Quickly Detecting Relevant Program Invariants

Michael D. Ernst[†], **Adam Czeisler**[†], **William G. Griswold**[‡], and **David Notkin**[†]
University of Washington Technical Report UW-CSE-99-11-01
November 15, 1999

[†]Dept. of Computer Science & Engineering
University of Washington
Box 352350, Seattle WA 98195-2350 USA
+1-206-543-1695
{mernst,czeisler,notkin}@cs.washington.edu

[‡]Dept. of Computer Science & Engineering
University of California San Diego, 0114
La Jolla, CA 92093-0114 USA
+1-619-534-6898
wgg@cs.ucsd.edu

http://www.cs.washington.edu/homes/mernst/daikon/

ABSTRACT

Explicitly stated program invariants can help programmers by characterizing 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 infer invariants from program traces by looking for patterns in and relationships among variable values. Although the original prototype recovered invariants from formally-specified programs, and the invariants it detected in other programs assisted programmers in a software evolution task, it suffered from reporting too many invariants, many of which were not useful. This paper considers four approaches for increasing the relevance of invariants reported by a dynamic invariant detector: suppressing implied invariants, limiting which variables are compared to one another, exploiting polymorphism, and ignoring unchanged values. We show that these approaches improve the output by removing many irrelevant invariants or by introducing additional relevant invariants. They also improve runtime by reducing the work done by the invariant detector.

1 INTRODUCTION

The Daikon project is exploring the use of dynamic methods for discovering likely program invariants, with a particular interest in supporting software evolution tasks [ECGN99]. The initial prototype demonstrated the feasibility of the approach, discovering invariants both for programs in which invariants were an intimate part of their construction, and for programs apparently written without thought about formal invariants. Furthermore, programmers found the detected invariants useful in modifying an undocumented C program. This paper describes significant improvements in two complementary dimensions: increasing both the performance of the underlying engine and also the relevance of the reported invariants. In particular, we report on four techniques.

Implication. 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. We prune the search space by suppressing implied invariants; for instance, if our engine determines that `x = y`, then no inference need be done for `y`, as invariants over `x` imply similar ones over `y`. Section 4 describes the details of 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 increase performance by reducing the number of invariants to be checked, and can reduce the number of irrelevant invariants reported. Section 5 compares two approaches to limiting the number of comparisons: using the declared program types of variables and using the flow-based types inferred via type inference by the Lackwit tool [OJ97].

Polymorphism Elimination. Variables declared polymorphically (such as Java's `Object` type) often contain only a single type at runtime. Knowledge of the runtime type permits instrumentation to extract fields or to print values more suitably, without suffering from the high costs of dealing with polymorphism at inference time. We address this issue via a two-pass technique that first reports the runtime class of objects and detects invariants over them, then feeds that information back to a second pass of instrumentation and invariant detection that takes advantage of the inferred types. Section 6 describes this technique, showing that it enables reporting of relevant but otherwise undetected invariants.

Repeated values. Only invariants that are statistically justified — relationships that do not appear to have occurred by chance alone — are reported. These tests depend on the number of samples seen. When a variable's value is traced repeatedly without the variable being reset — such as a variable that remains unchanged within a loop — then the number of samples is artifi-

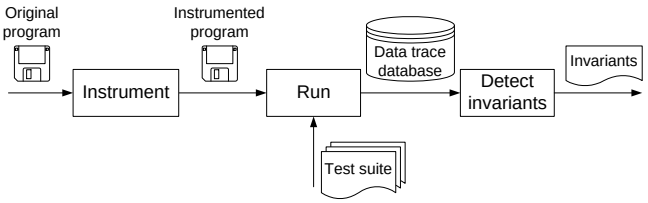

Figure 1: An overview of dynamic invariant inference as implemented by our tool, Daikon.

cially inflated and properties of the variable may be inappropriately reported. We address this problem by ignoring some instances of repeated values. Section 7 reports on the relative effectiveness of several rules for ignoring repeated values.

Our basic approach, including a concrete example of dynamic invariant extraction, is described in Section 2. Section 3 addresses the question, "What do we mean by relevance?", and also lays out our basic experimental method. The four techniques sketched above are addressed in the subsequent four sections. Section 8 briefly surveys related work, and Section 9 provides a synopsis of the results and concludes with a discussion of some future work.

2 BACKGROUND

Dynamic invariant detection [ECGN99] 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 inferring 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. 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 inference engine is language independent, and we currently have program instrumenters for C, Java, and Lisp.

