Dynamically Discovering Likely Program Invariants to Support Program Evolution

Michael D. Ernst, Jake Cockrell, William G. Griswold, *Member*, *IEEE*, and David Notkin, *Member*, *IEEE Computer Society*

Abstract—Explicitly stated program invariants can help programmers by identifying program properties that must be preserved when modifying code. In practice, however, these invariants are usually implicit. 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. This article reports three results. First, it describes techniques for dynamically discovering invariants, along with an implementation, named Daikon, that embodies these techniques. Second, it reports on the application of Daikon to two sets of target programs. In programs from Gries's work on program derivation, the system rediscovered predefined invariants. In a C program lacking explicit invariants, the system discovered invariants that assisted a software evolution task. These experiments demonstrate that, at least for small programs, invariant inference is both accurate and useful. Third, it analyzes scalability issues, such as invariant detection runtime and accuracy, as functions of test suites and program points instrumented.

Index Terms—Program invariants, formal specification, software evolution, dynamic analysis, execution traces, logical inference, pattern recognition.

1 Introduction

Invariants play a central role in program development. Representative uses include refining a specification into a correct program, statically verifying properties such as type declarations, and runtime checking of invariants encoded as `assert` statements.

Invariants play an equally critical role in software evolution. In particular, 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 alternative to expecting programmers to annotate code with invariants is to automatically infer invariants. 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. Fig. 1 shows the architecture of the Daikon invariant detector. As with other dynamic approaches, such as testing and profiling, the accuracy of the inferred invariants depends in part on the quality and completeness of the test cases; additional test cases might provide new data from which more accurate invariants can be inferred.

The inference of invariants from program traces and its application to software evolution raises a number of technical questions. How can invariants be detected? Can the inference process be made fast enough? What kind of test suite is required to infer meaningful invariants? What techniques can be used to minimize irrelevant invariants that are unlikely to aid a programmer in the task at hand? How can the required information be extracted from program runs? Can programmers productively use the inferred invariants in software evolution? This article provides partial answers to these questions in the form of three results stemming from our initial experiences with this approach.

The first result is a set of techniques for discovering invariants from execution traces and a prototype invariant detector, Daikon, that implements these techniques. Invariants are detected from program executions by instrumenting the source program to trace the variables of interest, running the instrumented program over a set of test cases, and inferring invariants over both the instrumented variables and over derived variables that are not manifest in the original program. The essential idea is to test a set of possible invariants against the values captured from the instrumented variables; those invariants that are tested to a sufficient degree without falsification are reported to the programmer. Section 3 discusses the invariant detection engine; the discussion of instrumentation is deferred to Section 8.

The second result is the application of Daikon to two sets of target programs. The first set of programs appear in *The Science of Programming* [39]. These programs were derived from formal preconditions, postconditions, and loop invariants. Given runs of the program over randomly-generated

• *M.D. Ernst is with the Department of Electrical Engineering and Computer Science and the Laboratory for computer Science, Massachusetts Institute of Technology, 77 Massachusetts Ave., Cambridge, MA 02139. E-mail: mernst@lcs.mit.edu.*
• *J. Cockrell is with Macromedia, Inc., 101 Redwood Shores Parkway, Redwood City, CA 94065. E-mail: jcockrell@macromedia.com.*
• *W.G. Griswold is with the Department of Computer Science and Engineering, University of California, San Diego, 0114, La Jolla, CA 92093-0114. E-mail: wgg@cs.ucsd.edu.*
• *D. Notkin is with the Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle WA 98195-2350. E-mail: notkin@cs.washington.edu.*

Manuscript received 15 Nov. 1999; revised 21 Apr. 2000; accepted 1 May 2000.
Recommended for acceptance by D. Garlan.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 112142.

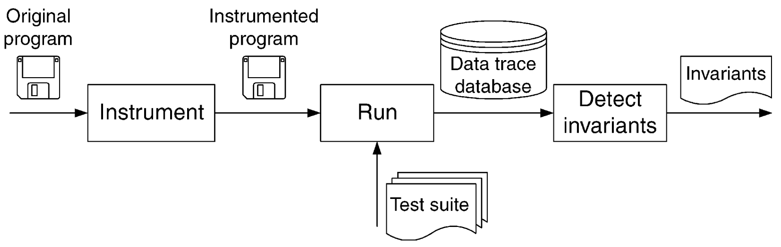

Fig. 1. Architecture of the Daikon tool for dynamic invariant inference.

inputs, Daikon discovers those same program properties, plus some additional ones (we introduce this result as motivation in Section 2). This first experiment demonstrates that dynamic invariant detection produces invariants that are accurate. The second set of programs—C programs, originally from Siemens [43] and modified by Rothermel and Harrold [72]—is not annotated with invariants, nor is there any indication that invariants were used explicitly in their construction. Section 4 shows how numeric invariants dynamically inferred from one of these programs assisted in understanding and changing it. This scenario also shows that dynamic invariant discovery is complementary to static techniques (which examine the program text but do not run the program). This second experiment demonstrates that dynamic invariant detection produces invariants that are useful.

The third result, presented in Section 5, is a quantitative analysis of scalability issues. The analysis demonstrates that inference running time is linearly correlated to the number of program points being traced, the square of the number of variables in scope at a program point, and the size of the test suite. Thus, choices of program points and variables over which to detect invariants can control invariant detection time. While there are many potential invariants, most of them are quickly falsified, contributing little to overall runtime. Experiments on test suite selection suggest that the set of invariants inferred tends to stabilize with growing test suite size, reducing the need for large test suites and, thus, limiting inference time. Section 6 correlates the number of invariants with program correctness. Section 7 discusses some initial work concerning the adequacy of automatically generated test suites for invariant inference.

Finally, Section 9 surveys related work, Section 10 discusses ongoing and future work, and Section 11 concludes the paper.

2 Rediscovery of Invariants

To introduce dynamic invariant detection and illustrate Daikon's output, we present the invariants detected in a simple program taken from *The Science of Programming* [39], 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 the exercise 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, computing 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 in this article, along with the complete Daikon output. Additionally, they illustrate a number of important issues in invariant detection.

As a simple example of invariant detection, consider a program that sums the elements of an array (Fig. 2). We transliterated this program to a dialect of Lisp enhanced with Gries-style control constructs such as nondeterministic conditionals. Daikon's Lisp instrumenter (Section 8) 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 (`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. Fig. 3 shows the output of the Daikon invariant detector given the data trace file.

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

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

Fig. 2. Gries array sum program (Program 15.1.1 [39, 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.

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

Fig. 3. Invariants inferred for the Gries array sum program (Fig. 2) over 100 randomly generated input arrays. Invariants are shown for the entry (precondition) and exit (postcondition) of the program, as well as the loop head (loop invariant). Daikon successfully rediscovered the invariants in the program's formal specification (Fig. 2); those goal invariants are boxed for emphasis. `B[ -1]` is shorthand for `B[ size(B)-1]`, the last element of array `B`, and `orig(`*var*`)` represents *var*'s value at the start of procedure execution. Invariants for elements of an array are listed indented under the array; in this example, no array has multiple elementwise invariants. The number of samples in the right-hand column is the number of times each program point was executed; the loop iterates multiple times for each test case, generating multiple samples. The counts of values, also 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 S = sum(B) indicates that variable `S` (and `sum(B)`) had only 96 distinct final values on those 100 executions.

This is neither the best nor most realistic test suite; it happens to be the first one we tried when testing Daikon. The results illustrate potential shortcomings of the approach and motivate improvements that handle them. Fig. 4 shows Daikon's output when the array sum program is run over a different test suite. Sections 3 and 7 discuss the selection of test suites.

The preconditions (invariants at the `ENTER` program point) of Fig. 3 record that `N` is the length of array `B`, that `N` falls between 7 and 13 inclusive, and that the array elements are always at least -100. The first invariant, $\mathrm{N} = \mathrm{size(B)}$, is crucial to the correctness of the program, yet was omitted from the formal invariants stated by Gries. Gries's stated

```
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)
```

Fig. 4. Invariants inferred for the Gries array sum program (Fig. 2) over an input set whose 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 Fig. 3. Invariants in Fig. 3 that were specific to that test suite do not appear in this output.

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

The postconditions (at the `EXIT` program point) include the Gries postcondition, $\mathrm{S} = \mathrm{sum(B)}$; Section 3.2 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. One of these additional invariants bounds the maximum value of the array elements, in complement to the minimum value noted in the precondition and postcondition invariants. Section 3.1 discusses why it is reported as a loop invariant but not in the preconditions or postconditions and [27] shows how to eliminate such invarients.

In Fig. 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 $\mathrm{N} = \mathrm{size(B)}$. Second are properties of the test suite, such as $\mathrm{N} \in [7..13]$. These invariants provide valuable information about the data set and can help validate a test suite or indicate the usage context of a function or other computation. Third are extraneous, probably uninteresting invariants, such as $\mathrm{N} \neq \mathrm{B}[-1]$, which are further discussed in Section 10.1 and eliminated by [27].

In this example, Daikon detected $\mathrm{N} = \mathrm{size(B)}$ 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 $\mathrm{s} = (\sum \mathrm{j} : 0 \leq \mathrm{j} < \mathrm{size(B)} : \mathrm{b[j]})$. The same code could be used in a different way, to sum part of an array with precondition $\mathrm{N} \leq \mathrm{size}(\mathrm{B})$ 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 article, static analysis could produce the same result. However, static analysis cannot report true but undecidable properties or properties of the program context. Furthermore, static analysis of language features such as pointers remains beyond the state of the art because of the difficulty of representing the heap, which forces precision-losing approximations. Dynamic analysis does not suffer these drawbacks, so it complements static analysis.) Section 4 shows Daikon's application to a more realistic program that was constructed without the use of formal invariants. Before that, however, Section 3 describes how Daikon operates.

