Quickly Detecting Relevant Program Invariants

Michael D. Ernst†, **Adam Czeisler**†, **William G. Griswold**‡, and **David Notkin**†

†Dept. of Computer Science & Engineering
University of Washington
Box 352350, Seattle WA 98195-2350 USA
{mernst,czeisler,notkin}@cs.washington.edu

‡Dept. of Computer Science & Engineering
University of California San Diego, 0114
La Jolla, CA 92093-0114 USA
wgg@cs.ucsd.edu

ABSTRACT

Explicitly stated program invariants can help programmers by characterizing certain aspects of program execution and identifying program properties that must be preserved when modifying code. Unfortunately, these invariants are usually absent from code. Previous work showed how to dynamically detect invariants from program traces by looking for patterns in and relationships among variable values. A prototype implementation, Daikon, accurately recovered invariants from formally-specified programs, and the invariants it detected in other programs assisted programmers in a software evolution task. However, Daikon suffered from reporting too many invariants, many of which were not useful, and also failed to report some desired invariants.

This paper presents, and gives experimental evidence of the efficacy of, four approaches for increasing the relevance of invariants reported by a dynamic invariant detector. One of them — exploiting unused polymorphism — adds desired invariants to the output. The other three — suppressing implied invariants, limiting which variables are compared to one another, and ignoring unchanged values — eliminate undesired invariants from the output and also improve runtime by reducing the work done by the invariant detector.

1 INTRODUCTION

Previous research explored the use of dynamic methods for discovering likely program invariants, with a particular interest in supporting software evolution tasks [ECGN]. A prototype implementation, Daikon, demonstrated the feasibility of dynamically detecting invariants, or properties that hold at a particular program point. The approach is to run the program of interest, examine the values that the program computes, and postulate and check potential invariants over those values, reporting those that are true for the test suite and that also satisfy some other conditions.

Daikon's output was accurate: it rediscovered formal specifications from which a set of programs had been derived. Daikon's output was also useful: programmers who were modifying an undocumented C program (apparently written without thought for formal invariants) found the dynamically detected invariants helpful in their modification task. This paper describes four techniques that improve the relevance — usefulness to programmers — of the reported invariants and the performance of the underlying engine. The first technique adds desired but previously missing invariants, and the latter three eliminate undesirable invariants while simultaneously improving runtime.

Polymorphism Elimination. Variables declared polymorphically (as with Java's `Object` type or any other base class) often contain only a single type at runtime. Daikon uses declared types to avoid the costs of managing polymorphism at invariant detection time, but it cannot exploit runtime types to extract specific fields. We address this issue via a two-pass technique. A first pass detects invariants over the runtime class of objects; the resulting information is fed back into a second pass. Section 5 describes this technique, showing that it enables reporting of relevant but otherwise undetected invariants specific to the variable values' run-time types.

Redundant invariants. Not all invariants are worth reporting. For instance, if two invariants $x \neq 0$ and x in $[7..13]$ are determined to be true, there is no sense in reporting both because the latter implies the former. Furthermore, not all invariants are worth computing. As an example, once Daikon determines that $x = y$, then no inference need be done for `y`, as invariants over `x` imply similar ones over `y`. Pruning both the computation and the reporting of implied invariants reduces the size of the output without removing any information from it. Section 6 details our approach to implied invariants and reports its efficacy in practice.

Comparability. Not all variables can be sensibly compared. For instance, numerically comparing a boolean to a non-null pointer results in the accurate but useless invariant that the boolean is always less than the pointer value. Restricting comparability can both increase performance by reducing the number of potential invariants to be checked

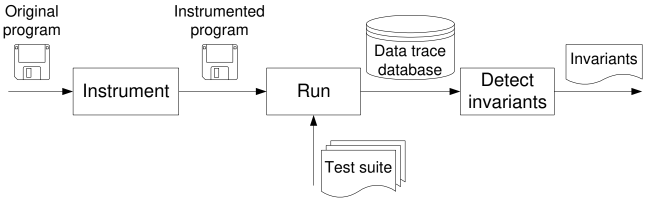

Figure 1: An overview of dynamic invariant detection as implemented by the Daikon tool.

and reduce the number of irrelevant invariants reported. Section 7 compares four approaches to limiting the number of comparisons. These approaches use declared types and value flow information computed by the Lackwit tool [OJ97].

