Dynamically Discovering Likely Program Invariants

Michael D. Ernst

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

2000

Program Authorized to Offer Degree: Computer Science and Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Michael D. Ernst

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chairperson of the Supervisory Committee:

David Notkin

Reading Committee:

Craig Chambers

Pedro M. Domingos

William G. Griswold

David Notkin

Date:

University of Washington

Abstract

Dynamically Discovering Likely Program Invariants

by Michael D. Ernst

Chairperson of the Supervisory Committee:
Professor David Notkin
Computer Science and Engineering

This dissertation introduces dynamic detection of program invariants, presents techniques for detecting such invariants from traces, assesses the techniques' efficacy, and points the way for future research.

Invariants are valuable in many aspects of program development, including design, coding, verification, testing, optimization, and maintenance. They also enhance programmers' understanding of data structures, algorithms, and program operation. Unfortunately, explicit invariants are usually absent from programs, depriving programmers and automated tools of their benefits.

This dissertation shows how invariants can be dynamically detected from program traces that capture variable values at program points of interest. The user runs the target program over a test suite to create the traces, and an invariant detector determines which properties and relationships hold over both explicit variables and other expressions. Properties that hold over the traces and also satisfy other tests, such as being statistically justified, not being over unrelated variables, and not being implied by other reported invariants, are reported as likely invariants. Like other dynamic techniques such as testing, the quality of the output depends in part on the comprehensiveness of the test suite. If the test suite is inadequate, then the output indicates how, permitting its improvement. Dynamic analysis complements static techniques, which can be made sound but for which certain program constructs remain beyond the state of the art.

Experiments demonstrate a number of positive qualities of dynamic invariant detection and of a prototype implementation, Daikon. Invariant detection is accurate — it rediscovers formal specifications — and useful — it assists programmers in programming tasks. It runs quickly and produces output of modest size. Test suites found in practice tend to be adequate for dynamic invariant detection.

Table of Contents

List of Figures

Acknowledgments

Portions of this dissertation were previously published at ICSE 99 [ECGN99, Ern99], at ICSE 2000 [ECGN00], and in IEEE *TSE* [ECGN01]. The largest overlaps are with IEEE *TSE* (Chapters 2, 3, and 6) and ICSE 2000 (Chapter 4). The material included in this dissertation has been updated, extended, and rewritten.

I have been fortunate to have Jake Cockrell, Adam Czeisler, William G. Griswold, Yoshio Kataoka, and David Notkin as collaborators in this research. It would not have been brought to its current state without their contributions at every stage. As for me, when my colleagues present their results and stimulate me with thought-provoking exchanges, I work more effectively and with a lighter heart.

I have been especially fortunate to have David Notkin as my advisor. After previous stints as a graduate student at MIT, a researcher at Microsoft Research, and a lecturer at Rice University, at UW I have been happiest and most productive, and David is a major reason for that. He has provided timely and effective guidance and advice on technical and other matters, but the key to his success is that he truly puts his students first. And, his unflagging faith in me and support of my work made more of a difference than I would have ever predicted. In 2000, David won the University of Washington Distinguished Graduate Mentor Award, only the second such honor ever presented.

Bill Griswold acted as a second advisor for this research. His involvement was central from inception — the idea arose during a conversation with him — through all subsequent stages. I am grateful for his many contributions.

Craig Chambers and Dan Weld advised me in previous work at the University of Washington. Their insights have made me a more effective researcher and so contributed to my success in this project, in addition to the direct feedback and guidance they provided on it.

My Ph.D. Supervisory Committee (Brian Bershad, Craig Chambers, Pedro Domingos, Bill Griswold, Butler Lampson, David Notkin, and Mani Soma) provided valuable counsel on both research and writing. I am particularly grateful for the closer guidance of the Reading Committee: Craig Chambers, Pedro Domingos, Bill Griswold, and David Notkin.

My implementation builds on the work of others, whose artifacts greatly eased my task. Rob O'Callahan provided the Lackwit and Ajax tools. EDG donated their C/C++ parser. Dave Shields, Philippe Charles, and IBM Corporation made the Jikes Java compiler into an open-source project that I used both as a Java compiler and as the basis of the Daikon front end for Java. My research has also benefited indirectly from a large collection of other tools, from Emacs and the TeX text formatter (and LaTeX macro package) to GCC, Java, Perl, and Python.

My experiments use programs graciously shared by David Flanagan, Daniel Hoffman, Daniel Jackson, Rustan Leino, Nick Mathewson, Gregg Rothermel, Aung Thaung, Mark

Allen Weiss, and Hongwei Xi.

Many of my colleagues commented on my ideas, put forward suggestions, critiqued drafts, directed me to related work, offered tools, or otherwise enriched my research. I am grateful to Greg Badros, A.J. Bernheim Brush, Brian Bershad, Mike Burrows, Craig Chambers, John Clark, Jake Cockrell, Geoff Cohen, Adam Czeisler, Pedro Domingos, Oren Etzioni, John Field, Michael Gorlick, Ron Graham, Bill Griswold, Zack Ives, Daniel Jackson, Kalon Jelen, Josh Kataoka, Chris Laffra, Butler Lampson, Tessa Lau, Rustan Leino, Vass Litvinov, Doug Lorch, David Madigan, Todd Millstein, George Necula, James Noble, David Notkin, Jon Nowitz, Rob O'Callahan, Denise Pinnel, Rachel Pottinger, Stuart Russell, Jared Saia, Vibha Sazawal, Mani Soma, Neil Spring, V. C. Sreedhar, Aung Thaung, Frank Tip, Nigel Tracey, Steve Wolfman, and Hongwei Xi.

Aung Thaung performed the analysis of student programs reported in Section 2.3.

Craig Kaplan carved the Daikon logo.

The anonymous ICSE 99, ICSE 2000, and IEEE *TSE* referees improved both the research and the presentation, as did audiences at a variety of talks. I received useful feedback from attendees at the ICSE 99 Doctoral Symposium, notably Gregory Abowd, Bill Griswold, and Mary Shaw.

This research was supported in part by NSF grant CCR-9970985, an IBM Graduate Fellowship, and gifts from Edison Design Group, Microsoft Corporation, and Toshiba Corporation.

I have received so much generous assistance that it seems inevitable that despite my best intentions, I have inadvertently omitted someone from these lists. If so, accept my apologies and be assured that I am grateful for your help.

Chapter 1

Introduction

A program invariant is a property that is true at a particular program point or points, such as might be found in an `assert` statement, a formal specification, or a representation invariant. Examples include $\mathsf{y} = 4 * \mathsf{x} + 3$; $\mathsf{x} > \mathsf{abs(y)}$; array a contains no duplicates; $\mathsf{n} = \mathsf{n.child.parent}$ (for all nodes n); $\mathsf{size(keys)} = \mathsf{size(contents)}$; and graph g is acyclic.

Invariants explicate data structures and algorithms and are helpful for programming tasks from design to maintenance. As one example, they identify program properties that must be preserved when modifying code. Despite their advantages, invariants are usually missing from programs. An alternative to expecting programmers to fully annotate code with invariants is to automatically infer likely invariants from the program itself. This research focuses on dynamic techniques for discovering invariants from execution traces. A dynamic detector of program invariants examines variable values captured during execution over a test suite and reports properties and relationships that hold over those values.

This chapter discusses how to obtain invariants (Section 1.1), introduces dynamic invariant detection (Section 1.2), discusses two issues having to do with the use of a dynamic technique (test suites in Section 1.3 and usage properties in Section 1.4), lists some uses for invariants (Section 1.5), lists the contributions of the dissertation (Section 1.6), and gives a roadmap to the remainder of the document (Section 1.7).

1.1 Ways to obtain invariants

I contend that most programmers have invariants in mind, consciously or unconsciously, when they write or otherwise manipulate programs: they have an idea of how the system works or is intended to work, how the data structures are laid out and related to one another, and the like. Regrettably, these notions are rarely written down, and so most programs are completely lacking in formal or informal invariants and other documentation.

An alternative to expecting programmers to annotate code with invariants is to automatically infer invariants. Invariant detection recovers a hidden part of the design space: the invariants that the programmer had in mind. This can be done either statically or dynamically.

Static analysis examines the program text and reasons over the possible executions and runtime states. The most common static analysis is dataflow analysis, with abstract interpretation as its theoretical underpinning. The results of a conservative, sound analysis are guaranteed to be true for all possible executions; it is most appropriate when correctness is crucial, as for compilers and some other systems whose consumer is not a human.

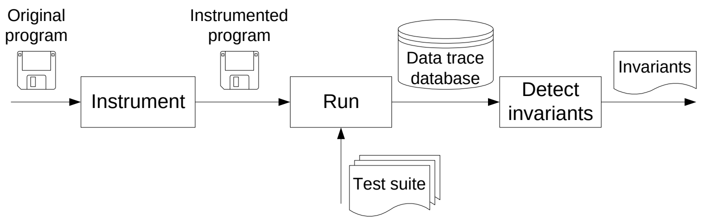


Figure 1.1: Architecture of the Daikon tool for dynamic detection of program invariants.

Static analysis has a number of limitations. It cannot report true but undecidable properties or properties of the program context. Static analysis of programs using language features such as pointers remains beyond the state of the art because the difficulty of representing the heap forces precision-losing approximations and produces weak results. Dynamic analysis, which runs the program, examines the executions, and reports properties over those executions, does not suffer these drawbacks and so complements static analysis.

1.2 Dynamic invariant detection

This research focuses on the dynamic discovery of invariants: the technique is to execute a program on a collection of inputs and infer invariants from captured variable traces. Figure 1.1 shows the high-level architecture of the Daikon invariant detector, which is named after an Asian radish.

Dynamic invariant detection discovers likely invariants from program executions by instrumenting the target program to trace certain variables, running the instrumented program over a test suite, and inferring invariants over both the instrumented variables and derived variables not manifest in the program.

All of the steps are fully automatic (except selecting a test suite). The currently-existing instrumenters are source-to-source translators for C, Java, and Lisp; we use the terms "instrumenter" and "front end" interchangeably.

The inference step tests possible invariants against the values captured from the instrumented variables. Properties that are satisfied over all the values, and that also satisfy other tests, such as being statistically justified, not being over unrelated variables, and not being implied by other reported invariants, are reported as likely invariants.

1.3 Running the program

Dynamic analysis requires executing the target program. A test suite's benefits outweigh its costs even in the absence of dynamic invariant detection, for it enables other dynamic techniques such as (regression) testing. Indeed, a program lacking a test suite (or that cannot be run or never has) is likely to have many problems that should be addressed using standard techniques before invariants are detected over it.

As with other dynamic approaches such as testing and profiling, the accuracy of the inferred invariants depends in part on the quality and comprehensiveness of the test cases. Additional test cases might provide new data from which more accurate invariants can be inferred. The inferred invariants can also validate the test suite by revealing properties true when executing it. Thus, users know whether a test suite is adequate and, if not, are directly informed how to improve it.

In practice, standard test suites have performed adequately, and detected invariants are relatively insensitive to the particular test suite, so long as it is large enough (see Chapter 6). However, we are not yet sure of the precise properties that make a test suite good for invariant detection. These are not necessarily the same as make a test suite good for detecting bugs. A test suite that strives for efficiency by executing each statement a minimal number of times would be bad for invariant detection, which requires multiple executions as a basis for generalization so that there is statistical support for the inferences.

Like any dynamic analysis, dynamic invariant detection cannot guarantee the completeness or soundness of its results. It is not complete because there are infinitely many potential invariants that could be reported. However, it is complete for the set of invariants it checks; Daikon's grammar is given in Section 3.2 and extended later in the dissertation, most notably in Section 4.3. It is not sound because the test suite may not fully characterize all executions: a property that held for the first 10,000 executions does not necessarily hold on the next one. However, the technique is sound over the training data (all the data presented to it).

Although dynamic invariant detection is not complete or sound, it is useful, which is a considerably more important property. (This characterization fits other tools, such as testing, Purify [HJ92], ESC [DLNS98], PREfix [PRE99], and many more.) Additionally, it is complementary to other techniques; it can shore up the weaknesses of static analysis while static analysis covers for its deficiencies.

1.4 Functional invariants and usage properties

Because the results of dynamic invariant detection depend on the particular test suite, not all reported invariants will be true for every possible execution of the program. The Daikon output can be generally classified into functional invariants and usage properties.

A functional invariant depends only on the code for a particular data structure or function, and the invariant is universally true for any use of that entity. Usage properties, on the other hand, result from specific usage of a data structure or function; they depend on the context of use and the test suite. Because it operates on traces, Daikon cannot distinguish between these classes, which are intermingled in its output. (The heuristics of Chapter 4, and notably the statistical checks of Section 4.5, can help to separate them, as can varying the test suite.)

People cannot necessarily discriminate between the two, either. The distinction between functional invariants and usage properties is clear only at a system's entry and exit — for a standalone Java or C program, at the beginning and end of the `main` routine. At other points in the program, the differences are less clear. For instance, a precondition stating that a procedure's arguments are valid can be viewed as a functional invariant of the procedure or as a usage property of the callers, which take care not to supply illegal values. (In the formal specification literature, a precondition, if met, guarantees that the postcondition will be true upon exit. All preconditions and postconditions reported by Daikon satisfy this criterion, though Daikon's preconditions will not in general be weakest preconditions.)

Usage properties are useful in explicating program operation or the limited contexts in which a function is called or a data structure is used. Because the distinction between functional invariants and usage properties is irrelevant to working programmers, we will henceforth set aside the goal of detecting only the former.

1.5 Uses for invariants

Invariants are useful, to humans and to tools, in all aspects of programming, including design, coding, testing, optimization, and maintenance. This section lists some uses of invariants, to motivate why programmers care about them and why extracting them from programs is a worthwhile goal.

Write better programs. A number of authors have noted that better programs result when invariants are used in their design [Gri81, LG86]. Invariants precisely formalize the contract of a piece of code, clarifying its intended operation and indicating when it is complete. Thinking about code formally can result in more disciplined design and implementation, but even informal use of invariants can help programmers [HHJ$^+$87, HHH$^+$87]. Other authors suggest making invariants an essential part of implementation, refining a specification into a program [CM88, BG93, FM97]. Although these uses are valuable, this dissertation focuses on uses of invariants in already-constructed programs.

Document code. Invariants characterize certain aspects of program execution and provide valuable documentation of a program's operation, algorithms, and data structures. As such, they assist program understanding, some level of which is a prerequisite to every program manipulation. Documentation that is automatically extracted from the program is guaranteed to be up-to-date, unlike human-written information that may not be updated when the code is.

Refine documentation. Automatically inferred invariants are useful even for code that is already documented with comments, `assert` statements, or specifications. The inferred invariants can check or refine programmer-supplied invariants; program self-checks are often out-of-date, ineffective, or incorrect [LCKS90]. Furthermore, human cross-checks are weak because different people tend to make the same mistakes [KL86].

Check assumptions. Invariants can be inserted into a program as `assert` statements for further testing or to ensure that detected invariants are not later violated as code evolves. Program types are another such assumption that can be checked at compile time, at run time, or both.

Avoid bugs. Invariants can protect a programmer from making changes that inadvertently violate assumptions upon which the program's correct behavior depends. The near absence of explicit invariants in existing programs makes it all too easy for programmers to introduce errors while making changes. An invariant that is established at one point is likely to be depended upon elsewhere, but if the original invariant is not documented, much less the dependence, then it is easy for a programmer to violate it, introducing a bug in a distant part of the program. Program maintenance introduces errors [OC89, GKMS], and anecdotally, many of these are due to violating invariants.

Helping programmers avoid introducing bugs was the original motivation for dynamic invariant detection. This activity is as valuable as detecting bugs (even if harder to quantify and less glamorous), because preventing a problem is easier and cheaper than setting it right later on.

Either statically or dynamically specified invariants serve for the uses listed above (except the first one, use in program design). For the remaining uses, dynamically detected invariants can be even more useful than static ones.

Form a spectrum. A program spectrum [AFMS96, RBDL97, HRWY98] is any measurable property of a program or its execution. Examples include the set of lines (or paths) executed by a program, the size of the output, runtime, or static properties such as cyclomatic complexity [McC76] of the source. Informally, a spectrum can be thought of as a summary or a hash code; differences between program spectra can indicate or characterize differences between programs, inputs, or executions. Dynamically detected invariants also form a program spectrum, changes to which can indicate properties of a changed program or input and be used just as other spectra are.

Locate unusual conditions. Unusual or exceptional conditions may indicate bugs or special cases that should be brought to a programmer's attention. A nearly-true invariant could indicate a situation requiring special care or an input anomaly.

Validate test suites. Dynamically detected invariants can reveal as much about a test suite as about the program itself, because the properties reflect the program's execution over the test suite. An invariant might reveal that the program manipulates only small values, or only positive ones, or that some variables are always in a particular relationship to one another. These properties may indicate insufficient coverage of program values (and states) and inadequate exercising of program behavior, even if the suite covers every line or path of the program. These invariants can assist in test case generation in one of two different ways. New tests can intentionally violate invariants cite1998:ase:tracey, improving the suite by broadening its value coverage (similar to but broader than operator coverage [CR99], which requires that two variables must have different values). Alternately, new tests can respect invariants over program runs, so that the test suite characterizes the actual, correct usage of the program or component in practice.

Optimize common cases. Profile-directed compilers optimize programs using information gathered on previous runs. If a particular value or condition is common, cheap to test, and permits useful specialization when it holds, then a compiler can (among other techniques) insert checks and branch to a specialized version of the code that assumes the condition. As one example, variable aliasing is a pointer equality test at runtime, and the specialized versions of a routine for the aliased and not-aliased case can both be more efficient than the general case.

The low-level execution information used in profile-directed compilation (usually just the most common values for single variables) can be augmented with higher-level invariants to enable better optimization for the common case. Invariants that refer to program structures rather than memory locations or registers could permit entire data structure operations to be avoided or optimized.

Bootstrap proofs. Theorem-proving, dataflow analysis, model-checking, and other automated or semi-automated mechanisms can verify the correctness of a program with respect to a specification, confirm safety properties such as lack of bounds overruns or null dereferences, establish termination or response properties, and otherwise increase confidence in an implementation. However, it can be tedious and error-prone for people to specify the properties to be proved, and current systems have trouble postulating them; some researchers consider that task harder than performing the proof [Weg74, BLS96]. Dynamically detected program invariants could be fed into an automated system, relieving the human of the need to fully hand-annotate their programs to supply properties for validation — a task that few programmers are either skilled at or enjoy.

1.6 Hypothesis and contributions

The thesis of this research is that dynamic invariant detection — discovering likely program invariants via a dynamic analysis that examines values computed by a program in order to find properties over, and relationships among, them — is an effective and efficient technique for discovering invariants which can aid programmers and tools in a variety of tasks.

At the beginning of this research, this hypothesis was very much in doubt; many researchers said that it was a foolhardy approach unlikely to succeed. A few of the potential hurdles are as follows. It might have been the case that no dynamic technique for extracting properties from data traces existed. A dynamic technique might have been too inefficient, or the volume of trace data might have made the technique computationally intractable. The technique might not have computed sufficiently accurate invariants. The technique might have computed so many inaccurate invariants that they swamped the accurate ones, rendering the latter impossible or excessively difficult to pick out. Test suites found in practice might have been inappropriate for invariant detection, and good ones might have been prohibitively costly to construct. The resulting invariants might have been too test-suite-specific, not approximating the true static invariants. The detected invariants might not have been good for any tasks, even if accurate. People might not have been able to use the invariants in practice.

The first contribution of this research is the notion of dynamic invariant detection and a technique for dynamically detecting likely program invariants. This research area had not previously been clearly enunciated or pursued in a sustained fashion.

The second contribution is enhancements to the basic invariant detection techniques to improve its performance, the domains of invariants detected, and the relevance, or usefulness to programmers, of its output. A straightforward implementation of dynamic invariant discovery would suffer from poor performance and report too few desirable invariants and too many undesirable ones.

The third contribution is two prototype implementations which demonstrate that dynamic invariant detection is feasible. Although the idea of dynamic invariant detection is simple, considerable engineering was required in the design and implementation of these systems. The invariant detectors are complemented by front ends which enable them to work over C, Lisp, and Java programs. All of this code is publicly available.

The fourth contribution is evaluation of the techniques and implementations. A series of experiments show the qualities and benefits of dynamically detected invariants. The invariants are accurately capture what programmers consider important about their own code. The invariants are useful, assisting programmers in a variety of ways during software modification tasks. The implementation and the resulting invariants are scalable, and generally available test suites are adequate for invariant detection.

1.7 Outline

The remainder of this dissertation is organized as follows.

Chapter 2 introduces and motivates dynamic invariant detection and the Daikon tool by describing the results of three experiments. The first experiment shows that Daikon is accurate: given programs which were derived from formal specifications, it recovered (and improved) those specifications. Other experiments later in the dissertation show similar results. The chapter's second experiment demonstrates that Daikon is useful: its output assisted programmers in modifying a C program by explicating data structures, revealing bugs, preventing introduction of other bugs, showing specialized procedure use, indicating poor test suite coverage, validating changes, and more. The third experiment shows a correlation between invariants detected in student programs and the grades awarded those programs. The chapter also lists and briefly describes a number of other experiments.

Chapter 3 describes the basic technique for dynamic detection of invariants: check each of a selection of potential invariants over all variables or combinations thereof. The chapter also lists the invariants built into Daikon and describes how to infer object invariants.

Chapter 4 presents, and gives experimental evidence of the efficacy of, five approaches for increasing the relevance — the usefulness to a programmer in performing a task — of invariants reported by a dynamic invariant detector. Two of them (adding implicit values and exploiting unused polymorphism) add desired invariants to the output. The other three (statistical confidence checks, suppressing redundant invariants, and limiting which variables are compared to one another) eliminate undesired invariants from the output and also improve runtime by reducing the work done by the invariant detector.

Chapter 5 extends dynamic invariant detection to recursive data structures. A first technique, linearization, traverses implicit collections and records them explicitly as arrays in the program traces, making them available to the basic Daikon invariant detector. A second technique, data splitting, splits data traces into multiple parts based on predicates Daikon chooses, then detects invariants in each part. This enables discovery of conditional invariants that are not universally satisfied, but whose truth is dependent on some other condition. The chapter discusses several policies and a mechanism for computing such invariants.

Chapter 6 contains three experimental results. The first analyzes the time and space costs of dynamic invariant inference, which grow modestly with the number of program points and variables instrumented, number of invariants checked, and number of test cases run. The second result shows that relatively small test suites enable effective invariant inference. The third examines the feasibility of automatically generating test suites for invariant detection.

Chapter 7 discusses the implementation of the Daikon invariant detector. These include its design goals and file formats; details of program instrumentation; architecture of the invariant detector proper; and how users can extend it.

Chapter 8 surveys related work. This can be divided into three main categories: other dynamic techniques for inferring invariants, including machine learning; static techniques for inferring invariants; and checking invariants.

Chapter 9 suggests avenues for future work, including new varieties of invariants to check, implementation improvements, user interfaces to control invariant detection and and display invariants, and experimental evaluation of invariant detection. All of these topics will help invariant detection scale to larger programs and to use in real-world programming tasks.

Finally, Chapter 10 concludes with a summary of the contributions and a set of lessons learned.

Chapter 2

Sample applications of dynamic invariant detection

To introduce dynamic invariant detection and illustrate Daikon's output, this chapter presents several applications of invariant detection. Section 2.1 shows that Daikon is accurate: given formally-specified programs from which the formal specifications had been erased, Daikon produced (and improved) those invariants. Section 2.2 demonstrates that dynamic invariant detection is useful — a more important quality than accuracy. Daikon's output assisted programmers in modifying a C program. Section 2.3 shows a correlation between invariants detected in student programs and the grades awarded those programs. Finally, Section 2.4 briefly describes a number of other codebases to which Daikon has been applied.

2.1 Rediscovery of formal specifications

This section presents the invariants detected in a simple program taken from *The Science of Programming* [Gri81], a book that espouses deriving programs from specifications. Unlike typical programs, for which it may be difficult to determine the desired output of invariant detection, many of the book's programs include preconditions, postconditions, and loop invariants that embody the properties of the computation that the author considered important. These specifications form a "gold standard" against which an invariant detector can be judged. Thus, these programs are ideal initial tests of our system.

Daikon successfully reports all the formally-specified preconditions, postconditions, and loop invariants in Chapters 14 and 15 of the book. (After this success, we did not feel the need to continue with the following chapters.) Chapter 14 is the first containing formally-specified programs; previous chapters present the underlying mathematics and methodology. These programs perform simple tasks such as searching, sorting, changing multiple variables consistently, GCD, and the like. We did not investigate a few programs whose invariants were described via pictures or informal text rather than mathematical predicates.

All the programs are quite small, and we built simple test suites of our own. These experiments are not intended to be conclusive, but to be a good initial test. The programs are small enough to show in full, along with the complete Daikon output. Additionally, they illustrate a number of important issues in invariant detection, so we will periodically return to them for pedagogical reasons.

As a simple example of invariant detection, consider a program that sums the elements of an array (Figure 2.1). We transliterated this program to a dialect of Lisp enhanced with Gries-style control constructs such as nondeterministic conditionals (Figure 2.2). Daikon's Lisp instrumenter (Section 7.3) added code that writes variable values into a data trace file; this code was automatically inserted at the program entry (`ENTER`), at the loop head

```
$i, s := 0, 0;$
do $i \neq n \rightarrow$
    $i, s := i + 1, s + b[i]$
od
```

Precondition: $\mathsf{n} \geq 0$
Postcondition: $\mathsf{s} = (\sum \mathsf{j} : 0 \leq \mathsf{j} < \mathsf{n} : \mathsf{b}[\mathsf{j}])$
Loop invariant: $0 \leq \mathsf{i} \leq \mathsf{n}$ and $\mathsf{s} = (\sum \mathsf{j} : 0 \leq \mathsf{j} < \mathsf{i} : \mathsf{b}[\mathsf{j}])$

Figure 2.1: Gries array sum program (Program 15.1.1 [Gri81, p. 180]) and its formal specification. The program sums the values in array b (of length n) into result variable s. The statement $i, s := 0, 0$ is a parallel (simultaneous) assignment of the values on the right-hand side of the := to the variables on the left-hand side. The **do-od** form repeatedly evaluates the condition on the left-hand side of the $\rightarrow$ and, if it is true, evaluates the body on the right-hand side; execution of the form terminates when the condition evaluates to false.

```
;; Gries's The Science of Programming, page 180, Program 15.1.1
(defun p180-15.1.1 (b n)
  (declare (type (array integer 1) b)
           (type integer n))
  (let ((i 0) (s 0))
    (declare (type integer i s))
    (do-od ((/= i n)
            (psetq i (+ i 1)
                   s (+ s (aref b i))))))
```

Figure 2.2: Lisp transliteration of Gries array sum program (Figure 2.1). The `do-od` and `psetq` Lisp macros implement Gries-style program semantics [Gri81]. Daikon's Lisp instrumenter uses the type declarations to specify the format of the data trace file; the variable types are mentioned in the book's text but absent from its code. The Daikon distribution includes Lisp transliterations of all the programs from Gries's Chapters 14 and 15. It is available from `http://www.cs.washington.edu/homes/mernst/daikon` or from the author on request.

(`LOOP`), and at the program exit (`EXIT`). We ran the instrumented program on 100 randomly-generated arrays of length 7 to 13, in which each element was a random number in the range −100 to 100, inclusive. Figure 2.3 shows the output of the Daikon invariant detector given the data trace file.

(This test suite was the first one we tried when testing Daikon. Figure 4.3 (page 38) shows Daikon's output when the array sum program is run over a different test suite. Chapter 6 (page 69) discusses the selection of test suites.)

The preconditions (invariants at the `ENTER` program point) of Figure 2.3 record that `N` is the length of array `B`, that `N` falls between 7 and 13 inclusive, and that the array elements fall between −100 and 100 inclusive. The first invariant, $\mathsf{N} = \mathsf{size}(\mathsf{B})$, is crucial to the correctness of the program, yet was omitted from the formal invariants stated by Gries.

```
15.1.1:::ENTER                   100 samples
   N = size(B)                     (7 values)
   N in [7..13]                    (7 values)
   B                               (100 values)
     All elements in [-100..100]   (200 values)

15.1.1:::EXIT                    100 samples
   N = I = orig(N) = size(B)       (7 values)
   B = orig(B)                     (100 values)
   S = sum(B)                      (96 values)
   N in [7..13]                    (7 values)
   B                               (100 values)
     All elements in [-100..100]   (200 values)

15.1.1:::LOOP                    1107 samples
   N = size(B)                     (7 values)
   S = sum(B[0..I-1])              (452 values)
   N in [7..13]                    (7 values)
   I in [0..13]                    (14 values)
   I <= N                          (77 values)
   B                               (100 values)
     All elements in [-100..100]   (200 values)
   B[0..I-1]                       (985 values)
     All elements in [-100..100]   (200 values)
```

Figure 2.3: Invariants inferred for the Gries array sum program (Program 15.1.1 [Gri81], Figures 2.1 and 2.2), which sums array `B` of length `N` into variable `S`. This is the complete Daikon output, for uniformly-distributed arrays of length between 7 and 13 and elements between −100 and 100.

Invariants are shown for the entry (precondition) and exit (postcondition) of the program, as well as the loop head (the loop invariant). Daikon successfully rediscovered the invariants in the program's formal specification (Figure 2.1); those goal invariants are boxed for emphasis.

At the exit, `orig(`*var*`)` represents *var*'s value at the corresponding entry. Invariants for elements of an array are listed indented under the array; in this particular output, no array has multiple elementwise invariants. The number of samples (listed to the right of each program point name) is the number of times the program point was executed; the loop iterates multiple times for each test case, generating multiple samples. The counts of values, in the right-hand column, indicate how many distinct variable values were encountered. For instance, although the program was exited 100 times, the boxed postcondition $\mathsf{S} = \mathsf{sum}(\mathsf{B})$ indicates that there were only 96 distinct final values for variable `S` (and for `sum(B)`) on those 100 executions.

Gries's stated precondition, $\mathsf{N} \geq 0$, is implied by the boxed output, $\mathsf{N} \in [7\,..\,13]$, which is shorthand for $\mathsf{N} \geq 7$ and $\mathsf{N} \leq 13$.

The postconditions (at the `EXIT` program point) include the Gries postcondition, $\mathsf{S} = \mathsf{sum}(\mathsf{B})$; Section 4.3 describes inference over functions such as sum. In addition, Daikon discovered that `N` and `B` remain unchanged; in other words, the program has no side effects on those variables.

The loop invariants (at the `LOOP` program point) include those of Gries, along with several others. Since Gries provided loop invariants, recovering them was a reasonable goal for this experiment. However, the remainder of this dissertation will not consider loop invariants. Such local invariants may be easier for a programmer to infer and less relevant for most program changes. A loop invariant primarily relates to code in the loop, while a function pre- or post-condition or an object invariant may have larger-reaching dependences.

In Figure 2.3, invariants that appear as part of the formal specification of the program in the book are boxed for emphasis. Invariants beyond those can be split into three categories. First are invariants erroneously omitted from the formal specification but detected by Daikon, such as $\mathsf{N} = \mathsf{size}(\mathsf{B})$. Second are properties of the test suite, such as $\mathsf{N} \in [7..13]$. These invariants provide valuable information about the data set and can help validate a test suite or indicate the contexts in which a function or other computation is used. Third are extraneous, probably uninteresting invariants, which do not appear in this example (see Chapter 4, especially Figure 4.4).

In this example, Daikon detected $\mathsf{N} = \mathsf{size}(\mathsf{B})$ because that property holds in the test cases, which were written to satisfy the intent of the author (as made clear in the book). To express this intent, the postcondition should have been $\mathsf{s} = (\sum \mathsf{j} : 0 \leq \mathsf{j} < \mathsf{size}(\mathsf{B}) : \mathsf{b}[\mathsf{j}])$. The same code could be used in a different way, to sum part of an array, with precondition $\mathsf{N} \leq \mathsf{size}(\mathsf{B})$ and the existing postcondition. A different test suite could indicate such uses of the program.

The fact that Daikon found the fundamental invariants in the Gries programs — including crucial ones not specified by Gries — demonstrates the potential of dynamic invariant detection. For this toy program, which was small enough to exhaustively discuss in this section, static analysis could produce the same result, but this is not true in general.

2.2 Invariant use in program modification

While Section 2.1 demonstrated dynamic invariant detection's accuracy, that property is little consolation unless the invariants are useful to programmers or to tools. This section reports an experiment in which inferred invariants were of substantial assistance in understanding, modifying, and testing a program that contains no explicitly-stated invariants. To determine whether and how derived invariants aid program modification, two programmers working as a team modified a program. The programmers used both Daikon's output and traditional tools.

This section lays out the task, describes the programmers' activity in modifying the program, and discusses how the use of invariants is qualitatively different from more traditional styles of gathering information about programs.

2.2.1 The task

The Siemens `replace` program takes a regular expression and a replacement string as command-line arguments, then copies an input stream to an output stream while replacing any substring matched by the regular expression with the replacement string. The `replace` program consists of 563 lines of C code and contains 21 procedures. The program has no comments or other documentation, which is regrettably typical for real-world programs. The `replace` program comes from the Siemens suite [HFGO94] as modified by Rothermel

```
...
else if ((arg[i] == CLOSURE) && (i > start))
{
    lj = lastj;
    if (in_set_2(pat[lj]))
        done = true;
    else
        stclose(pat, &j, lastj);
}
...
```

Figure 2.4: Function `makepat`'s use of constant `CLOSURE` in Siemens program `replace`.

and Harrold [RH98]. These programs are used in research on program testing and come with extensive test suites.

The regular expression language of `replace` includes Kleene-* closure [Kle56] but omits Kleene-+ closure, so we decided that this would be a useful and realistic extension. In preparation for the change, we instrumented and ran `replace` on 100 randomly selected test cases from the 5542 provided with the Siemens suite. (We used a small test suite because of fears that Daikon would not scale to larger ones. These fears were unjustified, and yet the small test suite was adequate for our purposes; see Chapter 6 for details on both points.) Given the resulting trace, Daikon produced invariants at the entry and exit of each procedure. We provided the output to the programmers making the change, who then worked completely independently of us. As described below, they sometimes used the dynamically detected invariants and sometimes found traditional tools and techniques more useful.

The programmers in the study were two software engineering professors who are colleagues of the author. As such, they have several qualities that may not be characteristic of all programmers: they are far from full-time programmers, for they spend most of their time on advising, managing, and teaching; they were already familiar with the notion of invariants, even if they did not always use them in their own work; they were motivated to make the experiment succeed; and they provided helpful and detailed feedback on on their mental processes and on the successes and failures of the tool and approach. We believe the experiment was fair, but it remains to be validated by additional repetitions with other subjects.

2.2.2 Performing the change

The programmers began by studying the program's call structure and high-level definitions (essentially a static analysis) and found that it is composed of a pattern parser, a pattern compiler, and a matching engine. To avoid modifying the matching engine and to minimize changes to the parser, they decided to compile an input pattern of the form ⟨*pat*⟩`+` into the semantically equivalent ⟨*pat*⟩⟨*pat*⟩`*`.

The initial changes were straightforward and were based on informal program inspection and manual analysis. In particular, simple text searches helped the programmers find how "`*`" was handled during parsing. They mimicked the constant `CLOSURE` of value `'*'` with the

```
void stclose(pat, j, lastj)
char    *pat;
int     *j;
int     lastj;
{
    int jt;
    int jp;
    bool        junk;

    for (jp = *j - 1; jp >= lastj ; jp--)
    {
        jt = jp + CLOSIZE;
        junk = addstr(pat[jp], pat, &jt, MAXPAT);
    }
    *j = *j + CLOSIZE;
    pat[lastj] = CLOSURE;
}
```

Figure 2.5: Function `stclose` in Siemens program `replace`. This was the template for the new `plclose` function (Figure 2.7).

new constant `PCLOSURE` (for “plus closure”) of value `'+'` and made several simple changes, such as adding `PCLOSURE` to sets that represent special classes of characters (in functions `in_set_2` and `in_pat_set`).