Daikon infers invariants at specific program points such as loop heads and procedure entries and exits. 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 we compute, where x, y, and z are variables, and a, b, and c are computed constants:

- invariants over any variable, such as being constant ($x = a$), taking its values from a small set ($x \in \{a, b, c\}$), etc.
- invariants over a single numeric variable, such as in a range ($a \leq x \leq b$), never zero, modulus ($x \equiv a \pmod b$), etc.
- invariants over two numeric variables, such as a linear relationship ($y = ax + b$), an ordering comparison ($x \leq y$), functions ($x = fn(y)$) for built-in unary functions, or combinations of invariants over a single numeric variable ($x + y \equiv a \pmod b$), etc.
- invariants over three numeric variables, such as a linear relationship ($z = ax + by + c$), etc.
- invariants over a single sequence variable, such as minimum and maximum sequence values, lexicographical ordering, element ordering, invariants over all sequence elements treated as a single large collection, 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 ($i \in s$).

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. To check for a potential invariant, we examine 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 invariants over `a[lasti]` may be of interest, even if that expression does not appear in the program text. Derived variables are treated just like other variables by the inference engine, 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 engine can report the invariant $i < size(A)$ 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.

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

15.1.1:::EXIT    100 samples
  B = B_orig                           (96 values)
  N = I = N_orig = 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])                   (95 values)
  N >= 0                               (24 values)
  I >= 0                               (36 values)
  I <= N                               (363 values)
```

Figure 2: Invariants inferred for Gries's Program 15.1.1 [Gri81], which sums array `B` into variable `S`. The boxed invariants were specified by the program's author. See Figure 5 for C source code.

Example. As a simple example of Daikon's output, consider a program that sums the elements of an array into another variable. Our 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 the resulting output of our invariant detector at the three program points. This output contains all the invariants specified by the programmer, who derived the programs from the formal specification. It also adds additional invariants, such as that `N` is the length of array `B` (which is crucial to the correctness of the routine but was incorrectly omitted) and that the routine does not modify its arguments (which could be an aid to understanding the program).

3 RELEVANCE and METHOD

An invariant is *relevant* if it productively assists a programmer in the programming activity at hand. Therefore, relevance is inherently contingent on the particular task, as well as the programmer's capabilities, working style, and knowledge of the code base. No automatic system can know this context. A consequence of this is that Daikon computes and reports some invariants that the user does not find relevant. This reduces the benefit to the user and increases Daikon's runtime.

There are at least two ways to improve the relevance of the invariants that are computed and reported. First, the programmer — who *is* privy to much of the context — could control the invariant inference process. Second, heuristics could be added to Daikon to improve the relevance of the reported invariants. In this paper, we investigate the second approach, which lessens the initial burden on the programmer.

Program	NBNC LOC	Num Procs
calc	163	5
replace	516	21
tcas	136	9
tot_info	274	7

Figure 3: Basic data for 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.

The subjective definition of relevance complicates assessment of our techniques for improving the relevance of reported invariants. In general, we report a combination of quantitative and qualitative measurements for each of our four techniques. Two of the techniques — polymorphism elimination and implication — inherently improve relevance. Polymorphism elimination reports additional invariants that are possibly of interest that were not previously computed; implication removes invariants from the output without losing any information *per se*. The other two techniques are more heuristic in nature, requiring more careful qualitative assessment of the differences in invariants reported using them. In principle, our analysis is inspired by notions in information retrieval (see Section 8): we manually classify each each reported invariant as relevant or not relevant based on our own judgment, informed by careful examination of the program and the test cases.

Each of the four techniques are applied to as many as five bodies of code. The first body is programs taken from *The Science of Programming* [Gri81]; we include all the formally-specified programs in chapters 14 and 15. These programs were derived from formal specifications, so we know what an ideal invariant detector would report for them. All the programs are quite small, and we built simple test suites on our own. The other four programs are originally from Siemens [HFGO94] and were subsequently modified by Rothermel and Harrold [RH98]. These programs — `calc`, `replace`, `tcas`, and `tot_info` — are provided with extensive test suites and represent modest-size, broadly available programs written without thought for formal invariants. Figure 3 shows some basic statistics on these four Siemens programs.

We generally performed our measurements for each of the techniques in the presence of the "best" version of the other techniques. This provides a more reasonable baseline against which to evaluate the improvement due to a given technique. In a few cases this was not possible for mechanical reasons. For example, we could only assess polymorphism elimination for Java programs, but our implementation of Lackwit-style type inference for Java is still underway.