Repeated values. Only invariants that are statistically justified — relationships that do not appear to have occurred by chance alone — should be reported. These statistical confidence tests depend on the set of values obtained at a particular program point. When a variable's value is repeatedly examined without intervening variable assignments — such as a variable that is examined at a loop head but remains unchanged within a loop — then the number of samples is artificially inflated and properties of the variable may be inappropriately reported. Section 8 reports on the relative effectiveness of several rules for ignoring some instances of repeated values.

Dynamic invariant detection is described in Section 2, along with a concrete example of Daikon's output. Section 3 explains the relevance metric and Section 4 lays out the experimental method. The four techniques sketched above for improving relevance are further explained and experimentally justified in Section 5, which discusses adding invariants that are missing but desired, and Sections 6–8, which discuss eliminating undesired invariants. Section 9 briefly surveys related work. Finally, Section 10 recapitulates the results and discusses future work.

2 BACKGROUND

Dynamic invariant detection [ECGN] discovers likely invariants 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 then postulating and checking invariants over both the instrumented variables and derived variables not manifest in the program (Figure 1). 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 or to another tool. As with other dynamic approaches such as profiling, the accuracy of the inferred invariants depends in part on the quality and completeness of the test cases. The Daikon invariant detector is language independent, and program instrumenters exist for C, Java, and Lisp.

Daikon detects invariants at specific program points such as loop heads and procedure entries and exits; each program point is treated independently. The invariant detector is provided with a variable trace that contains, for each execution of a program point, the values of all variables in scope at that point. Each of a set of possible invariants is tested against various combinations of traced variables. The following lists the classes of invariants Daikon computes, where x, y, and z are variables, and a, b, and c are computed constants:

- invariants over any variable, such as being constant ($\mathsf{x} = \mathsf{a}$), taking its values from a small set ($\mathsf{x} \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$), etc.
- invariants over a single numeric variable, such as being in a range ($\mathsf{a} \leq \mathsf{x} \leq \mathsf{b}$), non-zero, modulus ($\mathsf{x} \equiv \mathsf{a} \pmod{\mathsf{b}}$), etc.
- invariants over two numeric variables, such as a linear relationship ($\mathsf{y} = \mathsf{ax} + \mathsf{b}$), an ordering relationship ($\mathsf{x} \leq \mathsf{y}$), functions ($\mathsf{x} = \mathsf{fn}(\mathsf{y})$) for built-in unary functions, combinations of invariants over a single numeric variable ($\mathsf{x} + \mathsf{y} \equiv \mathsf{a} \pmod{\mathsf{b}}$), etc.
- invariants over three numeric variables, such as a linear relationship ($\mathsf{z} = \mathsf{ax} + \mathsf{by} + \mathsf{c}$), functions, etc.
- invariants over a single sequence variable, such as minimum and maximum sequence values, lexicographical ordering, element ordering, invariants holding for all elements in the sequence, etc.
- invariants over two sequence variables, such as an elementwise linear relationship, lexicographic comparison, subsequence relationship, etc.
- invariants over a sequence and a numeric variable, in particular membership ($\mathsf{x} \in \mathsf{y}$).

For each variable or tuple of variables, each potential invariant is tested. Each potential unary invariant is checked for all variables, each potential binary invariant is checked over all pairs of variables, and so forth. A potential invariant is checked by examining each sample (i.e., tuple of values for the variables being tested) in turn. As soon as a sample not satisfying the invariant is encountered, that invariant is known not to hold and is not checked for any subsequent samples. Because false invariants tend to be falsified quickly, the cost of computing invariants tends to be proportional to the number of invariants discovered. All the invariants are inexpensive to test and do not require full-fledged theorem-proving.