They then studied the use of `CLOSURE` in function `makepat`, since `makepat` would have to handle `PCLOSURE` analogously. The basic code in `makepat` (Figure 2.4) determines whether the next character in the input is `CLOSURE`; if so, it calls the “star closure” function, `stclose` (Figure 2.5) under most conditions (and the exceptions should not differ for plus closure). The programmers duplicated this code sequence, modifying the copy to check for `PCLOSURE` and to call a new function, `plclose`. Their initial body for `plclose` was a copy of the body of `stclose`.

To determine appropriate modifications for `plclose`, the programmers studied `stclose`. The initial, static study of the program determined that the compiled pattern is stored in a 100-element array named `pat`. They speculated that the uses of array `pat` in `stclose`’s loop manipulate the pattern that is the target of the closure operator, adding characters to the compiled pattern using the function `addstr`.

The programmers wanted to verify that the loop was indeed entered on every call to `stclose`. Since this could depend on how `stclose` is called, which could depend in turn on unstated assumptions about what is a legal call to `stclose`, they decided to examine the invariants for `stclose` rather than attempt a global static analysis of the program. The initialization and exit condition in `stclose`’s loop imply the loop would not be entered if `*j` were equal to `lastj`, so they examined the invariants inferred for those variables on entry to `stclose`:

$$
\begin{aligned}
*\mathsf{j} &\geq 2 \\
\mathsf{lastj} &\geq 0 \\
\mathsf{lastj} &\leq *\mathsf{j}
\end{aligned}
$$

```
int addstr(c, outset, j, maxset)
char    c;
char    *outset;
int     *j;
int     maxset;
{
    bool        result;
    if (*j >= maxset)
        result = false;
    else {
        outset[*j] = c;
        *j = *j + 1;
        result = true;
    }
    return result;
}
```

Figure 2.6: Function `addstr` in Siemens program `replace`.

The third invariant implies that the loop body may not be executed (if $\mathsf{lastj} = *\mathsf{j}$, then `jp` is initialized to `lastj-1` and the loop body is never entered), which was inconsistent with the programmers' initial belief.

To find the offending values of `lastj` and `*j`, they queried the trace database for calls to `stclose` in which $\mathsf{lastj} = *\mathsf{j}$, since these are the cases when the loop is not entered. (Daikon includes a tool that takes as input a program point and a constraint and produces as output the tuples in the execution trace database that satisfy — or, optionally, falsify — the constraint at the program point.) The query returned several calls in which the value of `*j` is 101 or more, exceeding the size of the array `pat`. The programmers soon determined that, in some instances, the compiled pattern is too long, resulting in an unreported array bounds error. This error was apparently not noticed previously, despite a test suite of 5542 test cases.

Excluding these exceptional situations, the loop body in `stclose` always executes when the function is called, increasing the programmers' confidence that the loop manipulates the pattern to which the closure operator is being applied. To allow them to proceed with the Kleene-+ extension without first fixing this bug, we recomputed the invariants without the test cases that caused the improper calls to `stclose`.

Studying `stclose`'s manipulation of array `pat` (Figure 2.5) more carefully, they observed that the loop index is decremented, and `pat` is both read and written by `addstr`. Moreover, the closure character is inserted into the array not at the end of the compiled pattern, but at index `lastj`. Looking at the invariants for `pat`, they found $\mathsf{pat} \neq \mathsf{orig(pat)}$, which indicates that `pat` is always updated. To determine what `stclose` does to `pat`, they queried the trace database for values of `pat` at the entry and exit of `stclose`. For example:

Test case: `replace "ab*" "A"`
  values of parameter `pat` for calls to `stclose`:
    in value: `pat` = "cacb"

```
void plclose(pat, j, lastj)
char    *pat;
int     *j;
int     lastj;
{
    int jt;
    int jp;
    bool        junk;

    jt = *j;
    addstr(CLOSURE, pat, j, MAXPAT);
    for (jp = lastj; jp < jt; jp++)
    {
        junk = addstr(pat[jp], pat, j, MAXPAT);
    }
}
```

Figure 2.7: Function `plclose` in the extended `replace` program. It was written by copying `stclose` (Figure 2.5), then modifying the copy.

out value: `pat` = "ca*cb"

This suggests that the program compiles literals by prefixing them with the character `c` and puts Kleene-* expressions into prefix form. (I independently discovered this fact through careful study of the program text, though the programmers were not aware of this.) In the compiled pattern ca*cb, ca stands for the character `a`, cb stands for the character `b`, and * modifies cb.

The negative indexing and assignment of * into position `lastj` moves the closed-over pattern rightward in the array to make room for the prefix *. For a call to `plclose` the result for the above test case should be `cacb*cb`, which would match one or more instances of character `b` rather than zero or more. The new implementation of Kleene-+ requires duplicating the previous pattern, rather than shifting it rightward, so the Kleene-+ implementation can be a bit simpler. After figuring out what `addstr` is doing with the address of the index passed in (it increments the index unless the array bound is exceeded), the programmers converged on the version of `plclose` in Figure 2.7.

To check that the modified program does not violate invariants that should still hold, they added test cases for Kleene-+ and we recomputed the invariants for the modified program. As expected, most invariants remained unchanged, while some differing invariants verified the program modifications. Whereas `stclose` has the invariant $*\mathsf{j} = \mathsf{orig}(*\mathsf{j}) + 1$, `plclose` has the invariant $*\mathsf{j} \geq \mathsf{orig}(*\mathsf{j}) + 2$. This difference was expected, since the compilation of Kleene-+ replicates the entire target pattern, which is two or more characters long in its compiled form.

2.2.3 Invariants for `makepat`

In the process of changing `replace`, the programmers also investigated several invariants discovered for function `makepat` (among others). In determining when `stclose` is called — to learn more about when the new `plclose` will be called — the `makepat` invariants showed them that parameter `start` (tested in Figure 2.4) is always 0, and parameter `delim`, which controls the outer loop, is always the null character (character 0). These invariants indicated that `makepat` is used only in specialized contexts, saving considerable effort in understanding its role in pattern compilation. The programmers reported doing mental partial evaluation to understand the specific use of the function in the program.

The programmers had hypothesized that both `lastj` and `lj` in `makepat` should always be less than local `j` (i.e., `lastj` and `lj` refer, at different times, to the last generated element of the compiled pattern, whereas `j` refers to the next place to append). Although the invariants for `makepat` confirmed this relation over `lastj` and `j`, no invariant between `lj` and `j` was reported. A query on the trace database at the exit of `makepat` returned several cases in which `j` is 1 and `lj` is 100, which contradicted the programmers' expectations and prevented them from introducing bugs based on a flawed understanding of the code.

Another inferred invariant was $\mathsf{calls(in\_set\_2)} = \mathsf{calls(stclose)}$. Since `in_set_2` is only called in the predicate controlling `stclose`'s invocation (see Figure 2.4), the equal number of calls indicates that none of the test cases caused `in_set_2` to return `false`. Rather than helping modify the program, this invariant indicates a property of the particular 100 test cases we used. It suggests a need to run `replace` on more of the provided test cases to better expose `replace`'s special-case behavior and produce more accurate invariants.

2.2.4 Invariant uses

In the task of adding the Kleene-+ operator to the Siemens `replace` program, dynamically detected invariants played a number of useful roles.

Explicated data structures. Invariants and queries over the invariant database helped explicate the undocumented structure of compiled regular expressions, which the program represents as strings.

Confirmed and contradicted expectations. In function `makepat`, the programmers expected that $\mathsf{lastj} < \mathsf{j}$ and $\mathsf{lj} < \mathsf{j}$. The first expectation was confirmed, increasing their confidence in their understanding of the program. The second expectation was refuted, permitting them to correct their misunderstanding and preventing them from introducing a bug based on a flawed understanding.

Revealed a bug. In function `stclose`, the programmers expected that $\mathsf{lastj} < *\mathsf{j}$ (this `*j` is unrelated to `j` in `makepat`). The counterexample to this property evidenced a previously undetected array bounds error.

Showed limited use of procedures. Two of the parameters to function `makepat` were the constant zero. Its behavior in that special case — which was all that was required in order to perform the assigned task — was easier to understand than its full generality.

Demonstrated test suite inadequacy. The number of invocations of two functions (and the constant return value of one of them, which the programmers noticed later) indicated that one branch was never taken in the small test suite. This indicated the need to expand the test suite.

Validated program changes. Differences in invariants over `*j` in `stclose` and `plclose` showed that in one respect, `plclose` was performing as intended. The fact that invariants over much of the rest of the program remained identical showed that unintended changes had not been made, nor had changes in modified parts of the program inadvertently affected the computations performed by unmodified parts of the program.

2.2.5 Discussion

Although the use of dynamically detected invariants was convenient and effective, everything learned about the `replace` program could have been detected via a combination of careful reading of the code, additional static analyses (including lexical searches), and selected program instrumentation such as insertion of `printf` statements or execution with a debugger. Adding inferred invariants to these techniques provides several qualitative benefits, however.

First, inferred invariants are a succinct abstraction of a mass of data contained in the data trace. The programmer is provided with information — in terms of manifest program variables and expressions at well-defined program points — that captures properties that hold across all runs. These invariants provide substantial insight that would be difficult for a programmer to extract manually from the trace or from the program using traditional means.

Second, inferred invariants provide a suitable basis for the programmer's own, more complex inferences. The reported invariants are relatively simple and concern observable entities in the program. Programmers might prefer to be told "`*j` refers to the next place to append a character into the compiled pattern," but this level of interpretation is well beyond current capabilities. However, the programmer can examine the program text or perform supporting analyses to better understand the implications of the reported invariants. For example, the presence of several related invariants indicating that `*j` starts with a 0 value and is regularly incremented by 1 during the compilation of the pattern allowed the programmers to quickly determine the higher-level invariant. The basic nature of reported invariants do not render them useless.

Third, the programmers reported that seeing the inferred invariants led them to think more in terms of invariants than they would have otherwise. They believed that this helped them to do a better job and make fewer errors than they would have otherwise, even when they were not directly dealing with the Daikon output.

Fourth, invariants provide a beneficial degree of serendipity. Scanning the invariants reveals facts that programmers would not have otherwise noticed and almost surely would not have thought to check. An example, even in this small case, is the expectation that the program was correct, because of its thousands of tests; dynamic invariant detection helped find a latent error (where the index exceeded the array bounds in some cases). This ability to draw human attention to suspicious but otherwise overlooked aspects of the code is a strength of this approach. A programmer seeking one specific piece of information or aiming to verify a specific invariant and uninterested in any other facts about the code may be able to use dynamic invariant detection to advantage, but will not get as much from it as a programmer open to other, possibly valuable, information.

Finally, two tools provided with Daikon proved useful. Queries against the trace database help programmers delve deeper when unexpected invariants appear or when expected invariants do not appear. For example, expectations regarding the preconditions for `stclose` were contradicted by the inferred invariants, and the clarifying information was provided by supporting data. This both helped discover a bug and simplified an implementation. The other tool, an invariant comparator, reveals how two sets of invariants differ, enabling comparison on programs, versions of a program, test suites, or settings of the invariant detector. It verified some aspects of the correctness of the program change.

No technique can make it possible to evolve systems that were previously intractable to change. But our initial experience with inferred invariants shows promise in simplifying evolution tasks.

2.3 Invariants and program correctness

This section compares invariants detected across a large collection of programs written to the same specification. We found that correct versions of programs give rise to more invariants than incorrect programs.

We examined 424 student programs from a single assignment for the introductory C programming course at the University of Washington (CSE 142, Introductory Programming I). The grades assigned to the programs approximate how well they satisfy their specification; they are not a perfect measure of adherence to the specification because points may be deducted for poor documentation, incorrectly formatted output, etc.

The programs all solve the problem of fair distribution of pizza slices among computer science students. Given the number of students, the amount of money each student possesses, and the number of pizzas desired, the program calculates whether the students can afford the pizzas. If so, then the program calculates how many slices each student may eat, as well as how many slices remain after a fair distribution of pizza.

We manually modified the programs to use the same test suite, to remove user interaction, and to standardize variable names. Invariant detection was performed over 200 executions of each program, resulting in 3 to 28 invariants per program. From the invariants detected in the programs that received perfect grades, we selected 8 relevant invariants, listed in Figure 2.8. The list does not include trivial invariants such as $\mathsf{slices\_per} \geq 0$, indicating that students never receive a negative number of slices, as well as uninteresting invariants such as $\mathsf{slices} \leq \mathsf{pizza\_price} + 75$, which is an artifact of the 200 test cases. These invariants can be valuable in understanding test suites and some aspects of program behavior, but that was not our focus in this experiment.

Figure 2.9 displays the number of relevant invariants that appeared in each program. There is a relationship between program correctness (as measured by the grade) and the number of relevant invariants detected: low-grade programs tend to exhibit fewer relevant invariants, while high-grade programs tend to exhibit the most.

```
people in [1..50]
pizzas in [1..10]
pizza_price in {9, 11}
excess_money in [0..40]
slices = 8 * pizza
slices = 0 (mod 8)
slices_per in {0, 1, 2, 3}
slices_left <= people - 1
```

Figure 2.8: The 8 relevant invariants of the student pizza distribution programs. The first two variables are the program inputs; the test suite used up to 50 people trying to order up to 10 pizzas. Every program satisfied these two invariants. The problem specified that pizzas cost \$9 or \$11. In the test suite, there is up to \$40 left after paying for the pizzas (the maximal possible number of pizzas is not necessarily ordered) and each person receives no more than three slices. The last invariant embodies the requirement that there be fewer leftover pizza slices than people eating.

	Invariants detected					
Grade	2	3	4	5	6	Total
12	4	2	0	0	0	6
14	9	2	5	2	0	18
15	15	23	27	11	3	79
16	33	40	42	19	9	143
17	13	10	23	27	7	80
18	16	5	29	27	21	98
Total	90	82	126	86	40	424

Figure 2.9: Relationship between grade and the number of goal invariants (those listed in Figure 2.8) found in student programs. For instance, all programs with a grade of 12 exhibited either 2 or 3 invariants, while most programs with a grade of 18 exhibited 4 or more invariants. A grade of 18 was a perfect score, and none of the 424 programs exhibited more than 6, or fewer than 2, of the 8 relevant invariants.

The correlation between program correctness and the number of relevant invariants detected is not perfect. The main reason for the discrepancy was that some programs calculate key values in a `printf` statement and never store them in a variable. Indeed, the programs were specified (and graded) in terms of their output rather than for computing and returning or storing values. Programs with a more algorithmic or data-structure bent, or performing less trivial computations, would probably be more likely to return or store their results, exposing them to invariant inference.

2.4 Other experiments

Many of the examples presented in this dissertation are small, so that they can be presented in a small amount of text and fully understood by the reader. (The dissertation returns to these examples multiple times for the same reasons.) A system that works only on small programs is of little interest, however, since almost any technique (even those that educators and researchers blanch to consider) can be successfully applied to a small program. Daikon is not useful only on small programs, but also on ones that challenge other techniques. (The current implementation does have some limitations in scaling to large programs, but we are confident they can be overcome; see Section 9.2 on page 104.)

This section lists ten sets of programs used in testing or validating Daikon. (A number of other programs have been less carefully examined, and users have reported results on yet more programs.) Most of the programs have some variety of specification ranging from full formal specifications, to textual descriptions, to `assert` statements. Such programs predominate because they have a clear set of goal invariants and so no qualitative assessment, which could be debated by skeptics, need be done. In some cases, however, we have performed that kind of assessment, since that matters most to real programmers. Performing more case studies and laboratory experiments is an important area of future work.

The ten sets of programs include student solutions to two programming assignments (for UW CSE 142 and MIT 6.170), programs used in research on testing (from Siemens and Hoffman), programs used in research on program checkers (from Xi and ESC), sample programs from textbooks (by Gries, Weiss, and Flanagan), and an AI planning system (Medic). These programs are described below.

UW CSE 142 These 424 programs, written in C and generally less than 100 lines long, are solutions to a problem in the University of Washington's introductory programming course. See Section 2.3, page 19.

MIT 6.170 These 174 programs, written in Java and ranging from 1500 to 5000 lines long, are solutions to a problem in MIT's sophomore-level Software Engineering Laboratory. See Section 5.5, page 66.

Siemens These seven programs, written in C and ranging from 200 to 700 lines, were originally from Siemens [HFGO94] and were subsequently modified by Rothermel and Harrold [RH98]. These programs are used in research on program testing, so they come with extensive test suites. They have minimal documentation and represent small but realistic programs written without thought for formal invariants. Our experiments focused on four of them: `replace` (string pattern replacement; 21 procedures, 516 non-blank non-comment lines of code (NBNC LOC)), `schedule` (process scheduling; 17 procedures, 304 NBNC LOC), `tcas` (aircraft collision avoidance; 9 procedures, 136 NBNC LOC), and `tot_info` (combine statistics from multiple tables; 7 procedures, 274 NBNC LOC). These programs appear in Section 2.2 and Chapters 4 and 6.

Hoffman These two programs together comprise 2100 lines of Java code (with common code counted just once). They manipulate and test implementations of a header-table and a cursor-window. A header-table consists of a base table, plus a column header subtable of the same width as the base table and a row header subtable of the same height as the base table. A cursor-window is a view of a part of a table (and possibly the appropriate parts of its headers) in a window that may be too small to display the entire table and that is automatically updated when elements are inserted or deleted. The programs are examples for validation of testing via (combinations of) boundary value domains, which are more efficient than testing the full cross-product of domains, and dependent domains, which extend an n-tuple into a set of $(n+1)$-tuples [HS99].

The supplied test drivers check the programs to ensure they are operating correctly. However, the stated object invariant was not always valid. For instance an invariant was stated over `TestTuple` objects and tested by function `CWRDriverCP.oracle`. Because this condition was not checked by the `TestTuple` constructor, it should have been guaranteed by all callers. However, function `ProcessCP.processTuple` builds noncompliant `TestTuples`. Direct manipulations of non-encapsulated objects violated abstractions in some other cases as well.

Our experiments also illustrated deficiencies in the test suite. For instance, there were always 3 column headers and 3 row headers in the header-tables that the test suites constructed.

Xi Hongwei Xi provided these programs, which are written in Xanadu, a Java-like language annotated with explicit invariants in the form of dependent types [Pfe92, XP98, XP99]. Because no Xanadu interpreter or compiler yet exists, we transliterated a short binary search program to Lisp and verified that Daikon detected the dependent types.

ESC This 543-line Java program converts Java source to HTML for viewing via a web browser. It is used as a test case by the Extended Static Checking (ESC) project at Compaq Systems Research Center [Det96, LN98, DLNS98]. ESC checks a form of specification intended to strike a compromise between the ease of use and understanding of types and the power and completeness of full formal specifications. The program, which is a modified version of Hannes Marais's `Java2HTML`, contains these annotations, many of which Daikon was also able to detect.

Gries These programs are examples in a textbook that espouses deriving programs from specifications [Gri81]. The programs are less than 20 lines each, fully specified (for the most part), written in a Dijkstra-like syntax, and lacking test cases. These programs are used in Sections 2.1, 4.5, and 4.6, and incidentally in Section 7.2.

Weiss These Java programs are examples in a data structures textbook [Wei99]. There are 74 files and 7000 lines of code in all, but most of the programs involve only a few files. The programs implement simple, well-understood data structures. See Sections 4.4 and 5.4.

Flanagan These programs are examples in a Java reference and tutorial [Fla99, Fla97]. The suite contains 136 files and 13,000 lines of code. There are 57 runnable programs (containing a `main` routine); 21 of these require user interaction (we used automated inputs for the command-line programs and omitted those with graphical user interfaces) and 8 more enter an infinite loop. Additionally, 9 other files contain Java errors and so cannot be run.

These programs are intended to demonstrate Java programming constructs. They are so simple, and do so little, that few interesting invariants arise. However, they were quite useful in shaking out bugs in the Daikon front end for Java. (The Mauve test suite [Mau] and a number of other programs not mentioned in this list served similar purposes.)

Medic The Medic planner [EMW97] is a 13,000-line Lisp program. It solves a planning problem by converting it into a conjunctive-normal-form formula in propositional logic, finding satisfying variable assignments for the variables in the formula, and using that assignment to construct a plan solving the original problem. The program is annotated with `assert` statements, and Daikon was able to rediscover many of the asserted properties. This particular program was hand-instrumented, as it was examined quite early and we wished to focus attention on the locations with assertions, which would permit assessment of Daikon. Invariants detected at arbitrary locations in the code would have been unreasonably difficult to verify by hand.

The observant (or critical) reader will note that we did not run Daikon on itself. Experiments on programs and test suites that we did not write ourselves are most compelling, because they suggest that the results may generalize to other programs. Thus, we have striven to use others' programs and test inputs whenever possible. An experiment on a program and test suite that we controlled ourselves should not convince anyone — so we did not bother to perform or report it.

Chapter 3

Invariant discovery technique

Dynamic invariant detection occurs in three steps (Figure 1.1, page 2): the program is instrumented to write data trace files, the instrumented program is run over a test suite, and then the invariant detector reads the data traces, generates potential invariants, and checks them, reporting appropriate ones. This chapter explains the third step, the invariant detector proper. (Chapter 7 covers instrumentation, and Section 1.3 and Chapter 6 discuss test suites.)

Section 3.1 gives the basic technique for invariant detection: checking each of a selection of potential invariants over all variables or combinations thereof. This technique is (deliberately) simple, but a naive implementation would fail for a number of reasons; subsequent chapters refine the technique. Section 3.2 lists the invariants Daikon checks and justifies those choices. Section 3.3 lays out a simple extension for inference of object invariants, which hold at multiple program points.

3.1 Inferring invariants

Daikon detects invariants at specific program points such as procedure entries and exits. The instrumented program provides Daikon, for each execution of a program point, the values of variables in scope.

Let x, y, and z be variables and a, b, and c be computed constants. Daikon checks for unary invariants involving a single variable (such as $\mathsf{x} = \mathsf{a}$ or $\mathsf{x} \equiv \mathsf{a} \pmod{\mathsf{b}}$), binary invariants over two variables (such as $\mathsf{x} \leq \mathsf{y}$ or $\mathsf{x} \in \mathsf{y}$), and ternary invariants (such as $\mathsf{x} = \mathsf{ay} + \mathsf{bz} + \mathsf{c}$ or $\mathsf{x} = \mathsf{fn}(\mathsf{y}, \mathsf{z})$ for standard functions fn). Section 3.2 lists the invariants (actually, templates for invariants) predefined in the current implementation of Daikon.

For each tuple of variables up to arity 3, each potential invariant is instantiated and tested. For instance, given variables x, y, and z, each potential unary invariant is checked for x, for y, and for z, each potential binary invariant is checked for $\langle x, y\rangle$, for $\langle x, z\rangle$, and for $\langle y, z\rangle$, and each potential ternary invariant is checked for $\langle x, y, z\rangle$.

A potential invariant is checked by examining each sample in turn; a sample is a tuple of values for the instrumented variables at a program point, stemming from one execution of that program point. As soon as a sample not satisfying the invariant is encountered, the invariant is known not to hold and is not checked for any subsequent samples (though other invariants may continue to be checked).

As a simple example, consider the C code

```
int inc(int *x, int y) {
  *x += y;
  return *x;
}
```

At the procedure exit, value tuples might include (the first line is shown for reference)

$$
\begin{array}{lccccccccl}
\langle & \texttt{orig(x),} & \texttt{orig(*x),} & \texttt{orig(y),} & \texttt{x,} & \texttt{*x,} & \texttt{y,} & \texttt{return} & \rangle \\
\\
\langle & 4026527180, & 2, & 1, & 4026527180, & 3, & 1, & 3 & \rangle \\
\langle & 4026527180, & 3, & 1, & 4026527180, & 4, & 1, & 4 & \rangle \\
\langle & 146204, & 13, & 1, & 146204, & 14, & 1, & 14 & \rangle \\
\langle & 4026527180, & 4, & 1, & 4026527180, & 5, & 1, & 5 & \rangle \\
\langle & 146204, & 14, & 1, & 146204, & 15, & 1, & 15 & \rangle \\
\langle & 4026527180, & 5, & 1, & 4026527180, & 6, & 1, & 6 & \rangle \\
\langle & 4026527180, & 6, & 1, & 4026527180, & 7, & 1, & 7 & \rangle \\
 & & & & \vdots & & & &
\end{array}
$$

This value trace admits invariants including $\mathsf{x} = \mathsf{orig(x)}$, $\mathsf{y} = \mathsf{orig(y)} = 1$, $*\mathsf{x} = \mathsf{orig}(*\mathsf{x}) + 1$, and $\mathsf{return} = *\mathsf{x}$.

All three aspects of this process are inexpensive: instantiation and checking a sample are fast, and there are usually few checks. Instantiation usually requires just an object allocation. As a more complicated example the linear relationship $\mathsf{x} = \mathsf{ay} + \mathsf{bz} + \mathsf{c}$ with unknown coefficients a, b, and c and variables x, y, and z has three degrees of freedom. Consequently, three linearly independent tuples of $\langle x, y, z\rangle$ values are sufficient to determine the coefficients. (For reasons of numerical stability, the implementation does not work in exactly that way [Lov86].) Instantiation can be thought of as generalization from a few values. Once an invariant is instantiated, checking is cheap: usually constant-time and at worst linear in the size of the new sample. (When the variables are scalars, their values have constant size; for array variables, it may be necessary to examine all the elements, for instance to compute sums, minima, maxima, subsequence relationships, and the like.) Checking manipulates actual values and requires no theorem-proving. This process may update computed constants; for example a common modulus (variable b in $\mathsf{x} \equiv \mathsf{a} \pmod{\mathsf{b}}$) is the greatest common divisor of the differences among list elements. Finally, false invariants tend to be falsified quickly and need not be further checked. When a sample that does not satisfy the invariant, and cannot be reconciled with it, is encountered, the offending invariant is discarded. For instance, the third tuple in the trace above invalidates the potential invariant $\mathsf{orig(x)} = 4026527180$. Therefore, the cost of computing invariants tends to be proportional to the number of invariants discovered, which is small — usually just a few per variable (see also Section 6.1).

To reduce source language dependence, simplify the implementation, and improve error checking, Daikon supports only these forms of data: integral number (including characters and booleans), floating-point number (only minimally supported, because they introduce complications of floating-point error but few issues specific to invariant detection), sequence of integer, sequence of floating-point, and string (separated from integral sequence to permit more specific and efficient checks). All trace values must be converted into one of these forms. For example, an array `A` of tree nodes (each with a left and a right child) would be converted into two arrays: `A.left` containing (object IDs for) the left children, and `A.right` for the right children. This design choice avoids the inference-time overhead of interpretation of data structure information. Because declared types are also recorded (in a separate file), mapping all program types to this limited set does not conflate different types. Invariants over the original objects can be recovered from Daikon's output because it computes invariants across the arrays, such as finding relationships over the ith element in each. For example, $\mathsf{a[i].left} < \mathsf{a[i].right}$ is reported as $\mathsf{a.left[i]} < \mathsf{a.right[i]}$, which a postprocessing step could easily convert to the former representation by referring to the original program type declarations.

Invariants can be viewed as forming a partial order based on subsumption (logical implication). Section 4.6 describes how the implementation takes advantage of these relationships to improve both performance and the intelligibility of the output. Perhaps additional advantage could be gained by further formalizing this partial order.

3.2 List of invariants

This section lists the invariants Daikon checks. Actually, these are invariant templates that are instantiated by binding their variables to variables in a data trace file.

The invariants are as follows, where x, y, and z are variables, and a, b, and c are computed constants:

- invariants over any variable:
  - constant value: $\mathsf{x} = \mathsf{a}$ indicates the variable is a constant
  - uninitialized: $\mathsf{x} = \mathsf{uninit}$ indicates the variable is never set
  - small value set: $\mathsf{x} \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ indicates the variable takes on only a small number of different values
- invariants over a single numeric variable:
  - range limits: $\mathsf{x} \geq \mathsf{a}$, $\mathsf{x} \leq \mathsf{b}$, and $\mathsf{a} \leq \mathsf{x} \leq \mathsf{b}$ (printed as `x in [a..b]`) indicate the minimum and/or maximum value
  - nonzero: $\mathsf{x} \neq 0$ if the variable is never set to 0
    See Section 4.5 for details on when such an invariant is reported.
  - modulus: $\mathsf{x} \equiv \mathsf{a} \pmod{\mathsf{b}}$ indicates that $x \bmod b = a$ always
  - nonmodulus: $\mathsf{x} \not\equiv \mathsf{a} \pmod{\mathsf{b}}$, reported only if $x \bmod b$ takes on every value besides a
- invariants over two numeric variables:
  - linear relationship: $\mathsf{y} = \mathsf{ax} + \mathsf{b}$
  - ordering comparison: $\mathsf{x} < \mathsf{y}$, $\mathsf{x} \leq \mathsf{y}$, $\mathsf{x} > \mathsf{y}$, $\mathsf{x} \geq \mathsf{y}$, $\mathsf{x} = \mathsf{y}$, $\mathsf{x} \neq \mathsf{y}$
  - functions: $\mathsf{y} = \mathsf{fn(x)}$ or $\mathsf{x} = \mathsf{fn(y)}$, for fn a built-in unary function (absolute value, negation, bitwise complement)
  - invariants over $x+y$: any invariant from the list of invariants over a single numeric variable, such as $\mathsf{x} + \mathsf{y} \equiv \mathsf{a} \pmod{\mathsf{b}}$

  - invariants over $x - y$: as for $x + y$.
    This subsumes ordering comparisons and can permit inference of properties such as $\mathsf{x} - \mathsf{y} > \mathsf{a}$, which Daikon prints as `x > y+a`.

- invariants over three numeric variables:

  - linear relationship: $\mathsf{z} = \mathsf{ax} + \mathsf{by} + \mathsf{c}$, $\mathsf{y} = \mathsf{ax} + \mathsf{bz} + \mathsf{c}$, or $\mathsf{x} = \mathsf{ay} + \mathsf{bz} + \mathsf{c}$
  - functions: $\mathsf{z} = \mathsf{fn}(\mathsf{x}, \mathsf{y})$, for *fn* a built-in binary function (min, max, multiplication, and, or, greatest common divisor; comparison, exponentiation, floating point rounding, division, modulus, left and right shifts).
    The other permutations of $\langle x, y, z \rangle$ are also tested. Additional functions are trivial to add.

- invariants over a single sequence variable:

  - range: minimum and maximum sequence values, ordered lexicographically
    For instance, this can indicate the range of string or array values.
  - element ordering: whether the elements of each sequence are non-decreasing, non-increasing, or equal
    In the latter case, each sequence contains (multiple instances of) a single value, though that value may differ from sequence to sequence.
  - invariants over all sequence elements (treated as a single large collection): for example, in Figure 2.3 (page 11), all elements of array `B` are at least -100

  The sum invariants of Figure 2.3 do not appear here because `sum(B)` is a derived variable, which is described in Section 4.3, page 32.

- invariants over two sequence variables:

  - linear relationship: $\mathsf{y} = \mathsf{ax} + \mathsf{b}$, elementwise
  - comparison: $\mathsf{x} < \mathsf{y}$, $\mathsf{x} \leq \mathsf{y}$, $\mathsf{x} > \mathsf{y}$, $\mathsf{x} \geq \mathsf{y}$, $\mathsf{x} = \mathsf{y}$, $\mathsf{x} \neq \mathsf{y}$, performed lexicographically
  - sub-sequence relationship: x a sub-sequence of y or vice versa
  - reversal: x is the reverse of y

- invariants over a sequence and a numeric variable:

  - membership: $\mathsf{i} \in \mathsf{s}$

3.2.1 Why these invariants?

We produced the list of potential invariants by proposing a basic set of invariants that seemed natural and generally applicable, based on our programming and specification experience. We later added other invariants we found helpful in analyzing programs; we did this only between experiments rather than biasing experiments by tuning Daikon to specific programs. We also removed from our original list some invariants that turned out to be less useful in practice than we had anticipated. The list does not include all the invariants that programmers might find useful. For instance, it omits linear relationships over four or more variables, nor does Daikon test every data structure for the red-black tree invariant. Omitting such invariants controls cost and complexity: Section 6.1 notes that the number of invariants checked can significantly affect Daikon's runtime. In general, we balanced performance and the likely general utility of the reported invariants. Over time we expect to modify Daikon's list of invariants, based on comments from users and on improvements in the underlying inference technology. (Users can easily add their own general-purpose or domain-specific invariants and derived variables — see Section 7.8, page 90.) Even the current list is useful: it enabled the successful detection of invariants in a number of experiments.

3.3 Object invariants

Object invariants — also known as class invariants or representation invariants — are data structure well-formedness conditions that indicate the required relationship among components of the data structure. They are typically enforced by the implementation of the data structure abstraction, but in languages lacking strong encapsulation, the environment (the remainder of the program) may be trusted to preserve the property. Often, a data structure's operations are guaranteed to work only if its object invariant holds, and the data structure abstraction's contract includes maintaining the object invariant, so that clients or future data structure operations can depend on it.

The object invariant for a string class that maintains both a null-delimited content and an explicit length might be string.content[string.length] = '\0', indicating that the character at the specified length is the terminator. (Another invariant over that data structure would state that no earlier character is the terminator.) For a node in a sorted tree, the object invariant might state node.left.value $\leq$ node.right.value. Figure 5.6 (page 62) shows another object invariant discovered by Daikon that indicates that a linked list class contains a dummy header node at the beginning of the representation in order to make insertion and deletion at the head of the list easier.

Object invariants hold at multiple program points, not just one. Daikon detects them by aggregating the abstraction's entry and exit points. The instrumenter aggregates variables in scope at the entry and exit of every public method (in other words, the fields of the object and/or class, for Daikon omits global variables from consideration for this purpose) into a new, synthetic program point. Invariants detected at that program point are object invariants.

At each public routine entry and exit, Daikon may re-detect the object invariants. (Section 5.5 shows an example where statistical justification tests prevented this.) However, it is more useful to the programmer to report them in one place (to emphasize that they hold everywhere and to reduce clutter), and it may be more efficient to test them just once. (Daikon does not yet implement that optimization.)

In addition to object invariants, Daikon detects (static) class invariants. These involve only static class variables that have the same value for all objects (but may change at runtime). These could differ from (a subset of) the object invariants if stricter conditions held for public static methods than public methods. In practice, we have not seen such differences.

Chapter 4

Improving the relevance of reported invariants

A straightforward implementation of the invariant detection technique of Chapter 3 would not produce the results described in Chapter 2. It would omit some of those results and include other, undesirable properties. Consequently, its output would be less valuable to programmers than we would prefer.

This chapter presents, and gives experimental evidence of the efficacy of, five approaches for increasing the relevance (the usefulness to a programmer in performing a task) of invariants reported by a dynamic invariant detector. Two of them — adding implicit values and exploiting unused polymorphism — add desired invariants to the output. The other three — statistical confidence checks, suppressing redundant invariants, and limiting which variables are compared to one another — eliminate undesired invariants from the output. These techniques also improve performance by reducing the work done by the invariant detector.

Section 4.1 briefly introduces the techniques. Section 4.2 explains the relevance metric and lays out the experimental method. The five techniques for improving relevance are further explained and experimentally justified in Sections 4.3–4.7.

4.1 Overview

This section briefly describes the five techniques for improving invariant relevance. They are treated in detail later in this chapter.

Implicit values (Section 4.3) Important properties may hold over quantities not explicitly stored in program variables, such as the size of a data structure or the largest value it contains. Daikon introduces so-called derived variables to represent these values. This permits ordinary invariant detection to report relationships involving these variables. Section 4.3 describes the technique and lists the derived variables. Section 4.3.1 describes how to avoid introducing too many derived variables.

