def Cons_def FGlobal12Cons_def prems Let_def
       asig_internals_def Global1_asig_def Cons_asig_def
       Global1_internal_def Cons_internal_def mapGlobal1_def mapCons_def)

  (* For enablement:  *)
  apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def
         Cons_def FGlobal12Cons_def prems Global1_enablement_def
         Cons_enablement_def Global1_effect_def Cons_effect_def
         Global1_state.make_def Cons_state.make_def)

  (* For posteffect relation:  *)
  apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def Cons_def
         FGlobal12Cons_def prems Global1_effect_def Cons_effect_def
         Global1_state.make_def Cons_state.make_def)
done
qed
qed


(* For enabled  ( Global1.assignVal b v) *)
theorem FGlobal12Cons_trans_assignVal:
assumes
  p0: "enablement_of Global1 sGlobal1 ( Global1.assignVal b v)"
  and p1: " FGlobal12Cons sGlobal1 sCons"
  and p2: "reachable Global1 sGlobal1"
shows "∃  betaCons . correspExec  Global1 Cons FGlobal12Cons mapGlobal1
           mapCons sGlobal1 sCons [ ( Global1.assignVal b v)] betaCons"
proof (- )
(* Proof entry available *)
show "?thesis"
proof (cases  "(¬(b ∈ (Global1.succeeded sGlobal1)))")
{
  (* True case *)
  case True
  def  betaCons == "[] ::  Cons_action list"
  show "?thesis"
    proof (rule exI)
    show "correspExec  Global1 Cons FGlobal12Cons mapGlobal1 mapCons
           sGlobal1 sCons [ ( Global1.assignVal b v)] betaCons"
    apply (simp add: correspExec_def)
    apply (insert p0 p1)
    apply (auto )

    (* For trace equality:  *)
    apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def Cons_def
           FGlobal12Cons_def prems Let_def asig_internals_def Global1_asig_def
           Cons_asig_def Global1_internal_def Cons_internal_def mapGlobal1_def mapCons
    (* For enablement:  *)
    apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def
           Cons_def FGlobal12Cons_def prems Global1_enablement_def
           Cons_enablement_def Global1_effect_def Cons_effect_def
           Global1_state.make_def Cons_state.make_def)

    (* For posteffect relation:  *)
    apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def Cons_def
           FGlobal12Cons_def prems Global1_effect_def Cons_effect_def
           Global1_state.make_def Cons_state.make_def)
  done
  qed
}
{
  (* Elseif case *)
  case False
  show "?thesis"
  proof (cases  "(∃ b:: Ballot . ((b ∈ (Global1.succeeded sGlobal1)) &
        (¬((sub (Global1.val sGlobal1) b) = nil))))")
  {
    (* True case *)
    case True
    def  betaCons == "[] ::  Cons_action list"
    show "?thesis"
      proof (rule exI)
      show "correspExec  Global1 Cons FGlobal12Cons mapGlobal1 mapCons
            sGlobal1 sCons [ ( Global1.assignVal b v)] betaCons"
      apply (simp add: correspExec_def)
      apply (insert p0 p1)
      apply (auto )

      (* For trace equality:  *)
      apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def
         Cons_def FGlobal12Cons_def prems Let_def asig_internals_def
          Global1_asig_def Cons_asig_def Global1_internal_def
          Cons_internal_def mapGlobal1_def mapCons_def)

      (* For enablement:  *)
      apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def
         Cons_def FGlobal12Cons_def prems Global1_enablement_def
         Cons_enablement_def Global1_effect_def Cons_effect_def
         Global1_state.make_def Cons_state.make_def)

      (* For posteffect relation:  *)
      apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def
         Cons_def FGlobal12Cons_def prems Global1_effect_def Cons_effect_def
         Global1_state.make_def Cons_state.make_def)
    done
    qed
  }
  {
    (* False case *)
    case False
    def  betaCons == "[( Cons.chooseVal v)] ::  Cons_action list"
    show "?thesis"
      proof (rule exI)
      show "correspExec  Global1 Cons FGlobal12Cons mapGlobal1 mapCons
            sGlobal1 sCons [ ( Global1.assignVal b v)] betaCons"
      apply (simp add: correspExec_def)
      apply (insert p0 p1)
      apply (auto )