Derived variables. To enable reporting of invariants regarding components or properties of aggregates, Daikon represents such entities as additional variables available for inference. For instance, if array `a` and integer `lasti` are both in scope, then properties over `a[lasti]` may be of interest, even though it is not a variable and may not even appear in the program text. Derived variables are treated just like other variables by the invariant detector, permitting the engine to infer invariants that are not hardcoded into its list. For instance, if `size(A)` is derived from sequence `A`, then the sys-

```
15.1.1:::ENTER  100 samples
  N = size(B)
 [N >= 0]

15.1.1:::EXIT    100 samples
  B = B_orig
  N = I = N_orig = size(B)
 [S = sum(B)]
  N >= 0

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

Figure 2: Invariants inferred for Gries's Program 15.1.1 [Gri81], which sums array B into variable S. The program's author specified the boxed invariants; they are the (successfully obtained) goal. See Figure 5 for C source code.

tem can report the invariant $\mathsf{i} < \mathsf{size}(\mathsf{A})$ without hardcoding a less-than comparison check for the case of a scalar and the length of a sequence.

Invariant confidence. An invariant is reported only if there is adequate evidence of its plausibility. In particular, if there are an inadequate number of samples of a particular variable, patterns observed over it may be mere coincidence. Consequently, for each detected invariant, Daikon computes the probability that such a property would appear by chance in a random input. The property is reported only if its probability is smaller than a user-defined confidence parameter.

Example. As a simple example of invariant detection, consider a program that sums the elements of an array. An automatic instrumenter 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 this program on 100 arrays generated from an exponential distribution. Figure 2 shows Daikon's output. This output contains all the invariants specified by the programmer, who derived the programs from the formal specification. It also adds invariants, such as that N is the length of array B (which is crucial to the correctness of the routine but was incorrectly omitted from the specification) and that the routine does not modify its arguments (which could aid understanding the program). More details are available elsewhere [ECGN].

3 RELEVANCE

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

Program	NBNC LOC	Procedures
replace	516	21
tcas	136	9
tot_info	274	7

Figure 3: Size of the Siemens programs. The second column is the number of non-blank, non-comment source lines. The third is the number of procedures in the program.

To improve invariant relevance, the programmer — who *is* privy to much of the context — could control the invariant inference process. Alternately, heuristics could be added to Daikon to improve the relevance of the reported invariants in most cases. This paper focuses on the second approach, which lessens the initial burden on the programmer.

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

We manually classified reported invariants as potentially relevant or not relevant based on our own judgment, informed by careful examination of the program and the test cases. We judged an invariant to be potentially relevant if it expressed a property that was necessarily true of the program or expressed a salient property of its input, and if we believed that knowledge of that property would help a programmer to understand the code. We made every effort to be fair and objective in our assessments. The relevance of a set of invariants is the percentage of the set's members that are potentially relevant.

4 METHOD

This paper reports experiments over two bodies of code. The first is all the formally-specified programs in chapters 14 and 15 (the first chapters that contain such programs) of *The Science of Programming* [Gri81]. These programs were derived from formal specifications, which represent what the author considered important about those programs. All the programs are quite small, and we built simple test suites of our own. The second body of code is three programs used in testing research, originally from Siemens [HFGO94] and subsequently modified by Rothermel and Harrold [RH98]. These programs — replace, tcas, and tot_info — come with extensive test suites and represent small but realistic programs written without thought for formal invariants. Figure 3 gives the sizes of these programs.

We measured results for each technique while using the best version of the other techniques. This provides a fair baseline against which to evaluate the improvement due to a given technique. In a few cases this was not possible. For example, C lacks polymorphism, so we could only assess polymorphism elimination for Java programs; however, our implementation of Lackwit-style comparability checking for Java is still underway.

5 POLYMORPHISM ELIMINATION

Polymorphism permits functions and containers to manipulate objects of multiple runtime types. Polymorphism also enables code sharing and reuse and provides flexibility for future change, among other benefits. Variables that are declared polymorphically — as with Java's `Object` type or any other base type — often contain objects of only a single runtime type. (Consider a polymorphic list that a particular program uses to hold Integer objects.) In these cases, it is desirable to detect properties over the runtime values that would not be sensible for arbitrary objects of the declared type.

However, to reduce Daikon's complexity and increase its performance, the data trace format for a program is statically determined during instrumentation. Consequently, Daikon cannot directly find invariants over polymorphic variables, whose exact type and data fields are unknown until runtime.

A simple two-pass technique permits Daikon to detect runtime-type-specific invariants over polymorphic variables. The front end causes data traces to include the runtime types of polymorphic variables. Such values are treated like any other variable that was explicitly present in the source code. If an invariant is discovered over the class (such as it being a specific type whenever it is non-`null`), then that information is fed back into the system via a source-code comment. The front end reads these comments and treats the variables as having the specified types. In particular, fields specific to the annotated type can be accessed and provided to the invariant detector. This technique adds invariants over quantities that would not be accessible otherwise.

We assessed this technique on the first five Java programs from a data structures text [Wei99]. The data structures include polymorphic linked lists, stacks and queues (implemented using both linked lists and arrays), and trees. The test cases provided with the programs manipulate sorted collections of `MyInteger` objects. (`MyInteger` implements the `Comparable` interface, whereas Java 1.1's `Integer` does not.) On the first pass, Daikon was unable to detect the sortedness of the collections, because it was provided only the hashcodes and classes of the elements. The second pass, however, reported additional relations such as the following object invariant for class `LinkedList`.