Polymorphism elimination (Section 4.4) Variables declared polymorphically (as with Java's `Object` type or any other base class) often contain only a single type at runtime in practice. Daikon uses declared types to avoid the costs of managing polymorphism at invariant detection time, but this prevents it from examining fields specific to the runtime type. A two-pass technique solves this problem. The first pass detects invariants over the runtime class of objects; this refined type information is fed into the second invariant detection pass. Section 4.4 describes this technique, showing that it enables reporting of relevant but otherwise undetected invariants specific to the variable values' run-time types.

Statistical justification (Section 4.5) Only invariants that are statistically justified — relationships that do not appear to have occurred by chance alone — should be reported. These statistical confidence tests depend on the set of values obtained at a particular program point. Section 4.5 describes these tests.

When a variable's value is repeatedly examined without intervening variable assignments — such as a variable that is examined at a loop head but remains unchanged within a loop — then the number of samples is artificially inflated and properties of the variable may be inappropriately reported. Section 4.5.1 reports on the relative effectiveness of several rules for ignoring some instances of repeated values.

Redundant invariants (Section 4.6) Logically implied invariants are not worth reporting. For instance, if two invariants `x` $\neq$ `0` and `x in [7..13]` are determined to be true, there is no sense in reporting both because the latter implies the former. Furthermore, implied invariants are not even worth computing. As an example, once Daikon determines that `x` = `y`, then no inference need be done for `y`, as invariants over `x` imply similar ones over `y`. Pruning both the computation and the reporting of implied invariants reduces the size of the output without removing any information from it. Section 4.6 details our approach to implied invariants and reports its efficacy in practice.

Incomparable variables (Section 4.7) Not all variables can be sensibly compared. For instance, numerically comparing a boolean to a non-null pointer results in the accurate but useless invariant that the boolean is always less than the pointer value. Restricting which variables are compared to one another increases performance by reducing the number of potential invariants to be checked and reduces the number of irrelevant invariants reported. Section 4.7 compares four approaches to limiting the number of comparisons. These approaches use declared types and value flow information computed by the Lackwit tool [OJ97].

4.2 Relevance

We call an invariant *relevant* if it assists a programmer in a programming activity. The relevance of a set of invariants is the percentage of the set's members that are potentially relevant. Relevance is inherently contingent on factors such as the particular task and the programmer's capabilities, working style, and knowledge of the code base. Because no automatic system can know this context, Daikon omits some invariants that the user might find helpful and reports some invariants that the user does not find helpful. This reduces the benefit to the user.

The usefulness of a set of invariants depends on the absolute and relative number of relevant invariants, their organization and presentation, the amount of time it takes to compute them (on-demand or incremental invariant computation complicates matters further), and other factors.

To improve invariant relevance, the programmer — who *is* privy to much of the context — could control the invariant inference process. Alternately, heuristics could be added to

Daikon to improve the relevance of the reported invariants in most cases. This chapter pursues the second approach, which lessens the burden on the programmer.

We manually classified reported invariants as potentially relevant or not relevant based on our own judgment, informed by careful examination of the program and the test cases. (This process invariably improved our understanding of the programs and test suites, as we came across invariants that surprised us or looked invalid. This is supporting evidence that invariants assist program understanding.) We judged an invariant to be potentially relevant if it expressed a property that was necessarily true of the program or expressed a salient property of its input, and if we believed that knowledge of that property would help a programmer to understand the code and perform a task. We made every effort to be fair and objective in our assessments.

For our evaluation, the absolute relevance, usefulness, or value of a set of invariants, is less important than the improvement in invariant relevance or in efficiency. We evaluated each technique while using the best version of the other techniques, in order to provide a fair baseline against which to evaluate the improvement. In a few cases this was not possible. For example, C lacks polymorphism, so we could only assess polymorphism elimination for Java programs; however, our implementation of Lackwit-style comparability checking for Java is still underway.

The subjective definition of relevance complicates assessment of techniques for improving the relevance of reported invariants. We report a combination of quantitative and qualitative measurements for each technique.

4.3 Implicit values

Computing invariants over manifest program variables can be inadequate for a programmer's needs. For instance, invariants relating the size of a collection to other (explicit or implicit) quantities should be reported. As another example, if array `a` and integer `lasti` are both in scope, then `a[lasti]` may be of interest, even though it is not a variable and may not even appear in the program text. Daikon adds certain "derived variables" (actually expressions) to the variables it is supplied in the data trace file.

These derived variables can be computed from the trace file, so they are introduced by the invariant detector proper. Some other synthetic variables are output directly by the front end. These include data structure components (e.g., `point.x` and `point.y`), cyclicity of data structures, and any other information the front end wishes to add. Daikon has no way of distinguishing them from actual program variables, except possibly from knowledge about how the front end chooses their names. (The current implementation takes advantage of such information in limited circumstances.)

Daikon treats derived variables just like other variables, permitting it to infer invariants that are not hard-coded into its list. For instance, if `size(A)` is derived from sequence `A`, then the system can report the invariant $\mathsf{i} < \mathsf{size}(\mathsf{A})$ without hard-coding a less-than comparison check for the case of a scalar and the length of a sequence. Thus, the implementation can report compound relations that we did not necessarily anticipate.

The derived variables are the following:

- derived from any sequence `s`:

  - length (number of elements): `size(s)`
  - extremal elements: `s[0]`, `s[1]`, `s[size(s)-1]`, `s[size(s)-2]`
    The latter two are reported as `s[-1]`, `s[-2]` for brevity, where the negative indices suggest indexing from the end rather than the beginning of the sequence. Including the second and penultimate elements (in addition to the first and last) accommodates header nodes and other distinguished uses of extremal elements.

- derived from any numeric sequence `s`:

  - sum: `sum(s)`
  - minimum element: `min(s)`
  - maximum element: `max(s)`

- derived from any sequence `s` and any numeric variable `i`:

  - element at the index: `s[i]`, `s[i-1]` (as in the `a[lasti]` example above)
    Both the element at the specified index and the element immediately preceding it are introduced as derived variables because programmers sometimes use a maximum (the last valid index) and sometimes a limit (the first invalid index).
  - sub-sequences: `s[0..i]`, `s[0..i-1]`
    The notation `s[a..b]` indicates the portion of `s` spanning indices `a` to `b`, inclusive. As in the above case, two sub-sequences are introduced because numbers may indicate a maximum valid index or a length.

- derived from function invocations: number of calls so far
  Daikon computes this from a running count over the trace file.

Many possible derived variables are not of general interest. For example, we do not want to run a battery of tests on x^y for every x and y, even if such a derived variable might be useful in a few situations. Just as with the list of invariants (Section 3.2.1, page 27), we do not and could not have a complete list of derived variables. Rather, the current set aims to be basic, general, and useful. Section 7.8 (page 90) shows how users can add new derived variables to Daikon.

Adding these derived variables to Daikon was a success, for it began producing helpful and important new invariants. Many derived variables may be added, however, and this has two potential drawbacks. First, it could slow down the system with many invariants to check over the new variables. This issue is treated in Section 4.3.1, below. Second, introducing many potential invariants inevitably increases the number of irrelevant invariants reported, even if only a small percentage of potential invariants are irrelevantly reported. The techniques of Sections 4.5, 4.6, and 4.7 quash this potential difficulty.

4.3.1 Limiting variable derivation

Deriving variables from other derived variables could eventually create an arbitrary number of new variables. In order to avoid overburdening the system (and introducing baroque, unhelpful variables), Daikon halts derivation after a fixed number of iterations, limiting the depth of any potential derivation and the number of derived variables. This depth defaults to 2 in the current implementation.

More importantly, derived variables are introduced only when previously-computed invariants indicate they will be sensible. This requires interleaving invariant detection and variable derivation rather than performing all variable derivation before any invariant inference.

In particular, derived variables are not introduced until invariants have been computed over previously-existing variables, and derived variables are introduced in stages rather than all at once. For instance, for sequence `A`, the derived variable `size(A)` is introduced and invariants are computed over it before any other variables are derived from `A`. If $\mathsf{j} \geq \mathsf{size(A)}$, then there is no sense in creating the derived variable `A[j]`. When a derived variable is only sometimes sensible, as when `j` is only sometimes a valid index to `A`, no further derivations are performed over `A[j]`. Likewise, `A[0..size(A)-1]` is identical to `A`, so it need not be derived.

Derived variables are guaranteed to have certain relationships with other variables; for instance, `A[0]` is a member of `A`, and `i` is the length of `A[0..i-1]`. Daikon does not compute or report such tautologies. Likewise, whenever two or more variables are determined to be equal, one of them is chosen as canonical and the others are removed from the pool of variables to be derived from or analyzed, reducing both computation time and output size.

4.4 Polymorphism elimination

Polymorphism permits functions and containers to manipulate objects of multiple runtime types. Polymorphism also enables code sharing and reuse and provides flexibility for future change, among other benefits. Variables that are declared polymorphically — as with Java's `Object` type or any other base type — often contain objects of only a single runtime type. As an example, consider a polymorphic list that a particular program uses to hold only integers (Figure 4.1). It is desirable to detect properties over the runtime values that would not be sensible for arbitrary objects of the declared type, such as that the list is sorted, which is not meaningful for an arbitrary list of `Objects`. This example also demonstrates the need to know the runtime type of a variable in order to extract its fields. Field `obj.value` is not meaningful for an arbitrary object `obj`, but needs to be examined when `obj` is actually of type `MyInteger`.

Daikon statically determines a program's data trace format during instrumentation, based on declared variable types. (See Section 7 for details and a justification of this choice, which trades off simplicity and robustness against flexibility.) Consequently, Daikon cannot directly find invariants over polymorphic variables, whose exact type and data fields are unknown until runtime.

Daikon's technique for detecting runtime-type-specific invariants over polymorphic variables involves two passes of invariant detection. For each variable in scope at a program point, the front end ordinarily causes the variable's object ID, its known fields (based on its declared type), and its run-time class to be written to a data trace file when the program point is executed. Even if the declared type has no fields, as for Java's `Object`, the variable's run-time class is presented to the invariant detector like any other value that was explicitly present in the source code. If Daikon detects invariants over the run-time class (such as the object being of a specific class whenever it is non-`null`), the user can annotate the source code with a comment indicating a more accurate refined type. For instance, the `ListNode` declaration of Figure 4.1 would be rewritten as

```
class ListNode { /*refined_type: MyInteger*/ Object element;
                 ListNode next; ... }
```

The user currently inserts the annotations. Automating this step might be worthwhile in the future.

A second pass of instrumentation and invariant detection takes advantage of the refined type annotations in the source code. The front end treats annotated variables as having the specified types. In particular, fields specific to the annotated type can be accessed and provided to the invariant detector. This is sound, if the program is run over the same inputs and is deterministic. In any event, Daikon catches errors raised by instrumentation code, so the program's behavior is guaranteed not to be modified. The second run of invariant detection is presented with a larger number of variables, permitting reporting of previously undetectable invariants over those variables.

While this technique primarily adds new invariants to Daikon's output (as mentioned on page 30), it can also remove irrelevant or obvious invariants from the output, in conjunction with the comparability enhancements of Section 4.7. In particular, if two variables are declared to be of the same (polymorphic) type but are subsequently discovered to have different run-time classes, then there is no need to compare them, such as reporting that they are never equal to one another.

We assessed polymorphism elimination on the first five Java programs from a data structures text [Wei99]. The data structures include polymorphic linked lists, stacks and queues (implemented using both linked lists and arrays), and trees. The test cases provided with the programs manipulate sorted collections of `MyInteger` objects. (The use of `MyInteger` is not gratuitous: it implements the `Comparable` interface, whereas Java 1.1's `Integer` does not.) See Figure 4.1 for an example; other classes were similar. On the first pass, Daikon did not detect the sortedness of the collection, because it was provided only the hashcodes and classes of the elements; it did detect invariants over the run-time class. The second pass reported additional relations such as the object invariant shown in Figure 4.1.

The name of the variable in the object invariant requires some explanation. (Chapter 5 on pointer-based collections gives more details.) For recursive fields such as `next`, the notation `header.closure(next)` is the collection of `ListNode` objects reachable from `header` by following `next` fields. A field reference applied to a collection indicates the collection made up by taking that field reference for each element of the original collection. (As an example, `myarray.myfield` indicates the array formed by taking the `myfield` field of every element of `myarray`.) Thus, `header.closure(next).element.value` is the collection of `value` fields of elements reachable from `header`. The result is a sortedness invariant over a realistic and useful collection.

Routines that manipulate `LinkedListItr` iterators, whose `current` fields point to list elements, report the stronger (strictly increasing) sortedness invariant

Declarations:

```
class LinkedList { ListNode header; ... }
class ListNode { Object element;
                 ListNode next; ... }
class MyInteger { int value; ... }
```

Sample code:

```
LinkedList myList = new LinkedList();
for (int i=1; i<=10; i++)
  myList.add(new MyInteger(i));
```

`LinkedList` object invariant reported by Daikon:

header.closure(next).element.value is sorted by $\leq$

Figure 4.1: Simplified code [Wei99] and a detected invariant for a polymorphic linked list class. The object invariant states that for every `ListNode` object, the `value` fields of the elements reachable from the `LinkedList`'s `header` are sorted.

current.closure(next).element.value is sorted by $<$

Figure 4.1 reports a $\leq$ rather than a $<$ relationship because the header element is not used. Other invariants in the output indicate that it is always zero (see Figure 5.6, page 62) and that the list proper can contain zero.

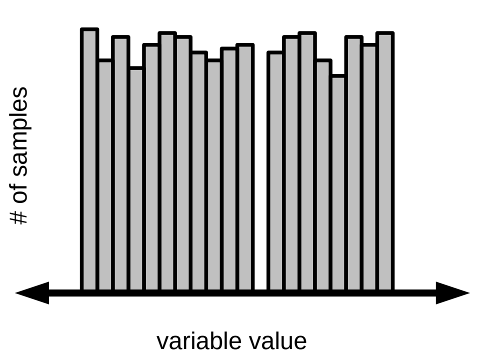
In other data structures, Daikon found similar invariants (such as sortedness of a tree or membership in a collection); see Chapter 5 for details.

4.5 Invariant confidence

Daikon reports only invariants that pass a statistical confidence test; properties that could easily have occurred by chance are not reported, as they are likely to be accidents of the data. As an example, suppose that over an entire test suite, a program point was executed just three times, on which variable `x` had values 7, -42, and 22. An invariant detector could report $x \neq 0$, $x \leq 22$, or $x \geq -42$. The data satisfy but do not justify these invariants.

As another example, suppose that $0 < y < 10$ and $0 < z < 10$. Given three $\langle y, z\rangle$ pairs, the invariant $y \neq z$ should not be reported, even if it is true for those three pairs: they do not support such a generalization. If there are 10,000 pairs with `y` never equal to `z`, then the relationship is likely to be more than a coincidence.

Reporting spurious invariants of this sort would make the output less useful and could discourage programmers. One simple solution to the problem is to use a better test suite. A larger, more complete test suite is likely to include counterexamples to coincidental properties that hold in smaller test sets. However, generating ideal test suites is difficult, and good invariant detection output is desirable even for deficient test suites. Another approach would perform invariant detection multiple times over parts of the data, accepting only invariants



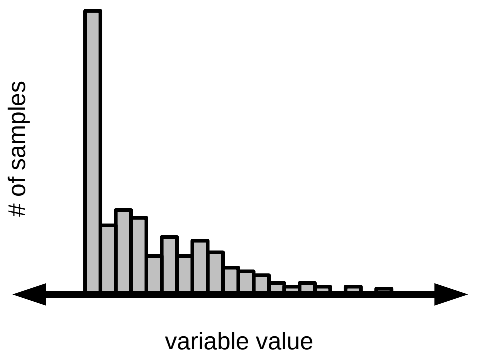


Figure 4.2: Example observed histograms of (nearly) uniform and (nearly) exponential distributions. Each graph shows the number of samples for a particular variable value — that is, the number of times the variable had that value when the program point was executed. The histograms show all samples over a test suite: no variable values larger or smaller than those shown ever occurred.

that always appear. This works in some cases, but is poorly motivated and computationally expensive; Daikon takes another tack.

For each detected invariant, Daikon computes the probability that such a property would appear by chance in a random input. If that probability is smaller than a user-specified confidence parameter, then the property appears to be non-coincidental and is reported. In other words, Daikon assumes a distribution and performs a statistical test in an attempt to discredit the null hypothesis, which states that the observed values were generated by chance from the distribution. If the null hypothesis is rejected at a certain level of confidence, then the observed values are non-coincidental and their unusual property is worth reporting.

The confidence level does not indicate whether the invariant is (or is likely to be) true over all possible inputs, because the test suite may not fully characterize the program's actual execution environment [Goe85]; rather, it is used to decide whether a particular invariant is worth reporting to the user. Additionally, since the actual distribution of variable values is not known, the exact value of this confidence is less important than its order of magnitude and comparisons among confidences. Daikon often tests against the uniform distribution. Although this is rarely the true distribution, it is as good a default as any, it works in practice, and exactitude is not necessary. Finally, reporting an invariant does not guarantee that the invariant is relevant (useful to a programmer for a specific task).

The confidence limit should be quite strict; a value such as .01 — only invariants that are no more than 1% likely to have occurred by chance are reported — is too lenient. If the system checks millions of potential invariants, then reporting thousands of spurious invariants is likely to be unacceptable. Use of smaller confidence limits has worked well in practice, eliminating undesired test suite artifacts while retaining desired invariants.

Each invariant in Daikon implements its own confidence test. Here are two examples:

non-zero. Suppose the reported values for variable x fall in a range of size r that includes 0, but that $x \neq 0$ for all test cases. If the values are uniformly distributed, then the probability that a single instance of x is not 0 is $1 - \frac{1}{r}$. Given s samples, the probability that x is never 0 is $(1 - \frac{1}{r})^s$; if this probability is less than a user-defined confidence

```
15.1.1:::ENTER                  100 samples
  N = size(B)                     (24 values)
  N >= 0                          (24 values)

15.1.1:::EXIT                   100 samples
  B = orig(B)                     (96 values)
  N = I = orig(N) = size(B)       (24 values)
  S = sum(B)                      (95 values)
  N >= 0                          (24 values)

15.1.1:::LOOP                   986 samples
  N = size(B)                     (24 values)
  S = sum(B[0..I-1])              (858 values)
  N >= 0                          (24 values)
  I >= 0                          (36 values)
  I <= N                          (363 values)
```

Figure 4.3: Invariants inferred for the Gries array sum program (Figure 2.1, page 10), which sums array `B` into variable `S`. This is the complete Daikon output, for exponentially-distributed arrays. Array lengths and element values were chosen from exponential distributions, but with the same expected array lengths and element values as the uniform distributions used in Figure 2.3 (page 11).
Compared to Figure 2.3, no irrelevant bounds are reported, and the output is almost identically the desired formal specification.

level, then the invariant $\mathsf{x} \neq 0$ is reported. For example, see Figure 4.2. If the gap in the left-hand histogram is at zero, then $\mathsf{x} \neq 0$ would likely be justified. Several other tests, including $\mathsf{x} \neq \mathsf{y}$, linear relationships, and (non)modulus, are analogous.

range limits. Ranges for numeric variables (such as $\mathsf{x} > 0$ or $\mathsf{c} \in [32..126]$) can be statistically justified in two different ways. A limit is reported if the several values near the range's extrema all appear about as often as would be expected, so the distribution appears to be uniform and to stop at the observed minimum or maximum. For instance, the left-hand histogram of Figure 4.2 seems to stop cleanly at its maximum, which is thus worth reporting. The right-hand histogram peters out to the right, so it would be rash to report the maximum value seen in this test suite as a hard bound.

The limit is also reported if the extremum appears much more often than would be expected, as if greater or lesser values have been clipped to that value. In Figure 4.2, this applies to the lower bound of the right-hand histogram.

The 100 random arrays used in the experiment of Figure 2.3 (page 11) happened to support both upper-bound and lower-bound inferences for the elements of array `B`. This was usually the case for 100 randomly-generated arrays from that distribution, but for one such test suite, only the lower bound

```
B                              (100 values)
  All elements >= -100         (200 values)
```

was reported. For larger test suites, both bounds were always inferred.

Figure 4.3 shows the result of running Daikon on a set of 100 arrays generated from an exponential rather than uniform distribution. The resulting Daikon output does not include the characteristics of the test suite that appeared in Figure 2.3.

The confidence heuristics are not guaranteed to work perfectly. The right-hand column (values and samples) in the Daikon output (see, for example, Figure 4.4) indicates how much data was seen, to permit users to make their own judgments.

The "values" column in the Daikon output gives intuition about support for an invariant; users can downgrade their confidence if they feel the heuristic is operating over too few values. Users can also request to see statistical confidences for each invariant, but these are not intuitively interpretable and users have not found them particularly helpful. Users can also display all invariants regardless of confidence; there are so many of them that this, too, is unhelpful.

4.5.1 Repeated values

The statistical tests fail to suppress some unjustified invariants, because multiple visits to a program point without assignment to a variable can cause the repeated values for the variable to be overweighted in the statistical tests. For example, additional samples for a loop-invariant variable could cause Daikon to report invariants inside the loop that are true but are not considered statistically justified outside the loop — even though the variable values are the same in both cases. As a concrete example, compare Figure 4.4, which shows the invariants for the Gries program using this rule, to Figure 2.3, which uses the "assignment" rule described below. Figure 4.4 contains three extra invariants at the loop head. Procedure invocations and other sorts of control flow cause similar anomalies.

This section compares five strategies for determining whether a particular sample of values should increase confidence in an invariant.

Always. Every sample contributes to confidence. This strategy is trivial to implement but performs unacceptably, as noted above.

Changed value. A sample contributes to invariant confidence when its value is different from the last time it was examined at the program point. This approach does not detect when a variable is recomputed and given the same value.

Assignment. A sample contributes to invariant confidence when the corresponding variable was assigned since the last time the program point was visited. This approach captures the intuitive notion that variable assignments are the semantically important events, and we expected it to perform best. This approach requires significant cooperation from the instrumenter. Chapter 7 describes how the instrumented program computes, and communicates to the invariant detector, whether a particular sample should contribute to invariant confidence.

```
15.1.1:::ENTER                    100 samples
  N = size(B)                       (7 values)
  N in [7..13]                      (7 values)
  B                                 (100 values)
    All elements in [-100..100]     (200 values)

15.1.1:::EXIT                     100 samples
  N = I = orig(N) = size(B)         (7 values)
  B = orig(B)                       (100 values)
  S = sum(B)                        (96 values)
  N in [7..13]                      (7 values)
  B                                 (100 values)
    All elements in [-100..100]     (200 values)

15.1.1:::LOOP                     1107 samples
  N = size(B)                       (7 values)
  S = sum(B[0..I-1])                (452 values)
  N in [7..13]                      (7 values)
  I in [0..13]                      (14 values)
  I <= N                            (77 values)
  B                                 (100 values)     ⋆
    All elements in [-100..100]     (200 values)     ⋆
  sum(B) in [-556..539]             (96 values)      ⋆
  B[0] nonzero in [-99..96]         (79 values)      ⋆
  B[-1] in [-88..99]                (80 values)      ⋆
  B[0..I-1]                         (985 values)     ⋆
    All elements in [-100..100]     (200 values)     ⋆
  N != B[-1]                        (99 values)      ⋆
  B[0] != B[-1]                     (100 values)     ⋆
```

Figure 4.4: Invariants inferred for the Gries array sum program (Figure 2.1, page 10), which sums array `B` into variable `S`. This is the complete Daikon output, for the same uniformly-distributed test inputs as used in Figure 2.3, except that repeated values are not suppressed: every sample contributes to invariant confidence.

`B[-1]` is shorthand for `B[size(B)-1]`, the last element of array `B`.

Compared to Figure 2.3, which used the "assignment" rule for determining when a sample contributes to confidence, several invariants are extraneous. The extraneous invariants (marked with ⋆) appear at the loop head but not elsewhere, even though array `B` does not change during the program's execution.

Random. A sample contributes to invariant confidence when its value changes and with probability $\frac{1}{2}$ otherwise.

Random proportionate to assignments. A sample contributes to invariant confidence when the value changes, and otherwise with a probability chosen so that the total number of contributing samples is the same as in the "assignment" strategy described above. This requires the same instrumentation as the assignment strategy. We do not consider this a reasonable approach. It has the same number of contributing

```
15.1.1:::ENTER                  100 samples
  N = size(B)                     (24 values)
  N >= 0                          (24 values)

15.1.1:::EXIT                   100 samples
  B = orig(B)                     (96 values)
  N = I = orig(N) = size(B)       (24 values)
  S = sum(B)                      (95 values)
  N >= 0                          (24 values)

15.1.1:::LOOP                   986 samples
  N = size(B)                     (24 values)
  S = sum(B[0..I-1])              (858 values)
  N in [0..35]                    (24 values)
  I >= 0                          (36 values)
  I <= N                          (363 values)
  B                               (96 values)     ⋆
    All elements in [-6005..7680] (784 values)    ⋆
  sum(B) in [-15006..21144]       (95 values)     ⋆
  B[0..I-1]                       (887 values)    ⋆
    All elements in [-6005..7680] (784 values)    ⋆
```

Figure 4.5: Invariants inferred for the Gries array sum program (Figure 2.1, page 10), which sums array B into variable S. This is the complete Daikon output, for the same exponentially-distributed test inputs as used in Figure 4.3, except that repeated values are not suppressed: every sample contributes to invariant confidence.

Compared to Figure 4.3, which used the "assignment" rule for determining when a sample contributes to confidence, the last three invariants are extraneous. The extraneous invariants (marked with ⋆) appear at the loop head but not elsewhere, even though array B does not change during the program's execution.

samples as the "assignment" rule, so comparing them reveals whether the assignment rule performs well because of the number of contributing samples or the specific set of them.

New value. A sample contributes to invariant confidence when its value has never been seen before at the program point. This effectively creates a flat distribution by considering just one sample per value. An implementation must maintain a collection of all previously-seen values. Daikon does not implement this heuristic.

We chose the "assignment" rule as the baseline for comparison. Although it requires greater programming effort and implementation overhead, it most closely captures our intuitive notion of when a sample is significant. If the dynamic execution path between two executions of a program point does not affect a variable's value, then the value of the variable is unrelated to behavior to be captured at the program point and should not increase invariant confidence. Consequently, Daikon should not treat each occurrence of the value at the instrumentation point as a separate, fresh instance of the value that contributes equal

	Always	Changed	Random	Random $\propto$
Added	33	23	36	26
relevant	0	4	0	0
irrelevant	33	19	36	26
Removed	10	9	14	14
relevant	6	1	6	6
irrelevant	4	8	8	8

Figure 4.6: Invariant differences due to handling of repeated values over the Siemens programs. Each column indicates the difference between invariants reported using the "assignment" rule and some other rule for whether a sample increases confidence. For instance, the "always" rule reported 33 irrelevant invariants not in the "assignment" output and omitted 6 relevant and 4 irrelevant invariants that appeared in the "assignment" output. The differences always represent less than 1% of the 5059 baseline invariants reported at more than 200 program points (5059 is the sum of the last three columns of the "reported" line of Figure 4.7).

weight to an invariant's confidence level.

As a performance optimization, Daikon can use modification information not only to produce more accurate confidence measures, but also to skip samples during invariant checking. An unmodified sample can be ignored since its values are the same as on the previous visit to the program point and hence the invariant being tested must (still) hold.

A particular sample can contribute to confidence for some invariants but not others. For binary and ternary invariants, a tuple of variables (for instance, $\langle x, y \rangle$ for a comparison $\mathsf{x} \leq \mathsf{y}$) is considered modified if any of its variables is modified.

An invariant is considered justified and reported only if three separate tests are satisfied. First, there must be a sufficient number of samples of the variables to be tested, regardless of how many contribute to confidence. Second, there must be a sufficient number of samples (exceeding a specific absolute bound) that contribute to confidence. (A variable such as a global constant that is set just once on program initialization is treated specially for the above tests. If there are many runs in the test suite, this isn't an issue.) Third, the statistical confidence in the invariant must exceed the user-specified bound. The first two tests are inexpensive and may prevent the invariant from being tested at all.

4.5.2 Results

We compared the rules listed above to assess their relative benefits. We omitted the Gries programs because they did not come with test suites, and our small tests might be better or worse than those constructed in practice by a tester. For each of the Siemens programs, we repeated invariant detection using each of the five rules listed above to determine which samples should contribute to invariant confidence. We then classified, by hand, each of the differences in the output as either relevant or irrelevant, according to the criteria of Section 4.2 (page 31).

Figure 4.6 presents the results of this analysis. Each rule for whether a sample adds confidence was compared to the baseline "assignment" rule. Among the techniques, only the “value” rule causes reporting of relevant invariants that are not justified according to the “assignment” rule. All the other rules miss some invariants reported by the “assignment” rule and add more irrelevant invariants than they prune. Because of its simplicity and lack of need for special runtime support, the “value” rule may be competitive with the “assignment” rule in practice, even if the latter usually produces a slightly more relevant set of invariants.

This experiment used 1000 test cases to create the data traces. When using only 300 test cases, there are 553 differing invariants, or over 10%, between the “assignment” rule and the other rules. Additionally, there are larger differences between the performance of the various rules. Larger test suites do not proportionately reduce the number of differences beyond those for 1000 test cases. In our experience, there are three reasons for this behavior (Section 6.2). (1) Beyond a certain size, expanding test suites has little impact on the accuracy of invariant detection or the specific invariants detected. (2) For test suites smaller than that cutoff, increasing test suite size greatly improves the accuracy of invariant detection, by providing counterexamples to undesirable invariants and providing increased confidence in desirable ones. (3) For test suites larger than the cutoff, which specific tests are chosen from a large pool of potential test cases has little effect on the detected invariants.

4.6 Redundant invariants

Invariants that are logically implied by other invariants need not be computed or reported. Eliminating implied invariants greatly reduces the time and space costs of invariant inference; in practice, without this improvement Daikon often fails to compute invariants even given a long runtime and a large physical and virtual memory. This technique also reduces the user’s burden of picking through reported invariants to find the ones of interest; implied invariants clutter the output without adding any new information.

Implication tests suppress redundancies at three stages in invariant detection. First, a derived variable is not introduced if it will be equal to another variable (the first element of array `a` is the same as the first element of array `a[0..i]`) or if it will be nonsensical (`a[i]` when `i` is known to be negative). Up to half of derived variables fall into one of these categories. Such savings are significant because invariant detection runtime is potentially cubic in the number of variables at a program point (see Section 6.1). Second, invariants whose truth or falsehood is known *a priori* are not checked. Suppressing false invariants has a relatively small effect, because false invariants tend to be falsified quickly and are not considered thereafter. Suppressing true invariants has a bigger payoff, since such invariants would be checked for all values computed by the target program over its test suite. In fact, most true invariants can be identified beforehand, and these can number an order of magnitude greater than the reported invariants. Third, some redundant invariants slip through these other tests, but are suppressed before being output. Invariants are instantiated in groups for the sake of efficiency, and invariants can only be suppressed by previously instantiated and checked invariants. Pruning these redundant invariants at output time removes about a quarter more of the potential output. This third stage is a user interface improvement only, while the first two improve both the output and the runtime; the savings are cumulative over the three stages.

	Gries	`replace`	`tcas`	`tot_info`	Stage
Variables	558	969	1438	625	
non-canonical	56	125	372	53	
missing	58	352	338	274	
canonical	444	492	728	298	
Derived vars	234	637	1078	420	
Suppressed	126	507	1198	40	(1)
Invariants	275162	540746	5497210	1010411	
falsified	272454	537284	5473523	1008386	
unjustified	1983	1749	10694	1091	
redundant	207	446	9985	130	(2)
reported	518	1247	3008	804	
Suppressed	2788	20186	1686543	101660	(3)
falsified	448	2648	8925	603	
redundant	2340	17538	1677618	101057	

Figure 4.7: Suppression of redundant computation and output via implication tests. "Gries" is formally specified textbook programs [Gri81]; the others are C programs [HFGO94, RH98]. The "Stage" column indicates whether the figures show improvement due to (1) avoiding derived variables, (2) avoiding checking invariants, or (3) avoiding reporting trivial invariants.

Daikon checks for redundancies at each appropriate stage of its computation. The checks do not use a general-purpose theorem-prover; for efficiency, each specific way a potential invariant can be implied by one or more other invariants is checked individually. The implicants (hypotheses) must be statistically justified in order to imply the implicand (conclusion). Invariants are indexed so that looking them up is very fast (see Section 7.7). The set of checks must be extended when new invariants or derived variables are added to the system; the consequence of a missing check is that some implied invariants appear in the output.

Figure 4.7 tabulates the effect of using implication to avoid work (primarily the top portion) and reduce the amount of output (primarily the bottom portion).

The top portion of the figure shows the total number of variables, including derived variables, summed over all program points. These are subdivided into variables that are non-canonical (because they are equal to another variable); "missing" variables that do not always have sensible values (for example, `p.left` if `p` can be null, `a[i]` if `i` can be out of the bounds of `a`, or variables that are encountered uninitialized); and the remaining canonical variables. The table separately lists the number of derived variables (each of which appears above as non-canonical, missing, or canonical) and the number of derived variables that were suppressed (i.e., not instantiated and not counted above) because an invariant implied that they would be non-canonical or missing.

The number of variables is the most important factor in the number of invariants checked (Section 6.1). Daikon's rules for using previously-computed invariants to suppress certain derived variables eliminate from 9% (40 out of 460 for `tot_info`) to 53% (1198 out of 2276 for `tcas`) of potential derived variables. (These numbers are underestimates because other derived variables could have been created from those in certain circumstances.) Together with suppressing invariants for non-canonical variables (variables that have been determined to be equal to another variable) and variables with possibly nonsensical values, these approaches substantially reduce the runtime of the system. This is particularly true because these suppressed variables would be likely to participate in invariants that would not be eliminated early. In fact, the improvement is so large as to be unmeasurable. With these optimizations disabled, Daikon is slowed down by orders of magnitude and eventually runs out of over 256MB of memory — despite the fact that it canonicalizes all data, so (for example) there are no two distinct integer arrays anywhere in the implementation that contain the same contents.

The bottom of Figure 4.7 first shows the total number of invariants that were instantiated and checked. These are subdivided into falsified invariants that do not hold for some runtime variable values, unjustified invariants that are not falsified but for which the statistical tests do not support reporting them, redundant invariants that are not falsified but are implied by some other non-falsified invariant, and the remaining invariants that are reported by the system. The "Suppressed" line gives the total number of invariants that were never instantiated, checked, or reported. The figure breaks this number down into those that were known *a priori* to be false and those that were known to be true but redundant.

Daikon's runtime is more dependent on the number of non-falsified invariants (which are necessarily checked against all samples at a program point) than the number of potential invariants. Thus, the number of suppressed invariants should be compared not to the total number of instantiated invariants, but to the number that are not falsified. Implication substantially reduces the number of costly, non-falsified invariants that must be checked. The smallest benefit came for the Gries programs, where the suppressed invariants account for 46% (2340 out of 5084) of the total non-falsified invariants that would have been computed otherwise; for `replace` it rises to 84% (17538 out of 20980), and the other programs are over 94%.

These figures, too, are underestimates; for instance, when iterating over all possible triples of variables, if one variable, or a combination of two variables, caused all invariants involving them to be suppressed, we did not iterate over the remaining variables to count the exact number of suppressed invariants (which would have depended on other factors in any event). As was the case above, exact runtime improvements resulting from these checks are unavailable because the system simply does not run in their absence.

Just as invariant detection is interleaved with variable derivation (Section 4.3.1, page 33), different sorts of invariants are detected at different times.

Daikon tests, for each variable: (1) whether the variable is a constant or can be missing from the trace, (2) whether the variable is equal to any other variable, (3) unary invariants, (4) binary invariants with each other variable, and (5) ternary invariants with each pair of other variables.