      (* For trace equality:  *)
      apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def
         Cons_def FGlobal12Cons_def prems Let_def asig_internals_def
         Global1_asig_def Cons_asig_def Global1_internal_def
         Cons_internal_def mapGlobal1_def mapCons_def)

      (* For enablement:  *)
      apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def
         Cons_def FGlobal12Cons_def prems Global1_enablement_def
         Cons_enablement_def Global1_effect_def Cons_effect_def
         Global1_state.make_def Cons_state.make_def)

      (* For posteffect relation:  *)
      apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def
         Cons_def FGlobal12Cons_def prems Global1_effect_def
         Cons_effect_def Global1_state.make_def Cons_state.make_def)
    done
    qed
  }
  qed
}
qed
qed


(* For enabled  ( Global1.vote i b) *)
theorem FGlobal12Cons_trans_vote:
assumes
  p0: "enablement_of Global1 sGlobal1 ( Global1.vote i b)"
  and p1: " FGlobal12Cons sGlobal1 sCons"
  and p2: "reachable Global1 sGlobal1"
shows "∃  betaCons . correspExec  Global1 Cons FGlobal12Cons mapGlobal1
       mapCons sGlobal1 sCons [ ( Global1.vote i b)] betaCons"
proof (- )
(* Proof entry available *)
def  betaCons == "[] ::  Cons_action list"
show "?thesis"
  proof (rule exI)
  show "correspExec  Global1 Cons FGlobal12Cons mapGlobal1
         mapCons sGlobal1 sCons [ ( Global1.vote i b)] betaCons"
  apply (simp add: correspExec_def)
  apply (insert p0 p1)
  apply (auto )
  (* For trace equality:  *)
  apply (simp add: traceOf_def p0 p1  betaCons_def
       Global1_def
         Cons_def FGlobal12Cons_def prems Let_def
       asig_internals_def Global1_asig_def Cons_asig_def
       Global1_internal_def Cons_internal_def mapGlobal1_def mapCons_def)

  (* For enablement:  *)
  apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def
         Cons_def FGlobal12Cons_def prems Global1_enablement_def
         Cons_enablement_def Global1_effect_def Cons_effect_def
         Global1_state.make_def Cons_state.make_def)

  (* For posteffect relation:  *)
  apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def
         Cons_def FGlobal12Cons_def prems Global1_effect_def
         Cons_effect_def Global1_state.make_def Cons_state.make_def)
done
qed
qed


(* For enabled  ( Global1.internalDecide b) *)
theorem FGlobal12Cons_trans_internalDecide:
assumes
  p0: "enablement_of Global1 sGlobal1 ( Global1.internalDecide b)"
  and p1: " FGlobal12Cons sGlobal1 sCons"
  and p2: "reachable Global1 sGlobal1"
shows "∃  betaCons . correspExec  Global1 Cons FGlobal12Cons mapGlobal1
       mapCons sGlobal1 sCons [ ( Global1.internalDecide b)] betaCons"
proof (- )
(* Proof entry available *)
show "?thesis"
proof (cases  "(b ∈ (Global1.succeeded sGlobal1))")
{
  (* True case *)
  case True
  def  betaCons == "[] ::  Cons_action list"
  show "?thesis"
    proof (rule exI)
    show "correspExec  Global1 Cons FGlobal12Cons mapGlobal1 mapCons
          sGlobal1 sCons [ ( Global1.internalDecide b)] betaCons"
    apply (simp add: correspExec_def)
    apply (insert p0 p1)
    apply (auto )

    (* For trace equality:  *)
    apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def
         Cons_def FGlobal12Cons_def prems Let_def asig_internals_def
         Global1_asig_def Cons_asig_def Global1_internal_def
         Cons_internal_def mapGlobal1_def mapCons_def)

    (* For enablement:  *)
    apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def
         Cons_def FGlobal12Cons_def prems Global1_enablement_def
         Cons_enablement_def Global1_effect_def Cons_effect_def
         Global1_state.make_def Cons_state.make_def)