3 Inferring Invariants

There are two principal challenges to inferring the invariants presented in the previous section: choosing what invariants to infer and performing the inference. A third challenge, capturing the program's behavior for inference, is discussed Section 8.

Daikon infers invariants at specific program points such as procedure entries and exits and, optionally, loop heads. The instrumented program provides Daikon, for each execution of such a program point, with the values of variables in scope. Daikon checks for invariants involving a single variable (a constraint that holds over its values) or multiple variables (a relationship among the values of the variables). 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: $\mathrm{x} = \mathrm{a}$ indicates the variable is a constant.
  - Uninitialized: $\mathrm{x} = \mathrm{uninit}$ indicates the variable is never set.
  - Small value set: $\mathrm{x} \in \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$ indicates the variable takes on only a small number of different values.
- Invariants over a single numeric variable:
  - Range limits: $\mathrm{x} \geq \mathrm{a}$, $\mathrm{x} \leq \mathrm{b}$, and $\mathrm{a} \leq \mathrm{x} \leq \mathrm{b}$ (printed as `x in [a..b]`) indicate the minimum and/or maximum value.
  - Nonzero: $\mathrm{x} \neq 0$ indicates the variable is never set to 0; see Section 3.1 for details on when such an invariant is reported.
  - Modulus: $\mathrm{x} \equiv \mathrm{a} \pmod{\mathrm{b}}$ indicates that $x \bmod b = a$ always holds.
  - Nonmodulus: $\mathrm{x} \not\equiv \mathrm{a} \pmod{\mathrm{b}}$ is reported only if $x \bmod b$ takes on every value besides a.
- Invariants over two numeric variables:
  - Linear relationship: $\mathrm{y} = \mathrm{ax} + \mathrm{b}$.
  - Ordering comparison: $\mathrm{x} < \mathrm{y}$, $\mathrm{x} \leq \mathrm{y}$, $\mathrm{x} > \mathrm{y}$, $\mathrm{x} \geq \mathrm{y}$, $\mathrm{x} = \mathrm{y}$, $\mathrm{x} \neq \mathrm{y}$.
  - Functions: $\mathrm{y} = \mathrm{fn}(\mathrm{x})$ or $\mathrm{x} = \mathrm{fn}(\mathrm{y})$, for *fn* one of Python's built-in unary functions (absolute value, negation, bitwise complement); additional functions are trivial to add.
  - Invariants over $x + y$: Any invariant from the list of invariants over a single numeric variable, such as $\mathrm{x} + \mathrm{y} \equiv \mathrm{a} \pmod{\mathrm{b}}$.
  - Invariants over $x - y$: As for $x + y$; this subsumes ordering comparisons and can permit inference of properties such as $\mathrm{x} - \mathrm{y} > \mathrm{a}$, which Daikon prints as `x > y + a`.
- Invariants over three numeric variables:
  - Linear relationship: $\mathrm{z} = \mathrm{ax} + \mathrm{by} + \mathrm{c}$, $\mathrm{y} = \mathrm{ax} + \mathrm{bz} + \mathrm{c}$, or $\mathrm{x} = \mathrm{ay} + \mathrm{bz} + \mathrm{c}$.
  - Functions: $\mathrm{z} = \mathrm{fn}(\mathrm{x}, \mathrm{y})$, for *fn* one of Python's built-in binary functions (min, max, multiplication, and, or, greatest common divisor; comparison, exponentiation, floating point rounding, division, modulus, left and right shifts); additional functions are trivial to add. The other permutations of $\langle x, y, z \rangle$ are also tested (three permutations for symmetric functions, listed before the parenthesis's semicolon, and six permutations for nonsymmetric functions).
- 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 nondecreasing, nonincreasing, 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 Fig. 3, all elements of array `B` are at least -100.

  The sum invariants of Fig. 3 do not appear here because `sum(B)` is a derived variable, which is described in Section 3.2.
- Invariants over two sequence variables:
  - Linear relationship: $\mathrm{y} = \mathrm{ax} + \mathrm{b}$, elementwise.
  - Comparison: $\mathrm{x} < \mathrm{y}$, $\mathrm{x} \leq \mathrm{y}$, $\mathrm{x} > \mathrm{y}$, $\mathrm{x} \geq \mathrm{y}$, $\mathrm{x} = \mathrm{y}$, $\mathrm{x} \neq \mathrm{y}$, performed lexicographically.
  - Subsequence relationship: x is a subsequence of y or vice versa.
  - Reversal: x is the reverse of y.
- Invariants over a sequence and a numeric variable:
  - Membership: $\mathrm{i} \in \mathrm{s}$.

For each variable or tuple of variables, 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). Thus, the cost of computing invariants tends to be proportional to the number of invariants discovered (see also Section 5).

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):

```
⟨  orig(x),     orig(*x),  orig(y),   x,           *x,  y,  return  ⟩
⟨  4026527180,  2,         1,         4026527180,  3,   1,  3       ⟩
⟨  146204,      13,        1,         146204,      14,  1,  14      ⟩
⟨  4026527180,  3,         1,         4026527180,  4,   1,  4       ⟩
⟨  4026527180,  4,         1,         4026527180,  5,   1,  5       ⟩
⟨  146204,      14,        1,         146204,      15,  1,  15      ⟩
⟨  4026527180,  5,         1,         4026527180,  6,   1,  6       ⟩
⟨  4026527180,  6,         1,         4026527180,  7,   1,  7       ⟩
                                   ⋮
```

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

The invariants listed above are inexpensive to test and do not require full-fledged theorem proving. For example, the linear relationship $\mathrm{x} = \mathrm{ay} + \mathrm{bz} + \mathrm{c}$ with unknown coefficients a, b, and c and variables x, y, and z has three degrees of freedom. Consequently, three (linearly independent) tuples of (x, y, z) values are sufficient to determine the coefficients, after which checking requires only a few arithmetic operations and an equality check. As another example of inexpensive checking, a common modulus (variable b in $\mathrm{x} \equiv \mathrm{a} \pmod{\mathrm{b}}$) is the greatest common divisor of the differences among list elements.

To reduce source language dependence, simplify the implementation, and improve error checking, Daikon supports only two forms of data: scalar number (including characters and booleans) and sequence of scalars; 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 *i*th element in each. For example, $\mathrm{a[i].left} < \mathrm{a[i].right}$ is reported as $\mathrm{a.left[i]} < \mathrm{a.right[i]}$, which a post-processing step could easily convert to the former representation by referring to the original program type declarations.

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 and that we believed would be generally useful; 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, Daikon does not yet follow arbitrary-length paths through recursive data structures (see Section 10 and [30]). Nor does Daikon compute invariants such as a linear relationship over four variables, nor test every data structure for the red-black tree invariant. Omitting such invariants controls cost and complexity: Section 5 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 domain-specific invariants and derived variables (Section 3.2) by writing a small amount of code.) Even the current list is useful: It enabled the successful detection of the Gries invariants and useful invariants in the Siemens suite (Section 4).

Invariants can be viewed as forming a lattice based on subsumption (logical implication). The implementation takes advantage of these relationships in order to improve both performance and the intelligibility of the output (see Section 10.1). Perhaps some additional advantage could be gained by further formalizing this lattice.

3.1 Invariant Confidence

Not all unfalsified invariants should be reported. If there are few unique value tuples at a program point (because the program point is executed few times or is frequently executed with the same variable values), then relationships over those few distinct variable values may be mere coincidences, even though the properties always held on the test runs. Reporting too many spurious invariants could discourage programmers from looking through the list for better-supported invariants.

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. Because generating ideal test suites is difficult (see also Sections 5.1.3 and 7) and to improve invariant detection output even for deficient test suites, Daikon includes a method for computing invariant confidences.

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 is considered noncoincidental 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 noncoincidental and their unusual property is worth reporting. (This probability limit is not a confidence on the likelihood that the reported invariants are correct over all possible inputs; rather, it is used to decide whether a particular invariant is worth reporting to the user.)

For the purposes of this article—in part to demonstrate spurious invariants like those of Fig. 3—we set the probability limit to 0.01, to report invariants that are no more than 1 percent likely to have occurred by chance. For actual use, we recommend a substantially smaller value: If the system checks millions of potential invariants, then reporting thousands of spurious invariants is likely to be unacceptable.

As a concrete example of a statistical test, suppose the reported values for variable x fall in a range of size r that includes 0 (suppose x ranges from $\frac{r}{2}$ to $-\frac{r}{2}-1$), 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 level, then the invariant $\mathrm{x} \neq 0$ is reported. Tests for $\mathrm{x} \neq \mathrm{y}$ and (non)modulus tests are analogous. As another example, ranges for numeric variables (such as $\mathrm{c} \in [32..126]$ or $\mathrm{x} > 0$) are also not reported unless they appear to be noncoincidental. A limit is reported if the several values near the range's extrema all appear about as often as would be expected (the distribution appears to be uniform and stops at the observed minimum or maximum), or if the extremum appears much more often than would be expected (as if greater or lesser values have been clipped to that value).

The 100 random arrays used in the experiment of Fig. 3 happened to support only one boundedness inference (all elements ≥ -100). On a second run, over 100 arrays selected from the same distribution, both bounds were inferred and, for larger test suites, both bounds were always inferred. Fig. 4 shows the result of running Daikon on a different set of 100 arrays; the output is almost precisely the Gries invariants.

In Figs. 3 and 4, some invariants are reported at the loop head, but not at the procedure entry or exit, even though the same array values were visible at all program points. The reason is that 100 samples were insufficient to support any inequality inferences, but the loop head is executed more times. We have subsequently enhanced our implementation to record whether each variable has been set since the last time the program point was encountered; counting only the first occurrence of a particular variable value eliminates all the extra loop invariants from Figs. 3 and 4. Details are reported in [27].