Invariants discovered earlier can save work later. For instance, if two variables are dynamically determined to be equal (that is, an equality invariant holds over them), then one of the variables is marked as non-canonical and not considered any further. After invariant inference is complete for a set of variables, then derived variables are introduced and inference is performed over them. This staging of derivation and inference, and the sub-staging of inference, is not a mere performance enhancement. The system simply does not run when they are omitted, for it is swamped in large numbers of derived variables and vast numbers of invariants to check; both memory use and runtime skyrocket.

In a few cases, staging of inference did not eliminate all implied invariants before they were introduced; often this was because some invariants are introduced simultaneously so they can be checked together rather than making multiple passes over (summaries of) the data. Removing these invariants reduced the size of the output by about a quarter on average. (See the "redundant" (stage 3) row in the second half of Figure 4.7.) This lessens the burden on the user of sifting through them without decreasing the information content of the output.

4.7 Variable comparability

Some variables have nothing to do with one another, so relationships over them are bound to be uninteresting. As an example, given a boolean and a non-null pointer the boolean would always be less than the (unsigned) pointer. While true, this property is useless. Even variables of the same declared type can be unrelated. Given integers `myweight` (in pounds) and `mybirthyear` (as a four-digit year), $\mathsf{myweight} < \mathsf{mybirthyear}$ will always hold but is completely uninteresting.

Restricting which variables are compared to one another causes some potential invariants not to be considered. This improves runtime and could improve or reduce relevance depending on whether the suppressed invariants are relevant or not. It has the potential disadvantage of reducing the serendipity that results from unanticipated invariants (perhaps over variables the programmer had not previously associated with one another).

We compared four methods for computing a comparability relation over variables:

Unconstrained. Consider all variables to be comparable to one another.

Source types. Variables are comparable if they have the same declared type.

Coerced types. Variables are comparable if their declared program types are coercible to one other. For example, C automatically coerces `int`s to `long`s, so this approach considers such variables comparable. The current prototype does interconvert integral and floating-point values.

Lackwit. Two variables are comparable if they can contain the same value or values that can interact via program expressions [OJ97]. For example, if `a=b` or `a+b` appears in the program, then `a` and `b` are given the same Lackwit type.

Consider the code in Figure 4.8. The unconstrained approach considers all the scalars (including array elements, indices, and addresses) comparable to one another. The source types approach makes `i` comparable to elements of `b`, and `s` comparable to `n`. However, `i` is not comparable to `n`, since they have different declared types. In this example, coerced source types are the same as unconstrained, since `int` and `long` can be coerced to each other.

```
// Return the sum of the elements of array b, which has length n.
long array_sum(int * b, long n) {
    long s = 0;
    for (int i=0; i<n; i++)
        s = s + b[i];
    return s;
}
```

Figure 4.8: Variable comparability sample code. This is a C transliteration of the Gries array sum program (Gries Program 15.1.1 [Gri81]). The original program appears in Figure 2.1 on page 10.

Lackwit's static analysis captures value flow (or ability to contain the same runtime value) via polymorphic type inference over a non-standard type system that unifies variables between which values can flow [OJ97]. Two variables are comparable if they participate in an expression. For instance, `i` is comparable to `n` because of "**`i<n`**" and `s` is comparable to elements of `b` because of "`s+b[i]`". Comparability relationships are extended transitively in a way that permits polymorphism in the Lackwit types.

This small example also demonstrates the potential downside of using type-inference-based comparability to guide invariant detection. If all elements of `b` are positive, then $\mathtt{i} \leq \mathtt{s}$, but that invariant would not be computed because it involves variables that are incomparable, according to Lackwit. Although it is easy to generate such examples, we have found none in real code and do not believe that they will be common in practice: Lackwit tends to capture programmers' intuitive definition of comparability, particularly since it operates interprocedurally and thus takes account of surrounding context. (In this particular example, the programmer could also indirectly but easily infer the unreported invariant because Daikon would directly report that all elements of `b` are positive.)

Daikon does not use Lackwit in the same way that a programmer would. Lackwit was developed to support reverse engineering [OJ97]. The analysis was designed to be scalable (in particular, computationally inexpensive even on large programs) and to handle complex language constructs such as aliasing and higher-order functions (so that languages such as C could be analyzed). The consumer is a human who interactively queries Lackwit or views graphical representations of its output, allowing programmers to answer questions about a program's structure and to find various anomalies such as abstraction violations, unused data structures, and memory leaks. By contrast, Daikon uses Lackwit's analysis in an automatic manner to reduce the cost and improve the quality of a dynamic analysis.

Using three Siemens programs (Section 2.2), we measured the number of variable pairs considered comparable by each technique (Section 4.7.2) and the differences among invariants produced using each of the approaches (Section 4.7.1).

4.7.1 Reduced Comparability

Compared to the unconstrained approach, how much do the three other approaches reduce the number of comparable variables that Daikon must consider?

Figure 4.9 lists, for each method, the average number (over the three Siemens programs) of variables comparable to a given variable. The unconstrained approach makes each vari-

Comparability	Other vars	Ratio
Unconstrained	8.8	1.00
Source types	4.5	.51
Coerced types	5.1	.58
Lackwit	0.6	.06

Figure 4.9: Average number of variables to which each variable is comparable. For a randomly chosen variable in the Siemens suite, this table indicates, for each of the four comparability relations, how many other variables the given variable is expected to be comparable to. The final column shows the ratios between each method and the unconstrained method.

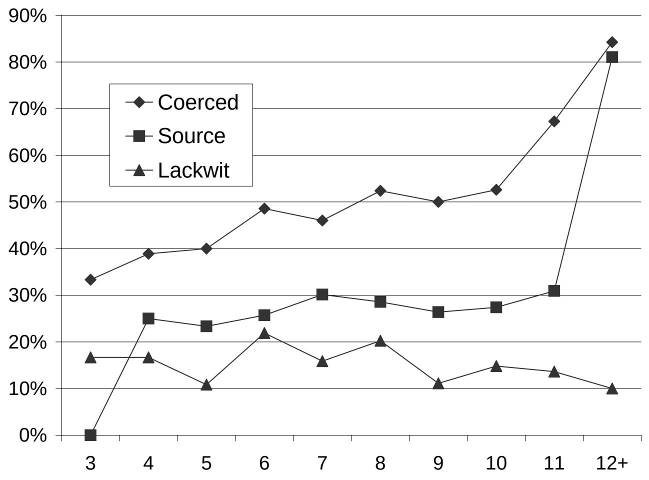


Figure 4.10: Reduction in comparability achieved by various static comparability analyses, graphed against number of variables in scope at a program point. The graph indicates, for a randomly chosen program point in the Siemens suite at which the specified number of variables is in scope, how much each comparability method reduces comparability, compared to the unconstrained case. The data of Figure 4.9 aggregate this information for all program points, regardless of number of variables. The numbers of variables do not include original values for parameters or other derived variables. The "12+" datapoint includes all program points with 12 or more variables in scope.

able comparable to every other variable, so at a program point with n variables in scope, each variable is comparable to $n-1$ others. For these programs, using program type constraints reduces the number of comparable variables 42–49% (depending on whether coercions are taken into account), while Lackwit reduces the number by 94%. Such comparisons have not previously been performed; they indicate quantitatively the effectiveness of Lackwit.

Figure 4.10 presents the same data, but broken down by the number of variables in scope at a program point. As the number of variables in scope grows, the constraints of source types or coerced source types become less effective (compared to unconstrained). In C, the number of types does not increase with function size, so larger functions have sharing of a fixed set of types. In contrast, the figure's Lackwit numbers tend to decrease as the number of variables increases. This suggests that the programs implicitly partition value flows; as

	Invariants		
Comparability	total	binary	Time
Unconstrained	100%	100%	100%
Source types	78%	61%	91%
Coerced types	78%	74%	96%
Lackwit	55%	27%	75%

Figure 4.11: Percentage of total and binary invariants reported, and time to compute all invariants, compared to unconstrained comparability.

the programs grow, the number of partitions tends to increase, as opposed to increasing the size of each partition.

4.7.2 Improved Relevance

Does the static reduction of variable comparability cause a corresponding reduction in the number of reported invariants? If so, are the removed invariants are likely to be irrelevant, and do more restrictive comparisons lead to performance improvements?

To address these questions for the three Siemens programs, we ran each program using the same trace files but with the four different comparability relations. For `replace` we used 3000 test cases randomly selected from the provided set, while the others used their full set of provided test cases, about 1500 cases each. Figure 4.11 shows the resulting data organized by comparability relation and averaged over the three programs. The total number of invariants, the number of binary invariants, and computation time are shown, all as percentages of unconstrained comparability. (Comparability does not directly affect unary invariants.)

Using Lackwit comparability significantly reduces the binary invariants reported (which in turn reduces the total invariants reported). On these programs, Lackwit comparability provides a significant reduction in invariants reported compared to any of the other comparability relations. The ratio of number of invariants reported when using Lackwit comparability to using unconstrained comparability ranges from .62 for `tot_info` to .03 for `replace`. The `tot_info` data are a bit unusual, since Daikon discovers only a small number of binary invariants (13 for unconstrained and eight for both source types and Lackwit). Lackwit types also provide a substantial performance improvement (which also has a large variance).

Our qualitative analyses compared the reported invariants for a given program and test suite across the four comparability relations. We focused primarily on `replace` because of our familiarity with it, which aids in making judgments about the potential relevance of reported invariants. The invariants removed as a result of using Lackwit comparability appear to be irrelevant for most likely programming tasks. For example, in procedure `amatch`, which contains two `char *` variables, `lin` and `pat`, the other three comparability relations (but not Lackwit) cause Daikon to report $\mathsf{lin} < \mathsf{pat}$. This invariant compares the pointer addresses: although values flow between elements within these arrays, the arrays themselves do not participate in any expressions. In procedure `makepat`, Lackwit comparability prevented an invariant between two unrelated boolean variables (`done` and `junk`) from being computed or reported, as it was for the other comparability relations.

These preliminary data suggest that computing invariants over only those variables that are considered comparable by the Lackwit typing mechanism is profitable in terms of relevance and performance.

Chapter 5

Invariants involving pointer-based collections

The techniques described so far are limited to finding invariants over scalars and arrays, and local properties of data structures. This chapter presents two simple techniques that enable discovery of invariants over collections of data represented by recursive data structures (including indirect links through tables, etc.). These techniques require minimal changes to the existing infrastructure.

The first technique, linearization, causes the instrumented code to traverse these implicit, potentially irregular collections and record them explicitly as arrays in the program traces. The basic Daikon invariant detector, whose design precludes direct representation of pointer structures and generic collections, can infer invariants over these trace elements. Examples include p.left $\in$ mytree and mytree is sorted.

The second technique, data splitting, splits data traces into multiple parts based on predicates Daikon chooses, then detects invariants in each part. This enables discovery of conditional invariants that are not universally satisfied, but whose truth is dependent on some other condition. Such invariants are necessary for reporting invariants over recursive data structures, which behave differently in their base and recursive cases, and are also useful in their own right. Examples include ptr $=$ null or $*$ptr $>$ i and if process.priority $<$ 0 then process.status $=$ active. Section 5.3 discusses several policies and a mechanism for computing conditional invariants.

The techniques are validated by successful application to two sets of programs: simple textbook data structures and student solutions to a weighted digraph problem.

As a motivating example, consider (part of) Daikon's output for a program that uses a linked list as a priority queue [Wei99]:

```
PriorityQueue:::CLASS
  prio.closure(next).rank is sorted by <=

void PriorityQueue.insert():::EXIT
  size(prio.closure(next)) = size(orig(prio.closure(next))) + 1
```

This output states that all elements reachable via `next` pointers from the root `prio` are sorted by their `rank` fields and that insertions increase the size of the priority queue. The output also contains a number of other invariants (some useful and some not); see Section 5.4 for details.

Section 5.1 distinguishes between local invariants, which are detected by the previously-described techniques, and global invariants that require a new approach. Section 5.2 discusses linearization for manipulating pointer-accessed collections. Section 5.3 describes how to create and test conditional invariants. Sections 5.4 and 5.5 experimentally assesses the effectiveness of these techniques. We evaluated them on the Weiss and 6.170 programs (described in Section 2.4). Daikon finds over 98% of the relevant invariants and reports less than 5% irrelevant invariants. In addition, the invariants identified several oversights in the design and implementation of the data abstractions. (Section 9.2 presents ongoing work in performing inference online, in conjunction with execution of the instrumented program. This optimization, which changes the architecture of the invariant detector, is necessitated by the potential size, and cost of traversing, pointer-directed data structures.)

5.1 Local and global invariants

Pointers present a challenge to invariant detection only in the context of recursive data structures, in which the system may have to traverse arbitrarily many links. Otherwise, a pointer can be treated as a record with two fields: the address and the content.

Invariants over pointer-based structures can be classified as either local invariants or global invariants.

A local invariant relates a small number of "syntactically close" objects. Syntactically close objects are reachable, from variables accessible in the same scope, within a fixed number of field dereferences, array accesses, or other operations. Examples of local invariants include $\mathsf{node.left.parent} = \mathsf{node}$, indicating that nodes point back to their parents, $\mathsf{emp.dept} = \mathsf{emp.manager.dept}$, indicating that an employee's manager is in the same department as the employee, and $\mathsf{a[i].rank} < \mathsf{a[j].rank}$, indicating that the `i`th element of the array takes precedence over the `j`th element. Because Daikon's instrumenters output object fields (up to a certain specified depth), the basic invariant detection techniques report local invariants.

A global invariant involves an arbitrary-size collection of objects, as in $\mathsf{x} \in \mathsf{mytree}$, $\mathsf{num\_used} < \mathsf{size(mylist)}$, and mytree is sorted. Local invariants do not always imply global ones; for instance, $\mathsf{a[i-1].rank} < \mathsf{a[i].rank}$ (for unconstrained `i`) implies that array `a` is sorted, but $\mathsf{p.left.rank} < \mathsf{p.rank} < \mathsf{p.right.rank}$ (for all `p`, when the specified elements exist) does not imply that the tree containing `p` is sorted. Therefore, global invariants must be checked explicitly. Daikon does so by explicitly representing the collection and performing invariant detection over it.

5.2 Invariants over collections

Although data structure traversal may be expensive, it need not be frequent, which can make the cost reasonable. When a data structure is no longer of interest (for example, all potential invariants over it have been falsified and are no longer being checked), then it need no longer be traversed; see also Section 9.2. Invariants could be checked on only some program point executions, which could be selected randomly and/or by backing off on checking after the invariant has been satisfied sufficiently many times. Dynamic invariant detection is necessarily unsound with respect to future program executions; this change makes the system unsound with respect to the test suite as well. In practice many dynamic analyses perform random sampling with good results. Traversal cost can be amortized by combining it with other traversals, such as that performed by garbage collection. Finally, data structure properties could be incrementally maintained. As an example, an analysis could indicate that tree rebalancing does not affect the order of its elements. Thus, no new traversal is needed after such an operation.

The Daikon invariant detector computes invariants only over scalars and arrays (see Section 3.1). This section extends Daikon to discover invariants over collections of data represented using structures other than arrays. The basic approach is to extend the instrumenter to find collections that are implicit in the program, linearize them, record them explicitly in the trace file as arrays, and then treat them like any other array. To linearize a collection, the instrumented program traverses the collection and writes out an array of the visited objects, which form the transitive closure of objects reachable via the collection's traversal operation.

This approach changes only the instrumenter. This design decision kept stable the most complex and error-prone component of Daikon, the invariant detector. Furthermore, as we show in Sections 5.4 and 5.5, this approach works quite well for our experiments to date. (Linearizing collections adds many variables to the data trace; Section 9.2 discusses strategies to reduce this cost.)

Linearizing an arbitrary-sized collection as an array requires selecting a root, determining a method for traversing the collection, and selecting the field or fields in the collection's objects to be written into the trace file. Program variables form the potential roots. When an object field leads to another object of the same type (e.g., `element.next`), the instrumenter outputs the two objects as successive elements in the same array. If there are multiple such fields (e.g., a `prev` field in addition to the `next` field), then multiple arrays are written into the trace file, one for each field. The linearized arrays are named `closure`*field*, where *field* is the name of the recursive field. Arrays are also created for each combination of fields. Two self-typed fields (such as `prev` and `next`, or `left` and `right`) might induce a tree, so the potential tree is linearized in-order, pre-order, and post-order.

As with other variables, fields with non-recursive types are also written out, to a user-specified depth. As it is linearizing collections, the instrumenter can write to the trace file additional information about the structures, such as whether they are cyclic, a dag, or a tree, which is computed during the data structure traversal but is not evident from the linearized form. This information is written as scalars that can be handled directly by Daikon.

Pointer-based data structures are no different than arrays in requiring traversal to determine some properties of the entire collection. Although conceptually similar, arrays do have several advantages. Accessing elements is cheap and syntactically and conceptually simple, and array size is often known *a priori* and arrays may not grow as large as arbitrary data structures.

Because of their prevalence, this section focuses on self-recursive object references, but the techniques can be generalized to other representations. Any entity that can serve as a proxy for a data object can operate as a pointer. For example, a Fortran program may create a pool of objects as an array, using the array indices to link the objects. Hash table keys are another example (see Section 5.5).

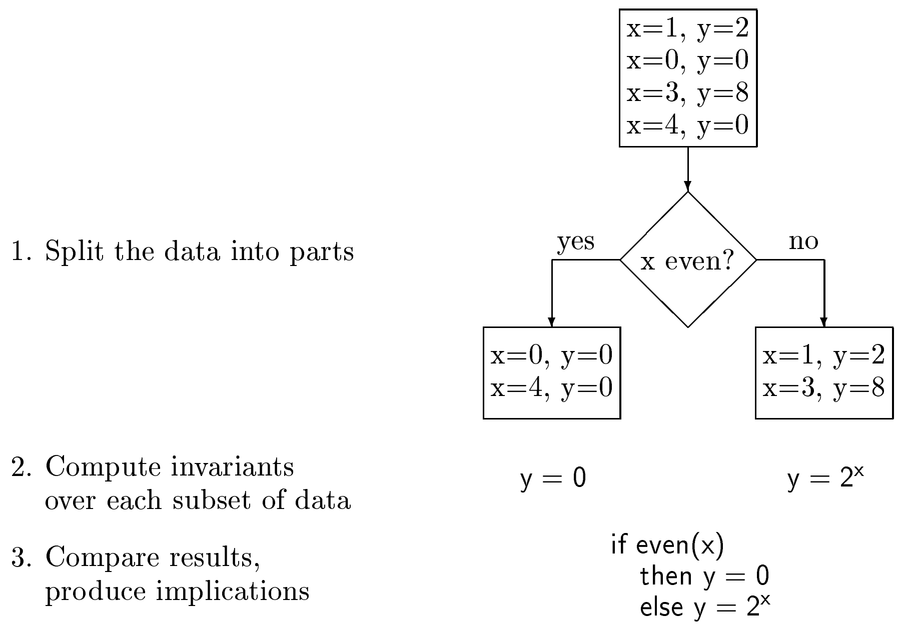

x=1, y=2
x=0, y=0
x=3, y=8
x=4, y=0
1. Split the data into parts
yes
x even?
no
x=0, y=0
x=4, y=0
x=1, y=2
x=3, y=8
2. Compute invariants over each subset of data
y = 0
y = 2^x
3. Compare results, produce implications
if even(x) then y = 0 else y = 2^x

Figure 5.1: Mechanism for computing conditional invariants.

5.3 Conditional invariants

Many important program properties are not universally true. For instance, the local invariant over a sorted binary tree, $\mathsf{p.left.value} < \mathsf{p.right.value}$, holds only if `p`, `p.left`, and `p.right` are non-null. Other examples include an absolute value routine with postcondition $\mathsf{if\ arg} < 0\ \mathsf{then\ result} = -\mathsf{arg\ else\ result} = \mathsf{arg}$ and a list deletion routine with postcondition $\mathsf{if\ x} \in \mathsf{orig(list)\ then\ size(list)} = \mathsf{size(orig(list))} - 1$, where `orig(list)` is the value of `list` on entry to the routine. Conditional invariants are particularly important for programs that manipulate recursive data structures, because different properties typically hold in the base case and in the recursive case.

Figure 5.1 illustrates the mechanism for detecting conditional invariants: split data traces into parts based on some predicate, perform invariant inference on each part, and report those sub-invariants, contingent on the splitting predicate. In the typical case, there are no significant differences between the invariants detected in the two parts of the data, so no conditional invariant is reported. This mechanism requires minimal change to the invariant detector. Splitting, like invariant detection itself, operates on each program point independently.

Section 5.3.1 presents policies for selecting the splitting criterion — the predicate that divides the data into parts. Section 5.3.2 solves a technical problem with modification bits, which must be propagated to each of the parts of the data.

There are many potential ways to split a program point's data; the system must decide how many of the candidate splits to use and how to combine them. For instance, given two splitting criteria p and q, there are at least 13 potential subparts of the data over which to perform invariant inference: the whole data (i.e., the condition *true*), four parts defined by one condition (p, $\neg p$, q, $\neg q$), and eight parts defined by both conditions ($p \wedge q$, $\neg(p \wedge q)$, $p \wedge \neg q$, $\neg(p \wedge \neg q)$, $\neg p \wedge q$, $\neg(\neg p \wedge q)$, $\neg p \wedge \neg q$, $\neg(\neg p \wedge \neg q)$).

Splitting a program point linearly increases work, because invariant inference is repeated for the subparts of the data. Combining splits may result in even finer, more useful invariants; but it always increases computation time. Daikon uses a single level of splitting on simple programs. For instance, in the example with splitting criteria p and q, it would examine the entire trace and the four subtraces for p, $\neg p$, q, and $\neg q$.

When splitting criteria are chosen *a priori*, it is easy to incrementally add new samples, but the samples are processed independently for each split (and for the whole, unsplit program point). It is not feasible to save work by performing invariant detection over the full data, then modify the results by removing some samples: in the absence of certain counterexamples, different invariants will be suppressed, checked, and detected, and different derived variables will be generated or suppressed. Two other approaches to eliminating repeated work might be advantageous, though Daikon does not yet support either one. The first is to start invariant inference from scratch for each part of the data but discard an invariant if it is no stronger than that detected over all the data (regardless of confidence over the subpart of the data). The second is to compute invariants over the parts and then combine them, eliminating the need for computation of invariants over the whole but causing difficulties in computing confidences for those invariants. A multi-level splitting strategy could examine information at shallower splits before combining slicing criteria.

5.3.1 Splitting policy

The splitting policy determines the predicate used for splitting data traces into parts. We considered the following splitting policies:

- A *static analysis* policy selects splitting conditions based on analysis of the program's source code. Daikon currently implements this policy.

- A *special values* policy compares a variable to preselected values chosen statically (such as null, zero, or literals in the source code) or dynamically (such as commonly-occurring values, minima, or maxima).

- A policy based on *exceptions to detected invariants* tracks variable values that violate potential invariants, rather than immediately discarding the falsified invariant. If the number of falsifying samples is moderate, those samples can be separately processed, resulting in a nearly-true invariant plus an invariant over the exceptions.

- A *random* policy could perform exhaustive or stochastic splitting over a sample of the data, then re-use the most useful splits on the whole data.

- A *programmer-directed* policy allows a user to select splitting conditions *a priori*.

The policies are described in more detail below.

Static analysis. Daikon currently splits data traces based on a simple examination of the program text. Each boolean condition used in the program (for instance, as a test in an `if` statement) is used as a splitting condition. Only conditionals that refer to variables in scope at the program points are of interest; this causes most splitting conditions to be used only for the single procedure in which they appeared.

A condition tested in one procedure may be meaningful for another procedure in which all those variables are in scope. Splitters are often invalid outside the procedures that contain them, especially if they reference local variables. However, we found that such splitters are sometimes serendipitously useful, for instance if there is another local variable of the same name and similar meaning.

A more sophisticated static analysis could perform a dataflow or other analysis in order to determine more complicated conditions, or conditions not explicit in the program text, that are likely to be true or to produce an interesting split. The next policy also includes an aspect of static analysis.

We implemented the static analysis policy of using boolean expressions in a method and pure boolean member functions. (The functions must be side-effect-free and non-exception-throwing; allocations are permitted so long as the new objects do not escape.) Daikon automatically generates and applies these splitting conditions. The results in Sections 5.4 and 5.5 demonstrate that not only is this policy easy to implement, it also works well for the programs we considered.

One reason for the success of static splitting may be that the methods in the programs we tested are relatively simple. For example, the `LinkedList` program has only a single conditional in its body and using this as a splitting condition led to the discovery of useful conditional predicates. More experiments are required to indicate whether the simple approach must be augmented for more complicated programs.

Special values. Variables often give additional information about, or guard access to, other values. For instance, quite different invariants are likely to hold when the tag of a union data structure takes on different values. A static analysis could identify values with special meaning, such as 0, `null`, and other values standing for missing data. Literals that appear in the source code may also serve as markers.

Dynamic analysis can also identify special values. A variable's initial value may be used before a data structure is initialized, and its final value can also have special meaning, such as being a complete, correct count. The program may take special action when extremal values such as a minimum or maximum are encountered; such extremal values may also appear statically, reducing the need for a dynamic analysis to detect them. A distribution's mode and other common values may represent a boundary case or a common case; in either event, there is a natural disjunction to be found. Finally, if a value takes on only a few distinct values, a complete split for each value may be worthwhile.

Exceptions to detected invariants. When an invariant is falsified during testing, Daikon discards it, testing it over no more value samples and not reporting it. This avoids reporting untrue invariants and improves runtime. However, it may be worthwhile not to discard invariants immediately, but to permit some exceptions to invariants before dismissing them entirely.

Such a strategy would cache exceptions to each invariant. If the exceptions exceeded a certain number or percentage of all encountered samples, then the invariant would be discarded as before. So long as there are a modest number of exceptions, they would be retained. Then, invariant detection could be performed over the exceptions to see under which conditions the original invariant does not hold.

This strategy permits multiple invariants of a particular variety to be active simultaneously. For instance, given three non-colinear $\langle x, y\rangle$ pairs, there are three different linear ($\mathsf{y} = \mathsf{ax} + \mathsf{b}$) relationships, each with a single exception.

Random splitting. Not all interesting data splits can be predicted *a priori* or discovered via heuristics. As a result, it may be worthwhile to perform either exhaustive splitting, or to try a number of random splits, over a subset of the data. The splits that produce conditioned invariants would be used on the full data. This potentially expensive approach could be quite useful in moderation; randomized algorithms often do well in other domains.

Furthermore, the random splits need not be perfect. For a condition that holds over part of the data trace, it is enough that it hold on one side of one of the random splits — that is, in one random group of samples — in order to be detected on the first pass. The condition would become the splitting criterion for the second (full) invariant detection pass, dividing the full data precisely. This strategy may be best for detecting properties that are usually true. For instance, if 90% of samples satisfy a property, then given 11 random groups of 10 samples each, it is over 99% likely that there exists a group all of whose samples satisfy the property. If half of all samples satisfy the property, then 4714 groups of 10 samples are needed in order to be 99% sure (or 2357 groups to be 90% sure) that all the samples of some group satisfy the property. These numbers can be computed from the formula

$$r = \log_{(1-p^s)}(1-c) \ ,$$

where p is the probability that a random sample satisfies the property, s is the size of each group, c is the desired confidence that all samples in some group satisfy the property, and r is the number of necessary groups (repetitions). Strictly speaking, this calculation should be adjusted to account for the possibility that every sample in all of the groups satisfies the property, which would also prevent the splitting condition from being detected on the first pass. This is accomplished by adjusting the confidence formula (from which the above formula for r was derived) from $c = 1 - (1-p^s)^r$ to $c = (1-(1-p^s)^r)(1-p^r s)$. Unless the property is very likely to hold and the number of groups is quite small, this adjustment is negligible. In 20 groups of 10 samples each, a property that holds over 99% of all samples is 87% likely to be falsified at least once.

value	assign	split
1	yes	A
1	no	B
1	no	B
1	no	A
1	no	B
2	yes	A
2	no	A
2	no	A
2	yes	A
2	no	B

(a) Original trace

Split A		Split B	
value	assign	value	assign
1	yes		
		1	no
		1	no
1	no		
		1	no
2	yes		
2	no		
2	no		
2	yes		
		2	no

(b) Naive split

Split A		Split B	
value	assign	value	assign
1	yes		
		1	**yes**
		1	no
1	no		
		1	no
2	yes		
2	no		
2	no		
2	yes		
		2	**yes**

(c) Adjusted modification bits

Figure 5.2: Modification bits and data splitting. The left-hand table shows a data trace, at a program point that was executed ten times, for a variable that was assigned three times (to values 1, 2, and 2). A splitting criterion (not shown) splits the data into two parts. The middle table shows the data split without any adjustment to the modification bits that indicate whether the variable was recently assigned; no modification bits are set for Split B. The right-hand table shows the modification bits adjusted so that each assignment sets (up to) one bit in each split.

Programmer-directed splitting. Programmers usually have intuitions about what properties are most important, are most likely to be of interest, or are most relevant to their particular task. Thus, programmers can supply extra information, such as which test case group a particular run came from, in order to compare such characteristics. Programmers could also directly propose splitting criteria. In particular, absent but expected invariants might suggest splitting predicates.

5.3.2 Repeated values

Daikon reports only invariants that pass a statistical confidence test. In order to avoid overweighting loop-invariant (and other unchanged) variables, the instrumentation tracks lvalue assignments and records a boolean value in the data trace, indicating whether the lvalue was assigned since the last visit to that particular program point (Section 7.5). A new sample adds confidence to an inferred invariant only if the boolean is set — that is, one of the invariant's lvalues has been assigned (Section 4.5.1).

When data traces are split in order to detect conditional invariants, all of the bits that are set may be assigned to one of the subparts. For an example, see Figure 5.2. This raises two problems. First, the modification bits no longer capture the intuition of a variable being recently set (and having one set bit per assignment), and so invariant confidence is skewed and desired invariants do not appear in the output. Second, the optimization that assumes that if the modification bit is not set, the value is the same as the previous one, is no longer justified.

Daikon avoids these problems by adjusting some modification bits, making each dynamic assignment result in a true modification bit in each subpart of the data. This is done without

Value	Assign	Split
1	yes	A
1	no	B
1	no	B
1	no	A
1	no	B
2	yes	A
2	no	A
2	no	A
2	yes	A
2	no	B

(a) Original trace

Assign Accum	
Split A	Split B
yes	yes
no	yes
no	no
no	no
no	no
yes	yes
no	yes
no	yes
yes	yes
no	yes

(b) Accumulator

Split A		Split B	
Value	Assign	Value	Assign
1	yes		
		1	yes
		1	no
1	no		
		1	no
2	yes		
2	no		
2	no		
2	yes		
		2	yes

(c) Modification bits

Assign Accum	
Split A	Split B
no	yes
no	**no**
no	no
no	no
no	no
no	yes
no	yes
no	yes
no	yes
no	**no**

(d) Accumulator

Figure 5.3: Accumulator for modification bits and data splitting. The final result of Figure 5.2(c), which is repeated here as the third table, is achieved by using an accumulator bit for each variable and subpart of the data. This example shows two accumulators, because there is only one variable and the data is split into two parts.

The left-hand table shows the original data trace. Whenever an modification bit is set in it, each of the corresponding accumulators is set, as illustrated in the second table. Modification bits are set in the sub-parts of the data (in the third table) according to the accumulator bits, then whatever accumulator bit was used is set to false (shown in the fourth table). The second and fourth tables show the same bits, but at different points in processing; the nth row of the right-hand table is the input used on the $(n + 1)$st row of processing. The result is that each bit that is set in the original trace sets at most one bit in each subpart of the data.

change to the instrumenter. The invariant detector maintains, for each subpart of the data, an accumulator bit per variable. The modification bits of the original trace are or-ed into the accumulator, and the accumulator is used (and cleared after use) for the modification bits in its part. (See Figure 5.3.) This implementation does not work if the samples are stored unordered in a database; it requires online processing or storing samples in order. Our experiments show that, across a range of test cases, few samples are exactly repeated, and there is temporal locality when they are repeated. So, using run-length encoding and interning of data structures, the additional memory overhead of an ordered representation is hardly noticeable, even if all samples are kept in memory.

Data splitting may cause invariants that were reported over the whole not to be reported over the parts. The missing invariants are not statistically justified over the smaller number of samples in the part.

5.4 Textbook data structures

This section reports the results of running Daikon over the first five data structures in a data structures textbook [Wei99]: linked lists, ordered lists, stacks represented by lists, stacks represented by arrays, and queues represented by arrays. (Analysis of other data structures gave similar results.) Before examining the reported invariants, we determined the desired output (the goal invariants that should be reported by an ideal invariant detector) by reading the book and the programs. We were able to objectively determine this "gold standard" because the programs are small, described in the textbook, and implement well-understood data structures. We extended Daikon with linearization and splitting, plus additional implication tests necessitated by those changes. However, we did not introduce new types of invariants for Daikon to check: the invariants already being checked were adequate for these pointer-based structures. (This suggests that a small, basic, general set of invariants may be broadly applicable; hand-tuning the set of potential invariants for each program is not necessary.)

This section reports what percentage of the reported invariants are relevant (like precision in information retrieval [Sal68]) and what percentage of the goal invariants are reported (like recall in information retrieval).

The textbook's implementation of the 5 data structures comprises 7 classes, ranging in size from 12 lines (for `ListNode`) to 65 lines (for `QueueAr`) of non-blank, non-comment code. (For comparison, the Sun JDK 1.2.2 `java.util.LinkedList` class is implemented in 673 lines, of which 255 are non-comment and non-blank. Its interface is richer but its functionality is essentially the same as the textbook's `LinkedList` class, so Daikon would perform similarly, for the same test suite.) For simplicity, this presentation assumes that the polymorphic data structures are rewritten to contain `Integer` objects; see Section 4.4 (page 34) for details about handling polymorphism.

Because the provided test suites are minimal at best, we augmented them with additional test cases. These additional tests are far from comprehensive, but they do exercise more of the code. Our test driver creates 32 instances of the specified data structure and performs between 0 and 15 insertions and between 0 and 15 deletions. Each inserted element is a random integer in the range 0–31.

Figure 5.4 tabulates Daikon's precision on the five textbook data structures. We manually determined the relevance of the reported invariants (see Section 4.2, page 31). Daikon reported at least 95% of the relevant invariants for each class.

Figure 5.5 shows data on goal invariants that Daikon failed to detect. Daikon reports most relevant invariants; its recall is always at least 98%. Daikon fails to report invariants for only two reasons. First, the invariant may be beyond Daikon's vocabulary, not expressible in the grammars of Sections 3.2 and 4.3. Second, the invariant may be detected but not reported, because the invariant is determined not to be relevant, so reporting it would be more likely to be unhelpful than helpful to a Daikon user. Sections 4.5–4.7 list the situations in which a detected invariant is not reported.

The remainder of this section qualitatively assesses the invariants detected on these five data structures, For brevity, we discuss each invariant only once, even if it was detected at multiple program points or in multiple data structures. Such invariants include usage properties, freedom from side effects, object invariants, invariants over helper classes, and others.

class	relevant	implied	irrelevant	precision
`LinkedList`	317	11	1	96%
`OrderedList`	201	5	5	95%
`StackLi`	184	8	1	95%
`StackAr`	159	0	0	100%
`QueueAr`	500	0	0	100%
`ListNode`	46	1	1	95%
`LinkedListItr`	185	8	0	95%

Figure 5.4: Invariants detected in textbook data structures [Wei99]. `ListNode` and `LinkedListItr` are used internally by the first three data structures. The "relevant" column counts the number of reported invariants that were relevant. The "implied" column counts the number of invariants that were implied by other invariants (the tests of Section 4.6, which were not fully implemented at the time of this experiment, could eliminate these). The "irrelevant" column counts the number of reported invariants that were irrelevant. The "precision" column is the ratio of the number of relevant invariants to the total number of reported invariants. These numbers do not include class invariants that were repeated at method entries and exits and tautological invariants that are necessarily true based on the subparts of variables being compared, all of which were removed by an automated postprocessing step.

class	relevant	missing	recall
`LinkedList`	317	3	99%
`OrderedList`	201	5	98%
`StackLi`	184	0	100%
`StackAr`	159	0	100%
`QueueAr`	500	10	98%
`ListNode`	46	0	100%
`LinkedListItr`	185	2	99%

Figure 5.5: Invariants not detected in textbook data structures [Wei99]. The "relevant" column is repeated from Figure 5.4. The "missing" column counts desired invariants that Daikon failed to report; see the text for discussion. The "recall" column is the ratio of relevant to the sum of relevant and missing.