```
header.closure(next).element.value
     is sorted by <=
```

An object invariant is a data well-formedness condition that is satisfied on entry to and exit from each public method. This particular invariant states for every `ListNode` object, the collection of `value` fields of the elements reachable from the `ListNode`'s `header` field is sorted.

The name of the variable requires some explanation. The relevant declarations are:

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

For recursive fields such as `next`, the front end outputs the reachable nodes as a collection. The notation `header.closure(next)` is the collection of `ListNode` objects reachable from `header` through `next` fields. A field reference applied to a collection indicates the collection made up by taking that field reference for each element of the original collection. As an example, `myarray.myfield` indicates the array formed by taking the `myfield` field of every element of `myarray`. Thus, `header.closure(next).element` is the collection of elements reachable from `header`. These have declared type `Object`, but a previous pass found them to be of runtime type `MyInteger`, which has field `value` of type `int`. The result is a sortedness invariant over a realistic and useful (albeit somewhat complicated to describe) collection.

In other data structures, Daikon found similar invariants (such as sortedness of a tree or membership in a collection); details are provided elsewhere [EGKN99].

6 REDUNDANT INVARIANTS

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

Redundancies can be suppressed at three stages in invariant detection. First, a derived variable — an expression that is treated like a variable by invariant detection — should not be introduced if it will be equal to another variable (the first element of array `a` is the same as the first element of array `a[0..i]`) or if it will be nonsensical (`a[i]` when `i` is known to be negative). Up to half of derived variables fall into one of these categories. Invariant detection runtime is potentially cubic in the number of variables at a program point because Daikon computes unary, binary, and ternary invariants, so such savings are significant. Second, invariants whose truth or falsehood is known *a priori* need not be checked. Suppressing false invariants has a relatively small effect, because false invariants tend to be falsified quickly and are not considered thereafter. Suppressing true invariants has a bigger payoff, since such invariants would be checked for all values computed by the target program over its test suite. In fact, most true invariants can be identified beforehand. Third, some redundant invariants may slip through these other tests and should be suppressed before being out-

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

Figure 4: Suppression of redundant computation and output. "Gries" is formally specified textbook programs [Gri81]; the others are C programs [HFGO94, RH98].

put; such tests prune away quite a bit more of the potential output. This last test is a user interface improvement only, while the first two improve both the output and the runtime.

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

Figure 4 illustrates the effect of using implication to avoid work (primarily the top portion) and reduce the amount of output (primarily the bottom portion).

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

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

The bottom of Figure 4 first shows the total number of invariants that were instantiated and checked. These are subdivided into falsified invariants that do not hold for some runtime variable values, unjustified invariants that are not falsified but for which the statistical tests do not support reporting them, redundant invariants that are not falsified but are implied by some other non-falsified invariant, and the remaining invariants that are reported by the system. The "Suppressed" line gives the total number of invariants that were never instantiated, checked, or reported. The figure breaks this number down into those that were known *a priori* to be false and those that were known to be true but redundant.