    (* For posteffect relation:  *)
    apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def
         Cons_def FGlobal12Cons_def prems Global1_effect_def
         Cons_effect_def Global1_state.make_def Cons_state.make_def)
  done
  qed
}
{
  (* Elseif case *)
  case False
  show "?thesis"
  proof (cases  "((sub (Global1.val sGlobal1) b) = nil)")
  {
    (* True case *)
    case True
    def  betaCons == "[] ::  Cons_action list"
    show "?thesis"
      proof (rule exI)
      show "correspExec  Global1 Cons FGlobal12Cons mapGlobal1
            mapCons sGlobal1 sCons [ ( Global1.internalDecide b)] betaCons"
      apply (simp add: correspExec_def)
      apply (insert p0 p1)
      apply (auto )

      (* For trace equality:  *)
      apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def
         Cons_def FGlobal12Cons_def prems Let_def asig_internals_def
         Global1_asig_def Cons_asig_def Global1_internal_def
         Cons_internal_def mapGlobal1_def mapCons_def)

      (* For enablement:  *)
      apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def
         Cons_def FGlobal12Cons_def prems Global1_enablement_def
         Cons_enablement_def Global1_effect_def Cons_effect_def
         Global1_state.make_def Cons_state.make_def)

      (* For posteffect relation:  *)
      apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def
         Cons_def FGlobal12Cons_def prems Global1_effect_def
         Cons_effect_def Global1_state.make_def Cons_state.make_def)
    done
    qed
  }
  {
    (* Elseif case *)
    case False
    show "?thesis"
    proof (cases  "(∃ b:: Ballot . ((b ∈ (Global1.succeeded sGlobal1)) &
  ((sub (Global1.val sGlobal1) b) ≠ nil)))")
{
  (* True case *)
  case True
  def  betaCons == "[] ::  Cons_action list"
  show "?thesis"
    proof (rule exI)
    show "correspExec  Global1 Cons FGlobal12Cons mapGlobal1 mapCons
          sGlobal1 sCons [ ( Global1.internalDecide b)] betaCons"
    apply (simp add: correspExec_def)
    apply (insert p0 p1)
    apply (auto )

    (* For trace equality:  *)
    apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def
     Cons_def FGlobal12Cons_def prems Let_def asig_internals_def
     Global1_asig_def Cons_asig_def Global1_internal_def
     Cons_internal_def mapGlobal1_def mapCons_def)

    (* For enablement:  *)
    apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def
     Cons_def FGlobal12Cons_def prems Global1_enablement_def
     Cons_enablement_def Global1_effect_def Cons_effect_def
     Global1_state.make_def Cons_state.make_def)

    (* For posteffect relation:  *)
    apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def
     Cons_def FGlobal12Cons_def prems Global1_effect_def Cons_effect_def
     Global1_state.make_def Cons_state.make_def)
  done
  qed
}
{
  (* False case *)
  case False
  def  betaCons == "[( Cons.chooseVal (val (sub (Global1.val sGlobal1) b)))] ::  C
  show "?thesis"
    proof (rule exI)
    show "correspExec  Global1 Cons FGlobal12Cons mapGlobal1
          mapCons sGlobal1 sCons [ ( Global1.internalDecide b)] betaCons"
    apply (simp add: correspExec_def)
    apply (insert p0 p1)
    apply (auto )

    (* For trace equality:  *)
    apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def
     Cons_def FGlobal12Cons_def prems Let_def asig_internals_def
     Global1_asig_def Cons_asig_def Global1_internal_def
     Cons_internal_def mapGlobal1_def mapCons_def)

    (* For enablement:  *)
    apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def
     Cons_def FGlobal12Cons_def prems Global1_enablement_def
```

```
         Cons_enablement_def Global1_effect_def Cons_effect_def
         Global1_state.make_def Cons_state.make_def)