5.4.1 Linked lists

Figure 5.6 displays some of the invariants detected over linked lists. Linked lists are implemented with a header node that is not part of the list proper. The object invariants indicate that there is always at least one `ListNode` reachable from `header`, which is implied by the fact that `header` is not null. Additionally, `header.element` (which is never used) is always set to 0.

Most of the `findPrevious` entry invariants are usage properties dependent on the particular program and test suite: elements are random integers between 0 and 31, and the maximum list size is 15. The first equality indicates that the test program always inserts

```
LinkedList:::CLASS
  header != null
  size(header.closure(next)) >= 1
  header.element = 0

LinkedListItr LinkedList.findPrevious(Object x):::ENTER
  p.current = header
  x <= 31
  x >= 0
  size(header.next.closure(next)) <= 15

LinkedListItr LinkedList.findPrevious(Object x):::EXIT
  x = orig(x)
  header = orig(header)
  header.closure(next) = orig(header.closure(next))
  header.closure(next).element = orig(header.closure(next).element)
  return != null
  return.current != null
  x != return.current.element
  return.current.closure(next) is a subsequence of header.closure(next)
  MISSING: return.current.next.element = x

void LinkedList.insert(Object x, LinkedListItr p):::EXIT
  x = header.next.element
  if (p != null && p.current != null)
    then size(orig(header.next.closure(next))) = size(header.next.closure(next)) - 1
    else header.closure(next) = orig(header.closure(next))

boolean LinkedList.isEmpty():::EXIT
  if (header.next == null)
    then return = true
    else return = false

void LinkedList.remove(Object x):::EXIT
  size(header.next.closure(next)) <= size(orig(header.next.closure(next)))
  MISSING: if (findPrevious(s) != null)
    then size(header.next.closure(next)) = size(orig(header.next.closure(next))) - 1
    else size(header.next.closure(next)) = size(orig(header.next.closure(next)))
```

Figure 5.6: Linked list invariants. This is a portion of Daikon's output for the `LinkedList` class [Wei99]. Invariants annotated by `:::CLASS` are object invariants that are valid at the entry and exit of every public method. The notation `closure(`*fieldname*`)` stands for the collection of objects reachable by following *fieldname* pointers; `orig(`*val*`)` stands for the value of *val* at entry to a procedure; and `size(`*val*`)` stands for the size of array or collection *val*.

at the beginning of the list. The routine's exit invariants indicate that it does not change `x`, `header`, or objects accessible from `header` or their fields. In other words, the routine has no side effects. Additionally, the routine always succeeds for this test suite: the return value is never null. The antepenultimate exit invariant indicates that the argument is never

```
OrderedList:::CLASS
  header.closure(next).element is sorted by <=
  MISSING: header.next.closure(next).element is sorted by <

void OrderedList.insert(Integer x):::EXIT
  size(header.next.closure(next)) >= size(orig(header.next.closure(next)))
```

Figure 5.7: Ordered list invariants. Only (some) differences from Figure 5.6 are shown.

returned. The penultimate one indicates that the return value points into the original list, and the final (missing) invariant is the basic contract of the procedure. Given a larger data structure depth for writing fields to the data trace file, Daikon finds that invariant; see the discussion of ordered lists (Section 5.4.2) for details.

The first `insert` invariant indicates that, for this program, insertion always occurs at the beginning of the list. The second one shows the conditions for successful insertion, in terms of the iterator `p`, which indicates where in the list to insert the new element. When insertion is successful, the list size increases by one. No size invariant is reported for unsuccessful insertion because the equality of the two collections implies that they have the same size; the redundant size invariant is automatically suppressed.

The `remove` invariant is an inequality over sizes because deletion is not always successful; additionally, it is not found to always occur at the beginning of the list, as was the case for insertion. Daikon does not, however, detect the exact predicate for determining when deletion was successful. The reason is that it currently splits only on conditions occurring in the program and zero-argument boolean member functions. Using unary boolean member functions, would add the desired condition. (The output would be most perspicuous if there were an `isMember` function, but the program lacks that.) Splitting on local variable `s` would also solve the problem.

5.4.2 Ordered lists

Most invariants over ordered lists are identical to those for arbitrary linked lists; some differences appear in Figure 5.7. The additional class invariant indicates that the linked list pointed to by the header is always sorted in terms of `element` values. The $\leq$ relationship results from the header element's value of 0, for the first element may also be 0. The strict $<$ ordering relation over the list proper is missed only because the default depth for outputting data structures does not derive the needed variable `header.next.next.closure(next)`. When we increased the depth (a command-line argument to the instrumenter) by 1, the desired invariant is produced, and likewise for `LinkedList.findPrevious`.

Some invariants that appeared in `LinkedList` do not appear in `OrderedList`. Insertion does not always occur because `OrderedList` permits no duplicate values, and insertion does not always occur at the list head. Daikon fails to find the condition predicate over the size of `insert`'s result for the same reasons as for `LinkedList.remove` above.

```
void StackLi.push(Object x):::EXIT
  x = topOfStack.element
  topOfStack.next = orig(topOfStack)
  topOfStack.next.closure(next) = orig(topOfStack.closure(next))
  size(topOfStack.closure(next)) = size(orig(topOfStack.closure(next))) + 1

Object StackLi.topAndPop():::EXIT
  return = orig(topOfStack.element)
  topOfStack = orig(topOfStack.next)
  topOfStack.closure(next) = orig(topOfStack.next.closure(next))
  size(topOfStack.closure(next)) = size(orig(topOfStack.closure(next))) - 1
```

Figure 5.8: Stack invariants (list representation).

```
StackAr():::EXIT
  theArray = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
  topOfStack = -1

boolean StackAr.isEmpty():::EXIT
  topOfStack >= 0
  return = false

void StackAr.push(Object x):::EXIT
  x = theArray[topOfStack]
  topOfStack >= 0
  orig(topOfStack) = topOfStack - 1
```

Figure 5.9: Stack invariants (array representation).

5.4.3 Stacks: list representation

The first three detected invariants in Figure 5.8 for stack `push` (for stacks implemented via linked lists) completely capture the operation's semantics; the `pop` invariants are symmetric. The fourth invariant for `push`, indicating that the stack grows by one after an insertion, does not explicitly appear in the output, as an artifact of choosing a canonical variable among equal variables. Instead, the output includes the invariants

```
  size(topOfStack.next.closure(next)) = size(topOfStack.closure(next)) - 1
  topOfStack.next.closure(next) = orig(topOfStack.closure(next))
```

and these imply the fourth invariant. Daikon explicitly reported the corresponding invariant for `pop`.

The first invariant at the exit of the `topAndPop` method captures the return of the top stack element. The second and third capture the notion of popping the stack. The final invariant indicates that the method decreases the size of the linked list by 1.

```
QueueAr():::EXIT
  currentSize = 0
  currentSize = front
  back = 15
  theArray = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Object QueueAr.dequeue():::EXIT
  return = theArray[front-1] = theArray[orig(front)]
  front = orig(front) + 1
  back = orig(back)

void QueueAr.enqueue(Object x):::EXIT
  x = theArray[currentSize-1] = theArray[back]
  back != orig(back)
  front = orig(front)
  currentSize = orig(currentSize) + 1
  MISSING: back = ((orig(back) + 1) mod DEFAULT_CAPACITY)
```

Figure 5.10: Queue invariants.

5.4.4 Stacks: array representation

Figure 5.9 shows invariants for a stack implemented by an array. The invariants for the exit point of the constructor `StackAr` show the initialization of the stack: all elements are zeroed and the top index takes an initial value. The `isEmpty` invariants reflect a shortcoming of the test suite: no tests were performed when the stack was empty. (Similarly, the stack never filled, which would result in more conditional invariants.) That the data structure is a stack is captured in the invariants on `push`: the element is inserted at the top of the stack, the top index is incremented, and the stack is non-empty after a push. The invariants for `pop` are similar.

5.4.5 Queues

Invariants inferred over queues implemented with an array representation appear in Figure 5.10. The constructor postconditions capture the initializations of the internal representation of the queue. The invariant $\mathsf{currentSize} = \mathsf{front}$ is accurate but coincidental, since in principle the initial length is always zero but another index could have been used to represent the front of the queue.

Collectively, the invariants at the exit of `QueueAr` say that the proper element is returned, the back index changes, and the front index remains unchanged. The invariant over the back index is accurate but too weak: we would prefer to find the missing modulus invariant. Adding differences and sums of variables as derived variables would cause this invariant to be detected (in a slightly different form). Our original prototype did so, but at the time of these experiments we had not yet added that to the current version, as we were unsure of its general applicability.

Another missing invariant is $0 \leq \mathsf{back} < \mathsf{theArray.length}$. This invariant, which indicates that the index of the last element of the queue is within the bounds of the array, is not satisfied by the code. Both the `Queue` constructor and its `makeEmpty` routine violate this property by setting `back` to -1. This is safe because queue insertion increments `back` before using it; however, those routines also reduce it modulo `theArray.length`, so this optimization saves only some portion of the modulus operation, and only once per data structure. The code would be improved by obeying the natural invariant throughout. (Other data structures in the book suffer from the same kind of problem.)

5.5 City map student programs

We also applied Daikon to Java programs written (to a single specification) by students in a software engineering course at MIT (6.170 Software Engineering Laboratory). The students were assigned to write code, formal specifications, and test cases for a city map implemented in terms of a weighted digraph. These programs are larger (1500–5000 lines) and more realistic, and the students wrote down representation invariants; however, we found that frequently those specifications were incomplete or were not satisfied by the code. Thus, we cannot report what percentage of desired invariants are reported, only what percentage of the reported invariants are relevant. We also present a qualitative analysis.

Although a weighted digraph is inherently recursive, no student implemented it via a recursive data structure. The implementations tended to use tables indexed by graph nodes; these were sometimes nested and sometimes not, and the indexed data varied as well. Determining a node's edges generally required at least one table lookup (with the node as the key). This approach to representing collections is not surprising given the availability of a library of efficiently implemented abstractions. More explicit representations of collections (for instance, as resizeable arrays) lessen the pressure for Daikon to handle recursively-defined collections; however, other properties (such as connectedness in the resulting graph) may become more difficult to infer (see Section 5.2).

The students were directed to achieve branch coverage with their test suites (and given a tool to test branch coverage), and most students stopped immediately upon achieving that goal. Thus, many methods were executed just a few times by their tests, and at many program points, no invariants were statistically justified. To report results consistent across the programs, we report object invariants for three classes (`Table`, `WtDigraph`, and `DistanceChart`) that appear in all the student programs. An object invariant holds at entry to and exit from all of the class's public methods.

Assessing the Daikon output for these programs is more difficult than for the textbook data structures considered in Section 5.4. The students' formal specifications were often spotty in quantity and quality, failing to note important properties of the code. Determining the true set of relevant invariants would require careful hand-analysis of the programs, which range from 1500–5000 lines of code. In any case, it would not be compelling to report Daikon's success in reporting invariants we had determined ourselves (even though, as was the case for the experiment of Section 5.4, we introduced no new invariants for Daikon to check). Thus, there is no "gold standard" against which to compare the Daikon output.

Figure 5.11 quantifies Daikon's output for the first four student programs we assessed in detail. The invariants listed as "relevant" were both stated by the students and also found by Daikon. They include uniqueness and non-nullness of collection elements, constraints

	relevant	implied	irrelevant	missing	added
Student 1	15	37	72	4	0
Student 2	13	24	27	10	1
Student 3	19	25	64	5	3
Student 4	13	30	48	10	7

Figure 5.11: Object invariants detected for 3 key classes (`Table`, `WtDigraph`, and `DistanceChart`) in student city map programs. The "relevant" column is the number of relevant invariants reported that also appear in the students' formal specifications. The "implied" column is the number of redundantly reported invariants that are implied by other (relevant) ones. The "irrelevant" column is the number of reported invariants that are not relevant. The "missing" column is invariants in the students' formal specifications that did not appear in Daikon's output. The "added" column is relevant invariants detected by Daikon that the students erroneously omitted from their formal specifications.

over data structure sizes, and the like.

The "added" invariants were missing from the student formal specifications but discovered by Daikon. As an example, some implementations of the `Table` abstraction used parallel `keys` and `values` arrays; Daikon always reported size(keys) = size(values), but some formally specified representation invariants omitted that property. Similarly, some but not all students explicitly noted that certain values should be non-null, a property that was also picked up by Daikon. Daikon also discovered some invariants of the test suites, such as size(nodes) $\geq$ size(edges.keys), which indicates that there were never more edges than nodes, and size(distanceChart.edges.keys) = 0 in `addCity`, which indicates that all cities were added before any highways.

The "implied" invariants are redundant, because given other reported invariants, they are necessarily true. Most implied invariants are suppressed from the output; that some remain is an easily corrected weakness of our special-purpose logical implication checker. We had not completed its implementation when we ran this experiment.

The "irrelevant" invariants are nearly all comparisons between incompatible runtime types (for instance, a city is discovered to be never equal to a highway). Daikon performs these comparisons because it respects the statically declared `Object` program types. One of two techniques would eliminate most or all of these irrelevant invariants: either using static analysis to determine that values cannot flow from one variable to the other (Section 4.7, page 46), or performing two stages of invariant detection, the first of which determines the actual runtime types (Section 4.4, page 34). So, applying known technologies will allow us to eliminate almost all of the "implied" and "irrelevant" invariants, cleaning up the Daikon output. That implementation work had not yet been completed at the time the statistics were gathered.

The "missing" numbers compare against student formal specifications and so are underestimates of the number of true invariants missed, because in most cases the students' formal specifications were incomplete. (In at least one case, a comment also indicated that the student believed a representation invariant held at a point in the constructor where it had not yet been established.)

The "missing" invariants fall into four categories. First, invariants stating that the graph is bidirectional or contains no self-edges require existential quantifiers or other mechanisms not currently supported by Daikon. These invariants are easy to state informally, but the programs' self-checks for these properties were sometimes a dozen lines long. Second, invariants about mutability — for example, "the keys are immutable" — would require an analysis of immutability as well as run-time types. A value can be immutable even if Daikon does not detect it as a constant; for instance, it might have different values on different invocations or program executions. Third, several invariants over the run-time types of objects were not detected due to polymorphism or inadequate test suites. Finally, the invariant that a collection contains no duplicates can be detected by Daikon, but in practice often was not because at runtime the specified collections were very small (for instance, maximum outdegree was often two) and thus were observed too few times for Daikon's statistical tests to permit the invariant to be reported. This is a deficiency of the test suites. (For some student programs, Daikon did report that collections, such as the global collection of all nodes, contained no duplicates or null entries.)

Detecting the invariants for these programs was inexpensive. The trace files ranged in size from 500KB to 15MB. For each of the four programs, the Java implementation of Daikon consumed approximately two minutes of processing time (on a 143MHz Sun Ultra 1 Model 140 running SunOS 5.5.1, running on the Classic JVM version 1.2.2 with no JIT compiler and all debugging assertions enabled) and reported between 700 and 1900 invariants at 59 to 83 program points.

Different student programs deal with exceptional conditions differently. For example, some data structures raise exceptions that clients are expected to catch, while others return a distinguished value in the case of nonsensical inputs or invalid commands. Additionally, some signaled exceptions only in the case of errors, while others used exceptions for ordinary control flow. The latter style obviated the need for many conditional invariants, because one condition held for all ordinary exits and another condition held for exceptional exits. The trace was already effectively split.

Chapter 6

Scalability and test suite selection

The time and space costs of dynamic invariant inference grow with the number of program points and variables instrumented, number of invariants checked, and number of test cases run. The cost of inference is hard to predict, however. For example, Daikon generates derived variables while analyzing traces, and which derived variables are introduced depends on the trace values. Also, Daikon stops testing for an invariant as soon as it is falsified, meaning that running time is sensitive to the order of variable value tuples. Finally, selection of test cases — both how many and which ones — impact what invariants are discovered. This chapter presents the results of several experiments to determine the costs of invariant inference (Section 6.1), the stability of the reported invariants as the test suite increases in size (Section 6.2), and the feasibility of automatically generating test suites (Section 6.3). These results enable users to predict and control runtime and output quality based on quantitative, observable factors.

6.1 Performance

Invariant detection time depends primarily on the number of variables, test suite size, and program size; these factors are multiplicative. Briefly, invariant detection time is:

- potentially cubic in the number of variables in scope at a program point (not the total number of variables in the program). Invariants involve at most three variables, so there are a cubic number of potential invariants. In other words, invariant detection time is linear in the number of potential invariants at a program point. However, most invariants are falsified very quickly, and only true invariants need be checked for the entire run, so invariant detection time at a program point is really linear in the number of true invariants, which is a small constant in practice.
- linear in the number of samples (the number of times a program point is executed), which determines how many sets of values for variables are provided to Daikon. This value is linearly related to test suite size; its cost can be reduced by sampling.
- linear in the number of instrumented program points, because each point is processed independently. In the default case, the number of instrumented program points is proportional to the size of the program, but users can control the extent of instrumentation to improve performance if they have no interest in libraries, intend to focus on part of the program, etc. Daikon's command-line parameters permit users to skip over arbitrary classes, functions, and program points.

Informally, invariant detection time can be characterized as

$$Time = O(\ (|vars|^3 \times falsetime\ +\ |trueinvs| \times |testsuite|)\ \times |program|\ )\ ,$$

where *vars* is the number of variables at a program point, *falsetime* is the (small constant) time to falsify a potential invariant, |*trueinvs*| is the (small) number of true invariants at a program point, |*testsuite*| is the size of the test suite, and |*program*| is the number of instrumented program points. The first two products multiply a number of invariants by the time to test each invariant.

The rest of this section fleshes out the intuition sketched above and justifies it via experiments. Section 6.1.1 describes the experimental methodology. Section 6.1.2 reports how the number of variables in scope at an instrumented program point affects invariant detection time, and Section 6.1.3 reports how the number of test cases (program runs) affects invariant detection time. Section 6.1.4 considers how other factors affect invariant detection time.

6.1.1 Methodology

We instrumented and ran the Siemens `replace` program on subsets of the 5542 test cases supplied with the program, including runs over 500, 1000, 1500, 2000, 2500, and 3000 randomly-chosen test inputs, where each set is a subset of the next larger one. We also ran over all 5542 test cases, but our initial prototype implementation ran out of memory, exceeding 180MB, for one program point over 3500 inputs and for a second program point over 4500 inputs. The implementation could reduce space costs substantially by using a different data representation or by not storing every tuple of values (including every distinct string and array value) encountered by the program. For instance, the system might only retain certain witnesses and counterexamples, for use by the query tool, to checked properties. The witnesses and counterexamples help to explicate the results when a user asks whether a certain property is satisfied in the trace database, as described in Section 2.2.2.

Daikon infers invariants over an average of 71 variables (6 original, 65 derived; 52 scalars, 19 sequences) per instrumentation point in `replace`. (The `replace` program has 21 procedures and so 42 instrumentation points, but one of the routines, which handles errors, was never invoked, so we omit it henceforth.) On average, 1000 test cases produce 10,120 samples per instrumentation point, and Daikon takes 220 seconds to infer the invariants for an average instrumentation point; for 3000 test cases there are 33,801 samples and processing takes 540 seconds.

We ran the experiments on a 450MHz Pentium II, using a version of Daikon written in the interpreted language Python [van97]. Daikon has not yet been seriously optimized for time or space, although at one point we improved performance by nearly a factor of ten by inlining two one-line procedures. In addition to local optimizations and algorithmic improvements, use of a compiled language such as C could improve performance by another order of magnitude or more.

6.1.2 Number of instrumented variables

The number of variables over which invariants are checked is the most important factor affecting invariant detection runtime. This is the number of variables in scope at a program point, not the total number of variables in the program, so it is generally small and should

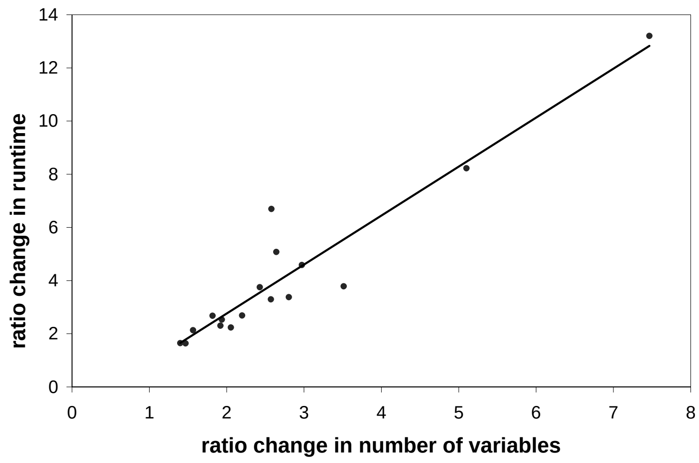


Figure 6.1: Change in invariant detection runtime versus change in number of variables. A least-squares trend line highlights the relationship; its R^2 value is over .89, indicating good fit. Each data point compares inference over two different sets of variables at a single instrumentation point, for invariant inference over 1000 program runs. (For 3000 test cases, the graph is similar, also with $R^2 = .89$.) If one run has v_1 variables and a runtime of t_1, and the other has v_2 variables and a runtime of t_2, then the x axis measures $\frac{v_2}{v_1}$ and the y axis measures $\frac{t_2}{t_1}$. The trendline equation is $y = 1.8x - .92$, indicating that doubling the number of variables tends to increase runtime by a factor of 2.5, while increasing the number of variables fivefold increases runtime by eight times.

grow very slowly with program size, as more global variables are introduced. On average, each of the 20 functions in `replace` has 3 parameters (2 pointers and 1 scalar), but those translate to 5 checked variables because, for arrays and other pointers, the address and the contents are separately presented to the invariant detector. On average there are two local variables (including the return value, if any) in scope at the procedure exit; `replace` uses no global variables. The number of derived variables is difficult to predict because it depends on the values of other variables, as described in Section 4.3. On average, about ten variables are derived for each original one; this number holds for a wide variety of relative numbers of scalars and arrays. In all of our statistics, the number of scalars or of sequences has no more (sometimes less) predictive power than the total number of variables.

Figure 6.1 plots growth in invariant detection time against growth in number of variables. Each data point compares invariant detection times for two sets of variables at every procedure exit in `replace` using a 1000-element test suite. One set of variables is the initial argument values, while the other set adds final argument values, local variables, and the return value. The larger set was from 1.4 to 7.5 times as large as the smaller one; this is the range of the x axis of Figure 6.1. The absolute number of variables ranges from 14 to 230. This choice of a variable sets for comparison is somewhat arbitrary; however, it can be applied consistently to all the program points, it produces a range of ratios of sizes for the two sets, and the results are repeatable for multiple test suite sizes. We used the same test suite for each run, and we did not compare inference times at different program points, because different program points are executed different numbers of times (have different sample sizes), generate different numbers of distinct values (have different value distribu-

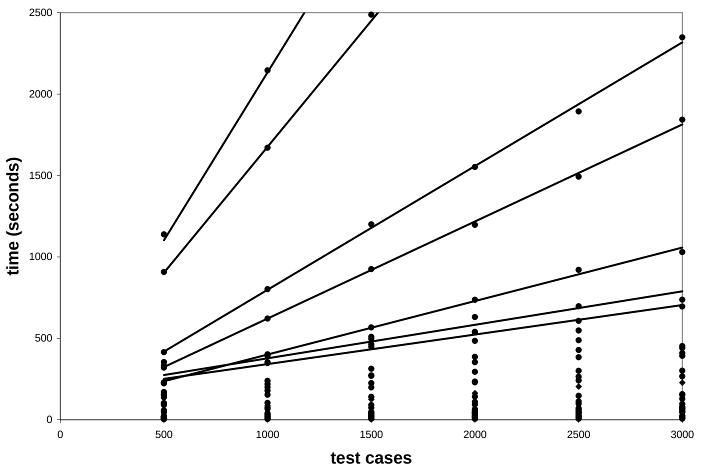


Figure 6.2: Invariant detection runtime vs. number of test cases (program runs). The plot contains one data point for each program point and test suite size — six data points per program point. Lines are drawn through some of these data sets to highlight the growth of runtime as test suite size increases. Figure 6.3 plots the slopes of these lines.

tions), and induce different invariants; our goal is to measure only the effect of number of variables.

Figure 6.1 indicates that, for a version of Daikon without ternary invariants (which had not yet been introduced when this experiment was run), invariant detection time grows approximately quadratically with the number of variables over which invariants are checked. (This is implied by the linear relationship over the ratios. When ratios $v_r = \frac{v_2}{v_1}$ and $t_r = \frac{t_2}{t_1}$ are linearly related with slope s, then $v_r = st_r - s + 1$ because $t_r = 1$ when $v_r = 1$, and thus $v \propto t^s$. For the 1000 test cases of Figure 6.1, the slope is 1.8, so $v \propto t^{1.8}$.) The quadratic growth is explained by the fact that the number of possible binary invariants (relationships over two variables) is also quadratic in the number of variables at a program point.

To verify our results, we repeated the experiment with a test suite of 3000 inputs. The results were nearly identical to those for 1000 test cases: the ratios closely fitted ($R^2 = .89$) a straight line with slope 2.1.

Figure 6.1 contains only 17 data points, not all 20. Our timing-related graphs omit three functions whose invariant detection runtimes were under one second, since runtime or measurement variations could produce inaccurate results. The other absolute runtimes range from 4.5 to 2100 seconds.

6.1.3 Test suite size

The effect of test suite size on invariant detection runtime is less pronounced than the effect of number of variables. Figure 6.2 plots growth in time against growth in number of test cases (program runs) for each program point. Most of these relationships are strongly linear: nine have R^2 above .99, nine others have R^2 above .9, and five more have R^2 above .85. The remaining twelve relationships have runtime anomalies of varying severity; the data points largely fall on a line, usually with a single exception. Although the timings are reproducible,

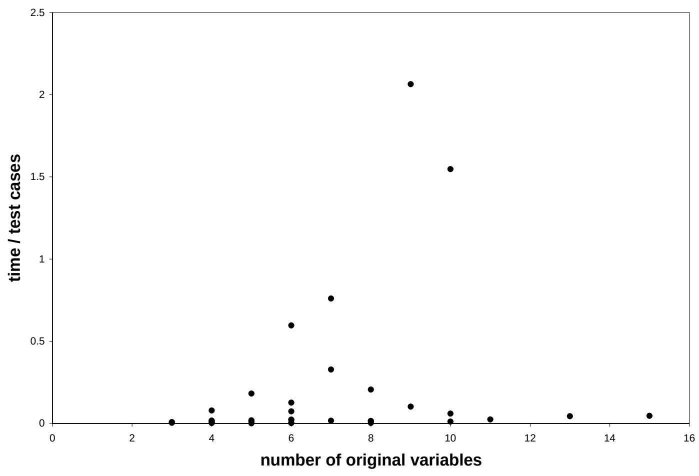


Figure 6.3: Growth of runtime with test suite size, plotted against number of source variables at a program point. There is no correlation between these values. The y axis plots slopes of lines in Figure 6.2.

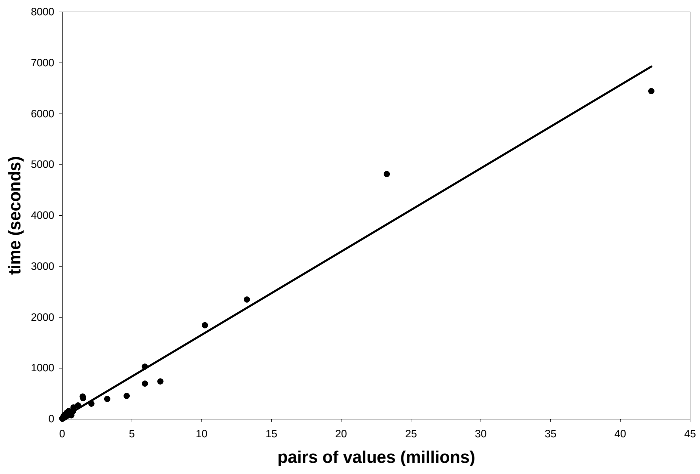


Figure 6.4: Number of pairs of values is the best predictor of invariant detection runtime ($R^2 = .94$). The number of pairs of values is the number of distinct $\langle x, y \rangle$ pairs, where x and y are the values of two different variables in a single sample (one particular execution of a program point). The number of pairs of variables is not predictable from (though correlated with) number of test inputs and number of variables.

we have not isolated a cause for these departures from linearity.

Although runtime is (for the most part) linearly related to test suite size, the divergent lines of Figure 6.2 show that the slopes of these relationships vary considerably. These slopes are not correlated with the number of original variables (the variables in scope at the program point), total (original and derived) variables, variables of scalar or sequence type, or any other measure we tested. Therefore, we know of no way to predict the slopes or the growth of runtime with test suite size.

6.1.4 Other factors

We compared a large number of factors in an attempt to find formulas relating them.

The best single predictor for invariant detection runtime is the number of pairs of values encountered by the invariant detector; Figure 6.4 plots that linear relationship. Runtime is also correlated with total number of values, with number of values per variable, with total number of samples, and with test suite size (as demonstrated above), but in none of those cases is the fit as good as with number of pairs of values, and it is never good enough for prediction. Runtime was not well-correlated with any other factors (or products or sums of factors) that we tried.

Although the number of pairs of values is a good predictor for runtime and is correlated with the number of values (but not with the ratio of numbers of scalar and sequence variables), it cannot itself be predicted from any other factors.

Unsurprisingly, the number of samples (number of times a particular program point is executed) is linearly related to test suite size (number of program runs). The number of distinct values is also well-correlated with the number of samples. The number of distinct variable values at each instrumentation point also follows an almost perfectly linear relationship to these measures, with about one new value per 20 samples. We expected fewer new values to appear in later runs. However, repeated array values are rare, and even a test suite of 50 inputs produced 600 samples per function on average, perhaps avoiding the high distinct-variable-values-per-sample ratio we expected with few inputs.

6.2 Invariant stability

A key question in invariant inference is what kind and how large a test suite is required to get a reliable, useful set of invariants. Too few test cases can result in both a small number of invariants, because confidence levels are too low, and more false invariants, because falsifying test cases were omitted. Running many test cases, however, increases inference times linearly, as demonstrated in Section 6.1.3.

To explore what test suite size is desirable for invariant inference, we compared the invariants detected on `replace` for different numbers of randomly selected test cases. Figures 6.5 and 6.6 chart the number of identical, missing, and different invariants reported between two test suites, where the smaller test suite is a subset of the larger. Missing invariants are invariants that were reported in one of the test suites but not in the other. Daikon always detects all invariants that hold over a test suite and are in its vocabulary: all invariants of the forms listed in Section 3.2, over program variables, fields, and derived variables of the forms listed in Section 4.3. Any invariant that holds over a test suite also holds over a subset of that test suite. However, a detected invariant may not be reported if it is not statistically justified (Section 4.5), if a stronger invariant that implies it was reported (Section 4.6), or if its variables are statically determined to be unrelated (Section 4.7). All comparisons of invariants in this dissertation are of reported invariants, which is the output the user sees.

Figures 6.5 and 6.6 separate the differences into potentially interesting ones and probably uninteresting ones. A difference between two invariants is considered uninteresting if it is a difference in a bound on a variable's range or if both invariants indicate a different small set

	Number of test cases			
	500	1000	1500	2000
Identical unary	2129	2419	2553	2612
Missing unary	125	47	27	14
Differing unary	442	230	117	73
interesting	57	18	10	8
uninteresting	385	212	107	65
Identical binary	5296	9102	12515	14089
Missing binary	4089	1921	1206	732
Differing binary	109	45	24	19
interesting	22	21	15	13
uninteresting	87	24	9	6

Figure 6.5: Invariant similarities and differences versus 2500 test cases for the Siemens `replace` program. The chart compares invariants computed over a 2500-element test suite with invariants computed over smaller test suites that were subsets of the 2500-element test suite.

of possible values (called "small value set" in Section 3.2); all other differences are classified as potentially interesting.

Some typical uninteresting invariant range differences are the following differences in invariants at the exit of function `putsub` when comparing a test suite of size 1000 to one of size 3000:

```
1000 tests:     s1 >= 0              (96 values)
3000 tests:     s1 in [0..98]        (99 values)

1000 tests:     i in [0..92]         (73 values)
3000 tests:     i in [0..99]         (76 values)
```

A difference in a bound for a variable is more likely to be a peculiarity of the data than a significant difference that will change a programmer's conception of the program's operation. In particular, that is the case for these variables, which are indices into arrays of length 100. The uninteresting category also contains variables taking on too few values to infer a more general invariant, but for which that set of values differs from one set of runs to another.

All other differences are reported in Figures 6.5 and 6.6 as potentially interesting. For example, when comparing a test suite of size 2000 to one of size 3000, the following difference is reported at the exit of `dodash`:

```
1000 tests:     *j >= 2              (105 values)
3000 tests:     *j = 0 (mod 2)       (117 values)
```

Such differences, and some missing invariants, may merit closer examination.

Examination of the output revealed that substantive differences in invariants, such as detecting result = i in one case but not another, are rare — far fewer than one per procedure on average. Most of the invariants discovered in one procedure but not in another were

	Number of test cases				
	500	1000	1500	2000	2500
Identical unary	2101	2254	2293	2314	2310
Missing unary	96	110	114	112	106
Differing unary	506	338	295	274	284
interesting	88	58	57	54	52
uninteresting	418	280	238	220	232
Identical binary	4881	6466	6835	6861	6837
Missing binary	3805	2833	2827	2933	2831
Differing binary	82	129	135	131	130
interesting	24	27	21	16	29
uninteresting	58	102	114	115	101

Figure 6.6: Invariant similarities and differences versus 3000 test cases for the Siemens `replace` program. The chart compares invariants computed over a 3000-element test suite with invariants computed over smaller test suites that were subsets of the 3000-element test suite.

between clearly incomparable or unrelated quantities (such as a comparison between an integer and an address, or between two elements of an array or of different arrays) or were artifacts of the particular test cases (such as adding $*\mathsf{i} \neq 5 \pmod{13}$ to $*\mathsf{i} \geq 0$). Other invariant differences result from different values for pointers and uninitialized array elements. For example, the minimum value found in an array might be -128 in one set of runs and -120 in another, even though the array should contain only (nonnegative) characters. Other nonsensical values, such as the sum of the elements of a string, also appeared frequently in differing invariants. The relevance enhancements of Chapter 4 had not yet been implemented when this experiment was performed.

6.2.1 Quantitative analysis

In Figures 6.5 and 6.6, the number of identical unary invariants grows modestly as the smaller test suite size increases. Identical binary invariants show a greater increase, particularly in the jump from 500 to 1000 test cases. Especially in comparisons with the 3000 case test suite, there are some indications that the number of identical invariants is stabilizing, which might indicate asymptotically approaching the true set of invariants for a program.

Inversely, the number of differing invariants is reduced as the smaller test suite size increases. Both unary and binary differing invariants drop off most sharply from 500 to 1000 test cases; differences with the 3000 case test set then smooth out significantly, perhaps stabilizing, while differences with the 2500 case test set drop rapidly. Missing invariants follow a similar pattern. The dropoff for unary invariants is largely due to fewer uninteresting invariants, while the dropoff for binary invariants is due to fewer interesting invariants.