Daikon's runtime is more dependent on the number of non-falsified invariants (which are necessarily checked against all samples at that program point) than the number of potential invariants. Thus, the number of suppressed invariants should be compared not to the total number of instantiated invariants, but to the number that are not falsified. Implication substantially reduces the number of costly, non-falsified invariants that must be checked. The smallest benefit came for the Gries programs, where the suppressed invariants account for 46% of the total non-falsified invariants that would have been computed otherwise; for `replace` it rises to 83%, and the remaining programs are over 94%.

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

In a few cases, staging of inference did not eliminate all implied invariants before they were introduced; often this was because some invariants are introduced simultaneously so they can be checked together rather than making multiple passes over (summaries of) the data. Removing these invariants reduced the size of the output by about a quarter on average. (See the fourth line ("redundant") in the second half

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

Figure 5: C code for Gries's Program 15.1.1 [Gri81].

of Figure 4.) This lessens the burden on the user of sifting through them without decreasing the information content of the output.

7 COMPARABILITY

Invariant discovery can report unexpected but potentially useful relationships that programmers otherwise might not consider. To increase the likelihood of such serendipity, the initial Daikon prototype compared all singletons, pairs, and triples of program variables (locals, parameters, and globals), derived variables, return values, and (for non-entry points) variables that represent the initial values of variables in scope.

Daikon did report invariants that were both useful and unexpected [ECGN]. It also reported some accurate but uninformative invariants. For example, for `replace` Daikon reports $\mathsf{done} < \mathsf{pat}[0]$ where `done` is a boolean and `pat` is a character array. That invariant is unlikely to be relevant to any programming task, because the variables have nothing to do with one another and are of different types.

Restricting which variables are compared to one another causes some potential invariants not to be considered. This reduces completeness, improves runtime, and could improve or reduce accuracy depending on whether the suppressed invariants are relevant or not.

We compared four methods for computing a comparability relation:

- *Unconstrained*. Consider all variables to be comparable to one another.
- *Source types*. Two variables are comparable if and only if they are declared to have the same type in the program.
- *Coerced types*. Two variables are comparable if their program types are coercible to one other. For example, C automatically coerces `int`s to `long`s, so this approach considers such variables comparable.
- *Lackwit types*. Two variables are comparable if they can contain the same value or values that can interact via program expressions [OJ97]. For example, if `a=b` or `a+b` appears in the program, then `a` and `b` are given the same Lackwit type.

Consider the code in Figure 5. The unconstrained approach considers all the scalars (including array elements, indices, and addresses) comparable to one another. The source types approach makes `i` and elements of `b` comparable, and `s` comparable to `n`; but (for example) `i` is not comparable to `n`, since they have different declared types. In this example, coerced source types are the same as unconstrained, since `int` and `long` can be coerced to each other.

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

This small example also demonstrates the potential downside of using type-inference-based comparability to guide invariant detection. If all elements of `b` are positive, then $\mathsf{i} \leq \mathsf{s}$, but that invariant would not be computed because it involves variables that are incomparable, according to Lackwit. Although it is easy to generate such examples, we have found few if any in real code and do not believe that they will be common in practice: Lackwit tends to capture programmers' intuitive definition of comparability, particularly since it operates interprocedurally and thus takes account of surrounding context. (Furthermore, in this particular example, the programmer could also indirectly but easily infer the unreported invariant because Daikon would report that all elements of `b` are positive.)

Using the Siemens programs described earlier, we measured both the number of variable pairs considered comparable by each technique and also the differences among invariants produced using each of the approaches.

7.1 Reduced Comparability