	Gries	calc	replace	tcas	tot_info
Variables	558	97	969	1438	625
non-canonical	56	8	125	372	53
missing	58	21	352	338	274
canonical	444	68	492	728	298
Derived vars	234	52	637	1078	420
Suppressed	126	32	507	1198	40
Invariants	275162	89928	540746	5497210	1010411
falsified	272454	89393	537284	5473523	1008386
not justified	1983	308	1749	10694	1091
redundant	207	57	446	9985	130
reported	518	170	1247	3008	804
Suppressed	2788	9384	20186	1686543	101660
falsified	448	49	2648	8925	603
redundant	2340	9335	17538	1677618	101057

Figure 4: Suppression of computation and output by implication for five bodies of code. "Gries" is programs from [Gri81]; the other four are C programs [HFGO94, RH98].

4 IMPLICATION

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 we often fail 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.

Figure 4 illustrates the effects 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. We separately list the number of derived variables (each of which may be non-canonical, missing, or canonical) and the number of derived variables that were suppressed (i.e., not instantiated) because an invariant implied that they would be non-canonical or missing.

The bottom of the figure first shows the total number of invariants that were instantiated and checked. These are subdivided into invariants that are falsified by some sample of variable values, invariants that are not falsified but for which the statistical tests do not support reporting them, invariants that are not falsified but are implied by some other non-falsified invariant, and invariants that are reported by the system. The antepenultimate line gives the total number of invariants that were suppressed and thus were not instantiated or checked. This number is broken down into those that were not instantiated because the engine knows that they would be shown to be false and into those that the engine knows would be shown to be true but redundant.

The number of variables is the most important factor in the number of invariants checked. Since we compute unary, binary, and ternary invariants, the number of potential invariants grows with the cube of the number of variables in scope at a program point. Our rules for using previously-computed invariants to suppress certain derived variables eliminated 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 (which have been determined to be equal to another variable) and variables with possibly non-sensical 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 that we are not able to measure it; when we disable any of these optimizations, our system is slowed down by orders of magnitude and eventually runs out of over 256MB of memory—despite the fact that we aggressively intern all data, so (for example) there are no two distinct integer arrays anywhere in the implementation that contain the same contents.

The bottom half of the table shows improvements due to implication that go beyond elimination of certain variables. One observation is that our system checks many invariants. Most of those are falsified quickly, so runtime is most dependent on the number of non-falsified invariants (which are necessarily checked against all samples at that program point). 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 again 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 three programs are all over 94%.

These, 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). Again, we are not able to provide runtime estimates for eliminating these checks, because the system simply does not run in their absence.

In a few cases, our staging of inference was not able to

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. However, we were able to substantially reduce the size of the output by removing these invariants. This lessens the burden on the user of sifting through them without decreasing the information content of the output.

5 COMPARABILITY

A major purpose of invariant inference is to report serendipitous relationships that the user was not previously aware of and might never have come up with independently. This argues for comparing as many variables as possible during invariant discovery.

At the same time, the sets of variables to compare can be very large, as they include singletons, pairs, and triples of program variables (locals, parameters, and globals), derived variables, return values (if appropriate), and (for exit points) variables that represent the initial values of variables in scope. To compare all possible combinations of variables in this set has two potential downsides.

First, the number of possible invariants grows with the cube of the number of variables. In practice, however, the cost of computing invariants is linear in the number of invariants reported, not the number checked [ECGN99]. This is because most invariants are falsified after only a small number of value tuples, and once an invariant is falsified, we prune it from the search.

Second, based in part on experience, comparing all combinations may cause accurate but uninformative invariants to be reported. For example, for `replace` our engine reports `done` < `pat[0]` where `done` is a boolean and `pat` is a character array. Although this is true and statistically justified — and thus reported — it represents an invariant that is unlikely to be relevant to any programming task.

For these reasons, we decided to explore alternative approaches for restricting which variables may be compared to one another. These approaches trade off the number of irrelevant invariants reported against serendipity and completeness. We considered four different methods for computing a comparability relation:

- *Unconstrained.* Consider all variables comparable to one another.
- *Source program types.* Two variables are comparable if and only if they are declared to have the same type in the program.
- *Coerced source program types.* Two variables are comparable if their program types are coercible to one other. For example, C automatically coerces `int` to `long`s, so this approach considers such variables comparable.
- *Lackwit types.* Two variables are comparable if they are unified by type inference over a type system that unifies variables between which values can flow [OJ97].