For `replace` and randomly selected test suites, there seems to be a knee somewhere between 500 and 1000 test cases: that is, the benefit per randomly-selected test case seems greatest in that range. Such a result, if empirically validated, could reduce the cost of selecting test cases, producing execution traces, and computing invariants.

Figures 6.5 and 6.6 paint somewhat different pictures of invariant differences. Differences are smaller in comparisons with the 2500-element test suite, while values tend to level off in comparisons with the 3000-element test suite. Only 2.5% of binary invariants detected for the 2000 or 2500 case test suites are not found identically in the other, and the number of invariants that differ is in the noise, though these are likely to be the most important differences. For comparisons against the 2500 test case suite, these numbers drop rapidly as the two test suites approach the same size. When the larger test suite has size 3000, more invariants are different or missing, and these numbers stabilize quickly. The 3000 case test suite appears to be anomalous: comparisons with other sizes show more similarity with the numbers and patterns reported for the 2500 case test suite. We did such comparisons for both smaller test suites and larger ones (the larger comparisons omitted the one or two functions for which our invariant database ran out of room for such large numbers of samples). Our preliminary investigations have not revealed a precise cause for the larger differences between the 3000 case test suite and all the others, nor can we accurately predict the sizes of invariant differences; further investigation will be required in order to fully understand these phenomena.

6.3 Automatically generated test suites

We have not yet characterized the properties of a test suite (besides size) that make it appropriate for dynamic invariant detection. Furthermore, it is desirable for test suite construction to be affordable. This section reports the quality of invariants resulting from test suites generated by two semi-automatic, relatively inexpensive methods: simple random test-case generation (Section 6.3.1) and grammar-driven test-case generation (Section 6.3.2).

For Siemens programs `replace`, `schedule`, and `tcas` (see Section 2.4), we compare invariants resulting from automatically generated test suites and (a random selection of) the hand-crafted test cases.

6.3.1 Randomly-generated test suites

The simplest method of generating test cases is to randomly generate inputs of the proper types. Random testing is cheap, but it has poor coverage and is most effective at finding highly peculiar bugs [Ham94].

Our randomly generated test suites failed to execute many portions of the program. Thus, Daikon did not produce many of the invariants resulting from the hand-crafted input cases. For example, random generation produces few valid input pattern strings for the `replace` program, so the functions that read and construct the pattern were rarely reached.

As another, more extreme, example, only one randomly produced input exercised the set of functions in the `tcas` program, so Daikon was unable to produce useful invariants. The `tcas` program begins by using a complex conditional to determine if its use is applicable to the current situation, as described by the 12 inputs. If the input values are not appropriate, it immediately exits.

For functions that were entered, the random test cases produced many invariants identical to the ones derived from the Siemens test cases and few additional ones. For example,

	Unary		Binary	
Program	identical	differing	identical	differing
`schedule`	1876	54	100	4
`tcas`	235	116	58	62
`replace`	437	391	1130	928

Figure 6.7: Number of identical and differing invariants between invariants produced from grammar-driven test cases and from the Siemens test cases for each of the 3 Siemens programs. Each test suite contained 100 test cases.

`schedule`'s function `init_prio_queue` adds processes to the active process queue. Daikon correctly produced the invariant $\mathsf{i} = \mathsf{num\_proc}$ at the end of its loop. Many of the discovered invariants were related to program behaviors that are largely independent of the procedure's actual parameters.

Random test cases did reveal how the program behaves with invalid inputs. For example, `tcas` performs no bounds checks on a statically declared fixed-sized array. When an index taken from the input was out of bounds, the resulting invariants showed the use of garbage values in determining the aircraft's collision avoidance response.

6.3.2 Grammar-generated test suites

Randomly generating test cases from a grammar that describes valid inputs holds more promise than fully random testing because it can ensure a large number of correct inputs, and biasing the grammar choices can produce more representative test cases. Compared to random test generation, the grammar-driven approach produced invariants much closer to those achieved with the Siemens test cases, but they also required more effort to produce.

The three programs had no specifications, so we derived grammars describing valid program inputs by looking at the source or at comments, when available. In general this was straightforward, although in some cases where input combinations could not occur together, we added explicit constraints to the generator. In the case of `replace`, we enhanced the generator to occasionally insert instances of the produced pattern in the target string, ensuring that substitution functions are exercised.

We also arranged for the grammars to produce some invalid inputs. In some cases introducing errors simplified the grammars; for example, permitting any character to fill a pattern format in `replace`'s test generation grammar, even when the pattern language prohibits regular expression metacharacters.

Table 6.7 compares the invariants produced from the grammar-driven test cases to invariants produced from the Siemens suite for each of the 3 programs, using 100 test cases. The grammar-driven test cases produced many of the invariants found with the Siemens test cases. Many of the differing invariants do not appear to be relevant (an inherently subjective assessment). In `replace`, many differing invariants resulted from the larger range of characters produced by the generator, compared to those of the Siemens test cases. Many other differing invariants are artifacts of erroneous or invalid input combinations produced by either the generated or Siemens test cases. However, some of the differences are significant, resulting from input combinations that the grammar-based generation method did not produce.

Although more investigation is required, there is some evidence that with reasonable effort in generating test cases we can derive useful invariants. In particular, grammar-driven test-case generators may be able to produce invariants roughly equivalent to those produced by a test suite designed for testing. A programmer need not build a perfect grammar-driven test-case generator, but rather one that exercises the program trace points of interest a sufficient number of times. The detected invariants indicate shortcomings of the test suite. Random selection of values within the constraints of the grammar is acceptable, even beneficial, for invariant inference. Furthermore, an imperfect grammar can help exercise error conditions that are needed to fully understand program behavior.

Chapter 7
Implementation

This chapter discusses details of the implementation of the Daikon invariant detector. Section 7.1 presents the design goals that led to its design, and Section 7.2 illustrates the format of the invariant detector inputs. Sections 7.3–7.5 discuss program instrumentation and the Daikon front ends for C and Java. The remainder of the chapter gives details about the invariant detector proper, including statistics about the implementation (Section 7.6), its data structures (Section 7.7), and how users can extend it (Section 7.8).

The Daikon implementation is available for download from `http://www.cs.washington.edu/homes/mernst/daikon`.

7.1 Design goals

Figure 7.1 diagrams Daikon's high-level architecture:

- A front end instruments the program so that in addition to performing its original computation, it also outputs, to a data trace file, the values of variables (and their fields) at particular program points. The decision about what fields to output is made statically, based on what fields the object is known to contain. The instrumenter also writes a separate declaration file (not shown in Figure 7.1) that describes the format of the data trace file.

- The user runs the instrumented program on a test suite. Each run of the program produces a data trace file.

- Daikon postulates potential invariants and checks those against the data traces. Daikon accepts an arbitrary number of trace files (and declaration files) as input, permitting aggregation of multiple program runs and production of a single set of invariants (which are generally superior to those from any single run).

- Daikon filters out invariants that are likely to be unhelpful to the programmer; see Chapter 4. The Daikon implementation interleaves filtering and invariant checking. Daikon also makes decisions about what derived variables to introduce (and so what invariants are possible) at invariant detection time.

The Daikon implementation is designed to be simple, robust, and language-independent. In order to achieve these goals, it sacrifices some efficiency and flexibility.

The data trace file must exactly conform to the format specified in the declaration file; variables may not be missing or appear out of order. Additionally, variables are written to

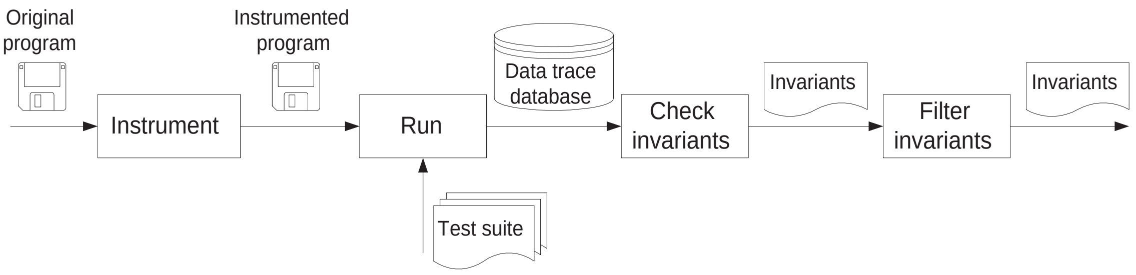


Figure 7.1: Expanded architecture of the Daikon tool for dynamic invariants detection (refined from Figure 1.1 on page 2). Although filtering can be conceptually viewed as postprocessing, for efficiency Daikon performs some filtering before, some during, and some after checking invariants.

the data trace file as integers, strings, or arrays thereof; no other types are supported. These decisions simplify the invariant detector (it need not special-case or interpret other types, or traverse data structures) and enable comprehensive error-checking of its input files. They also separate the instrumenter from the implementation of invariant detection, making it easy to produce multiple front ends and to debug the front ends and the invariant detector.

The format is somewhat constraining, however. Object and pointer values must be coerced to integer object IDs or addresses; this permits equality testing but prohibits other types of inference on the actual run-time values, particularly when the exact run-time type is not known *a priori* or is stricter than the declared type. Additionally, every variable that can ever appear in the data trace file must always appear in it, leading to inefficiencies when a variable is null and the values of its slots are not meaningful. (Such values are marked as "missing" in the data trace file.) Work to present data structures whole to the invariant detector is underway — see Section 9.2, page 104.

7.2 Data file format

For every instrumented program point, a trace file contains a list of tuples of values, one value per instrumented variable. (We refer to such a tuple as a sample.) For instance, suppose procedure `p` has two formal parameters, is in the scope of three global variables, and is called twelve times. When computing a precondition for `p` (that is, when computing an invariant at `p`'s entry point), the invariant detector would be presented a list of twelve samples, each sample being a tuple of five variable values (one for each visible variable). Daikon's instrumenters also output a modification bit for each value that indicates whether the variable has been set since the last time this program point was encountered. This permits Daikon to ignore garbage values in uninitialized variables and to prevent unchanged values encountered multiple times from over-contributing to invariant confidence (see Section 4.5.1, page 39 for details). Figure 7.2 shows an excerpt from a data trace file.

The instrumenter also creates, at instrumentation time, a declaration file describing the format of the data trace file. The declaration file lists, for each instrumented program point, the variables being instrumented, their types in the original program, their representations in the trace file, and the sets of variables that may be sensibly compared (see Section 4.7,

```
15.1.1:::ENTER
B = 92 56 -96 -49 76 92 -3 -88, modified
N = 8, modified

15.1.1:::LOOP
B = 92 56 -96 -49 76 92 -3 -88, modified
N = 8, modified
I = 0, modified
S = 0, modified

15.1.1:::LOOP
B = 92 56 -96 -49 76 92 -3 -88, unmodified
N = 8, unmodified
I = 1, modified
S = 92, modified
```

Figure 7.2: Data trace file example. These are the first three records in the data trace file for the Gries array sum program of Figures 2.1 and 2.2. The invariants of Figure 2.3 were derived from this trace file. `B` is an array of integers, and the other variables are integers. These records give variable values at the program entry and at the start of the first two loop iterations. The complete data trace file contains 1307 records: 100 for the program entry, 100 for the exit, and 1107 for the loop head.

page 46). Figure 7.3 shows part of a declaration file.

7.3 Program instrumentation

Daikon's input is a sequence of variable value tuples for every program point of interest to the programmer. Instrumentation inserted at the program points captures this information by writing out variable values each time the program point is executed. Daikon includes fully automatic instrumenters for C, Java, and Lisp.

The instrumenter's primary decisions are what program points to instrument and which variables to examine at those points. Presently the program points are procedure entries and exits. (The Lisp instrumenter can instrument loop heads, but that feature was added primarily to enable the experiment of Section 2.1, which recovered loop (and other) invariants from formally specified programs. Loop instrumentation will be added to the other front ends if a compelling application, such as generating loop invariants for program proving, emerges.) Module entries and exits are treated somewhat specially in order to detect object and class invariants: the instrumenter writes out data for a synthetic program point that takes values from the entries and exits of public routines (see Section 3.3). The current instrumenters write to the data trace all variables in scope, including global variables, procedure arguments, local variables, and (at procedure exits) the return value. Additionally, object fields (up to a user-specified depth) and other information such as cyclicity of data structures may appear in the data trace. For instance, a record `r` is output as an address (or object ID) and also gives rise to trace variables with the natural names `r.`*field1*, `r.`*field2*, etc., where *field1* and *field2* are the names of `r`'s fields.

```
DECLARE
P180-15.1.1:::ENTER              program point name
B                                variable name
int[]                              declared type
int[]                              representation type
(S B-element)[(B-index1 I N)]      comparable to
N                                variable name
int                                declared type
int                                representation type
(B-index1 I N)                     comparable to

DECLARE
P180-15.1.1:::LOOP-389           program point name
B                                variable name
int[]                              declared type
int[]                              representation type
(S B-element)[(B-index1 I N)]      comparable to
N                                variable name
int                                declared type
int                                representation type
(B-index1 I N)                     comparable to
I                                variable name
int                                declared type
int                                representation type
(B-index1 I N)                     comparable to
S                                variable name
int                                declared type
int                                representation type
(S B-element)                      comparable to
```

Figure 7.3: Declarations for the entry and loop head for the Gries array sum program of Figures 2.1 and 2.2. These declarations describe the format of the trace file of Figure 7.2. The italic text is not part of the text file, but explains the purpose of each line of the file.
For each program point, the trace file gives four pieces of information for each variable: its name, its declared type in the source code, its representation type in the data trace file, and a list of other variables to which it is comparable. In this example, the declared and representation types are always the same, but they differ when the variable's declared type is not representable as an integer, string, or sequence thereof. Variable `B-element` stands for the contents of the `B` array; variable `B-index1` stands for indices to the first (and only) dimension of array `B`; and the notation `(A B)[(I J K)]` indicates that the array is comparable to arrays `A` and `B` and that its index is comparable to integers `I`, `J`, and `K`.

An array of structures is converted into a set of parallel arrays (one for each structure slot, appropriately named to make their origin clear).

The Daikon front ends operate by source-to-source translation. The instrumenter parses the program source into an abstract syntax tree (AST), determines which variables are in scope at each instrumentable program point, inserts code at the program point to dump the variable values into an output file, inserts bookkeeping operations at various points in the program, and unparses the AST to a file as source code, which can be compiled and run in the standard way. Adding instrumentation to a program is fast (much faster than compilation). Object code instrumentation or binary rewriting [SE94, RVL+97] could permit improved precision (for instance, in determining exactly which memory locations have been accessed or hooking into the exact point where a variable is modified) and allow instrumentation of arbitrary binaries. However, most uses of invariant inference make most sense when a program is being modified, which requires access to the program source anyway. Source rewriting is simpler: for instance, it need not detect compiler-specific optimizations that reorder, rearrange, or inline code. Standard debugging tools can be used on instrumented source code without any special effort to maintain symbol tables, debugging source is easier and more portable than doing so for assembly, and instrumented source code is platform-independent.

For the relatively small, compute-bound programs we have examined so far, the instrumented code can be slowed down by an order of magnitude (or more, in some cases) because the programs become I/O-bound. We have not yet optimized trace file size or writing time. Another approach would be to perform invariant checking online rather than writing variable values to a file (Section 9.2).

We have implemented instrumenters for C, Lisp, and Java. Sections 7.4 and 7.5 discuss the C and Java front ends. The C front end predates the Java one and lacks some if its features; however, C is also a more difficult language to instrument. The Lisp instrumenter is conceptually similar to the C and Java front ends.

7.4 C instrumenter

Instrumenting C programs to output variable values requires care because of uninitialized variables, side effects in called procedures, uncertainty whether a pointer is a reference to an array or to a scalar, partially uninitialized arrays, and sequences not encoded as arrays. The Daikon front end for C, which is based on the EDG C front end [EDG95], manages these problems in part by maintaining runtime status information on each variable in the program and in part with simplifying assumptions. It does not track the status of C `unions`.

The instrumented program adds, for each variable in the original program, an associated status object whose scope is the same as that of the variable (for pointers, the `malloc` and `free` functions are instrumented to create and destroy status objects). The status object contains a modification timestamp, the smallest and largest indices used so far (for arrays and pointers into arrays), and whether a linked list can be made from the object (for structures; this is true if one of the slots has the same type as (or is a pointer to) the whole structure). When the program manipulates a variable, its status object may also be updated. For instance, an assignment copies status information from the source to the destination.

In order to provide procedures with accurate information about their parameters and to track modifications in called procedures, a variable and its status object are passed to (or returned from) a function together. If a variable is passed by reference, so is its status object; if a variable is passed by value, so its status; and if a function argument is not an lvalue (that is, if the argument is a literal, function call, or other non-assignable expression), then a dummy status object is created and passed by value. For instance, the function declaration and use

```
ele* get_nth_element(list* a_list, int n) { ... }

my_ele = get_nth_element(my_list, 4);
```

would be instrumented as

```
ele* get_nth_element(list* a_list, var_status *a_list_status,
                     int n, var_status n_status,
                     var_status *retval_status) { ... }

my_ele = get_nth_element(my_list, my_status, 4, dummy_status(), my_ele_status);
```

7.4.1 Tracking variable updates

The modification timestamp in a variable's status object not only prevents the writing of garbage values to the data trace file (an "uninitialized" annotation is written instead) but also prevents the instrumenter from dereferencing an uninitialized pointer, which could cause a segmentation fault. Daikon's problem is more severe than that faced by other tracers such as Purify [HJ92], which only examine memory locations that are referenced by the program itself. Code instrumented by Daikon examines and potentially dereferences all variables visible at a program point.

The modification timestamp is initially set to "uninitialized", then is updated whenever the variable is assigned. For instance, the statement `p = foo(j++);` becomes, in the instrumented version,

```
record_modification(&p_var_status),
record_modification(&j_var_status),
p = foo(j++, j_var_status);
```

The comma operator in C (used in the first two lines; the comma in the third line separates function arguments) sequentially evaluates its two operands, which allows the instrumented program to perform side effects in an arbitrary expression without introducing new statements that could affect the program's abstract syntax tree and complicate the source-to-source translator.

7.4.2 Pointers

C uses the same type, `T*`, for a pointer to a single object of type `T` and for (a pointer to) an array of elements of type `T`. An incorrect assumption about the referent of an arbitrary element of type `T*` can result in either loss of information (by outputting only a single element when the referent is actually an array) or in meaningless values or a program crash (by outputting an entire block of memory, interpreted as an array, when the referent is actually a single object). The Daikon front end for C discriminates the two situations with a simple static analysis of the program source. Any variable that is the base of an array indexing operation, such as a in a[i], is marked as an array rather than a scalar.

Even if a variable is known to point into an array, the size of that array is not available from the C runtime system. More seriously, many C programs allocate arrays larger than they need and use only a portion of them. Unused sections of arrays present the same problems to instrumentation as do uninitialized variables. To determine the valid portion of an array, a variable status object contains the smallest and largest integers used to index an array. This information is updated at each array index operation; for instance, the expression a[j] is translated to i[record_array_index(i_var_status, j)], where function record_array_index returns its second argument (an index) as well as updating its first argument (a variable status) by side effect. The minimal and maximal indices are used when writing arrays to the data trace file in order to avoid walking off the end (or the valid portion) of an array. Although this approach is not sound (for instance, it works well while an array-based implementation of a stack is growing, but irrelevant data can be output if the stack then shrinks), it has worked in practice. It always prevents running off the end of an array, because assigning to the array variable updates the variable status. For character arrays, the instrumenter assumes that the valid data is terminated by the null character '\0'. Although not universally true, this seems to work well in practice. (The programs we tested, and many but not all programs in practice, do not use character buffers which have explicit lengths rather than being null-terminated.)

When a structure contains a slot whose type is a pointer to the structure type, that structure can be used as a link — the building block for linked lists. Daikon cannot directly reason about such lists because of its limited internal data formats. The C instrumenter works around this limitation by constructing and outputting a sequence consisting of the elements reachable through that pointer (see Section 5.2).

7.5 Java instrumenter

The Daikon front end for Java is similar to that for C. The Java front end need not be concerned with determining array sizes, and its modification bit computation is more sophisticated. It is also more robust and complete.

The Java instrumenter rewrites the program to allocate extra space within objects, so that the timestamps are contained within the objects themselves. It maintains a timestamp per lvalue, which is updated when the lvalue is assigned to. It also maintains a timestamp per ⟨lvalue, program point⟩ pair. When execution reaches the program point, the lvalue is written to the trace file, along with a modification bit. The modification bit is true if the lvalue's timestamp is more recent than that of the ⟨lvalue, program point⟩ (in which case the latter timestamp is updated from the former); otherwise, the modification bit is false. Thus, the modification bit captures whether the lvalue was modified since the last time the program point was executed.

The instrumenter also introduces a number of additional functions into the class that assist with timestamp maintenance, printing data structures to a specified depth, and printing recursive data structures.

7.5.1 Implementation approach

The Daikon front end for Java is an extended version of Jikes [Jik], an open-source Java compiler written in C++ and originally from IBM Research. The extensions permit modifying the abstract syntax tree (AST) representation of the program and unparsing the AST into Java source. Jikes's parser and semantic analysis are used as is. The instrumenter operates as a source-to-source translator, rather than producing Java Virtual Machine (JVM) [MD97, LY99] bytecodes, for two reasons. First, the AST modifications violate Jikes's undocumented internal representation invariants in a way that does not hinder unparsing. However, it would be difficult to restore the invariants or to relieve the bytecode generator's dependence on them. Second, it is much easier to locate problems by examining the unparsed Java code than by debugging class files. The extensions total about 9000 lines of code, part of which is now included in the Jikes distribution (`http://oss.software.ibm.com/developerworks/opensource/jikes`), making it an attractive platform for Java analysis and transformation.

We had implemented two previous Java instrumenters using byte code rewriting (using first an academic project, JOIE [CCK98], and then a proprietary in-house commercial tool, IBM's Bobby), which can trivially determine lvalue modification points and which works on arbitrary class files for which source is not available. However, this approach was inadequate for technical and interface reasons. The technical reasons had to do with the immaturity of the technology of the time. The bytecode rewriters provided relatively low-level interfaces (though we implemented extensions to both byte code rewriters, permitting insertion of Java source code rather than individual byte codes), did little error checking (necessitating frequent hand-disassembly of erroneous class files), and interacted poorly with Java virtual machines (sometimes causing inexplicable JVM failures even when the rewriters' own checks succeeded).

The interface reasons have to do with the intended use of the resulting class files, which is to provide feedback to users. Reconstructing loop heads and live variables at those points from the class file required heuristics for each compiler's optimizations. Additionally, the bytecode rewriters erased debugging information from the class file, making the resulting files difficult to debug. (This shortcoming has since been corrected in Bobby.) Since typical Daikon users will have source code, it is reasonable to operate directly on it.

7.5.2 Timestamp maintenance

Conceptually, lvalue timestamps are updated whenever the lvalue is modified; in practice, the timestamp can be set just after the lvalue assignment, so long as no operations can observe the lvalue until the timestamp has been updated. As noted above, this is easy to arrange via bytecode or object instrumentation, but is a challenge for the Daikon front ends, which are source-to-source translators.

The Daikon front end for Java uses four different strategies for rewriting assignments in order to achieve the correct semantics.

Idempotent lhs timestamp setter. When the left-hand-side is idempotent and independent of the right-hand-side — that is, the left-hand side has no side effects and is guaranteed to give the same result no matter how often evaluated, either before or after the right-hand side is evaluated — then the left-hand side is replaced by a call to a method which sets the timestamp and returns the object itself. For instance,

```
     v = 7  ⇒  set_v_ts(now).v = 7
 lhs.v = 7  ⇒  lhs.set_v_ts(now).v = 7
  v[3] = 7  ⇒  set_v_ts(now).v[3] = 7
```

The instrumenter introduces methods such as `set_v_ts` for each field. This technique works for arbitrary lvalues and assignments (including "+=", "++", etc.), but does not work for static class variables, and the timestamp is set before the right-hand side is evaluated.

Idempotent rhs operation. When the left-hand side and the right-hand side have no side effects (or the left-hand side contains just one identifier, possibly with constant array indices), then an idempotent operation — adding zero to a number, anding true with a boolean, or concatenating `""` to a string) is performed either on the whole expression or on just the right-hand side. For instance,

```
    v = 7  ⇒  (v = 7) + set_v_ts_return_zero(now)
lhs.v = 7  ⇒  (lhs.v = 7) + lhs.set_v_ts_return_zero(now)
```

where method `set_v_ts_ret_zero` sets `v`'s timestamp and returns zero; the front end introduces such methods for each field. This technique only works for numeric, boolean, and String types — but those types cover all assignment operators except the simple assignment operator, which the next technique handles. It requires the left- and right-hand sides of the assignment to have no side effects; that restriction can be relaxed, but at a substantial increase in complexity.

Slot setter method. Ordinary assignments can be replaced by calls to methods which set both the timestamp and the lvalue. For instance,

```
    v = 7  ⇒  set_v(7)
lhs.v = 7  ⇒  lhs.set_v(7)
```

In the case of array element assignments, the call must include the array index (before the value argument, to preserve order of evaluation):

```
     v[3] = 7  ⇒  set_v_at_index(3, 7)
 lhs.v[3] = 7  ⇒  lhs.set_v_at_index(3, 7)
(lhs.v)[3] = 7  ⇒  lhs.set_v_at_index(3, 7)
```

This technique guarantees that the right-hand side is evaluated before the timestamp is set, which happens "at the same time as" the slot itself is set. The instrumented code is also relatively easy to read. However, this technique only works for ordinary assignments, not for operations such as "+=".

Top-level new variables. If none of the above techniques is applicable, Daikon ensures that an assignment (of any variety) occurs at top level in a statement, then introduces a new left-hand-side variable (to avoid multiple evaluation) and inline the above timestamp-setting methods. For instance, if the left-hand side is not a field access, then the following conversion is valid regardless of side effects on the left or right hand sides:

```
                v = rhs;  ⇒  v = rhs; v_timestamp = now;
a[foo = 4] = (foo = 5);  ⇒  a[foo = 4] = (foo = 5); a_timestamp = now;
```

The conversion works for field assignments when the left- and right-hand sides are side-effect-free:

```
lhs.v = rhs;  ⇒  lhs.v = rhs; lhs.v_timestamp = now;
```

In the general case, the conversion is as follows:

```
         lhs.v = rhs;  ⇒  Lhs_type newlhs = lhs;
                          newlhs.v = rhs;
                          newlhs.v_timestamp = now;
((a.b)[2]).c[3] = 7;  ⇒  B_elt_type newlhs = ((a.b)[2]);
                          newlhs.c[3] = 7;
                          newlhs.c_timestamp = now;
```

7.6 Implementation statistics

Two implementations of the Daikon invariant detector have been completed. The first prototype was written in about 6500 lines of Python; later, the system was rewritten from scratch in Java. This new implementation uses a slightly different architecture and takes advantage of Java features such as static typechecking, which catches errors much faster than debugging can. Some of the experiments reported in this dissertation used the initial prototype. Since those experiments were conclusive and the new system outperforms the original in all dimensions, there was no compelling reason to rerun them. Unless otherwise noted, in this dissertation "Daikon" refers to the current Java implementation.

Daikon consists of about 20,000 lines of Java code in about 100 files (600KB of source). These figures do not include front ends (the Daikon distribution includes instrumenters for C, Java, and Lisp, and users report writing others) or about 5000 lines (150KB) of general-purpose utilities written for this project. Additionally, Daikon takes advantage of a number of other tools such as Lackwit [OJ96], Ajax [O'C99a, O'C99b], EDG [EDG95], and Jikes [Jik], as well as libraries for regular expressions, options parsing, and the like.

7.7 Daikon data structures

At its core, the Daikon implementation manages a collection of potential invariants and a stream of samples (tuples of values for the variables in scope at a program point). Each sample is presented to each invariant at that program point. This updates state in the invariant, possibly invalidating it or changing its constants or statistical justification. Invalidated invariants are removed from further consideration.

Daikon's key data structure is the program point, represented by the abstract class `Ppt`. A program point includes `VarInfo` descriptions of the variables in scope, the views on the program point (including `PptSlice` slices which involve a subset of the variables and `PptConditional` conditional program points), and methods to process a new sample of variable values. A `PptSlice` object manages all invariants over specific variables. For instance, a program point with five variables in scope would initially have five unary slices, $\binom{5}{2} = 10$ binary slices, and so forth.

Daikon maintains a collection of `PptTopLevel` objects representing the true program points in the program. Each sample is presented to the appropriate `PptTopLevel` object, which in turn presents it to conditioned programs points and to the slices which deal with specific variables. A `PptSlice` object — say, one which deals with variables `x` and `y` — extracts the `x` and `y` values from the sample and supplies them to the binary invariants which the slice manages.

When an invariant is invalidated, it informs its containing slice, which removes it from its list of invariants. When the last invariant is eliminated from a slice, the slice is likewise removed from its containing program point.

Slices are indexed by variable so that they can be quickly looked up. Certain invariants are also indexed. These indices permit quick determination of whether a specified invariant is valid or not; they are used by the implication tests of Section 4.6. Rather than use a general-purpose theorem-prover, which would have to be able to manage millions of potential invariants, millions of tests over them, and rapid assertion and retraction of properties, the system checks specific implication relationships. Because of the limited number of invariants in the system, given a potential invariant, there are only a few ways that other invariants can imply it. The system looks up each of these potential invariants to determine whether enough of them exist (and are statistically justified) to imply the invariant in question.

7.8 Adding new invariants and derived variables

Users can easily add new invariants and derived variables to those provided by Daikon. Each of these tasks involves writing a single Java class file, then adding one line to a factory class so that it creates and installs the objects appropriately. The entire process takes from 20 minutes for a simple addition to longer if the checking or statistical tests are more sophisticated. Extending an existing invariant is even easier; for example, adding a new unary or binary function is a matter of implementing the function and adding a single line to the appropriate invariant.

Figure 7.4 shows the interface satisfied by an invariant over a single scalar variable. The eight such invariants in the Daikon distribution average 121 lines, inclusive of comments and blanks.

Figure 7.5 shows the interface satisfied by a derived variable dependent upon a single variable. The seven such derived variables in the Daikon distribution average 52 lines, inclusive of comments and blanks.

```
// Constructor; typical implementation is: super(ppt);
private InvariantName(PptSlice ppt);
// Typical implementation is: return new InvariantName(ppt);
public static InvariantName instantiate(PptSlice ppt);

// Add new samples with specified value and the modification bit set.
// count says how many samples have this value.
public void add_modified(int value, int count);

// The probability that the observed data could have happened by chance alone.
// 1 means certainly happened by chance; 0 means could never have happened by chance
protected double computeProbability();

// Printed representations. repr is low-level, format is high-level for user output.
public String repr();
public String format();
```

Figure 7.4: Interface for invariants over a single scalar (integer) variable, such as a lower bound, modulus constraint, or non-zero. Invariants over other types of variables, or over multiple variables, have a slightly different `add_modified` signature. Examples of implemented invariants can be found in the Daikon distribution.

```
// Constructor; typical implementation is: super(vi);
public DerivedVarName(VarInfo vi);

// Given values for other variables, compute derived variable's value and modification bit
public ValueAndModified computeValueAndModified(ValueTuple vt);

// Create a description of the new derived variable
protected VarInfo makeVarInfo();

// Optional; indicates whether this target variable should give rise to a derived variable
public static boolean applicable(VarInfo vi);
```

Figure 7.5: Interface for derived variables over a single variable; for example, `sum(a)`. Variables derived from multiple other sorts of variables (such as `a[i]`) have a slightly different `applicable` signature which takes multiple arguments. Examples of implemented derived variables can be found in the Daikon distribution.

Chapter 8
Related work

This chapter presents other dynamic (Section 8.1) and static (Section 8.2) approaches for determining invariants and considers checking of invariants (Section 8.3). Previously, Section 1.5 (page 4) discussed uses for program invariants.

8.1 Dynamic inference

Dynamic analysis runs a program or uses execution traces in order to infer properties of the program. This section first treats artificial intelligence approaches, noting that invariant detection can be cast as a machine learning problem, listing its domain characteristics that make previous techniques inapplicable, and comparing our approach to that taken by AI researchers. The section then discusses other (non-AI) dynamic analyses related to the problem of invariant detection.

8.1.1 Machine learning

Dynamic invariant detection can be viewed as a machine learning problem to be solved via techniques from artificial intelligence [RN95, Mit97]. The search space is the set of propositions at program points. (Daikon explores that space, but rather than performing a directed search, it checks all properties fitting its grammar (see Sections 3.2 and 4.3), with early results preventing the need to check other potential invariants.) The bias — also known as background knowledge, domain knowledge, feature predicates, or domain theory — is the particular properties that are checked and the heuristics regarding which ones are reported. Daikon generalizes from a training set and, like other learners, may over- or under-generalize, depending on the quality of its inputs.

Although the problem has many similarities with research in machine learning, artificial intelligence, data mining, and statistical and concept discovery, Daikon's solution to the problem differs from the techniques proposed to date in those areas. Those techniques, though often powerful in their own domains, are inadequate for detecting program invariants, largely because of a combination of special characteristics of that problem. No previously described technique handles all of these characteristics, which are described below.

Differences from machine learning

Not classification or clustering. Most machine learning research solves problems of classification or clustering. Classification places examples into one of a set of predefined categories, and the categories require definitions or, more commonly, a training set.

For example, decision tree learning is primarily applicable to classification. Clustering groups similar examples and separates dissimilar ones, under some domain-specific similarity metric. By contrast, invariant detection seeks higher-order relationships and descriptions of the data; these properties do not fit neatly into the traditional machine learning categories. By contrast, the relationships detected by machine learning are typically restricted to functions.

No negative examples. Most concept learning systems, such as inductive logic programming, must be trained on a set of examples marked with correct answers before they can produce useful results. One potential danger is overgeneralization, an extreme example of which is outputting the simplest possible hypothesis — the concept "true," which fits all positive examples. To avoid overgeneralization, learners require counterexamples in the training set (this is the traditional approach), finding the minimal positive generalization of the examples, or adding an inductive bias or domain-specific background knowledge that guides the search. As one example, description logics find least common subsumers (essentially, simplest descriptions) that cover no negative examples, iterating until all positive examples are covered [CH94].

Counterexamples are not available for detecting program invariants: generating conforming counterexamples would require knowledge of the properties to be exhibited, which is the problem to be solved. Only trivial counterexamples that violate a statically detected property are available, and they are of little use [BG93].

A related approach, reinforcement learning, requires experimental control, in which a trainer or the environment rewards or penalizes an agent for each action it takes. An invariant detector performs observational rather than experimental discovery: it cannot ask whether a given set of variable values is possible or, given a subset of variable values, the values of the other variables.

While no counterexamples are available, program execution can cheaply generate arbitrarily many positive examples. (It may need to do so in order to exercise sufficiently many program paths, as in testing.) This wealth of data is no challenge for data mining but may overwhelm techniques designed to be run on a few dozens or hundreds of instances (see below for an example).

No noise. Statistical model-fitting approximately characterizes a cloud of data or identifies trends in it. Regression learns a function over $n-1$ variables producing the nth, again in the presence of noise. These and other statistical approaches are closer to the goal of invariant detection, but the are not applicable to finding general relationships over variables.

Learning approaches such as Bayesian learning, and some algorithms based on PAC learning, assume there is noise in the input data, which is distributed according to some (unknown) probability or distribution. As a result, a hypothesis that inaccurately classifies some of the training data is acceptable or even beneficial, because such inaccuracies help to avoid overtraining or prevent the learner from being fooled by noise. Program traces contain no noise: they indicate the exact values of all instrumented variables at a program point. While any learner can misclassify additional data, Daikon's output characterizes the training set perfectly.

Intelligible output. The primary goal of this research is to help programmers understand programs. The detected invariants must be comprehensible and useful, even if fed into another tool rather than presented to people. Some AI approaches, such as neural networks, can produce artifacts that predict results but have little explicative power, nor is it possible to know under what circumstances they will be accurate. Others present intelligible results.