Compared to the unconstrained approach, how does each of the three other approaches fare in reducing the number of comparable variables that the Daikon engine must consider?

Figure 6 lists, for each method, the average number (over the three Siemens programs) of variables comparable to a given variable. The unconstrained approach makes each variable comparable to every other variable, so at a program point with n variables in scope, each variable is comparable to $n - 1$ others. For these programs, using program type constraints reduces the number of comparable variables by roughly 30%, while Lackwit reduces the number by nearly 90%.

Figure 7 presents the same data, but broken down by the number of variables in scope at a program point. As the number of variables in scope grows, the constraints of source types or coerced source types become less effective (compared to unconstrained).

Comparability	Pairs	Ratio
Unconstrained	10.3	1.000
Source types	7.1	.685
Coerced types	7.9	.763
Lackwit	1.2	.116

Figure 6: Average number of other variables to which a variable is comparable. For a randomly chosen variable in the Siemens suite, this table indicates, for each of the four comparability relations, how many other variables the given variable is expected to be comparable to. The final column shows the ratios between each method and the unconstrained method.

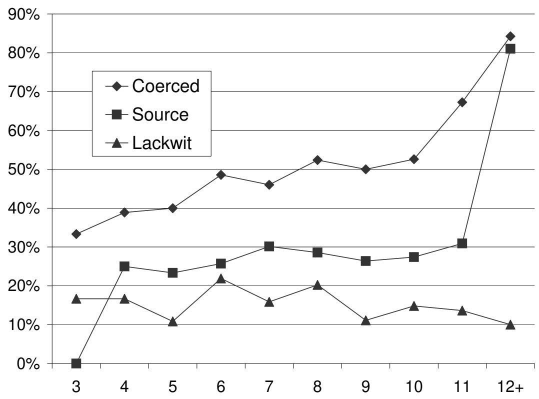

Figure 7: Reduction in comparability achieved by various static comparability analyses, graphed against number of variables in scope at a program point. The graph indicates, for a randomly chosen program point in the Siemens suite at which the specified number of variables is in scope, how much each comparability method reduces comparability, compared to the unconstrained case. The data of Figure 6 aggregate this information for all program points, regardless of number of variables. The numbers of variables do not include original values for parameters or other derived variables. The "12+" datapoint includes all program points with 12 or more variables in scope.

This is not too surprising, since the number of types does not increase as a function grows (at least in a language like C), so there is more sharing of a set of types as programs grow. In contrast, the Lackwit values tend to decrease as the number of variables increases. This suggests that the programs implicitly partition value flows; as the programs grow, the number of partitions tends to increase.

7.2 Improved Relevance

The key questions are whether the static reduction of variable comparability leads to a corresponding reduction in the size of the reported output and, if so, whether the removed invariants are likely to be irrelevant. In addition, more restrictive comparisons might lead to performance improvements.

To address these questions for the three Siemens programs, we ran each program using the same trace files but with the four different comparability relations. For `replace` we used

Comparability	Total	Binary	Time
Source types	78%	61%	91%
Coerced types	78%	74%	96%
Lackwit	55%	27%	75%

Figure 8: Percentage of total and binary invariants reported, and time to compute all invariants, compared to unconstrained comparability.

3000 test cases randomly selected from the provided set, while the others used their full set of provided test cases, about 1500 cases each. Figure 8 shows the resulting data organized by the comparability relation and averaged over the three programs. The total number of invariants, the number of binary invariants, and computation time are shown, all as percentages of unconstrained comparability. (Comparability does not directly affect unary invariants.)

Quantitatively, Lackwit significantly reduces the binary invariants reported (which in turn reduces the total invariants reported). The variance of improvement is high: the most limited (but still significant) reduction is for `tot_info`, where Lackwit reports 62% of unconstrained; the most extreme is for `replace`, which reports just under 3% for Lackwit as compared to unconstrained. The `tot_info` data are a bit unusual, since Daikon discovers only a small number of binary invariants (13 for unconstrained and eight for both source types and Lackwit). In any case, on these programs, Lackwit comparability provides a significant reduction in invariants reported compared to any of the other comparability relations. A substantial performance improvement (which also has a large variance) is also achieved using Lackwit types.

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

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