3.2 Derived Variables

Computing invariants over manifest program variables can be inadequate for a programmer's needs. For instance, if array `a` and integer `lasti` are both in scope, then `a[lasti]` may be of interest, even though that expression is not a source code variable and may not appear in the program text.

Therefore, Daikon adds certain "derived variables" (actually expressions) to the list it is given as input. These 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).
  - Subsequences: `s[0..i]`, `s[0..i-1]`, where the notation `s[a..b]` indicates the portion of `s` spanning indices `a` to `b`, inclusive. As in the above case, two subsequences 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.

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 $\mathrm{i} < \mathrm{size(A)}$ 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.

Variable derivation and invariant inference can also avoid unnecessary work by examining previously-computed invariants. Therefore, 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 $\mathrm{j} \geq \mathrm{size(A)}$, 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.

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.

4 USE OF INVARIANTS

As discussed in Section 2, dynamic invariant detection accurately rediscovered the known invariants for the Gries programs. This section reports on a second experiment that indicates that inferred invariants can be 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 (from the Siemens suite [43] as modified by Rothermel and Harrold [72]) using both traditional tools and invariants produced by the prototype invariant detector Daikon.

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.

4.1 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 regular expression language of replace includes Kleene-* closure [55] 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 test cases randomly selected from the 5,542 provided with the Siemens suite. 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.

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

Fig. 5. Function makepat's use of constant CLOSURE in Siemens program replace.

4.2 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 $\langle pat \rangle$+ into the semantically equivalent $\langle pat \rangle \langle pat \rangle$*.

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 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 (Fig. 5) determines whether the next character in the input is CLOSURE; if so, it calls the "star closure" function, stclose (Fig. 6), 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 conditions in stclose's loop imply the loop would not be entered if

```
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;
}
```

Fig. 6. Function `stclose` in Siemens program `replace`. This was the template for the new `plclose` function (Fig. 8).

`*j` were equal to `lastj`, so they examined the invariants inferred for those variables on entry to `stclose`:

$$*\mathrm{j} \geq 2$$
$$\mathrm{lastj} \geq 0$$
$$\mathrm{lastj} \leq *\mathrm{j}.$$

The third invariant implies that the loop body might not be executed (if $\mathrm{lastj} = *\mathrm{j}$, 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 $\mathrm{lastj} = *\mathrm{j}$, 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 5,542 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` (Fig. 6) more carefully, they observed that the loop index is decremented and `pat` is both read and written by `addstr` (Fig. 7). 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 $\mathrm{pat} \neq \mathrm{orig(pat)}$, 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`"
    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. (One of the authors independently discovered this fact through careful study of the program text.) 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 Fig. 8.

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 $*\mathrm{j} = \mathrm{orig}(*\mathrm{j}) + 1$, `plclose` has the invariant $*\mathrm{j} \geq \mathrm{orig}(*\mathrm{j}) + 2$. 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.

```
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;
}
```

Fig. 7. Function `addstr` in Siemens program `replace`.

4.3 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 Fig. 5) 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 in order 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

$$\text{calls}(\text{in\_set\_2}) = \text{calls}(\text{stclose}).$$

Since `in_set_2` is only called in the predicate controlling `stclose`'s invocation (see Fig. 5), 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 (see also Section 5).

4.4 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 $\text{lastj} < \text{j}$ and $\text{lj} < \text{j}$. 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 $\text{lastj} < *\text{j}$ (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

```
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);
    }
}
```

Fig. 8. Function `plclose` in the extended `replace` program. It was written by copying `stclose` (Fig. 6), then modifying the copy.

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.

4.5 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. However, adding inferred invariants to these techniques provides several qualitative benefits.

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 zero value and is regularly incremented by one 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, the inferred invariants contradicted expectations regarding the preconditions for `stclose` and clarifying information was provided by supporting data. This both revealed a bug and simplified an implementation. The other tool, an invariant comparator, reveals how two sets of invariants differ, enabling comparison of 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 both by concisely summarizing the program trace data and providing a means for querying the trace database for additional insight.

5 Scalability

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. However, the cost of inference is hard to predict. 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 in which variable value tuples are examined. Finally, selection of test cases—both how many and which ones—impact what invariants are discovered. This section presents the results of several experiments to determine the costs of invariant inference (Section 5.1) and the stability of the reported invariants as the test suite increases in size (Section 5.2). Based largely on the results of these experiments, Section 10 suggests ways to accelerate inference, improve scalability, and manage the reporting of invariants.

5.1 Performance

To gain insight on performance-related scalability issues, we measured invariant detection runtime over the Siemens `replace` program [43], [72]. We aimed to identify quantitative, observable factors that a user can control to manage the time and space requirements of the invariant detector.

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 count 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 5.1.1 describes the experimental methodology. Section 5.1.2 reports how the number of variables in scope at an instrumented program point affects invariant detection time and Section 5.1.3 reports how the number of test cases (program runs) affects invariant detection time. Section 5.1.4 considers how other factors affect invariant detection time. Because each instrumented program point is processed independently, program size affects invariant detection time only insofar as larger programs afford more instrumentation points and more global variables. This implies that analysis of a portion of a large program is no more difficult than complete analysis of a smaller program.

5.1.1 Methodology

We instrumented and ran the Siemens `replace` program on subsets of the 5,542 test cases supplied with the program, including runs over 500, 1,000, 1,500, 2,000, 2,500, and 3,000 randomly-chosen test inputs, where each set is a subset of the next larger one. We also ran over all 5,542 test cases, but our initial prototype implementation ran out of memory, exceeding 180MB, for one program point over 3,500 inputs and for a second program point over 4,500 inputs. (The `replace` program has 21 procedures (42 instrumentation points), but one of the routines, which performs error handling, was never invoked, so we omit it henceforth.) 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 4.2.

Daikon infers invariants over an average of 71 variables (6 original, 65 derived; 52 scalars, 19 sequences) per instrumentation point in `replace`. On average, 1,000 test cases produce 10,120 samples per instrumentation point and the current implementation of Daikon takes 220 seconds to infer the invariants for an average instrumentation point. For 3,000 test cases, there are 33,801 samples and processing takes 540 seconds.

We ran the experiments on a 450MHz Pentium II. Daikon is written in the interpreted language Python [79]. 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.

5.1.2 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 grow very slowly with program size, as more global variables are introduced. On average, each of the 20 functions in `replace` has three parameters (two pointers and one scalar), but those translate to five 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 3.2. 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.

Fig. 9 plots growth in invariant detection time against growth in number of variables. Each data point of Fig. 9 compares invariant detection times for two sets of variables at every procedure exit in `replace` using a 1,000-element

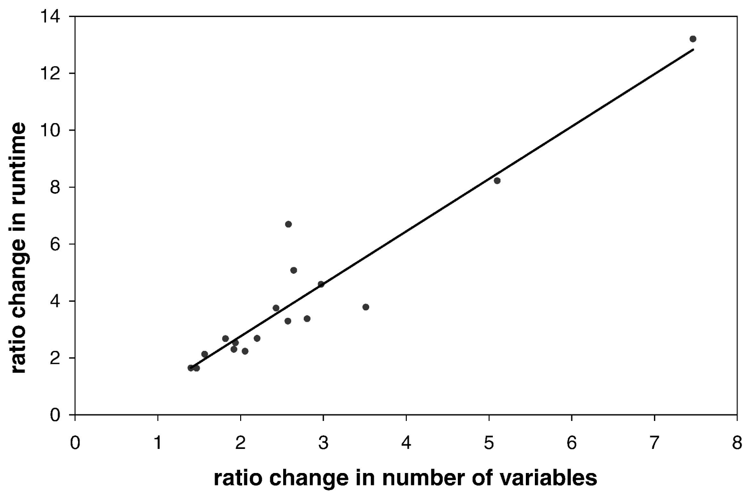

Fig. 9. Change in invariant detection runtime versus change in number of variables. A least-squares trend line highlights the relationship; its R^2 value is over 0.89, indicating good fit. Each data point compares inference over two different sets of variables at a single instrumentation point, for invariant inference over 1,000 program runs. (For 3,000 test cases, the graph is similar, also with $R^2 = 0.89$.) 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 $y = 1.8x - 0.92$, 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.

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 1.4 to 7.5 times as large as the smaller one; this is the range of the x axis of Fig. 9. The absolute number of variables ranges from 14 to 230. This choice of 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 distributions), and induce different invariants; our goal is to measure only the effect of number of variables.

Fig. 9 indicates that 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 1,000 test cases of Fig. 9, 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 3,000 inputs. The results were nearly identical to those for 1,000 test cases: The ratios closely fitted ($R^2 = 0.89$) a straight line with slope 2.1.

Fig. 9 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 2,100 seconds.

5.1.3 Test Suite Size

The effect of test suite size on invariant detection runtime is less pronounced than the effect of number of variables. Fig. 10 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 0.99, nine others have R^2 above 0.9, and five more have R^2 above 0.85. 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, we have not yet isolated a cause for these departures from linearity. We are in the midst of reimplementing Daikon and plan to repeat the experiment with the new implementation to see whether these aberrations remain.

Although runtime is (for the most part) linearly related to test suite size, the divergent lines of Fig. 10 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.

5.1.4 Other Factors

We compared a large number of factors in an attempt to find formulas relating them. Our hope was to relate runtime directly to factors under the user's control, such as number of test cases, so that users can predict invariant detection runtime.