Our emphasis on suppression of irrelevant output is similarly motivated. An invariant detector need not produce arbitrarily complex or general properties. Even if true, they would confuse, or be of no use to, the end user.

Invariant detection works in a man-made domain; because programmers have invariants in mind when writing the program, we should expect relatively simple and exact invariants to hold. In natural domains, fewer invariants may hold and they may be more complex or approximate.

As noted above, many learners produce relations that misclassify portions of the training set. Such inaccuracies are less likely to be acceptable in our domain, for they may mislead programmers, resulting in a loss rather than a gain in productivity.

Because of this new domain in which previous artificial intelligence techniques are at best partially applicable, this dissertation may be viewed as a research advance in AI as well as in software engineering. Much of this work was inspired, especially in its early stages, by AI and has been enriched both by borrowing from its techniques and by conversations with its practitioners. Generalizing learners or other AI techniques, or applying them to subproblems of invariant discovery, is a fruitful area for future research.

Related machine learning research

Artificial intelligence and machine learning research provides a number of techniques for extracting abstractions, rules, or generalizations from collections of data [Mit97]. Most closely related to our research is an application of inductive logic programming (ILP) [Qui90, Coh94]. ILP produces a set of Horn clauses (first-order if-then rules) that express the learned concepts. ILP requires counterexamples (which are not available in our domain) and background knowledge, and the resulting relations typically misclassify 10% or more of the training set. Bratko and Grobelnik use ILP to construct loop invariants from variable values on particular loop executions [BG93]. The technique is described by example application to a 5-line program which computes $\lfloor a/b \rfloor$ and $a \bmod b$. The multiplication and addition relations were added as background knowledge and negative examples were constructed by hand by modifying one of the variables. Several ILP systems were then able to infer the desired invariants. There has been no followup work or application to other programs.

In order to corroborate the practicality of ILP for inferring program properties, we experimented with FOIL [Qui90], version 6.4 [QCJ96], a learning program that can learn recursive first-order concepts, such as list membership. ("First-order" means the rules abstract over parameters rather than just applying to the particular instances in the input.) In one experiment, our goal concept was conjunctive normal form (CNF), which is used in the literature as an example. The results were mixed. In the end, the system learned the concept. However, that success required over 1000 positive and negative examples, with many iterations of rerunning the system after examining the output to determine what new examples were required (for instance, negative examples with atoms at the top level or with non-CNF formulas as subformulas). The input also had to contain positive and negative examples for the concept of member, since definitions cannot be input directly. The examples had to each be small in order for learning to complete; nonetheless, the system took hours to run. The output contained many irrelevant facts about the example formulas, and we had to modify the code for FOIL to prevent it from reordering relations in a way that prevented either computation by the system or comprehension by humans.

Numeric properties are not the strength of symbolic learners, so we tried using the regression system Cubist [Rul98] to find linear relationships among variables. While it found cases where a variable was constant and some (but not all) cases where two variables were equal, it did not find other simple linear relationships. The system is able to find piecewise-linear rules and works in the presence of noise, but we did not test those features.

Another related research area is programming by example (or programming by demonstration) [CHK$^+$93], whose goal is automation of repetitive user actions, such as might be handled by a keyboard macro recorder. That research focuses on discovery of simple repeated sequences in user input and in graphical user interfaces. The end result tends to be a sequence of commands (a program or algorithm) but generally isn't numeric. Most such systems require interaction with users, who indicate when to begin recording a macro or provide feedback about whether the system's suggested generalization is correct. Many of the systems are similar to macro recorders; others draw inspiration from teaching or programming. The systems' pragmatic focus downplays development of sophisticated generalization or inference subsystems.

Automatic programming [RW88a] is the generation of programs from high-level specifications. The Programmer's Apprentice [RW88b] automates certain repetitive, mundane, special-purpose tasks with the assistance of domain knowledge, pattern-matching, and a combination of special-purpose techniques with and general-purpose logical reasoning, each where it can contribute most.

Techniques for generating programs from behavioral examples in the form of $\langle$input, output$\rangle$ pairs can be extended to debugging [Sha81, Sha82, Sha83]. If the learned (or target) program operates incorrectly for an input, the problem is narrowed down by querying the user, for each procedure in the program, what the output should be, then searching for a small modification that corrects the behavior. This finds certain classes of simple errors; when generating new programs, the results are often not particularly elegant or easily understood.

The Bacon system [LSBZ87] performs concept discovery via depth-first search over the space of arithmetic expressions, with a number of extensions and heuristics. The system focuses on “scientific discovery” (numeric relationships like $v = at^2$ discovered by physical scientists) and aims to reproduce the techniques, successes, and failures of historical scientists. Various versions of the system use different techniques, but most assume experimental control.

Version spaces are a representation technique for sets of hypothesis [Mit78, Hir91, LDW00]. All potential hypothesis sets are arranged in a lattice ordered by generalization/specialization, and a region of the lattice (a set of hypothesis sets) is represented by its upper and lower fringes. New examples invalidate some hypothesis sets; this is represented by moving one boundary or the other, depending on whether the new example is a positive or negative example.

Evidence-based static branch prediction [CGJ$^+$97] uses machine learning to predict which branches will be taken based on the structure of the program and a corpus of training programs. The results are competitive with other static prediction mechanisms.

While the practical and empirical side of computational learning theory developed into today’s machine learning field, the theoretically motivated side based on the theory of computable functions came to be known as inductive inference. Several survey articles cover this ground [AS83, Ang92, Ang96].

8.1.2 Other dynamic approaches

Value profiling [CFE97, SS98, CFE99] addresses a subset of Daikon’s problem: detection of constant or near-constant variables or instruction operands. Such information can permit run-time specialization: the program branches to a specialized version if a variable value is as expected. Run-time disambiguation [Nic89, SHZ$^+$94, HSS94] is similar, though it focuses on pointer aliasing. Many optimizations are valid only if two pointers are known not to be aliased. Although efficient and precise static determination of that property is beyond the state of the art, it can be checked at runtime in order to use a specialized version of the code. For pairs of pointers that are shown by profiling to be rarely aliased, runtime reductions of 16–77% have been realized [Nic89]. Another approach to this particular problem is hardware support for checking for, and recovering from, conflicts during speculative execution [DC88, HS90, GCM$^+$94, SM97].

Other work is capable of finding more complicated invariants than constant or near-constant variables (though still a subset of Daikon’s invariants), such as ordering relationships among pairs of variables (e.g., $x < y$) [VH98] or simple linear patterns for predicting memory access strides, which permits more effective parallelization [Jon96, DPPA98].

Chimera [KF93] infers geometric relationships from the history of change operations used to transform one drawing into another. In order to reduce the computational load and the incidence of false positives, relationships are inferred over connected objects that a user has modified together. To enable accurate inference, the user must vary all of the degrees of freedom that are meant not to be constrained. Chimera then interactively maintains the inferred relationships as the user continues to work.

Graffiti [Lar99] exhaustively lists potential graph properties of the form “(sum of graph properties) $\leq$ (sum of graph properties)”. One example is “average distance (between any two vertices in a graph) $\leq$ independence number (of the graph).” It then checks these properties over a small collection of graphs (the collection is intentionally small, to control runtime) and also eliminates those implied by previously conjectured properties. Conjectures that are not falsified are publicly listed in the hopes that a mathematician will formally prove or disprove them. Related work is presented by Larson [Lar99] and Valdés-Pérez [VP98].

Spatial inference algorithms may be applicable to invariant detection. Jones [Jon96] used the Hough Transform, an image analysis algorithm, to infer reference patterns in sequences of pointer references.

Another approach to capturing and modeling run-time system behavior uses event traces, which describe the sequence of events in a possibly concurrent system, to produce a finite state machine (essentially, a grammar) generating the trace and thus modeling the system. The events tend to be manually identified but automatically extracted from the system's execution. In order to explicate system behavior, Cook and Wolf [CW98a, CW98b] produce a finite state machine that could generate the trace. They use statistical and other techniques to detect sequencing, conditionals, and iteration, both for concurrent programs and for business processes such as a customer bug report or code checkin. Users may need to correlate original and discovered models that have a different structure and/or layout, or may need to iteratively refine model parameters to improve the output. Verisoft [BG97] systematically explores the state space of concurrent systems using a synthesized finite state machine for each process. Andrews [And98] compares actual behavior against behavior of a user-specified finite-state model, indicating divergences between the two.

The Dynamic Dependence Analyzer [OHL$^+$97, LDB$^+$99] computes runtime dependences, advancing a synthetic timestamp only when dependences force it, and then constructs an optimistic parallel schedule from the synthetic timestamps. The Assure component of the KAP/Pro Toolset [Kuc] provides users similar dynamic dependence information. Bennett et al. [BLM$^+$99] suggest rewriting long transactions as two parts: a non-locking rehearsal phase performs modifications on local copies of objects, then a locking performance phase quickly redoes the side-effecting operations, skipping long computations, user input, interactions with other processes, etc. The performance phase requires values to satisfy certain user-specified properties that held during rehearsal, but the values need not be identical. It would be advantageous to automatically infer the required properties.

Eraser [SBN$^+$97] dynamically checks that all shared memory accesses follow a consistent locking discipline to ensure the absence of data races. Eraser maintains, for each memory location, a state in a finite state machine: virgin, exclusive, shared, or shared-modified (which indicates a race condition). It has found data races in a variety of programs.

Autoprogramming [BK76] is a variety of visual programming that permits users to construct computations by manipulating concrete values which are graphically displayed on a computer screen. The programmer is assumed to have the method well in mind and performs the operations by direct manipulation of the graphical representations. If an operation should be conditional, the user indicates the condition before performing the operation. The system then extends these traces to flowcharts, making loops big enough to eliminate ambiguity. If the inference is incorrect, the programmer supplies more traces or inserts into the existing traces.

Program spectra (specific aspects of program runs, such as event traces, code coverage, or outputs) [AFMS96, RBDL97, HRWY98, Bal99] can reveal differences in inputs or program versions. The set of invariants detected in a program could serve as another spectrum, as illustrated by the use of invariants to validate a change to the `replace` program (Section 2.2.4, page 17).

Database optimizations can speed up dynamically testing specified properties for all objects in a system [LHS97]; Daikon's query tool could use similar techniques.

8.2 Static inference

Work in formal methods [Hoa69, Dij76, DS90, CWA$^+$96] inspired our research; we wanted to find the dynamic analog to static techniques involving programmer-written specifications. (Just as the `assert` statement is the dynamic analog to static theorem-proving to verify a property, dynamic invariant detection is the dynamic analog to writing down a formal specification.) We have adopted the Hoare-Dijkstra school's notations and terminology, such as preconditions, postconditions, and loop invariants, even though an automatic system rather than the programmer produces these properties and they are not guaranteed, only likely, to be universally true. A number of authors note the advantages of knowing such properties and suggest starting with a specification before writing code [Gri81, LG86, Dro89] or refining a specification into a correct program [Gri81, CM88, BG93, FM97]. Despite these advantages, this approach is rarely applied in practice, necessitating techniques such as dynamic invariant inference.

Static analyses operate on the program text, not on particular test runs, and are typically sound but conservative. As a result, properties they report are true for any program run, and theoretically they can detect all sound invariants if run to convergence [CC77]. In particular, abstract interpretation (typically implemented as dataflow analysis) starts from a set of equations specifying the semantics of each program expression, then symbolically executes the program, so that at each point the values of all variables and expressions are available in terms of the inputs. (The static analysis uses an abstraction of runtime values; this makes the computation tractable but loses information. Here, we focus on abstractions that strive to maintain properties over variable values, though other types of abstraction are more common.) The solution is approached either as the greatest lower bound of decreasing approximations or as the least upper bound of increasing approximations. When the fixed point of the equations is reached (possibly after infinitely many iterations that compute improving approximations, or by reasoning directly about the fixed point), then the resulting properties are the optimal invariants: they imply every other solution.

In practice, static analyses suffer from several limitations. They omit properties that are true but uncomputable and properties that depend on how the program is used, including properties of its inputs or context. More seriously, static analyses are limited by uncertainty about properties beyond their capabilities and by the high cost of modeling program states; approximations that permit the algorithms to terminate introduce inaccuracies. For instance, accurate and efficient alias analysis is still beyond the state of the art [CWZ90, LR92, WL95] (see Section 8.2.2); pointer manipulation forces many static checkers to give up or to approximate, resulting in overly weak properties. In other cases, the resulting property may simply be the (infinite) unrolling of the program itself, which conveys little understanding because of its size and complexity. Because dynamic techniques can detect context-dependent properties and can easily check properties that stymie static analyses, the two approaches are complementary.

8.2.1 Tools based on static analysis

Some program understanding tools have taken the abstract interpretation/dataflow approach. For instance, full specifications can be constructed by extending a precondition (a specification on the inputs of a procedure) to its output. This approach is similar to abstract interpretation or symbolic execution, which, given a (possibly empty) precondition and an operation's semantics, determines the best postcondition. It also shares similarities with type inference's extension of partial type annotations to full ones; variable types are a variety of formal specification and documentation and whose checking can detect errors. Givan [Giv96a, Giv96b] takes this approach and permits unverified procedural implementations of specification functions to be used for runtime checking. No indication is provided of how many irrelevant properties are output. Gannod and Cheng [CG91, GC96] also reverse engineer (construct specifications for) programs via the strongest postcondition predicate transformer. User interaction is required to determine loop bounds and invariants. They also suggest ways to weaken conditions to avoid overfitting specifications to implementations, by deleting conjuncts, adding disjuncts, and converting conjunctions to disjunctions or implications [GC99]. This may result in a more understandable and even more accurate formal specification. Jeffords and Heitmeyer [JH98] generate state invariants for a state machine model from requirements specifications, by finding a fixed point of equations specifying events that cause mode transitions. Compared to analyzing code, this approach permits operation at a higher level of abstraction and detection of errors earlier in the software life cycle. Chan [Cha00] shows how to extend model checking techniques to model understanding, inferring temporal properties as well as checking them. Solutions to temporal-logic queries (temporal-logic formulas with single placeholders) are strongest invariants; users may need these complex formulas to be simplified or reduced.

Some formal proof systems generate intermediate assertions or auxiliary predicates for help in automatically proving a given goal formula. They may do so by forward propagation and generation of auxiliary invariants or by backward propagation and strengthening of properties [Weg74, GW75, KM76, BBM97]. As an example, given a complete specification and/or the initial state and that after the first iteration, Dunlop and Basili [DB84, DB85] use symbolic execution to generalize to a loop invariant for uniformly implemented condition-free loops that change one result variable in the same way as the loop index. In the case of array bounds checking [SI77, Gup90, KW95, NL98, XP98], the desired property is obvious. In general, though, it is considered harder to determine what property to check than to do the checking itself [Weg74, WS76, MW77, Els74, BLS96, BBM97]. Our research is directly applicable, since its goal is discovery or postulation of such properties, at any program point. However, our problem of choosing splitting conditions for the inference of disjunctive invariants presents similar challenges to those addressed by this work, for it too seeks to synthesize predicates that exploit special properties of the code.

ReForm [War96] takes the opposite approach: it semi-automatically transforms, by provably correct steps, a program into a specification. The Maintainer's Assistant [WCM89] uses programmer-directed interactive program transformation to derive specifications from code, to transform code into a logically equivalent form, and to prove program equivalence (two programs are equivalent if they can be transformed to the same specification or to one another).

The Illustrating Compiler [HWF90] heuristically infers, via compile-time pattern matching and type inference, the abstract datatype implemented by a collection of concrete operations, then graphically displays run-time data in a way that is natural for that datatype.

Staging and binding-time analyses determine invariant or semi-invariant values for use in partial evaluation [JGS93, LL93, Pal95].

Suzuki and Ishihata [SI77] check array bounds by using the weakest liberal precondition (wlp) to generate a potentially infinite chain of approximations to the general loop invariant. If a theorem-prover fails to prove a given property, then the wlp construction proceeds for one more iteration (potentially doubling the size of the goal formula) and theorem-proving is retried.

Dan et al. [DYKT89] describe a system for generating and verifying loop invariants. The system translates a Pascal program to Prolog, which is then verified. The user indicates which code paths to examine, and the system can detect arithmetic, multiplicative, and exponential patterns in loop-carried variables. It does this by symbolic execution and unification (via nondeterministic search over all possible substitutions) of the symbolic values on different loop iterations. For example. the sequence $\langle b, \frac{b}{2}, \frac{b-1}{2}, \frac{b}{4}, \frac{b-3}{4}\rangle$ would generalize to $\frac{b-x}{y}$ for unknown runtime values x and y.

The DISCOPLAN planner [GS98] demonstrates significant speedups over previous SAT-based planning systems by exploiting state constraints, such as that an object is not clear if some other object is on it.

Manna and Pnueli [MP95] overview work on computing temporal assertions statically for transition systems, in the context of model checking.

8.2.2 Pointer and shape analysis

Most pointer analysis research determines alias or points-to relations. Such information can be used to compute the may definitions of an assignment in static program slicing or to verify the independence of two pointer references to enable an optimization. The analysis results are applied to problems such as program optimization, program checking, debugging, and assisting changes to pointer-based programs. Precise pointer analysis is computationally difficult [LH88, LR92]. The high cost of flow-sensitive approaches [Wei80, JM81, JM82, CWZ90, HN90, LR92, HEGV93], has led to the development of flow-insensitive techniques [And94, Ste96, SH97], which are often nearly as precise for a fraction of the cost [HP98]. The same results may hold for context sensitivity [Ruf95]. For specific applications, contexts, or assumptions, efficient pointer analyses can be sufficiently accurate [Das00].

Shape analysis is a static analysis that infers properties of pointer structures that could be used by programmers as invariants. In particular, shape analysis produces a graph structure for a structure pointer reference that summarizes the abstract memory locations that it can reach [JM81, LH88, CWZ90, HHN92, SRW99]. ADDS [HHN92, GH96] propagates data structure shape descriptions through a program, cast as a traditional gen/kill analysis. These descriptions include the dimensionality of pointers and, for each pair of live pointer variables visible at a program point, reachability of one from the other and whether any common object is reachable from both. This information permits the determination of whether a data structure is a tree, a dag, or a cyclic graph, modulo approximations in the analysis. Other shape analyses have a similar flavor [SRW99]. Benedikt et al. [BRS99] propose a decidable regular-expression-based logic for describing paths through data structures,

The necessity to summarize actual properties in static approaches is akin to our choice to limit the depth to which we derive variables. Our depth limiting is similar to the simple static approach of k-limiting [JM81].

8.3 Checking invariants

A specification can be checked against an implementation either dynamically, by running the program, or statically, by analyzing it. (Specifications can also be checked directly, using techniques like those described above, for properties such as liveness or fairness.)

Dynamic approaches are simpler to implement and are rarely blocked by inadequacies of the analysis, but they slow down the program and check only finitely many runs. Numerous implementations of `assert` facilities exist, many motivated by making invariant assertion languages more expressive [GH93, Ros95, CBS98, KHB99], a topic that is often also taken up by research on static checking. Programmers tend to use different styles for dynamically- and statically-checked invariants; for instance, tradeoffs between completeness and runtime cost affect what checks a programmer inserts. Self-checking and self-correcting programs [BK95, WB97] double-check their results by computing a value in two ways or by verifying a value that is difficult to compute but easy to check. For certain functions, implementations that are correct on most inputs (and for which checking is effective at finding errors) can be extended to being correct on all inputs with high probability. Programmer-inserted checks are not always effective in detecting errors. In one study, out of 867 program self-checks, 34 were effective (located a bug, including 6 errors not previously discovered by n-way voting among 28 versions of a program), 78 were ineffective (checked a condition but didn't catch an error), 10 raised false alarms (and 22 new faults were introduced into the programs), and 734 were of unknown efficacy (never got triggered, and there was no known bug in the code they tested) [LCKS90].

Considerable research has addressed statically checking formal specifications [Pfe92, DC94, EGHT94, Det96, Eva96, NCOD97, LN98]; such work could be used to verify dynamically discovered likely invariants.

The LSL Checker checks the syntax and type consistency of LSL (Larch Shared Language) specifications, then generates LP proof obligations from their claims [GHG$^+$93]. LP, the Larch Prover, proves semantic claims about consistency (a theory does not contradict itself), theory containment (a specification has intended consequences), and relative completeness (a set of operators is adequately defined).

Model checking is a technique for checking properties of a formally specified system. Given a mathematical model of the system (typically a state transition diagram) and a property to prove (typically a formula in temporal logic), a model checker exhaustively explores the space of reachable states in order to determine whether a state satisfying or falsifying the goal property is reachable.

Although most checking research has addressed formal specifications, some realistic static specification checkers that connect these properties to code have recently been implemented. LCLint [EGHT94, Eva96] verifies that programs respect annotations in the Larch/C Interface Language [Tan94]. Although these annotations focus on properties such as modularity, which are already guaranteed in more modern languages such as C++, they also include pointer-based properties such as definedness, nullness, and allocation state.

ESC [Det96, LN98, DLNS98], the Extended Static Checker, permits programmers to write type-like annotations including arithmetic relationships and declarations about mutability; it catches array bound errors, nil dereferences, synchronization errors, and other programming mistakes. LCLint and ESC do not attempt to check full specifications, which remains beyond the state of the art, but are successful in their more limited domains where they emphasize fast checking of partial specifications which are easy for programmers to specify. Dependent types [Pfe92, Zen97] make a similar tradeoff between expressiveness and computability. Xi and Pfenning [XP99] state, "The main contribution of this paper lies in our language design, including the formulation of type-checking rules which makes the approach practical." For instance, the decision to limit the expressiveness of dependent types preserves checkability. Neither LCLint nor ESC is sound, but they do provide programmers substantial confidence in the annotations that they check. We are investigating integrating Daikon with one of these systems in order to explore whether it is realistic to annotate a program sufficiently to make it pass these checkers.

Several other projects aim to address code rather than specifications. LOOP [JvH$^+$98] converts a subset of Java to PVS[ORS92, ORSvH95], with human assistance. Properties can then be checked over the resulting PVS model. Bandera [CDH$^+$00] takes a similar approach of reusing existing, mature, efficient model checkers but supports multiple modeling formalisms and performs optimizations such as slicing to reduce the amount of code under consideration and supporting user abstractions to reduce the representations of program data. Java PathFinder [HP00] initially translated Java programs annotated with boolean invariants into Promela, the language of the SPIN model checker [Hol97], but more recently has switched to a special-purpose model-checker that works directly on Java bytecode.

ACL2 [KM97] is a theorem prover for a Lisp-like, quantifier-free, first-order mathematical logic based on recursively defined total functions. ACL2 is an extension of the Boyer-Moore system Nqthm [BM97]. ACL2 is written in Lisp and works over 170 Common Lisp functions (users can define more, subject to some constraints). This logic is designed to model hardware and software systems, and the models can be executed to corroborate their accuracy. It has been used on commercial hardware projects such as the Motorola CAP digital signal processor and the AMD5K86 floating-point division algorithm [BMJ96].

Lano [LB90] proposes using properties of monads and category theory to model a concurrent message-passing language and repeats several heuristics from Gries [Gri81] that may ease the extraction of Z [Spi88] specifications from programs,

Chapter 9

Future work

While this research has demonstrated some of the potential of dynamic invariant inference, many opportunities for extending the work remain. This chapter presents a few of the potential directions that such extensions could take. These include improving the implementation to reduce runtime and improve invariant quality (Sections 9.1 and 9.8); introducing additional varieties of invariant (Section 9.3); improving the user interface to give the programmer more control over the inference process (Section 9.4); evaluating the effectiveness of invariant detection for a variety of tasks (Sections 9.5 and 9.6); and characterizing what test suites are best for invariant inference (Section 9.7).

9.1 Scaling

This dissertation has demonstrated the accuracy, usefulness, and efficiency of dynamic invariant detection for modest programs of several thousands of lines. A key question of any research in programming, from software process to compiler analysis, is "Does it scale?" There are two equally important aspects to scaling: scaling the technology and scaling the utility.

Scaling the technology means making the system operate on large programs, where a large program may have many instrumentation points, large data structures, and/or long runtimes. Chapter 6 showed that Daikon's runtime grows (essentially) linearly with program and test suite size and that relatively small test suites enable good invariant detection. Additionally, not all of a program need be examined, because programmers typically aren't interested in all the program points in a program. However, it is clear that the current prototype will not handle large programs without change. We have profiled the system to identify its weaknesses, which center around I/O and storage of data. For one thing, it reads and stores all data before processing it. Section 9.2 outlines an approach we believe will solve these problems. Section 5.2 gave further suggestions for reducing the cost of traversing and checking large data structures.

Making invariant detection useful (as opposed to merely possible) for large programs requires enabling users to cope with invariants at many program points; see Section 9.4. Large programs may have qualitatively different invariants, and uses for invariants, than do small programs; only experiments and case studies can settle this question. Large programs will not necessarily fare badly in this analysis; for example, in large programs object invariants (which are relatively inexpensive to test) may be more interesting than function pre- and post-conditions. For programs with many small procedures — and thus many potential instrumentation points — that is likely to be the case, too. The goal of this or any software engineering research is to learn something or aid some task(s). It can be a success even if it does not perfectly reveal all information or assist every programmer and task.

9.2 Incremental, online processing

The approaches outlined in this dissertation perform adequately for modest programs, but we do not expect them to work without change on large programs. Based upon profiling Daikon to determine its bottlenecks, this section outlines one approach to using online, incremental invariant inference to scale Daikon to larger, more realistic pointer-based programs.

The major hurdle to making invariant detection scale is the large number of possibly sizable values maintained by the target program. Large data structures are costly for the instrumented code to traverse and output; for instance, an instrumented operation has cost at least $O(n)$, where n is the size of the visible data. Large data structures are also costly for the invariant detector to input and examine, for the same reasons. Additionally, wide data structures result in a large number of fields being output to the data trace. Invariant detection itself has a modest cost (see Section 6.1).

To speed up tracing and inference, we can eliminate I/O costs by running the invariant detector online in cooperation with, and in the same address space or virtual machine as, the program under test. Daikon can then directly examine the program's data structures. As noted in Section 3.1 (page 25), this change loses some advantages of simplicity and language-independence while gaining performance, more precise information, and other benefits such as the ability to handle polymorphism in one pass.

The initial prototypes retained the full data traces in memory. When testing invariants online, Daikon need not store all data values indefinitely. However, the program point data structures (see Section 7.7, page 89) will store some values, either transiently or permanently, for three reasons. First, a sufficient pool of values must be accumulated initially to permit instantiating all viable invariants in a staged fashion, then discarded. Staging permits simpler invariants to be tested first so that redundant invariants can be suppressed before being instantiated; see Section 4.3.1 (page 33) for details. Second, it is more efficient to collect values and process them together — say, in blocks of 100 or 1000 — than individually. Third, some data values will be retained permanently to indicate why each invariant was falsified and to (probabilistically) answer other user queries. All other values can be discarded after potential invariants are checked over them.

The online approach requires that when an invariant is falsified, then other invariants and variable derivations that were suppressed by its presence must be reinstated. This is not necessary for a batch algorithm that can examine all data at each stage before proceeding to the next one. Accumulating a moderate amount of data before proceeding, as proposed above, will mitigate this cost by reducing the number of falsifications.

Beyond eliminating the I/O bottleneck and reducing memory costs, incremental processing can also reduce the amount of work performed by the instrumentation itself. When a variable is determined to be no longer of interest, then the instrumentation can be informed to stop recording its value, thus reducing its overhead. This is particularly important for pointer-directed collections that may be large and that the implementation would otherwise traverse. Given that most invariants are quickly falsified, we speculate that this will provide the largest speedup.

Integrating disjunction with incremental computation requires similar approaches to those outlined above. In particular, it is not possible to examine all data in order to determine splitting criteria.

Implementation of a detailed design for online operation is underway. Daikon already supports incremental inference. However, when the truth of an invariant suppresses derived variables or other invariants, those suppressed elements are not yet reinstated if the invariant is later falsified. Furthermore, invariant falsification is not linked to instrumentation, in order to selectively disable instrumentation when a variable is no longer of interest.

9.3 Extending domains and logical operators

The current invariant detection prototype, Daikon, infers invariants only over integers, arrays, and limited-depth record structures (Section 3.1). Section 9.2 discusses lifting this restriction via tighter integration with its front ends.

Daikon's invariants are universally quantified, as in "all array elements are zero." It should be extended to detect existentially quantified invariants like "the array always contains at least one zero element." This does not require a fundamental change: Daikon would still examine some values, generalize from them, and test the generalization over the remaining values.

Temporal invariants are another fruitful area for enhancement. These could indicate monotonicity or other properties of a variable's history or the fact that quantities vary together, even if their exact relationship cannot be determined. Many programs operate in stages. An array or data structure field may be used for one purpose at one point in a program and another purpose elsewhere. Many data structures have a mutable phase during which they are being constructed and an immutable phase during which they are only read. Periodically restarting invariant detection or doing special checking when long-standing invariants are violated can reveal such properties.

A final type of temporal invariant is liveness or safety properties, such as "if a request is made, it is eventually serviced" or "no response precedes its corresponding request." Formalisms such as LTL [MP91] are used in verifying programs, particularly concurrent ones. Such properties are an attractive target for Daikon because of the existence of model checkers that can automatically verify them [MP91, McM93, Hol97, DIS99, HP00].

9.4 User interface

A user interface can permit users to direct instrumentation and invariant inference, or it can display, and enable manipulation of, invariants.

User control over invariant detection can increase relevance (Chapter 4), since the user knows the intended use of the results. This potentially reduces serendipity, because unanticipated but useful invariants may not be computed or reported. Users can instrument only that part of the program that is of interest, specify certain variables to include or omit, and disable some invariants or derived variables. A user interested in data structure layout need not see numeric invariants, for example. Users can add special-purpose or domain-specific invariants and derived variables to Daikon's defaults (see Section 7.8). Users can also control performance by their choice of test suite. Fine-grained control of invariant detection increases the burden on the programmer. Our work has not emphasized such mechanisms, in order to reduce this burden. An experienced user who is sold on the technology will be willing to invest more effort in order to get a better result, but novices are unlikely to. Daikon does permits instrumentation to be suppressed for classes and functions specified via a command-line argument; likewise, users can specify detection of only class invariants, only procedure preconditions and postconditions, or both. An evaluation of user reactions to various control mechanisms might be interesting.

The utility of a user interface for organizing and managing computed invariants is clearer. Mechanisms for filtering of uninteresting invariants might be similar to those above. The interface could also group invariants by variable, category, or predicted usefulness to help a programmer find a relevant invariant more quickly. Displaying invariants in the context of the program's source code could assist understanding or user interaction. For example, clicking on a program point or variable could show the invariants at that program point or for that variable. Such querying can overcome the difficulty of finding an invariant in the current prototype's output, either because of the verbosity of the output or because that invariant is implied and so does not occur (for an example, see Section 5.4.3, page 64). Invariants could also be computed on demand, possibly improving response time. We have already seen that interactive query tools and computing differences among sets of invariants are useful to programmers.

Daikon reports properties of a program's implementation and data representations. Although this is extremely useful to an implementer of a data structure or service, programmers implementing clients would probably prefer reports in terms of the program's abstractions rather than its concrete realization. Interpreting programmer intent is beyond the ability of computers today and for some time to come, and programmers are able to use information about representations to learn about abstractions. However, it might be possible to improve this output, perhaps abbreviating or suppressing invariants over private members.

9.5 Evaluation

Further experimental evaluation of dynamic invariant detection — applying it to more and bigger programs and especially to a variety of tasks performed by a variety of users, including those listed in Section 1.5 — is the most important aspect of future work, because in the final analysis utility for users is more important than the underlying technology.

Example questions to be answered include the following: How steep is the learning curve for use of the tool and for use of invariants in understanding code? What tasks are invariants useful for, and which tasks are not aided by the availability of invariants? Are invariants more useful to certain programmers than to others? Does the availability of invariants change the way programmers think about their work? How can dynamically-detected invariants complement other software engineering tools and techniques?

Because of the high cost of effectively replicating large-scale programmer studies, we propose to begin with observational studies of programmers in the research environment, followed by a standard case study of the invariant detector on a more substantial system. In the latter, external users will work with their own programs, solving the real-life software engineering tasks. The (possibly) unanticipated ways they use the tool, and their experiences in doing so, will help evaluate it. Feedback will focus our attention on specific solutions that are likely to enable broader and more effective use of the invariants technology. For example, it will help us select from among the many potential extensions to Daikon listed in this chapter.

Users who are interested in experimenting with the prototype invariant detection tool, Daikon, are encouraged to do so and to contact the author with any comments, questions, or suggestions at `mernst@cs.washington.edu`. Daikon is available for download at `http://www.cs.washington.edu/homes/mernst/daikon`.

9.6 Proving

Like any dynamic analysis, dynamic invariant detection is unsound. The resulting properties are valid for the test cases but not guaranteed to hold over arbitrary executions. Additionally, systems that check user annotations or specifications, such as ESC and LCLint, require a substantial upfront investment for the user to annotate the program. Integrating Daikon with a program checker could solve both problems. The reported invariants would be used as an initial program annotation, and the checker would verify their soundness or report which ones could not be statically proved. This line of research presents at least two challenges. First, the detected invariants may need to be weakened, because they may include properties too strong to be proved, due to inadequacies of the prover or properties of the environment when are inaccessible to the prover. A partially-annotated program could trigger even more warning messages than an unannotated one. Second, the invariants may need to be broken into groups that will be separately checked, particularly if the prover gives little feedback on what blocked the proof, in which case a failure to simultaneously prove a large set of properties reveals little. Rushby [Rus99] gives some examples of manually splitting a state space into parts and then determining how to modify invariants to make them checkable.

9.7 Test suites

Using Daikon requires execution of a test suite, and the output depends on the quality of the tests. Chapter 6 began to explore the relationship between Daikon and the test suite, and a number of open questions remain.

Characterizing what test suites are good for invariant detection is an important open question. The test suite can be measured in terms of coverage, size, or other metrics, and the invariants in terms of stability (as in Section 6.2), obtaining desired invariants, etc. We are particularly interested in concrete direction for users interested in construct good test suites (either by hand or semi-automatically) or predicting the quality of invariants generated by a particular test suite. One aspect of test suite construction is determining what types of test case are most crucial to falsifying undesired invariants or providing support for desired invariants. A related task is test suite minimization: reducing a test suite to the minimal size that still produces useful invariants.

The techniques for improving invariant relevance (Chapter 4) might be obviated by the use of better test suites. For instance, if two variables really aren't related, then given an adequate test suite, perhaps every possible relationship will be violated by at least one test cases. It is unclear to what extent huge, complete test suites a tradeoff against other techniques for improving relevance.

Daikon can verify the breadth or value coverage of test suites. This could reveal unusual properties of test suites, permitting improvement of test suites based on the invariants detected. Such feedback could be used either manually or by an automatic tool. Test suites could also be constructed to avoid violating invariants which demonstrate actual, correct usage of a module. Useful research might include a study of existing test suites, watching users improve test suites, or building an automatic test suite improver.

Some of the test-suite-size experiments of Section 6.1 evidenced departures from linearity. Resolving the cause of such aberrations would eliminate a nagging question about the system's scalability.

9.8 Other directions

Additional improvements to the relevance of Daikon's output may be required. For instance, Daikon could perform dynamic slicing (trace run-time dependences to indicate exactly which values depend on which others) to compute a finer-grained comparability measure. Experiments may demonstrate the effectiveness of a less precise but cheaper (to implement and run) technique for computing modification bits than maintaining assignment timestamps at runtime. Daikon could also suppress statically obvious invariants that are directly implied by a single atomic statement in the program. For instance, presently Daikon would report the invariant $\mathtt{x} = \mathtt{y} + 1$ at a program point immediately following the assignment `x=y+1`. Such a static analysis must not be too strong, but should only eliminate invariants that are obvious to a programmer.