```
15.1.1:::LOOP   986 samples
  N = size(B)
  S = sum(B[0..I-1])
  N in [0..35]
  I >= 0
  I <= N
  B
    All elements in [-6005..7680]
  sum(B) in [-15006..21144]
  B[0..I-1]
    All elements in [-6005..7680]
```

Figure 9: Invariants inferred for the loop head of Gries's Program 15.1.1 [Gri81], with every sample contributing to invariant confidence. Compared to Figure 2, which used the "assignment" rule for determining when a sample contributes to confidence, the last three invariants (which appear in at the loop head but not elsewhere, even though array B does not change during the program's execution) are extraneous. The invariants for the procedure entry and exit are unchanged from Figure 2.

8 REPEATED VALUES

Daikon reports only invariants that pass a statistical confidence test; properties that could easily have occurred by chance are not reported, as they are likely to be accidents of the data. In other words, an invariant is reported only if the null hypothesis, which states that the observed data are not unusual, can be rejected with a user-specified confidence.

For instance, given $0 < x < 10$ and $0 < y < 10$, if there are only three $\langle x, y \rangle$ pairs, then the invariant $x \neq y$ should not be reported, even if it is true for those three pairs: they do not support such a generalization. If there are 10,000 such pairs, but x was never equal to y, then the relationship is likely to be more than a coincidence.

Some unjustified invariants remain, however, because multiple visits to a program point without assignment to a variable can cause the repeated values for the variable to be overweighted in the statistical tests. For example, additional samples for a loop-invariant variable could cause Daikon to report invariants inside the loop that are true but are not considered statistically justified outside the loop — even though the variable values are the same in both cases. As a concrete example, compare Figure 9, which shows the invariants for the Gries program using this rule, to Figure 2, which uses the "assignment" rule described below. Figure 9 contains three extra invariants at the loop head because of undue confidence. Procedure invocations and other sorts of control flow cause similar anomalies.

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

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

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

Assignment. A sample contributes to invariant confidence if the variable was assigned since the last time the program point was visited. This approach requires significant cooperation from the instrumenter (see below for details).

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

Random proportionate to assignments. A sample contributes to invariant confidence when the value changes, and otherwise with a probability chosen so that the total number of contributing samples is the same as in the "assignment" strategy. Although not practical in that it requires the same instrumentation as the assignment strategy, normalizing for the number of contributing samples permits an assessment of the assignment strategy's choice of samples.

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

For the "assignment" rule, the instrumenter inserts code into the program under test to track variable assignments with a boolean bit vector indexed by program point. When a variable is assigned, all of its bits are set to true ("assigned since last visit to program point"). The instrumentation at a program point clears all of its bits for all the variables in scope. The instrumentation maintains the modification bits via a status object for every traced variable. To appropriately mirror the semantics of parameter passing on variables, the status object is passed by the same mode as its associated variable. Thus, if a parameter is pass-by-value, then it gets a new status object that is a copy of the incoming status; if the parameter is pass-by-reference, the status object is passed in by reference as well.

This approach readily solves the problem with loops, as a variable assigned outside the loop is counted the same for program points inside the loop as at the loop's entry and exit. However, it does not work as well for repeated function calls because each call counts as a unique assignment to its call-by-value parameters, even if the arguments are identical, be-

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

Figure 10: Number of differing invariants reported over the Siemens programs when using various rules for determining whether a sample increases confidence. The baseline for these measurements is the "assignment" rule; the four rules listed along the top of the table are compared to that baseline. See Section 8 for details.

cause the parameters are assigned outside the function's program points. Indeed, a number of undesirable invariants resulted from this limitation, which is corrected in Daikon's Java instrumenter. That instrumenter rewrites the program to allocate extra space within objects, so that the timestamps are contained within the objects themselves. It maintains a timestamp per lvalue, plus a timestamp per ⟨lvalue, program point⟩ pair.

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

We compared the rules listed above to assess their relative benefits. We omitted the Gries programs because they did not come with test suites, and the tests we constructed might be better or worse than those constructed in practice by a tester. For each of the Siemens programs, we repeated invariant detection using each of the five rules listed above to determine which samples should contribute to invariant confidence. We then classified, by hand, each of the differences in the output (a total of 165 differing invariants) as either relevant or irrelevant, according to the criteria of Section 3.

Figure 10 presents the results of this analysis. Each rule for whether a sample adds confidence was compared to the baseline "assignment" rule. The differences were approximately 3% as large as the full invariants (5059 invariants over more than 200 program points; this is the sum of the last three columns of the "reported" line of Figure 4).

Among the techniques, only the "value" rule causes reporting of relevant invariants that are not justified according to the "assignment" rule. All the other rules miss some invariants reported by the "assignment" rule and add more irrelevant invariants than they prune. Because of its simplicity and lack of need for special runtime support, the "value" rule may be competitive with the "assignment" rule in practice, even if the latter tends to result in a slightly more relevant set of invariants.

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

9 RELATED WORK