The best single predictor for invariant detection runtime is the number of pairs of values encountered by the invariant detector; Fig. 11 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),

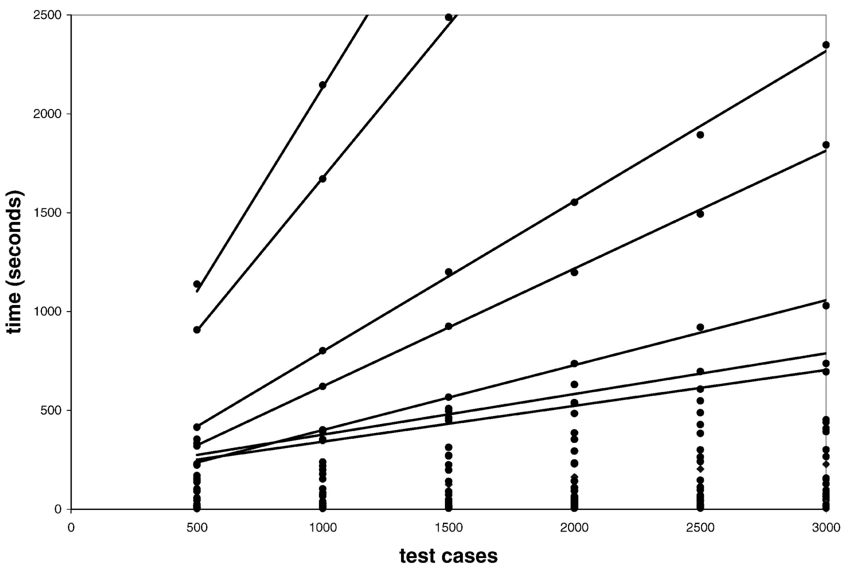

Fig. 10. Invariant detection runtime as a function of number of test cases (program runs). The plot contains one data point for each program point and test suite size—six data points per program point. Lines are drawn through some of these data sets to highlight the growth of runtime as test suite size increases.

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.

5.2 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 5.1.3.

To explore what test suite size is desirable for invariant inference, we compared, pairwise, the invariants detected on `replace` for different numbers of randomly selected test cases. Figs. 12 and 13 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, over program variables, fields, and derived variables of the forms listed in Section 3.2. 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 3.1) and in certain other circumstances (see Section 10.1 and [31]). All comparisons of invariants are of reported invariants, which is the output the user sees.

Figs. 12 and 13 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 of possible values (called "small value set" in Section 3); 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 1,000 to one of size 3,000:

```
1,000 tests:  s1 >= 0          (96 values)
3,000 tests:  s1 in [0..98]    (99 values)

1,000 tests:  i in [0..92]     (73 values)
3,000 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.

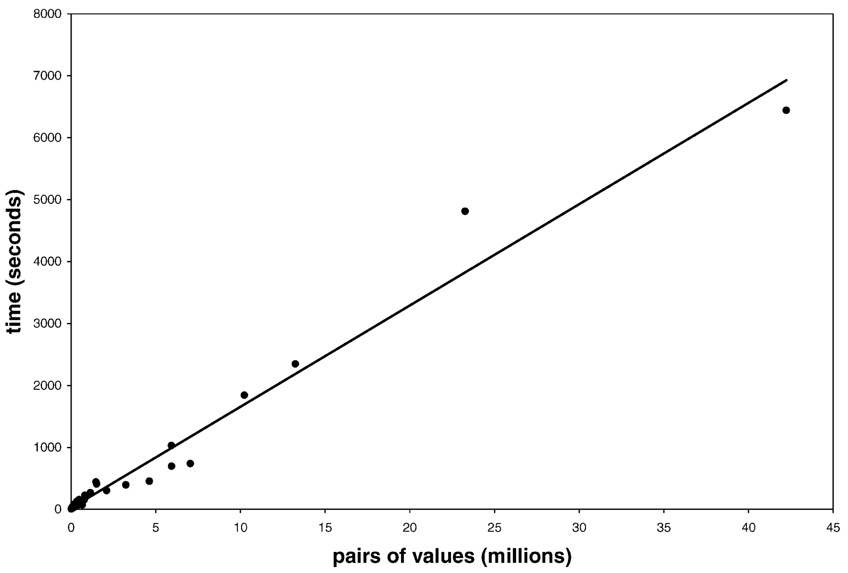

Fig. 11. Number of pairs of values is the best predictor of invariant detection runtime ($R^2 = 0.94$). The number of pairs of values is the number of distinct $\langle x, y \rangle$ pairs, where x and y are the values of two different variables in a single sample (one particular execution of a program point). The number of pairs of variables is not predictable from (though correlated with) number of test inputs and number of variables.

All other differences are reported in Figs. 12 and 13 as potentially interesting. For example, when comparing a test suite of size 2,000 to one of size 3,000, the following difference is reported at the exit of dodash:

1,000 tests:	`*j >= 2`	`(105 values)`
3,000 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 $\mathrm{result} = \mathrm{i}$ 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 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 $*\mathrm{i} \neq 5 \pmod{13}$ to $*\mathrm{i} \geq 0$). 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. Important future directions of research will include reporting, or directing the user to, more relevant invariants and determining which invariant differences are significant and which can be safely ignored.

In Figs. 12 and 13, 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 1,000 test cases. Especially in comparisons with the 3,000 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. (Daikon found all the invariants Gries listed (Section 2) and other experiments have had similar results.)

	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

Fig. 12. Invariant similarities and differences versus 2,500 test cases for the Siemens `replace` program. The chart compares invariants computed over a 2,500-element test suite with invariants computed over smaller test suites that were subsets of the 2,500-element test suite.

	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

Fig. 13. Invariant similarities and differences versus 3,000 test cases for the Siemens `replace` program. The chart compares invariants computed over a 3,000-element test suite with invariants computed over smaller test suites that were subsets of the 3,000-element test suite.

$$
\begin{array}{l}
\text{people} \in [1..50] \\
\text{pizzas} \in [1..10] \\
\text{pizza\_price} \in \{9, 11\} \\
\text{excess\_money} \in [0..40] \\
\text{slices} = 8 * \text{pizza} \\
\text{slices} = 0\ (mod\ 8) \\
\text{slices\_per} \in \{0, 1, 2, 3\} \\
\text{slices\_left} \leq \text{people} - 1
\end{array}
$$

Fig. 14. The eight 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.

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 1,000 test cases; differences with the 3,000 case test set then smooth out significantly, perhaps stabilizing, while differences with the 2,500 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 1,000 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.

Figs. 12 and 13 paint somewhat different pictures of invariant differences. Differences are smaller in comparisons with the 2,500-element test suite, while values tend to level off in comparisons with the 3,000-element test suite. Only 2.5 percent of binary invariants detected for the 2,000 or 2,500 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 2,500 test case suite, these numbers drop rapidly as the two test suites approach the same size. When the larger test suite has size 3,000, more invariants are different or missing, and these numbers stabilize quickly. The 3,000 case test suite appears to be anomalous: Comparisons with other sizes show more similarity with the numbers and patterns reported for the 2,500 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 memory for such large numbers of samples). Our preliminary investigations have not revealed a precise cause for the larger differences between the 3,000 case test suite and all the others, nor can we accurately predict the sizes of invariant differences; further investigation is required in order to understand these phenomena.

Grade	Invariants detected					Total
	2	3	4	5	6	
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

Fig. 15. The relationship between grade and the number of goal invariants (Fig. 14) found in student programs. For instance, all programs with a grade of 12 exhibited either two or three goal invariants, while most programs with a grade of 18 exhibited four or more invariants. A grade of 18 was a perfect score, and none of the 424 programs exhibited more than six, or fewer than two, of the eight relevant invariants.

6 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 eight relevant invariants, listed in Fig. 14. The list does not include trivial invariants such as $\text{slices\_per} \geq 0$, indicating that students never receive a negative number of slices, as well as uninteresting invariants such as $\text{slices} \leq \text{pizza\_price} + 75$, 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 the focus of this experiment.

Fig. 15 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 more.

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 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.

7 Test Suites for Invariant Discovery

So far, Daikon has produced adequate invariants from randomly generated tests (for the Gries programs, Section 2) and from preexisting test suites (for the Siemens programs, Section 4). However, 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 semiautomatic, relatively inexpensive methods: simple random test-case generation (Section 7.1) and grammar-driven test-case generation (Section 7.2).

For Siemens programs `replace` (string pattern replacement), `schedule` (process scheduling), and `tcas` (aircraft collision avoidance), we compare invariants resulting from automatically generated test suites and (a random selection of) the hand-crafted test cases from Siemens [43] as modified by Rothermel and Harrold [72].

7.1 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 [42].

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.

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, `schedule`'s function `init_prio_queue` adds processes to the active process queue. Daikon correctly produced the invariant $i = num\_proc$ 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 specified by the input was out of bounds, the resulting invariants showed the use of garbage values in determining the aircraft's collision avoidance response.

7.2 Grammar-Generated Test Suites

Randomly generating test cases from a grammar that describes valid inputs holds more promise than fully random testing. The grammar 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 in which to perform replacements, 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, we permitted any character to fill a pattern format in `replace`'s test generation grammar, even when the pattern language prohibits regular expression metacharacters.

The table in Fig. 16 compares the invariants produced from the grammar-driven test cases to invariants produced from the Siemens suite for each of the three 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 executes the program trace points sufficiently often. 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

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

Fig 16. Number of identical and differing invariants between invariants produced from grammar-driven test cases and from the Siemens test cases for each of the three Siemens programs. Each test suite contained 100 test cases.

exercise error conditions that are needed to fully understand program behavior.

8 Program Instrumentation

Daikon's input is a sequence of variable value tuples for every program point of interest to the programmer. Instrumentation inserted at each of 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.

8.1 Data File Format

At each program point of interest, the instrumented program writes to a data trace file the values of all variables in scope, including global variables, procedure arguments, local variables, and (at procedure exits) the return value. 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 [68] (see Section 10.1).

For every instrumented program point, then, the trace file contains a list of tuples of values, one value per instrumented variable. 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 engine would be presented a list of twelve elements, each element 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 overcontributing to invariant confidence (see Section 10.1 for details). Fig. 17 shows an excerpt from a data trace file.

In languages like C with explicit pointers (or in Java when the JVM gives access to an object ID), references are output both as an address (or object ID) and as a content (an object or array). This permits comparisons over both the references and over contents.

As noted in Section 3, Daikon operates only over scalar numbers (including characters and booleans) and arrays of numbers. Thus, values must be converted into one of these forms. For instance, a record `r` is converted into a collection of variables with the natural names `r.a`, `r.b`, etc. An array of structures is converted into a set of parallel arrays (one for each structure slot, appropriately named to make their origin clear).

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).