        (* For posteffect relation:  *)
        apply (simp add: traceOf_def p0 p1  betaCons_def Global1_def
         Cons_def FGlobal12Cons_def prems Global1_effect_def
         Cons_effect_def Global1_state.make_def Cons_state.make_def)
      done
      qed
    }
    qed
  }
  qed
}
qed
qed

theorem FGlobal12Cons_trans:
  "isFwdSim_trans Global1 Cons FGlobal12Cons mapGlobal1 mapCons"
proof (rule isFwdSim_transRule)
fix  sGlobal1 aGlobal1 sCons
assume
  p0: "enablement_of Global1 sGlobal1 aGlobal1"
  and p1: " FGlobal12Cons sGlobal1 sCons"
  and p2: "reachable Global1 sGlobal1"
show "∃ betaCons . correspExec  Global1 Cons FGlobal12Cons
      mapGlobal1 mapCons sGlobal1 sCons [ aGlobal1] betaCons"
  apply (cases  aGlobal1)
  apply (insert prems)
  apply (simp_all add:  FGlobal12Cons_trans_init)
  apply (simp_all add:  FGlobal12Cons_trans_fail)
  apply (simp_all add:  FGlobal12Cons_trans_decide)
  apply (simp_all add:  FGlobal12Cons_trans_makeBallot)
  apply (simp_all add:  FGlobal12Cons_trans_abstain)
  apply (simp_all add:  FGlobal12Cons_trans_assignVal)
  apply (simp_all add:  FGlobal12Cons_trans_vote)
  apply (simp_all add:  FGlobal12Cons_trans_internalDecide)
done
qed
```

Bibliography

[AS87] Bowen Alpern and Frederick Schneider. Recognizing safety and liveness. *Distributed Computing*, pages 117–126, 1987.

[Att99] Paul C. Attie. Liveness preserving simulation relations. In *Proceedings of the ACM Symposium on Distributed Computing (PODC)*, Atlanta, GA, 1999.

[BGL02] Andrej Bogdanov, Stephen J. Garland, and Nancy A. Lynch. Mechanical translation of I/O automaton specifications into first-order logic. In *22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems*, Houston, TX, November 2002.

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

[CC77a] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, 1977.

[CC77b] Patrick M. Cousot and Radhia Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. In *Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages*, pages 1–12, Rochester, NY, August 1977.

[CC92] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming*, pages 269 –295, Leuven, Belgium, 1992. LNCS 631, Springer-Verlag.

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

[Che98] Anna E. Chefter. A simulator for the IOA language, May 1998. Master of Engineering and Bachelor of Science in Computer Science and Engineering Thesis.

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

[Dea00] Laura G. Dean. Improved simulation of I/O automata. Master of engineering thesis, Massachusetts Institute of Technology, Massachusetts Institute of Technology, Cambridge Massachusetts, September 2000.

[Dod02] Nii Dodoo. Selecting predicates for conditional invariant detection using cluster analysis. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, 2002.

[DPLS$^+$02] Roberto De Prisco, Nancy Lynch, Alex Shvartsman, Nicole Immorlica, and Toh Ne Win. *A Formal Treatment of Lamport's Paxos Algorithm*, 2002. In progress.

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

[ECGN01a] Michael Ernst, Jake Cokrell, William G. Grisworld, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, 2001.

[ECGN01b] 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.

[GC96] Gerald C. Gannod and Betty H.C. Cheng. Strongest postcondition semantics as the formal basis for reverse engineering. *Journal of Automated Software Engineering*, 3(1/2):139–164, June 1996.

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

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

[GL00a] Stephen J. Garland and Nancy A. Lynch. Using I/O automata for developing distributed systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 285–312. Cambridge University Press, 2000.

[GL00b] Stephen J. Garland and Nancy A. Lynch. Using I/O automata for developing distributed systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 13, pages 285–312. Cambridge University Press, USA, 2000.

[Gol90a] Kenneth J. Goldman. *Distributed Algorithm Simulation using Input/Output Automata*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, July 1990. Also,[Gol90b].

[Gol90b] Kenneth J. Goldman. Distributed algorithm simulation using input/output automata. Technical Report MIT/LCS/TR-490, MIT Laboratory for Computer Science, Cambridge, MA, September 1990. Also, PhD Thesis [Gol90a].

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

[Gro02] Alex Groce. Personal correspondance. Personal correspondance, 2002.

[Har02] Michael Harder. Improving test suites via generated specifications. Technical Report 848, MIT Laboratory for Computer Science, Cambridge, MA, June 4, 2002. Revision of author's Master's thesis.

[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.