Integration of additional artificial intelligence techniques with Daikon, either by extending them to Daikon's domain or by application to a subpart of it, is another promising line of research.

Chapter 10

Assessment and conclusion

This dissertation has introduced dynamic detection of program invariants, presented techniques for detecting such invariants from traces, and assessed the techniques' efficacy. Section 1.6 (page 6) presented the hypothesis and contributions of the research, which are briefly recapped here.

Dynamic invariant detection obtains invariants from programs, providing programmers the benefits of invariants — in program design, coding, verification, testing, optimization, and maintenance — even when those invariants are not explicitly written down. Given runtime values of variables, an invariant detector reports properties that hold over a test suite.

The dissertation also describes how to improve these invariants by adding implicit quantities to explicit program variables, by eliminating unused polymorphism, by statistically testing invariants, by avoiding comparing unrelated variables, by suppressing logically implied quantities. It also extends invariant detection to pointer-based data structures by linearizing implicit collections into arrays and by detecting conditional invariants, which are not universally satisfied, by examining only parts of the data.

A prototype invariant detector, Daikon, accurately rediscovered and improved formal specifications which had been erased. Daikon also usefully aided programmers in understanding and modifying software. Daikon runs quickly and produces output of modest size. Test suites found in practice tend to be adequate for dynamic invariant detection.

The Daikon tool is available for download from `http://www.cs.washington.edu/homes/mernst/daikon`.

10.1 Lessons learned

The key result of this dissertation is that dynamic invariant inference is both feasible and promising for assisting humans and other tools in a variety of tasks. This was not at all obvious at the beginning of the research: most observers considered the approach exceedingly unlikely to succeed. This result does not necessarily improve performance on some existing benchmark, as a new algorithm, compiler optimization, or computer architecture might do. Rather, it enables a new way of working that provides non-incremental benefits. For instance, rather than showing a way to find bugs, it permits users to avoid introducing them in the first place. It automatically documents undocumented programs, providing information that would otherwise be difficult to obtain. There are doubtless more uses for invariants than listed in this dissertation or imagined by the author.

Here are some lessons taught by this research that may be helpful to others in their work. Many of the lessons are not new — their successful application is reflected in other tools and results — even if they have not always been explicitly stated.

Weak formalisms can dominate strong ones. A theoretically ideal analysis is sound and complete. In the real world, it is also infeasible. A weakened version is likely to be of more use than the strong, pure original. Full formal specifications, which fully describe the contract of a procedure, are infeasible to create (by hand or automatically) or to test (in most cases), and programmers often do not find them useful. By contrast, partial specifications such as types are easy to specify, check, and understand and have been widely adopted. This research weakens formal specifications by finding only certain varieties of invariants, but it can detect them fully automatically and they are comprehensible to people.

Partial solutions are useful. In order to help a programmer, a tool or environment need not solve every problem, nor need its solutions be perfect. The goal should be to provide some useful information, not to produce all of it. Inadequacy for one task is an irrelevant criticism if users happily use the tool for a different purpose. Dynamic invariant detection is of no obvious use for a number of aspects of the software development process, its scope is relatively limited, and it can fail to report even properties in its domain. However, it is still useful to programmers.

Unsoundness is little hindrance. For certain applications, an incorrect result is disastrous. For instance, a compiler that depends on an unsound analysis can perform optimizations that incorrectly change the behavior of the program. People are much more resilient. They can perform sanity checks on information they are provided, dismissing or investigating that which fails to make sense. They can understand the context and limitations of the information, using it only in appropriate ways. Finally, even information that is not strictly true can be a great aid in understanding a system or performing a specific task. Daikon's output is only guaranteed to be correct over the test suites for which it was run; however, users find it to be of use anyway. Further, they are as often grateful to learn the usage properties of the test suite as the functional invariants of the code.

Expect the unexpected. Unanticipated information can dispel misunderstandings or draw a programmer's attention to, or raise suspicions about, a part of the code he or she was not previously considering. This serendipity is particularly valuable when assumptions are not explicitly recognized by the programmer, for otherwise the reported properties would likely have been overlooked or never discovered.

Daikon detects code bugs. I anticipated that Daikon would be of no help in finding bugs in programs. Daikon reports properties of the program without making value judgments over them. However, human users, who expect certain invariants, can compare their preconceptions with Daikon's output. Discrepancies are bugs either in the programmer's understanding or in the program.

As a corollary, programmers are too focussed on debugging. When told that Daikon had been used to detect bugs, many programmers disregarded all other uses for invariants and pigeon-holed the invariant detection as a debugging technique. This may have been because debugging is their primary programming activity. They would be better served by understanding and using techniques — including dynamic invariant detection — for preventing bugs from being introduced in the first place, rather than continuing with their current strategy.

Formal specifications are buggy. Since writing a formal specification is as hard as writing a program, specifications are as error-prone as programs. Although programmers often think harder about specifications than programs, specifications are rarely validated either by formal techniques or by execution. Daikon can indicate incorrect formal specifications by reporting different properties, then supplying counterexamples on demand. It did so for nearly every program with a non-validated formal specification that it was supplied, including those in the MIT 6.170, Gries, and Hoffman test suites (see Section 2.4, page 20).

Filtering is harder than generation. In the initial stages of this research, I anticipated that the biggest challenge would be to produce any kind of reasonable output. In fact, that turned out to be easy, and even missing invariants were relatively straightforward to add, though the engineering required was often nontrivial. The problem was reporting far too many uninteresting invariants, which smothered the worthwhile ones and made using the tool difficult and tedious. Creating and implementing mechanisms to eliminate irrelevant invariants was crucial to the success of the tool and consumed a fair amount of time and energy.

Typical test suites are adequate. Because detected invariants characterize specific (sets of) executions of the target program was run, the quality of Daikon's output depends on the test cases. We do not know what makes a test suite good for invariant detection, and the most prevalent variety, test suites for bug detection, can be poor if they have been minimized. However, experiments show that moderate-size test suites found in practice produce good invariants. Furthermore, an inadequate test suite "fails" in one of two ways: no (or few) invariants are reported, indicating the test suite should be expanded because it is too small to statistically justify any invariants, or the reported invariants are properties of the specific data in the test suite and the computations arising from them. In the latter case, this precise characterization of the test suite indicates how to expand and improve it. In other words, when running Daikon, you can't lose. Either good invariants result, or Daikon indicates how to improve the test suite.

Brute force works. Daikon's basic approach is very simple: every potential invariant, at every instrumented program point, over every set of variables, is instantiated and checked. There can easily be millions of such invariants. However, while this checking could theoretically be very expensive, in practice it is cheap: there are few true invariants, and most false invariants are falsified quickly. Optimizations such as suppression of implied quantities by staged invariant inference and variable derivation

(Section 4.3.1, page 33) are required in order to achieve acceptable performance, but at base the system performs exhaustive search.

There are no undocumented programs. Programs lacking documentation are the bane of software engineering; considerable effort has been expended on understanding and manipulating them. Armed with invariant detection, a programmer never need face an undocumented program again, for the invariant detector can automatically generate a useful — albeit stylized, relatively low-level, and limited — variety of comments.

Cooperate with people. Daikon checks and reports basic, general, relatively low-level invariants over numbers and simple data structures such as arrays. These invariants cannot communicate concepts in terms of program abstractions, nor can they describe high-level properties. However, Daikon has reported accurate and useful invariants in a variety of programs. The key to this success is that, given hints about properties that are true, programmers are very good at extrapolating to higher-level properties about program abstractions. Furthermore, surprisingly simple basic invariants serve this purpose. Daikon exploits synergy between the computer and the human: the system does what it is good at (quickly checking many low-level invariants) and the people do what they are good at: understanding the implications and extracting the most important parts.

You can teach an old dog new tricks. Users of Daikon reported thinking in qualitatively new ways, keeping invariants in mind not only when viewing its output but also in other aspects of their task. This caused them to think more formally and correctly about their code. These particular users were familiar with the notions of formal specifications, though they did not use them in practice. This result suggests that tools such as Daikon can be powerful motivators in changing programmers' behavior and habits, inducing them to use techniques that college education and other experience had not. Daikon had this success in part because it does not require a large commitment from its users and because it uses a limited but useful subset of the formalism that is easier to understand and more effective than the whole. It provides real benefits rather than merely a mathematical theory.

Be a little bit stupid. The proposal for this research met considerable skepticism. It seemed clear to most people that the problem was undecidable and that an unsound, incomplete, computationally intractable approach had no chance of success. Pursuing the work regardless of these objections was somewhat foolhardy. Nonetheless, it produced worthwhile results, even if the jury is still out on the final impact.

To recover full, provably sound formal specifications or to use only a naive and straightforward implementation approach would have been doomed to failure. The current limited success required taste in choosing the goal and the tasks to which it would be applied, and it required ingenuity in implementation choices and in refining the output. In other words, you have to be a little bit smart as well: stupid alone does not win the day.

Bibliography

[AFMS96] David Abramson, Ian Foster, John Michalakes, and Rok Socič. Relative debugging: A new methodology for debugging scientific applications. *Communications of the ACM*, 39(11):69–77, November 1996.

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. DIKU report 94/19.

[And98] James H. Andrews. Testing using log file analysis: Tools, methods and issues. In *Proceedings of the 13th Annual International Conference on Automated Software Engineering (ASE'98)*, pages 157–166, Honolulu, Hawaii, October 1998.

[Ang92] Dana Angluin. Computational learning theory: Survey and selected bibliography. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, pages 351–369, Victoria, BC, Canada, May 1992.

[Ang96] Dana Angluin. A 1996 snapshot of computational learning theory. *ACM Computing Surveys*, 28(4es):216–216, December 1996.

[AS83] Dana Angluin and Carl H. Smith. Inductive inference: Theory and methods. *ACM Computing Surveys*, 15(3):237–269, September 1983.

[Bal99] Thomas Ball. The concept of dynamic analysis. In *Proceedings of the 7th European Software Engineering Conference and the 7th ACM SIGSOFT Symposium on the Foundations of Software Engeneering*, pages 216–234, September 6–10, 1999.

[BBM97] Nicolaj Bjørner, Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, February 1997.

[BG93] Ivan Bratko and Marko Grobelnik. Inductive learning applied to program construction and verification. In José Cuena, editor, *Knowledge Oriented Software Design: Extended Papers from the IFIP TC 12 Workshop on Artificial Intelligence from the Information Processing Perspective, AIFIPP '92, Madrid, Spain, 14-15 September, 1992*, pages 169–182. North-Holland, 1993.

[BG97] Bernard Boigelot and Patrice Godefroid. Automatic synthesis of specifications from the dynamic observation of reactive programs. In *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97)*, pages 321–333, Twente, April 1997.

[BK76] Alan W. Biermann and Ramachandran Krishnaswamy. Constructing programs from example computations. *IEEE Transactions on Software Engineering*, 2(3):141–153, September 1976.

[BK95] Manuel Blum and Sampath Kannan. Designing programs that check their work. *Journal of the Association for Computing Machinery*, 42(1):269–291, January 1995.

[BLM$^+$99] Brian Bennett, Avraham Leff, Thomas A. Mikalsen, James T. Rayfield, Isabelle Rouvellou, Bill Hahm, and Kevin Rasmus. An OO framework and implementation for long running business processes, 1999.

[BLS96] Saddek Bensalem, Yassine Lakhnech, and Hassen Saidi. Powerful techniques for the automatic generation of invariants. In *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, pages 323–335, New Brunswick, NJ, USA, July/August 1996.

[BM97] Robert S. Boyer and J. Strother Moore. *A Computational Logic Handbook.* Academic Press, 2nd edition, 1997.

[BMJ96] B. Brock, M. Kaufmann, and JS. Moore. ACL2 theorems about commercial microprocessors. In M. Srivas and A. Camilleri, editors, *First international conference on formal methods in computer-aided design*, volume 1166 of *Lecture Notes in Computer Science*, pages 275–293, Palo Alto, CA, USA, November 1996. Springer Verlag.

[BRS99] Michael Benedikt, Thomas Reps, and Mooly Sagiv. A decidable logic for describing linked data structures. In *ESOP: 8th European Symposium on Programming*, 1999.

[CBS98] Edwin C. Chan, John T. Boyland, and William L. Scherlis. Promises: Limited specifications for analysis and manipulation. In *Proceedings of the 20th International Conference on Software Engineering*, pages 167–176, Kyoto, Japan, April 19–25, 1998.

[CC77] 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.

[CCK98] Geoff A. Cohen, Jeffrey S. Chase, and David L. Kaminsky. Automatic program transformation with JOIE. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 167–178, Berkeley, CA, June 15–19, 1998. USENIX Association.

[CDH$^+$00] James Corbett, Matthew Dwyer, John Hatcliff, Corina Păsăreanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, Limerick, Ireland, June 7–9, 2000.

[CFE97] Brad Calder, Peter Feller, and Alan Eustace. Value profiling. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-97)*, pages 259–269, San Jose, CA, December 1–3, 1997.

[CFE99] Brad Calder, Peter Feller, and Alan Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*, 1, March 1999. http://www.jilp.org/vol1/.

[CG91] Betty H.C. Cheng and Gerald C. Gannod. Abstraction of formal specifications from program code. In *Proceedings of the 3rd International Conference on Tools for Artificial Intelligence TAI '91*, pages 125–128, San Jose, CA, November 1991.

[CGJ$^+$97] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems*, 19(1):188–222, January 1997.

[CH94] William W. Cohen and Haym Hirsh. Learning the CLASSIC description logic: Theoretical and experimental results. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourth International Conference (KR94)*, pages 121–133, Bonn, Germany, 1994. Morgan Kaufmann.

[Cha00] William Chan. Temporal-logic queries. In *Computer Aided Verification, 12th International Conference, CAV 2000 Proceedings*, pages 450–463, Chicago, IL, USA, July 15–19, 2000.

[CHK$^+$93] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch What I Do: Programming by Demonstration.* MIT Press, Cambridge, MA, 1993.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, 1988.

[Coh94] William W. Cohen. Grammatically biased learning: learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 68:303–366, August 1994.

[CR99] Juei Chang and Debra J. Richardson. Structural specification-based testing: Automated support and expenmental evaluation. In *Proceedings of the 7th European Software Engineering Conference and the 7th ACM SIGSOFT Symposium on the Foundations of Software Engeneering*, pages 285–302, N. Y., September 6–10 1999.

[CW98a] Jonathan E. Cook and Alexander L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, July 1998.

[CW98b] Jonathan E. Cook and Alexander L. Wolf. Event-based detection of concurrency. In *Proceedings of the ACM SIGSOFT '98 Symposium on the Foundations of Software Engineering*, pages 35–45, Orlando, FL, November 1998.

[CWA$^+$96] Edmund M. Clarke, Jeannette M. Wing, Rajeev Alur, Rance Cleaveland, David Dill, Allen Emerson, Stephen Garland, Steven German, John Guttag, Anthony Hall, Thomas Henzinger, Gerard Holzmann, Cliff Jones, Robert Kurshan, Nancy Leveson, Kenneth McMillan, J. Moore, Doron Peled, Amir Pnueli, John Rushby, Natarajan Shankar, Joseph Sifakis, Prasad Sistla, Bernhard Steffen, Pierre Wolper, Jim Woodcock, and Pamela Zave. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, White Plains, NY, June 20–22, 1990.

[Das00] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, Vancouver, BC, Canada, June 18–23, 2000.

[DB84] Douglas D. Dunlop and Victor R. Basili. A heuristic for deriving loop functions. *IEEE Transactions on Software Engineering*, 10(3):275–285, May 1984.

[DB85] Douglas D. Dunlop and Victor R. Basili. Generalizing specifications for uniformly implemented loops. *ACM Transactions on Programming Languages and Systems*, 7(1):137–158, January 1985.

[DC88] Henry G. Dietz and C. H. Chi. CRegs: A new kind of memory for referencing arrays and pointers. In *Proceedings of Supercomputing 1988*, pages 360–367, Orlando, Florida, November 1988.

[DC94] Matthew B. Dwyer and Lori A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of SIGSOFT '94 Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 62–75, New Orleans, LA, USA, December 6–9, 1994.

[Det96] David L. Detlefs. An overview of the Extended Static Checking system. In *Proceedings of the First Workshop on Formal Methods in Software Practice*, pages 1–9, January 1996.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

[DIS99] Claudio Demartini, Radu Iosif, and Riccardo Sisto. A deadlock detection tool for concurrent Java programs. *Software — Practice and Experience*, 29(7):577–603, June 1999.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, December 18, 1998.

[DPPA98] Srikrishna Devabhaktuni, Dain Petkov, Georgi Peev, and Saman Amarasinghe. Softspec: Software-based speculative parallelism via stride prediction, November 1998.

[Dro89] Geoff Dromey. *Program Derivation: The Development of Programs From Specifications.* Addison-Wesley, Sydney, Australia, 1989.

[DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, New York, 1990.

[DYKT89] Seiichiro Dan, Takahira Yamaguchi, Osamu Kakusho, and Yoshikazu Tezuka. Program verification system with synthesizer of invariant assertions. *Systems and Computers in Japan*, 20(1):1–13, 1989.

[ECGN99] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Pro-*

ceedings of the 21st International Conference on Software Engineering, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

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

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

[EDG95] Edison Design Group. *C++ Front End Internal Documentation*, version 2.28 edition, March 1995. http://www.edg.com.

[EGHT94] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Proceedings of SIGSOFT '94 Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 87–97, New Orleans, LA, USA, December 6–9, 1994.

[Els74] Bernard Elspas. The semiautomatic generation of inductive assertions for proving program correctness. Interim Report Project 2686, Stanford Research Institute, Menlo Park, CA, July 1974.

[EMW97] Michael D. Ernst, Todd D. Millstein, and Daniel S. Weld. Automatic SAT-compilation of planning problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1169–1176, Nagoya, Aichi, Japan, August 23–29, 1997.

[Ern99] Michael D. Ernst. Research summary for dynamic detection of program invariants. In *Proceedings of the 21st International Conference on Software Engineering*, pages 718–719, Los Angeles, CA, USA, May 19–21, 1999.

[Eva96] David Evans. Static detection of dynamic memory errors. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 44–53, Philadelphia, PA, May 21–24, 1996.

[Fla97] David Flanagan. *Java Examples in a Nutshell: A Tutorial Companion to Java in a Nutshell.* O'Reilly and Associates, Sebastopol, CA, September 1997.

[Fla99] David Flanagan. *Java in a Nutshell: A Desktop Quick Reference.* O'Reilly and Associates, Sebastopol, CA, 3rd edition, September 1999.

[FM97] Pascal Fradet and Daniel Le Métayer. Shape types. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 27–39, Paris, France, January 15–17, 1997.

[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.

[GC99] Gerald C. Gannod and Betty H. C. Cheng. A specification matching based approach to reverse engineering. In *Proceedings of the 21st International Conference on Software Engineering*, pages 389–398, Los Angeles, CA, USA, May 19–21, 1999.

[GCM$^+$94] David M. Gallagher, William Y. Chen, Scott A. Mahlke, John C. Gyllenhaal, and Wen-mei W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–193, San Jose, California, October 4–7, 1994.

[GH93] Michael Golan and David R. Hanson. DUEL — a very high-level debugging language. In *Proceedings of the Winter 1993 USENIX Conference*, pages 107–117, San Diego, California, USA, January 25–29, 1993.

[GH96] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–15, St. Petersburg Beach, FL, January 21–24, 1996.

[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.* Springer-Verlag, New York, NY, 1993.

[Giv96a] Robert Givan. Inferring program specifications in polynomial-time. In *Proceedings of the Third International Symposium on Static Analysis, SAS '96*, pages 205–219, Aachen, Germany, September 1996.

[Giv96b] Robert Lawrence Givan Jr. *Automatically Inferring Properties of Computer Programs.* PhD thesis, MIT, Cambridge, MA, June 1996.

[GKMS] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey Siy. Predicting fault incidence using software change history. http://www.bell-labs.com/org/11359/projects/decay/papers/.

[Goe85] Amrit L. Goel. Software reliability models: Assumptions, limitations, and applicability. *IEEE Transactions on Software Engineering*, SE-11(12):1411–1423, December 1985.

[Gri81] David Gries. *The Science of Programming.* Springer-Verlag, New York, 1981.

[GS98] Alfonso Gerevini and Lenhart K. Schubert. Inferring state constraints for domain-independent planning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 905–912, Madison, WI, July 26–30, 1998.

[Gup90] Rajiv Gupta. A fresh look at optimizing array bound checking. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 272–282, White Plains, NY, USA, June 1990.

[GW75] Steven M. German and Ben Wegbreit. A synthesizer of inductive assertions. *IEEE Transactions on Software Engineering*, 1(1):68–75, March 1975.

[Ham94] Dick Hamlet. Random testing. In *Encyclopedia of Software Engineering*. John Wiley and Sons, 1994.

[HEGV93] Laurie J. Hendren, Maryam Emami, Rakesh Ghiya, and Clark Verbrugge. A practical context-sensitive interprocedural alias analysis framework for C compilers. ACAPS Technical Memo 72, McGill University School of Computer Science, Advanced Compilers, Architectures, and Parallel Systems Group, Montreal, Quebec, July 24, 1993.

[HFGO94] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, pages 191–200, Sorrento, Italy, May 16–21, 1994.

[HHH$^+$87] C. A. R. Hoare, I. J. Hayes, Jifeng He, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Corrigenda: "Laws of programming". *Communications of the ACM*, 30(9):770, September 1987. See [HHJ$^+$87].

[HHJ$^+$87] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–686, August 1987. See corrigendum [HHH$^+$87].

[HHN92] Laurie J. Hendren, Joseph Hummel, and Alexandru Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 249–260, San Francisco, CA, June 17–19, 1992.

[Hir91] Haym Hirsh. Theoretical underpinnings of version spaces. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 665–670, Sydney, Australia, August 1991.

[HJ92] Reed Hastings and Bob Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In *Proceedings of the Winter 1992 USENIX Conference*, pages 125–138, San Francisco, California, January 20–24, 1992.

[HN90] Laurie J. Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Special Issue: Formal Methods in Software Practice.

[HP98] Michael Hind and Anthony Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *5th International Symposium on Program Analysis, SAS'98*, pages 57–81, September 1998.

[HP00] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[HRWY98] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. An empirical investigation of program spectra. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '98)*, pages 83–90, Montreal, Canada, June 16, 1998.

[HS90] Ben Heggy and Mary Lou Soffa. Architectural support for register allocation in the presence of aliasing. In *Proceedings of Supercomputing '90*, pages 730–739, November 1990.

[HS99] Daniel Hoffman and Paul Strooper. Tools and techniques for Java API testing, May 14, 1999.

[HSS94] Andrew S. Huang, Gert Slavenburg, and John Paul Shen. Speculative disambiguation: A compilation technique for dynamic memory disambiguation. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 200–210, Chicago, Illinois, April 18–21, 1994.

[HWF90] Robert Henry, Kenneth M. Whaley, and Bruce Forstall. The University of Washington Illustrating Compiler. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 223–246, White Plains, NY, USA, June 1990.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, 1993.

[JH98] Ralph Jeffords and Constance Heitmeyer. Automatic generation of state invariants from requirements specifications. In *Proceedings of the ACM SIGSOFT '98 Symposium on the Foundations of Software Engineering*, pages 56–69, Orlando, Florida, November 3–5, 1998.

[Jik] Jikes open-source Java compiler. http://oss.software.ibm.com/developerworks/opensource/jikes/.

[JM81] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of Lisp-like structures. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, Englewood Cliffs, N.J., 1981.

[JM82] Neil D. Jones and Steve S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 66–74, January 1982.

[Jon96] Richard W. M. Jones. A strategy for finding the optimal data placement for regular programs. Master's thesis, Department of Computing, Imperial College, 1996.

[JvH$^+$98] Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, Ulrich Hensel, and Hendrik Tews. Reasoning about Java classes. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, pages 329–340, Vancouver, BC, Canada, October 18–22, 1998.

[KF93] David Kurlander and Steven Feiner. Inferring constraints from multiple snapshots. *ACM Transactions on Graphics*, 12(4):227–304, October 1993.

[KHB99] Murat Karaorman, Urs Holzle, and John Bruno. jContractor: A reflective Java library to support design by contract. In *Reflection'99: The Second International Conference on Meta-Level Architectures and Reflection*, Saint Malo, France, July 19–21, 1999.

[KL86] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, 12(1):96–109, January 1986.

[Kle56] S. C. Kleene. Representation of events in nerve nets and finite automata. In Claude E. Shannon and John McCarthy, editors, *Automata Studies, Annals of Math. Studies 34*, pages 3–40. Princeton, New Jersey, 1956.

[KM76] Shmuel Katz and Zohar Manna. Logical analysis of programs. *Communications of the ACM*, 19(4):188–206, April 1976.

[KM97] Matt Kaufmann and J. Strother Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997.

[Kuc] Kuck & Associates. KAP/Pro toolset. http://www.kai.com.

[KW95] Priyadarshan Kolte and Michael Wolfe. Elimination of redundant array subscript range checks. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 270–278, La Jolla, CA, June 18–21, 1995.

[Lar99] C. E. Larson. Intelligent machinery and mathematical discovery. http://www.math.uh.edu/~clarson/, October 1, 1999.

[LB90] Kevin C. Lano and Peter T. Breuer. From programs to Z specifications. In John E. Nicholls, editor, *Z User Workshop: Proceedings of the Fourth Annual Z User Workshop, Oxford 1989*, Workshops in Computing, pages 46–70, Oxford, 1990. Springer-Verlag.

[LCKS90] Nancy G. Leveson, Stephen S. Cha, John C. Knight, and Timothy J. Shimeall. The use of self checks and voting in software error detection: An empirical study. *IEEE Transactions on Software Engineering*, 16(4):432–443, 1990.

[LDB$^+$99] Shih-Wei Liao, Amer Diwan, Robert P. Bosch, Jr., Anwar Ghuloum, and Monica S. Lam. SUIF Explorer: An interactive and interprocedural parallelizer. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'99)*, pages 37–48, Atlanta, Georgia, May 1999.

[LDW00] Tessa Lau, Pedro Domingos, and Daniel S. Weld. Version space algebra and its application to programming by demonstration. In *ICML*, Stanford, CA, June 2000.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* MIT Press, Cambridge, MA, 1986.

[LH88] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 21–34, Atlanta, Georgia, June 1988.

[LHS97] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. Query-based debugging of object-oriented programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 304–317, Atlanta, GA, USA, October 1997.

[LL93] Mark Leone and Peter Lee. Deferred compilation: The automation of run-time code generation. Technical Report CS-93-225, Carnegie Mellon University, School of Computer Science, December 1993.

[LN98] K. Rustan M. Leino and Greg Nelson. An extended static checker for Modula-3. In *Compiler Construction: 7th International Conference, CC'98*, pages 302–305, Lisbon, Portugal, April 1998.

[Lov86] László Lovász. *An Algorithmic Theory of Numbers, Graphs and Convexity.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1986.

[LR92] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, San Francisco, CA, June 1992.

[LSBZ87] Pat Langley, Herbert A. Simon, Gary L. Bradshaw, and Jan M. Zytkow. *Scientific Discovery: Computational Explorations of the Creative Process.* MIT Press, Cambridge, MA, 1987.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, Reading, MA, USA, 2nd edition, 1999.

[Mau] Mauve test suite for the Java class libraries. http://sourceware.cygnus.com/mauve/.

[McC76] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.

[McM93] Kenneth L. McMillan. *Symbolic model checking.* Kluwer Academic Publishers, 1993.

[MD97] Jon Meyer and Troy Downing. *Java Virtual Machine.* O'Reilly and Associates, Sebastopol, CA, 1997.

[Mit78] Tom M. Mitchell. *Version Spaces: An Approach to Concept Learning.* PhD thesis, Department of Computer Science, Stanford University, Stanford, CA, December 1978. Stanford University Technical Report, HPP-79-2.

[Mit97] Tom M. Mitchell. *Machine Learning.* McGraw-Hill Series in Computer Science. WCB/McGraw-Hill, Boston, MA, 1997.

[MP91] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer Verlag, New York, 1991.

[MP95] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[MW77] James H. Morris, Jr. and Ben Wegbreit. Subgoal induction. *Communications of the ACM*, 20(4):209–222, April 1977.

[NCOD97] Gleb Naumovich, Lori A. Clarke, Leon J. Osterweil, and Matthew B. Dwyer. Verification of concurrent software with FLAVERS. In *Proceedings of the 19th International Conference on Software Engineering*, pages 594–595, Boston, MA, May 1997.

[Nic89] Alexandru Nicolau. Run-time disambiguation: Coping with statically unpredictable dependencies. *IEEE Transactions on Computers*, 38(5):663–678, May 1989.

[NL98] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 333–344, Montreal, Canada, 17–19 June 1998.

[OC89] Mitsuru Ohba and Xiao-Mei Chou. Does imperfect debugging affect software reliability growth? In *Proceedings of the 11th International Conference on Software Engineering*, pages 237–244, Pittsburgh, PA, USA, May 15–18, 1989.

[O'C99a] Robert O'Callahan. The design of program analysis services. Technical Report CMU-CS-99-135, Carnegie Mellon University School of Computer Science, Pittsburgh, PA, June 1999.

[O'C99b] Robert O'Callahan. Optimizing a solver of polymorphism constraints: SEMI. Technical Report CMU-CS-99-135, Carnegie Mellon University School of Computer Science, Pittsburgh, PA, June 1999.

[OHL$^+$97] Jeffrey Oplinger, David Heine, Shih-Wei Liao, Basem A. Nayfeh, Monica S. Lam, and Kunle Olukotun. Software and hardware for exploiting speculative parallelism with a multiprocessor. Technical Report CSL-TR-97-715, Stanford University, Computer Systems Laboratory, February 1997.

[OJ96] Robert O'Callahan and Daniel Jackson. Practical program understanding with type inference. Technical Report CMU-CS-96-130, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1996.

[OJ97] Robert O'Callahan and Daniel Jackson. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 19th International Conference on Software Engineering*, pages 338–348, Boston, MA, May 1997.

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

[ORSvH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995. Special Section—Best Papers of FME (Formal Methods Europe) '93.

[Pal95] Jens Palsberg. Comparing flow-based binding-time analyses. In *TAPSOFT '95: Theory and Practice of Software Development*, pages 561–574, Aarhus, Denmark, May 22–26, 1995.

[Pfe92] Frank Pfenning. Dependent types in logic programming. In Frank Pfenning, editor, *Types in Logic Programming*, chapter 10, pages 285–311. MIT Press, Cambridge, MA, 1992.

[PRE99] PREfix/Enterprise. http://www.intrinsa.com/, 1999.

[QCJ96] Ross Quinlan and Mike Cameron-Jones. FOIL 6.4. ftp://ftp.cs.su.oz.au/pub/, February 5, 1996.

[Qui90] J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[RBDL97] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the Sixth European Software Engineering Conference and Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 97)*, pages 432–449, Zurich, Switzerland, September 22–25, 1997.

[RH98] Gregg Rothermel and Mary Jean Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, 24(6):401–419, June 1998.

[RN95] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[Ros95] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.

[Ruf95] Erik Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–22, La Jolla, CA, June 1995.

[Rul98] RuleQuest Research. Data mining with Cubist. http://www.rulequest.com/cubist-info.html, 1998.

[Rus99] John Rushby. Integrated formal verification: Using model checking with automated abstraction, invariant generation, and theorem proving. In *Fifth International SPIN Workshop*, Trento, Italy, June 1999.

[RVL$^+$97] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and J. Bradley Chen. Instrumentation and optimization of Win32/Intel executables using Etch. In *The USENIX Windows NT Workshop*, pages 1–7, Seattle, WA, August 11–13, 1997. USENIX.

[RW88a] Charles Rich and Richard C. Waters. Automatic programming: Myths and prospects. *IEEE Computer*, 21(8):40–51, August 1988.

[RW88b] Charles Rich and Richard C. Waters. The Programmer's Apprentice: A research overview. *IEEE Computer*, 21(11):10–25, November 1988.

[Sal68] Gerard Salton. *Automatic Information Organization and Retrieval.* McGraw-Hill, 1968.

[SBN$^+$97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, pages 27–37, October 5–8, 1997.

[SE94] Amitabh Srivastava and Alan Eustace. ATOM — a system for building customized program analysis tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, Orlando, FL, USA, June 1994.

[SH97] Marc Shapiro and Susan Horwitz. The effects of precision on pointer analysis. In *Proceedings of the Fourth International Symposium on Static Analysis, SAS '97*, pages 16–34, Paris, France, September 8-10, 1997.

[Sha81] Ehud Y. Shapiro. An algorithm that infers theories from facts. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence (IJCAI-81)*, pages 446–451, Vancouver, BC, August 1981.

[Sha82] Ehud Y. Shapiro. Algorithmic program diagnosis. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 299–308. ACM, ACM, January 1982.

[Sha83] Ehud Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, Cambridge, MA, 1983.

[SHZ$^+$94] Bogong Su, Stanley Habib, Wei Zhao, Jian Wang, and Youfeng Wu. A study of pointer aliasing for software pipelining using run-time disambiguation. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-97)*, pages 112–117, San Jose, CA, November 30–December 2, 1994.

[SI77] Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 132–143, Los Angeles, CA, January 17–19, 1977.

[SM97] J. Gregory Steffan and Todd C. Mowry. The potential for thread-level data speculation in tightly-coupled multiprocessors. Technical Report SCRI-TR-350, Department of

Electrical and Computer Engineering, University of Toronto, Toronto, Canada, February 1997.

[Spi88] J. M. Spivey. *Understanding Z.* Cambridge Tracts in Theoretical Computer Science 3. Cambridge University Press, 1988. ISBN 0-521-33429-2.

[SRW99] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 105–118, San Antonio, TX, January 20–22, 1999.

[SS98] Avinash Sodani and Gurindar S. Sohi. An empirical analysis of instruction repetition. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 35–45, San Jose, CA, October 4–7, 1998.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg Beach, FL, January 21–24, 1996.

[Tan94] Yang Meng Tan. Formal specification techniques for promoting software modularity, enhancing documentation, and testing specifications. Technical Report MIT-LCS//MIT/LCS/TR-619, Massachusetts Institute of Technology, Laboratory for Computer Science, June 1994.

[van97] Guido van Rossum. *Python Reference Manual*, release 1.5 edition, December 31, 1997.

[VH98] Mandana Vaziri and Gerard Holzmann. Automatic detection of invariants in Spin. In *SPIN98: Papers from the 4th International SPIN Workshop*, nov 1998.

[VP98] Raúl E. Valdés-Pérez. Why some machines do science well. In *International Congress on Discovery and Creativity*, Ghent, Belgium, 1998.

[War96] Martin P. Ward. Program analysis by formal transformation. *The Computer Journal*, 39(7):598–618, 1996.

[WB97] Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, November 1997.

[WCM89] Martin Ward, Frank W. Calliss, and Malcolm Munro. The Maintainer's Assistant. In *Proceedings of the International Conference on Software Maintenance*, pages 307–315, Miami, FL, 1989.

[Weg74] Ben Wegbreit. The synthesis of loop predicates. *Communications of the ACM*, 17(2):102–112, February 1974.

[Wei80] William E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, January 1980.

[Wei99] Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java.* Addison Wesley Longman, 1999.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, La Jolla, CA, June 1995.

[WS76] Ben Wegbreit and Jay M. Spitzen. Proving properties of complex data structures. *Journal of the ACM*, 23(2):389–396, April 1976.

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, Canada, 17–19 June 1998.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, TX, January 1999.

[Zen97] Christoph Zenger. Indexed types. *Theoretical Computer Science*, 187:147–165, 1997.

Vita

Michael D. Ernst holds the S.B. and S.M. degrees from MIT, and the M.S. degree from the University of Washington. He has previously been a full-time lecturer at Rice University and a researcher at Microsoft Research. His primary technical interest is programmer productivity, encompassing software engineering, program analysis, compilation, and programming language design. However, he has also published in artificial intelligence, theory, and other areas of computer science.