```
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
```

Fig. 17. The first three records in the data trace file for the Gries array sum program of Fig. 2, from which the invariants of Fig. 3 were derived. `B` is an array of integers and the other variables are integers. These records give variable values at program entry and at the start of the first two loop iterations. The complete data trace file contains 1,307 records.

8.2 Instrumentation Approach

The Daikon front ends add instrumentation to a program by source-to-source translation. Each instrumenter operates by parsing the program source into an abstract syntax tree (AST), determining which variables are in scope at each program point, inserting code at the program point to dump the variable values into an output file, and unparsing the AST to a file as source code, which can be compiled and run in the standard way. Adding instrumentation to a program is much faster than compiling it. Although instrumenting a program by modifying its object code would permit improved precision (for instance, in determining exactly which memory locations have been accessed or hooking into the exact point at which a variable is modified) and allow instrumentation of arbitrary binaries, it offers substantially greater obstacles to an implementation. For example, 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 entirely platform-independent. Invariant inference makes most sense when a program is being modified, which requires access to the program source anyway. Source code instrumentation also simplifies instrumenting just part of a system (such as only certain files).

For the relatively small, compute-bound programs we have examined so far, the instrumented code can be slowed down by more than an order of magnitude 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.

We have implemented instrumenters for C, Lisp, and Java. Section 8.3 discusses the C front end, which was used for the experiment described in Section 4. The Lisp instrumenter, which was used for the experiment described in Section 2, is similar, though simpler in some respects (for instance, it need not be concerned with determining array sizes nor avoiding segmentation faults). The Java front end is discussed elsewhere [31].

8.3 Instrumenting C Programs

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 [28], manages these problems in part by maintaining runtime status information on each variable in the program and in part with simplifying assumptions.

The instrumented program contains, 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 accurate information about procedure parameters and to track modifications in called procedures, a variable and its status object are passed to (or returned from) a procedure 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 nonassignable 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);
```

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 [45], 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.

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 a variable 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 expression `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. (Actually, the sequence of structures is converted into a collection of sequences, one per structure slot, as described in Section 3.)

9 Related Applications and Techniques

This section discusses uses of program invariants, presents other dynamic and static approaches for determining invariants, and considers how discovered invariants might be checked by other methods.

9.1 Other Applications of Invariants

This article has focused on the dynamic inference of invariants for applications in software evolution. Invariants, however, have many uses in computer science. Dynamically inferred invariants also could be used in many situations that declared or statically inferred invariants can and, in some cases, the application of dynamic ones may be more effective.

Invariants provide valuable documentation of a program's operation and data structures. Discovered 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. They can also double-check existing documentation or `assert` statements, particularly since program self-checks are often ineffective [58]. Additionally, a nearly-true invariant may indicate a bug or special case that should be brought to the programmer's attention.

Invariants may assist in test-case generation or validate a test suite. As observed in Section 4, invariants in the resulting program runs can indicate insufficient coverage of certain program states. Dynamic invariants form a program spectrum [71], which can help assess the impacts of change on software.

Detected invariants could bootstrap or direct a (manual or automatic) correctness proof. This would make Daikon sound and would help bootstrap users who do not wish to fully hand-annotate their programs before taking advantage of theorem-provers or other static verifiers. The low-level execution information used in profile-directed compilation could be augmented with higher-level invariants to enable better optimization for the common case.

9.2 Dynamic Inference

9.2.1 Machine Learning

Artificial intelligence research provides a number of techniques for extracting abstractions, rules, or generalizations from collections of data [64]. Most relevant to our research is an application of inductive logic programming (ILP) [70], [14], which produces a set of Horn clauses (effectively, first-order if-then rules) that express the learned concepts, to construct invariants from variable values on particular loop executions [3].

Traditional AI and machine learning techniques are not applicable to our problem for a variety of reasons mostly relating to the nature of the training sets. First, most learning systems, including ILP, must be trained on a set of examples marked with correct answers before they can produce useful results. Also, to preclude the generation of hypotheses that overgeneralize the training data, learning systems often apply additional techniques such as supplying counterexamples in the training set, adding domain-specific knowledge, or requiring an extra inference step to find the minimal positive generalization of the initial hypothesis. We have no access to counterexamples. Our domain knowledge comes in the limited form of fixed classes of hypotheses (invariants) to test. Second, we do not have the experimental control required by learning systems to perform reinforcement learning, in which a trainer or the environment rewards or penalizes an agent for each action it takes. In other words, we are performing observational rather than experimental discovery. Third, learning approaches, such as Bayesian and PAC learning, assume there is noise in the input data and, hence, inaccuracies in classification are acceptable or even beneficial. Our inputs contain no noise: We know the exact values of all instrumented variables at a program point. Accuracies that are considered quite good in some subfields are not acceptable in our domain. (Our approach characterizes the training set perfectly; either approach can misclassify additional data.) Fourth, our research focuses on comprehensibility of the resulting invariants and usefulness to programmers. Approaches like 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. Finally, most AI research addresses problems different from ours. Clustering, for example, groups similar examples under some domain-specific similarity metric. Classification places examples into one of a set of predefined categories and the categories require definitions or, more commonly, a training set. Regression attempts to learn a function over $n-1$ variables producing the nth, which is closer to our goal but still does not subsume finding other relationships among variables.

Another related area is programming by example (or programming by demonstration) [13], whose goal is automation of repetitive user actions, such as might be handled by a keyboard macro recorder. That research focuses on the discovery of simple repeated sequences in user input and on graphical user interfaces.

Dynamic invarient inference can be placed in the broad framework of concept discovery in artificial intelligence and it has a number of similarities with much of that work. For instance, it requires a good input set; irrelevant generalizations may result if the input set is too small or is not representative of the population of possible inputs. It generalizes over the data to find properties fitting a specified grammar; although it explores that space, it does not perform a directed search through it. And, it uses a bias—a choice of which properties are worth checking and reporting to the user.

While our particular problem has not been directly solved and many AI techniques are not applicable, we believe that generalizing these techniques, or applying them to subproblems of our task, can be fruitful.

9.2.2 Other Dynamic Approaches

Another approach to capturing and modeling runtime system behavior uses event traces, which describe the sequence of events in a possibly concurrent system, to produce a finite state machine generating the trace. Cook and Wolf [15], [16] use statistical and other techniques to detect sequencing, conditionals, and iteration, both for concurrent programs and for business processes. 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 [5] systematically explores the state space of concurrent systems using a synthesized finite state machine for each process. Andrews [2] compares actual behavior against behavior of a user-specified model, indicating divergences between the two.

Other dynamic analyses that examine program executions are used for software tasks from testing to debugging. Program spectra (specific aspects of program runs, such as event traces, code coverage, or outputs) [1], [71], [47] can reveal differences in inputs or program versions. The invariants detected in a program could serve as another spectrum.

Lencevicius et al. [61] applies database optimizations to the task of dynamically testing specified properties for all objects in a system; we could use similar techniques in our query tool.

Value profiling [10], [77], [11] addresses a subset of our problem: detection of constant or near-constant variables or instruction operands. Such information can permit runtime specialization: The program branches to a specialized version if a variable value is as expected. Runtime disambiguation [66], [74], [48] is similar, though it focuses on pointer aliasing. Many optimizations are valid only if two pointers are known not to be aliased. Although static determination of aliasing 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 percent have been realized [66]. Other work is capable of finding subsets of our invariants, such as ordering relationships among pairs of variables [80] or simple linear patterns for predicting memory access strides, which permits more effective parallelization [52], [24], [59].

9.3 Static Inference

Work in formal methods [46], [22], [17] inspired this research, which was motivated by a desire to find the dynamic analog to static techniques involving programmer-written specifications. 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 [39], [60], [25].

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 [9]. In particular, abstract interpretation (often 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 solution is approached either as the greatest lower bound of decreasing approximations or as the least upper bound of increasing approximations. The fixed point of the equations (possibly reached after infinitely many iterations that compute improving approximations, or by reasoning directly about the fixed point) is 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. 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 [18], [63], [85]; 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.

Some program understanding tools have taken the abstract interpretation/dataflow approach. Specifications can be constructed by extending 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. Givan [37], [38] takes this approach and permits unverified procedural implementations of specification functions to be used for runtime checking. No indication of how many irrelevant properties are output is provided. Gannod and Cheng [33], [12] 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 [34]. ADDS [44], [36] 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 [76]. Jeffords and Heitmeyer [51] 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.

Some formal proof systems generate intermediate assertions for help in proving a given goal formula by propagating known invariants forward or backward in the program [84], [41], [56], [19], [3]. In the case of array bounds checking [75], [40], [57], [67], [86], the desired property is obvious.

The Illustrating Compiler heuristically infers, via compile-time pattern matching and type inferencing, the abstract datatype implemented by a collection of concrete operations, then graphically displays the data in a way that is natural for that datatype [49].

ReForm [81] semiautomatically transforms, by provably correct steps, a program into a specification. The Maintainer's Assistant [83] uses program transformation techniques to prove equivalence of two programs (if they can be transformed to the same specification or to one another).

Other related work includes staging and binding-time analyses, which determine invariant or semi-invariant values for use in partial evaluation [50].

9.4 Checking Invariants

A specification can be checked against its implementation either dynamically, by running the program, or statically, by analyzing it. 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 and some research has addressed making invariant debugging and assertion languages more expressive or less restrictive [35], [73], [54], [8], a topic that is often 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 [7], [82] 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. Dynamic checks are not always effective in detecting errors. In one study, of 867 program self-checks, 34 were effective (located a bug, including six 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) [58].