[JR00] Daniel Jackson and Martin Rinard. The future of software analysis. In *The Future of Software Engineering*, Limerick, Ireland, 2000.

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

[KEGN01] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. Automated support for program refactoring using invariants. In *ICSM 2001, Proceedings of the International Conference on Software Maintenance*, pages 736–743, Florence, Italy, November 6–10, 2001.

[Lam74] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. In *Communications of the ACM*, 1974.

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

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

[Luh02] Christopher Luhrs. Translating from IOA to Isabelle. `http://www.mit.edu/people/cluhrs/index.html`, August 2002.

[LV95a] Nancy Lynch and Frits Vaandrager. Forward and backward simulations — Part I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

[LV95b] Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations – part I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

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

[Min01] Antoine Mine. The octagon abstract domain. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, Suttgart, Germany, October 2001.

[Mül98] Olaf Müller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory.* PhD thesis, Technische Universität München, Munich, Germany, 1998.

[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] Toh Ne Win and Michael Ernst. Verifying distributed algorithms via dynamic analysis and theorem proving. Technical Report 841, MIT, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 25, 2002.

[NE02b] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 232–242, Rome, Italy, 22, 2002.

[NE02c] Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking: An empirical evaluation. In *Proceedings of the ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE 2002)*, Charleston, SC, November 20–22, 2002.

[NEG$^+$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.

[Nim02a] Jeremy W. Nimmer. Automatic generation and checking of program specifications. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 2002.

[Nim02b] Jeremy W. Nimmer. Automatic generation and checking of program specifications. Technical Report 852, MIT Laboratory for Computer Science, Cambridge, MA, June 10, 2002. Revision of author's Master's thesis.

[NS94] Tobias Nipkow and Konrad Slind. I/O automata in Isabelle/HOL. In *Proceedings of the International Workshop on Types for Proofs and Programs*, pages 101–119, Båstad, Sweden, 1994.

[NS01] Toh Ne Win and Gustavo Santos. The IOA-Daikon connection: Enabling dynamic invariant discovery in IOA programs. `theory.lcs.mit.edu/tds/papers/Tohn`, September 2001.

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

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

[Pod03] Andreas Podelski. Software model checking with abstraction refinement. In *VMCAI'03, Fourth International Conference on Verification, Model Checking and Abstract Interpretation*, New York, New York, January 9–11, 2003.

[PPG$^+$96] Tsvetomir P. Petrov, Anna Pogosyants, Stephen J. Garland, Victor Luchangco, and Nancy A. Lynch. Computer-assisted verification of an algorithm for concurrent timestamps. In Reinhard Gotzhein and Jan Bredereke, editors, *Formal Description Techniques IX: Theory, Applications, and Tools* (FORTE/PSTV'96: Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification, Kaiserslautern, Germany, October 1996), pages 29–44. Chapman & Hall, 1996.

[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.

[RR00] J. Antonio Ramırez-Robredo. Paired simulation of I/O automata, September 2000. Master of Engineering and Bachelor of Science in Computer Science and Engineering Thesis, Massachusetts Institute of Technology, Cambridge, MA.

[SAGG$^+$93a] Jørgen F. Søgaard-Andersen, Stephen J. Garland, John V. Guttag, Nancy A. Lynch, and Anna Pogosyants. Computer-assisted simulation proofs. In Costas Courcoubetis, editor, *Computer-Aided Verification* (5th International Conference, CAV'93, Elounda, Greece, June/July 1993), volume 697 of *Lecture Notes in Computer Science*, pages 305–319. Springer-Verlag, 1993.

[SAGG$^+$93b] Jørgen F. Søgaard-Anderson, Stephen J. Garland, John V. Guttag, Nancy A. Lynch, and Anna Pogosyants. Computer-assisted simulation proofs. In Costas Courcoubetis, editor, *Fifth Conference on Computer-Aided Verification*, pages 305–319, Heraklion, Crete, June 1993. Springer-Verlag Lecture Notes in Computer Science 697.

[SGSAL98] Roberto Segala, Rainer Gawlick, Jørgen Søgaard-Andersen, and Nancy Lynch. Liveness in timed and untimed systems. *Information and Computation*, 141(2):119–171, March 1998.