Information Retrieval. One can reasonably consider Daikon's invariant discovery process as a form of information retrieval [Sal68]. Information retrieval applies a query to a corpus, returning likely matches to that query from the corpus. The conventional approach for assessing the effectiveness of an information retrieval technique is to measure *recall* and *precision*. Recall captures what portion of the true matches in the corpus are in the set of actual matches found by the given technique — essentially, how complete the retrieval is. Precision captures what portion of the actual matches are in the set of true matches — essentially, how pure the retrieval is. Like our approach for assessing relevance, recall and precision are based in qualitative analysis, since determining the true matches is generally subjective. Because the true set of relevant invariants is unknowable and potentially infinite, we cannot measure recall.

Lackwit. Daikon uses Lackwit's analysis to reduce the cost and improve the quality of a dynamic analysis. O'Callahan and Jackson developed Lackwit as a static technique to support reverse or re-engineering [OJ97]. The analysis was designed to be scalable (in particular, computationally inexpensive even on large programs) and to handle complex language constructs such as aliasing and higher-order functions (so that languages such as C could be analyzed). The use of the analysis is either query- or graphically-based, allowing programmers to answer questions about a program's structure and to find various anomalies such as abstraction violations, unused data structures, and memory leaks.

Daikon does not use Lackwit in the same way that a programmer would directly. Exploring how a variety of tools like these collectively aid programmers in managing evolving systems is a challenging task that is far beyond the scope of this paper.

Statistical Significance. We use statistical confidence tests to decide whether a specific invariant should be reported. An invariant is reported if the null hypothesis which states the observed distribution occurred by chance can be rejected at a certain level of confidence. These confidence levels do not indicate whether the invariant is (or is likely to be) actually true in practice, because there is no guarantee that the test suite fully characterizes the program's actual execution environment [Goe85]. Additionally, since the actual distribution of variable values is not known, the exact value of this confidence is less important than its order of magnitude and comparisons among confidences. Finally, reporting an invariant does *not* imply that the invariant is relevant (useful to a programmer for a specific task).

10 CONCLUSION

Dynamically inferring program invariants expands a programmer's ability to gather information pertinent to software evolution tasks. By combining this approach with existing static analysis techniques, a programmer may be able to gain the best of both the static and the dynamic worlds. Static analyses tend to be sound, but the state of the art does not accurately handle very large programs or all programming languages and features. In contrast, dynamic techniques tend to be more practical in terms of applicability to arbitrary programs and often seem to provide useful information despite their inherent unsoundness.

This paper describes four techniques for improving the performance of invariant detection and the relevance (usefulness to programmers) of the reported invariants. First, a simple two-pass technique finds invariants over variables with polymorphic types but which in practice contain values of more limited run-time types. Second, eliminating implied invariants makes the performance of the engine tractable and removes accurate invariants that clutter the output without adding any additional information. Third, type-based techniques can reduce the set of variables examined and thus the number of potential invariants checked; the eliminated invariants are highly likely to be irrelevant. More invariants are checked, with the concomitant performance costs, but relevant invariants are added to the output. Fourth, mechanisms for preventing a single variable value from being counted multiple times when testing statistical confidence prevent chance properties from being reported as justified invariants.

Elsewhere we report on how to discover invariants over recursive pointer-based structures and conditional invariants that are not universally true [EGKN99]; an example is $\mathsf{p} = \mathsf{NULL}$ or $\mathsf{p.left} \in \mathsf{mytree}$. We are also exploring a richer user interface for Daikon, which would support some graphical displays (when appropriate) and give the programmer more control over both instrumentation and display of invariants. We also intend to perform more and larger studies of how programmers use dynamically detected invariants in practice: it doesn't matter how fast or accurately they can be computed if they do not assist real programmers in real tasks. Initially, we plan to pursue a case study approach to answering this question.

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

ACKNOWLEDGMENTS

Rob O'Callahan provided the Lackwit tool, enhancements, bug fixes, and fruitful discussions. Jake Cockrell implemented the Daikon front end for C and an alternative implementation of the Lackwit type system, which provided us with significant insights. Yoshio Kataoka, visiting the University of Washington from Toshiba Corporation Research and Development, was an early user of Daikon and provided valuable suggestions and ideas. The anonymous referees provided helpful suggestions on the presentation. Support for the research comes from NSF grant CCR-9970985, an IBM Graduate Fellowship, and gifts from Edison Design Group, Microsoft Corporation, and Toshiba Corporation. Griswold is currently on sabbatical from UCSD at Xerox PARC.

REFERENCES

[ECGN] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*. To appear. A previous version appeared in *Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, May 19–21, 1999.

[EGKN99] Michael D. Ernst, William G. Griswold, Yoshio Kataoka, and David Notkin. Dynamically discovering pointer-based program invariants. Technical Report UW-CSE-99-11-02, University of Washington, Seattle, WA, November 16, 1999.

[Goe85] Amrit L. Goel. Software reliability models: Assumptions, limitations, and applicability. *IEEE Transactions on Software Engineering*, SE-11(12):1411–23, December 1985.

[Gri81] David Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.

[HFGO94] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, pages 191–200, May 1994.

[OJ97] Robert O'Callahan and Daniel Jackson. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 19th International Conference on Software Engineering*, pages 338–348, May 1997.

[RH98] Gregg Rothermel and Mary Jean Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, 24(6):401–419, June 1998.

[Sal68] Gerard Salton. *Automatic Information Organization and Retrieval*. McGraw-Hill, 1968.

[Wei99] Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java*. Addison Wesley Longman, 1999.