Considerable research has addressed statically checking formal specifications [69], [20], [65], [62], [53]; such work could be used to verify likely invariants discovered dynamically, making our system sound. Recently, some realistic static specification checkers have been implemented. LCLint [29], [32] verifies that programs respect annotations in the Larch/C Interface Language [78]. Although these focus on properties such as modularity, which are already guaranteed in more modern languages, they also include pointer-based properties, such as definedness, nullness, and allocation state. ESC [21], [62], [23], 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. (Dependent types [69], [88], [87] make a similar tradeoff between expressiveness and computability.) 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. (A partially-annotated program could trigger even more warning messages than an unannotated one.)

Although program checking is challenging, it can often be automated. Determining what property to check is considered even harder [84], [6]. Most research in this area has focused on generation of intermediate assertions: Given a goal to prove, systems such as STeP [3] attempt to find sufficiently strong auxiliary predicates to permit a proof to be performed automatically. They may do so by forward propagation and generation of auxiliary invariants or by backward propagation and strengthening of properties, as discussed above. Our research is directly applicable since its goal is discovery of properties at any program point.

10 Ongoing and Future Work

Early experience with dynamic inference of invariants has highlighted a number of issues that require further research. This section briefly discusses increasing the relevance of reported invariants, improving performance performance, enhancing the way users see and manage the reported invariants, and adding to the collection of checked invariants. There are many other interesting areas for investigation, such as evaluating and improving test suites and formally proving the detected likely invariants.

10.1 Increasing Relevance

A naive implementation of the techniques described in this article would run excessively slowly, produce many uninteresting invariants, and omit certain useful invariants. We call an invariant *relevant* if it assists a programmer in a programming task. Perfect relevance is unattainable even in the presence of ideal test suites since relevance depends on the task and the programmer's experience, knowledge of the underlying system, etc. However, we have developed four techniques that generally improve the relevance of dynamically detected invariants [27].

One of the techniques—exploiting unused polymorphism—uses a two-pass approach to add desired invariants to the output. Daikon respects declared types and accepts only integer and integer array inputs. However, runtime types can be detected by a first pass and this information, which may be more specific than declared polymorphic types, provided to a second pass which can manipulate objects in ways specific to their actual values.

The other three techniques remove irrelevant invariants. First, invariants that are logically implied by other invariants in the output can be suppressed, which cuts down the output without reducing its information content. Implications can also be exploited earlier in inference to save work. Second, variables that can be statically proven to be unrelated need not be compared. This saves runtime and also avoids reporting of coincidentally true but unhelpful and uninteresting properties. Third, variables which have not been assigned since the last time an instrumentation point was encountered can be ignored. Otherwise, they would contribute to confidence in an invariant even though no change has occurred (as for a loop-invariant value repeatedly encountered at a loop head).

10.2 Improving Performance

Some of the techniques for improving relevance, mentioned immediately above, aid performance by reducing the number of variables that are considered by the inference engine. But there are other ways to mitigate combinatorial blowups (in instrumentation output size, inference time, and number of results) due to the potentially large numbers of program points to instrument, variables to examine at each point, and invariants to check over those variables.

One such approach is to address the granularity of instrumentation, which affects the amount of data gathered and, thus, the time required to process it. Inferring loop invariants or relationships among local variables can require instrumentation at loop heads, at function calls, or elsewhere, whereas determining representation invariants or properties of global variables does not require so many instrumentation points; perhaps module entry and exit points would be sufficient. When only a part of the program is of interest, the whole program need not be instrumented; in the `replace` study, we often recomputed invariants over just a single procedure in order to make invariant detection complete faster. Similarly, the choice of variables instrumented at each program point also affects inference performance. When some are not of interest, they can be skipped and variables that cannot have changed since the last instrumentation point need not be reexamined. Finally, supplying fewer test cases results in faster runtimes at the risk of less precise output.

The inference engine can be directly sped up by checking for fewer invariants; this is particularly useful when a programmer is focusing on part of the program and is not interested in certain kinds of properties (say, ternary functions). Derived variables can likewise be throttled to save time or increased to provide more extensive coverage. More complicated derived variables may be added for complex expressions that appear in the program text; derived variables or invariants may also involve functions defined in the program.

Finally, as mentioned earlier, the Daikon implementation is written in the interpreted, object-oriented language Python [79] and we have not optimized the implementation in any significant way. Significant performance improvements appear to be feasible.

10.3 Viewing and Managing Invariants

It may be difficult, perhaps overwhelming, for a programmer to sort through a large number of inferred invariants. This was an issue with `replace`, in which Daikon reported dozens of invariants per program point, only some of which were useful for the particular task. The relevance improvements above should help significantly in this regard. However, additional tools for viewing and managing the invariants could also help.

As one example, we developed a tool that retrieves the variable–value tuples that satisfy or falsify a user-specified property. As another example, we are considering developing a text editor that can provide a list of invariants for the program point or variable at the cursor. A programmer could also be permitted to filter out classes of invariants.

Ordering the reported invariants according to category or predicted usefulness could also help a programmer find a relevant invariant more quickly. The invariant differencing tool can indicate how a program change has affected the computed invariants.

Selective reporting of invariants could also improve the performance of invariant inference. For example, if the user interface presents invariants on demand, the invariants could be computed on demand as well. In `replace`, for example, the average program point required 220 seconds of inference time. With an order of magnitude speed improvement due to implementation in a compiled language, combined with filtering of unwanted classes of invariants, perhaps over one or a few variables, on-demand inference time could be limited to a few seconds and the start-up costs for inference would be limited to running the test cases.

10.4 Richer Invariants

We are pursuing techniques that find and report more sophisticated invariants. At present, the two most critical improvements are discovering invariants over pointer-based recursive data structures, such as linked lists, trees, and graphs, and computing conditional (and disjunctive) invariants, such as $p = \mathrm{NULL}$ or $^*p > i$. These two improvements are symbiotic, as the trivial example shows. Our current design for handling pointer-based data structures is to linearize them, in a variety of ways, during instrumentation, and then look for invariants over the linearized sequences. Conditional invariants are detected by splitting the data trace into two parts, performing invariant inference over each part, and combining the results. The data can be split in a number of ways: random and exhaustive splitting of the traces; exceptions to invariants being tested; splitting on special values, such as common constants (like zero and one) or extremal values found earlier; and using static analysis to identify potential predicates for splitting. Preliminary results are reported elsewhere [30].

11 Conclusions

This research demonstrates the feasibility and effectiveness of discovering program invariants based on execution traces. This technique automatically detected all the stated invariants in a set of formally-specified programs; furthermore, the invariants detected in a C program proved useful in a software evolution task. The techniques and prototype implementation are adequately fast when applied to modest programs.

Working on evolution tasks with programs that we did not write gave us insights into the strengths and weaknesses of dynamic invariant detection, the specific techniques, and the Daikon tool. Moreover, the use of dynamically inferred invariants qualitatively affected programmers, encouraging them to think in terms of invariants where they might otherwise not have. With a variety of improvements, as discussed in the previous section, there is significant promise that the approach could be applicable to the evolution of larger systems.

This promise holds despite the fact that the invariant detector may discover invariants that are not universally true for all potential executions. A local static analysis can reveal useful invariants that are universally true of a function, no matter how it is used. A whole-program analysis can discover stronger properties of a function, in particular, properties that are dependent on the contexts in which function is called (as discussed in Section 9). A dynamic invariant detector can report yet stronger invariants that depend on the data sets over which the program may be run. The ability to achieve the last also admits invariants that are only true of the particular test suite chosen and these are not generally discernible by the user from the other kinds of invariants. Regardless, some inferred invariants are capable of pointing out flaws in the test suite and directing its improvement (as discussed in Section 4).

Focusing on the general task of software evolution as well as on the task-driven needs of a programmer has led to effective solutions. For instance, our technique need not find a complete specification or every interesting invariant, nor find only interesting or correct invariants. Rather, the technique must enable a programmer to evolve systems more effectively than before. This point of view has guided even the most technical aspects of the research. For instance, we chose a highly uniform design for the invariant engine, modeling only scalar integer variables and arrays of those scalars. All other types must be mapped to these types and nonvariable entities must be mapped to variables. Although the choices constrain what invariants the system infers, it provides useful invariants at an acceptable cost. Focusing on the software evolution task has also guided choices such as checking for a fixed set of invariant classes, computing confidence levels, and even the data we capture during instrumentation.

The Daikon invariant detector is available at http://sdg.lcs.mit.edu/~mernst/daikon/.

Acknowledgments

This is a revised and extended version of a paper that appeared in ICSE '99 [26].

Aung Thaung performed the analysis of student programs reported in Section 6. Vibha Sazawal provided valuable assistance with statistical analysis. Gregg Rothermel shared his modified versions of the Siemens test programs. Many of our colleagues provided comments on our ideas; we are particularly grateful for the early feedback of Craig Chambers, Oren Etzioni, Tessa Lau, David Madigan, and Jared Saia. Greg Badros, Craig Chambers, Tessa Lau, Todd Millstein, Jon Nowitz, Steve Wolfman, and the anonymous ICSE '99 and *IEEE Transactions on Software Engineering* referees improved this article by critiquing previous versions. Daniel Jackson, Vass Litvinov, George Necula, James Noble, and the referees suggested related work.

This work was supported by the US National Science Foundation grants CCR-9506779 and CCR-9508745, an IBM Cooperative Fellowship, and a gift from Edison Design Group [28]. This research was conducted while M.D. Ernst and J. Cockrell were at the University of Washington.

References

[1] D. Abramson, I. Foster, J. Michalakes, and R. Socic, "Relative Debugging: A New Methodology for Debugging Scientific Applications," *Comm. ACM*, vol. 39, no. 11, pp. 69–77, Nov. 1996.

[2] J.H. Andrews, "Testing Using Log File Analysis: Tools, Methods and Issues," *Proc. 13th Ann. Int'l Conf. Automated Software Eng. (ASE '98)*, pp. 157–166, Oct. 1998.

[3] N. Bjørner, A. Browne, and Z. Manna, "Automatic Generation of Invariants and Intermediate Assertions," *Theoretical Computer Science*, vol. 173, no. 1, pp. 49–87, Feb. 1997.

[4] I. Bratko and M. Grobelnik, "Inductive Learning Applied to Program Construction and Verification," *Knowledge Oriented Software Design: Extended Papers from the IFIP TC 12 Workshop Artificial Intelligence from the Information Processing Perspective, (AIFIPP '92)*, J. Cuena, ed., pp. 169–182, 1993.

[5] B. Boigelot and P. Godefroid, "Automatic Synthesis of Specifications from the Dynamic Observation of Reactive Programs," *Proc. Third Int'l Workshop Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97)*, pp. 321–333, Apr. 1997.

[6] S. Bensalem, Y. Lakhnech, and H. Saidi, "Powerful Techniques for the Automatic Generation of Invariants," *Proc. Eighth Int'l Conf. Computer Aided Verification (CAV)*, pp. 323–335, July/Aug. 1996.

[7] M. Blum, "Designing Programs to Check Their Work," *Proc. Int'l Symp. Software Testing and Analysis*, T. Ostrand and E. Weyuker, eds., p. 1, June 1993.

[8] E.C. Chan, J.T. Boyland, and W.L. Scherlis, "Promises: Limited Specifications for Analysis and Manipulation," *Proc. 20th Int'l Conf. Software Eng.*, pp. 167–176, Apr. 1998.

[9] P.M. Cousot and R. Cousot, "Automatic Synthesis of Optimal Invariant Assertions: Mathematical Foundations," *Proc. ACM Symp. Artificial Intelligence and Programming Languages*, pp. 1–12, Aug. 1977.

[10] B. Calder, P. Feller, and A. Eustace, "Value Profiling," *Proc. 27th Ann. Int'l Symp. Microarchitecture (MICRO-97)*, pp. 259–269, Dec. 1997.

[11] B. Calder, P. Feller, and A. Eustace, "Value Profiling and Optimization," *J. Instruction Level Parallelism*, vol. 1, Mar. 1999, http://www.jilp.org/vol1/.

[12] B.H.C. Cheng and G.C. Gannod, "Abstraction of Formal Specifications from Program Code," *Proc. Third Int'l Conf. Tools for Artificial Intelligence (TAI '91)*, pp. 125–128, Nov. 1991.

[13] *Watch What I Do: Programming by Demonstration*, A. Cypher, D.C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B.A. Myers, and A. Turransky eds., Cambridge, Mass.: MIT Press, 1993.

[14] W.W. Cohen, "Grammatically Biased Learning: Learning Logic Programs Using an Explicit Antecedent Description Language," *Artificial Intelligence*, vol. 68, pp. 303–366, Aug. 1994.

[15] J.E. Cook and A.L. Wolf, "Discovering Models of Software Processes from Event-Based Data," *ACM Trans. Software Eng. and Methodology*, vol. 7, no. 3, pp. 215–249, July 1998.

[16] J.E. Cook and A.L. Wolf, "Event-Based Detection of Concurrency," *Proc. ACM SIGSOFT '98 Symp. Foundations of Software Eng.*, pp. 35–45, Nov. 1998.

[17] E.M. Clarke, J.M. Wing, R. Alur, R. Cleaveland, D. Dill, A. Emerson, S. Garland, S. German, J. Guttag, A. Hall, T. Henzinger, G. Holzmann, C. Jones, R. Kurshan, N. Leveson, K. McMillan, J. Moore, D. Peled, A. Pnueli, J. Rushby, N. Shankar, J. Sifakis, P. Sistla, B. Steffen, P. Wolper, J. Woodcock, and P. Zave, "Formal Methods: State of the Art and Future Directions," *ACM Computing Surveys*, vol. 28, no. 4, pp. 626–643, Dec. 1996.

[18] D.R. Chase, M. Wegman, and F.K. Zadeck, "Analysis of Pointers and Structures," *Proc. SIGPLAN '90 Conf. Programming Language Design and Implementation*, pp. 296–310, June 1990.

[19] D.D. Dunlop and V.R. Basili, "A Heuristic for Deriving Loop Functions," *IEEE Trans. Software Eng.*, vol. 10, no. 3, pp. 275–285, May 1984.

[20] M.B. Dwyer and L.A. Clarke, "Data Flow Analysis for Verifying Properties of Concurrent Programs," *Proc. Second ACM SIGSOFT Symp. Foundations of Software Eng. (SIGSOFT '94)*, pp. 62–75, Dec. 1994.

[21] D.L. Detlefs, "An Overview of the Extended Static Checking System," *Proc. First Workshop Formal Methods in Software Practice*, pp. 1–9, Jan. 1996.

[22] E.W. Dijkstra, *A Discipline of Programming.* Englewood Cliffs, N.J.: Prentice-Hall, 1976.

[23] D.L. Detlefs, K. Rustan, M. Leino, G. Nelson, and J.B. Saxe, "Extended Static Checking," SRC Research Report 159, Compaq Systems Research Center, Dec. 1998.

[24] D. Bruening, S. Devabhaktuni, and S. Amarasinghe, "Softspec: Software-Based Speculative Parallelism," MIT/LCS Technical Memo, LCS-TM-606, Apr. 2000.

[25] G. Dromey, *Program Derivation: The Development of Programs from Specifications*, Addison-Wesley, 1989.

[26] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *Proc. 21st Int'l Conf. Software Eng.*, pp. 213–224, May 1999.

[27] M.D. Ernst, A. Czeisler, W.G. Griswold, and D. Notkin, "Quickly Detecting Relevant Program Invariants," *Proc. 22nd Int'l Conf. Software Eng.*, pp. 449-458, June 2000.

[28] Edison Design Group, *C++ Front End Internal Documentation*, version 2.28 ed., Mar. 1995, http://www.edg.com.

[29] D. Evans, J. Guttag, J. Horning, and Y.M. Tan, "LCLint: A Tool for Using Specifications to Check Code," *Proc. Second ACM SIGSOFT Symp. the Foundations of Software Eng. (SIGSOFT '94)*, pp. 87–97, Dec. 1994.

[30] M.D. Ernst, W.G. Griswold, Y. Kataoka, and D. Notkin, "Dynamically Discovering Pointer-Based Program Invariants," Technical Report UW-CSE-99-11-02, Univ. of Washington, Seattle, Wash., Nov. 1999.

[31] M.D. Ernst, "Dynamically Discovering Likely Program Invariants," PhD thesis, Dept. of Computer Science and Eng., Univ. of Washington, Seattle, Wash., Aug. 2000.

[32] D. Evans, "Static Detection of Dynamic Memory Errors," *Proc. SIGPLAN '96 Conf. Programming Language Design and Implementation*, pp. 44–53, May 1996.

[33] G.C. Gannod and B.H.C. Cheng, "Strongest Postcondition Semantics as the Formal Basis for Reverse Engineering," *J. Automated Software Eng.*, vol. 3, nos. 1-2, pp. 139–164, June 1996.

[34] G.C. Gannod and B.H.C. Cheng, "A Specification Matching Based Approach to Reverse Engineering," *Proc. 21st Int'l Conf. Software Eng.*, pp. 389–398, May 1999.

[35] M. Golan and D.R. Hanson, "DUEL—A Very High-Level Debugging Language," *Proc. 1993 USENIX Conf.*, pp. 107–117, Jan. 1993.

[36] R. Ghiya and L.J. Hendren, "Is It a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-Directed Pointers in C," *Proc. 23rd Ann. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, pp. 1–15, Jan. 1996.

[37] R. Givan, "Inferring Program Specifications in Polynomial-Time," *Proc. Third Int'l Symp. Static Analysis (SAS '96)*, pp. 205–219, Sept. 1996.

[38] R.L. Givan Jr., "Automatically Inferring Properties of Computer Programs," PhD thesis, Mass. Inst. of Technology, Cambridge, Mass., June 1996.

[39] D. Gries, *The Science of Programming*, New York: Springer-Verlag, 1981.

[40] R. Gupta, "A Fresh Look at Optimizing Array Bound Checking," *Proc. SIGPLAN '90 Conf. Programming Language Design and Implementation*, pp. 272–282, June 1990.

[41] S.M. German and B. Wegbreit, "A Synthesizer of Inductive Assertions," *IEEE Trans. Software Eng.*, vol. 1, no. 1, pp. 68–75, Mar. 1975.

[42] D. Hamlet, "Random Testing," *Encyclopedia of Software Eng.*, 1994.

[43] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria," *Proc. 16th Int'l Conf. Software Eng.*, pp. 191–200, May 1994.

[44] L.J. Hendren, J. Hummel, and A. Nicolau, "Abstractions for Recursive Pointer Data Structures: Improving the Analysis and Transformation of Imperative programs," *Proc. SIGPLAN '92 Conf. Programming Language Design and Implementation*, pp. 249–260, June 1992.

[45] R. Hastings and B. Joyce, "Purify: A Tool for Detecting Memory Leaks and Access Errors in C and C++ Programs," *Proc. USENIX Conf.*, pp. 125–138, Jan. 1992.

[46] C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," *Comm. ACM*, vol. 12, no. 10, pp. 576–583, Oct. 1969.

[47] M.J. Harrold, G. Rothermel, R. Wu, and L. Yi, "An Empirical Investigation of Program Spectra," *ACM SIGPLAN/SIGSOFT Workshop Program Analysis for Software Tools and Eng. (PASTE '98)*, pp. 83–90, June 1998.

[48] A.S. Huang, G. Slavenburg, and J.P. Shen, "Speculative Disambiguation: A Compilation Technique for Dynamic Memory Disambiguation," *Proc. 21st Ann. Int'l Symp. Computer Architecture*, pp. 200–210, Apr. 1994.

[49] R. Henry, K.M. Whaley, and B. Forstall, "The University of Washington Illustrating Compiler," *Proc. SIGPLAN '90 Conf. Programming Language Design and Implementation*, pp. 223–246, June 1990.

[50] N.D. Jones, C.K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation.* Prentice Hall Int'l, 1993.

[51] R. Jeffords and C. Heitmeyer, "Automatic Generation of State Invariants from Requirements Specifications," *Proc. ACM SIGSOFT '98 Symp. Foundations of Software Eng.*, pp. 56–69, Nov. 1998.

[52] R.W.M. Jones, "A Strategy for Finding the Optimal Data Placement for Regular Programs," master's thesis, Dept. of Computing, Imperial College, 1996.

[53] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews, "Reasoning About Java Classes," *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, pp. 329–340, Oct. 1998.

[54] M. Karaorman, U. Holzle, and J. Bruno, "jContractor: A Reflective Java Library to Support Design by Contract," Technical Report TRCS98-31, Univ. of Calif., Santa Barbara, Jan. 1999.

[55] S.C. Kleene, "Representation of Events in Nerve Nets and Finite Automata," *Automata Studies, Annals of Math. Studies 34*, C.E. Shannon and J. McCarthy, eds., pp. 3–40, 1956.

[56] S. Katz and Z. Manna, "Logical Analysis of Programs," *Comm. ACM*, vol. 19, no. 4, pp. 188–206, Apr. 1976.

[57] P. Kolte and M. Wolfe, "Elimination of Redundant Array Subscript Range Checks," *Proc. SIGPLAN '95 Conf. Programming Language Design and Implementation*, pp. 270–278, June 1995.

[58] N.G. Leveson, S.S. Cha, J.C. Knight, and T.J. Shimeall, "The Use of Self Checks and Voting in Software Error Detection: An Empirical Study," *IEEE Trans. Software Eng.* vol. 16, no. 4, pp. 432–443, 1990.

[59] S.-W. Liao, A. Diwan, R.P. Bosch, Jr., A. Ghuloum, and M.S. Lam, "SUIF Explorer: An Interactive and Interprocedural Parallelizer," *Proc. Seventh ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPOPP '99)*, pp. 37–48, May 1999.

[60] B. Liskov and J. Guttag, *Abstraction and Specification in Program Development.* Cambridge, Mass.: MIT Press, 1986.

[61] R. Lencevicius, U. Hölzle, and A.K. Singh, "Query-Based Debugging of Object-Oriented Programs," *Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pp. 304–317, Oct. 1997.

[62] K.R.M. Leino and G. Nelson, "An Extended Static Checker for Modula-3," *Proc. Compiler Construction: Seventh Int'l Conf. (CC '98)*, pp. 302–305, Apr. 1998.

[63] W. Landi and B.G. Ryder, "A Safe Approximate Algorithm for Interprocedural Pointer Aliasing," *Proc. SIGPLAN '92 Conf. Programming Language Design and Implementation*, pp. 235–248, June 1992.

[64] T.M. Mitchell, *Machine Learning.* McGraw-Hill Series in Computer Science, Boston, Mass.: WCB/McGraw-Hill, 1997.

[65] G. Naumovich, L.A. Clarke, L.J. Osterweil, and M.B. Dwyer, "Verification of Concurrent Software with FLAVERS," *Proc. 19th Int'l Conf. Software Eng.*, pp. 594–595, May 1997.
[66] A. Nicolau, "Run-Time Disambiguation: Coping with Statically Unpredictable Dependencies," *IEEE Trans. Computers,* vol. 38, no. 5, pp. 663–678, May 1989.
[67] G.C. Necula and P. Lee, "The Design and Implementation of a Certifying Compiler," *Proc. ACM SIGPLAN '98 Conf. Programming Language Design and Implementation,* pp. 333–344, June 1998.
[68] R. O'Callahan and D. Jackson, "Lackwit: A Program Understanding Tool Based on Type Inference," *Proc. 19th Int'l Conf. Software Eng.*, pp. 338–348, May 1997.
[69] F. Pfenning, "Dependent Types in Logic Programming," *Types in Logic Programming,* F. Pfenning, ed., chapter 10, pp. 285–311, 1992.
[70] J.R. Quinlan, "Learning Logical Definitions from Relations," *Machine Learning,* vol. 5, pp. 239–266, 1990.
[71] T. Reps, T. Ball, M. Das, and J. Larus, "The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem," *Proc. Sixth European Software Eng. Conf. and Fifth ACM SIGSOFT Symp. Foundations of Software Eng. (ESEC/FSE '97),* pp. 432–449, Sept. 1997.
[72] G. Rothermel and M.J. Harrold, "Empirical Studies of a Safe Regression Test Selection Technique," *IEEE Trans. Software Eng.,* vol. 24, no. 6, pp. 401–419, June 1998.
[73] D.S. Rosenblum, "A Practical Approach to Programming with Assertions," *IEEE Trans. Software Eng.,* vol. 21, no. 1, pp. 19–31, Jan. 1995.
[74] B. Su, S. Habib, W. Zhao, J. Wang, and Y. Wu, "A Study of Pointer Aliasing for Software Pipelining Using Run-Time Disambiguation," *Proc. 27th Ann. Int'l Symp. Microarchitecture (MICRO-97),* pp. 112–117, Nov./Dec. 1994.
[75] N. Suzuki and K. Ishihata, "Implementation of an Array Bound Checker," *Proc. Fourth Ann. ACM Symp. Principles of Programming Languages,* pp. 132–143, Jan. 1977.
[76] M. Sagiv, T. Reps, and R. Wilhelm, "Parametric Shape Analysis via 3-Valued Logic," *Proc. 26th Ann. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* pp. 105–118, Jan. 1999.
[77] A. Sodani and G.S. Sohi, "An Empirical Analysis of Instruction Repetition," *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII),* pp. 35–45, Oct. 1998.
[78] Y.M. Tan, "Formal Specification Techniques for Promoting Software Modularity, Enhancing Documentation, and Testing Specifications," Technical Report MIT/LCS/TR-619, Mass. Inst. of Technology, Laboratory for Computer Science, June 1994.
[79] G. van Rossum, *Python Reference Manual,* 1.5 ed., Dec. 1997.
[80] M. Vaziri and G. Holzmann, "Automatic Detection of Invariants in Spin," *SPIN 98: Papers from the Fourth Int'l SPIN Workshop,* Nov. 1998.
[81] M.P. Ward, "Program Analysis by Formal Transformation," *The Computer J.,* vol. 39, no. 7, pp. 598–618, 1996.
[82] H. Wasserman and M. Blum, "Software Reliability via Run-Time Result-Checking," *J. ACM,* vol. 44, no. 6, pp. 826–849, Nov. 1997.
[83] M. Ward, F.W. Calliss, and M. Munro, "The Maintainer's Assistant," *Proc. Int'l Conf. Software Maintenance,* pp. 307–315, 1989.
[84] B. Wegbreit, "The Synthesis of Loop Predicates," *Comm. ACM,* vol. 17, no. 2, pp. 102–112, Feb. 1974.
[85] R.P. Wilson and M.S. Lam, "Efficient Context-Sensitive Pointer Analysis for C Programs," *Proc. SIGPLAN '95 Conf. Programming Language Design and Implementation,* pp. 1–12, June 1995.
[86] H. Xi and F. Pfenning, "Eliminating Array Bound Checking Through Dependent Types," *Proc. ACM SIGPLAN '98 Conf. Programming Language Design and Implementation,* pp. 249–257, June 1998.
[87] H. Xi and F. Pfenning, "Dependent Types in Practical Programming," *Proc. 26th Ann. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* pp. 214–227, Jan. 1999.
[88] C. Zenger, "Indexed Types," *Theoretical Computer Science,* vol. 187, pp. 147–165, 1997.

Michael D. Ernst holds the SB and SM degrees from the Massachusetts Institute of Technology. He received the PhD degree in computer science and engineering from the University of Washington, prior to which he was a lecturer at Rice University and a researcher at Microsoft Research. He is an assistant professor in the Department of Electrical Engineering and Computer Science and in the Laboratory for Computer Science at the Massachusetts Institute of Technology. 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.

Jake Cockrell received the BS degree in computer science from the University of Virginia and the MS degree in computer science and engineering from the University of Washington. He currently works at Macromedia as an engineer on the Dreamweaver team.

William G. Griswold received the BA degree in mathematics from the University of Arizona in 1985 and the PhD degree in computer science from the University of Washington in 1991. He is an associate professor in the Department of Computer Science and Engineering at the University of California, San Diego. He is on the program committee for the 2000 International Conference on Software Engineering, an associate editor for *IEEE Transactions on Software Engineering*, and an officer of ACM SIGSOFT. His research interests include software evolution and design, software tools, and program analysis. He is a member of the IEEE and the IEEE Computer Society.

David Notkin received the ScB degree at Brown University in 1977 and the PhD degree at Carnegie Mellon University in 1984. He is the Boeing Professor of computer science and engineering at the University of Washington. Dr. Notkin received the US National Science Foundation Presidential Young Investigator Award in 1988, served as the program chair of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering, served as program cochair of the 17th International Conference on Software Engineering, chaired the steering committee of the International Conference on Software Engineering (1994-1996), served as charter associate editor of both *ACM Transactions on Software Engineering and Methodology* and the *Journal on Programming Languages*, serves as an associate editor of the *IEEE Transactions on Software Engineering*, was named as an ACM Fellow in 1998, serves as the chair of ACM SIGSOFT, and received the 2000 University of Washington Distinguished Graduate Mentor Award. His research interests are in software engineering in general and in software evolution in particular. He is a member of the IEEE Computer Society.