```
// 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], which sums array `B` into variable `S`.

Consider the C code of Figure 5. The unconstrained approach considers all the scalars, including array elements and indices, comparable to one another. The source types approach makes `i` and elements of `b` to be 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 uses polymorphic type inference to capture value flow (or ability to contain the same runtime value). 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]`". (These 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 inference. If all elements of `b` are positive, then $i \leq s$, but that invariant would not be computed because it involves incomparable variables. Although it is easy to generate such examples, we do not believe that they will be common in practice, and have not yet found any in real code: we believe that Lackwit tends to capture our intuitive definition of comparability, particularly since it operates interprocedurally and thus takes account of surrounding context. (In this particular example, Daikon would directly report that all elements of `b` are positive.)

Using the four Siemens programs described earlier, we performed two basic sets of measurements to assess the effectiveness of each of these approaches. The first set of measurements focuses on the degree to which each approach reduces the number of comparable variable pairs. The second set of measurements studies the invariants produced using each of these approaches.

Comparability	Pairs	Ratio
Unconstrained	11.7	1.000
Source types	6.3	.541
Coerced types	7.0	.602
Lackwit	0.6	.054

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 methods of prescribing comparability, 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.

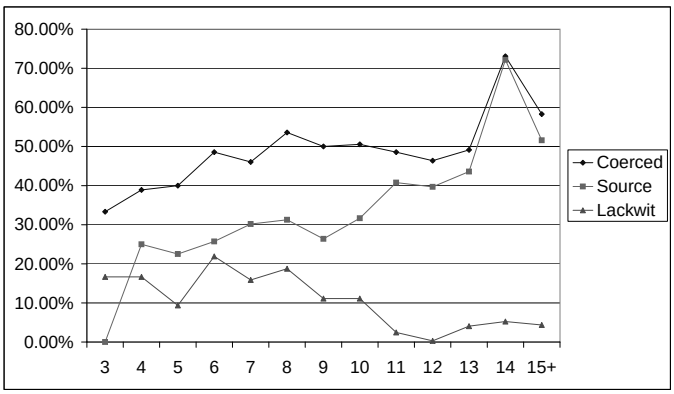

Figure 7: Reduction in comparability, graphed against number of variables at a program point. The graph indicates, for a randomly chosen program point at which the specified number of variables is in scope, how much each comparability method reduces comparability, compared to the unconstrained case. The numbers of variables do not include original values for parameters, nor derived variables. The "15" datapoint includes all procedures with 15 or more variables in scope.

Reduced Comparability

Compared to the original unconstrained approach, how does each of the three approaches above 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 Siemens programs) of other variables comparable to a given variable. The unconstrained approach makes each variable comparable to each other variable, so at a program point with n variables in scope, each variable is comparable to $n-1$ others. For the programs we analyzed, using program type constraints reduces the number of comparable variables by roughly a factor of two, while Lackwit reduces the number by a factor of about 20.

Figure 7 presents the same data, but broken down by the number of variables in scope at a program point. Source types and coerced types tend to grow (as a percentage of unconstrained) as the function size grows. 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 in the face of growth. In contrast, the Lackwit values tend to decrease as the number of variables increases. This seems to suggest that the programs we analyzed have some kind of implicit partitioning of value flows; as the programs grow, the number of partitions tends to increase as opposed to increasing the size of each partition.

Improved Relevance

The static reduction of variables to compare to one another is valuable, but the key questions are whether it reduces the size of the reported output and, if so, whether the removed invariants are likely to be irrelevant.

We report some basic data addressing these questions for the four Siemens programs. We could not produce the full analysis over the four comparability methods, since our initial prototype hardcodes a prohibition on comparing scalar variables with different source program types, although all addresses can be compared. (The prototype under development eliminates this restriction.) Thus, our analysis of the reported invariants is limited to assessing how well source types work compared to Lackwit types.

For each of the four Siemens programs, we ran the same trace files through Daikon for the source type and Lackwit comparability relations. For `calc`, we ran over the 13 test cases provided as part of its automatic test harness. For the other three programs, we ran over 300 random test cases from the provided set. We also assessed the runtime improvement resulting from checking invariants over fewer comparable variables; the ratios of inference times using Lackwit comparability to those using source type comparability were 1.19 for `calc`, 0.63 for `replace` 0.73 for `tcas`, and 0.42 for `tot_info`. The `calc` ratio may be skewed by measurement errors in the low absolute running times.

For two programs, `calc` and `tot_info`, using the Lackwit comparability method returned an identical set of invariants as for source types. This is not especially surprising, since `calc` only returns a single binary invariant even in the unconstrained case, and `tot-info` returns only eight binary invariants. (Comparability does not directly affect unary invariants, of course.)

In the other two programs, however, there are significant differences in the invariants computed using Lackwit's comparability method versus the source types method. For `replace`, the unconstrained case computes 548 total invariants, of which 255 are binary invariants. The Lackwit method on `replace` reports 343 total and 50 binary invariants; that is, Lackwit reports 63% of the total invariants reported by unconstrained, and 20% of the binary invariants. For `tcas`, the percentages are 39% (745 vs. 1907) and 21% (305 vs. 1467), respec-

tively.

For `tcas`, one unary invariant was reported that was not reported in the unconstrained case. The Lackwit case reported:

```
Other_Capability in {0, 1}
tcas_equipped in {0, 1}
```

Unconstrained reported:

```
Other_Capability = tcas_equipped
Other_Capability in {0, 1}
```

In the Lackwit case, the two variables happened to be equal, but since they are not comparable using Lackwit, they weren't compared and the equality invariant wasn't reported. The unconstrained case did compare these, but reported them in a different form (removing the `tcas_equipped in {0, 1}` because it is implied by the two invariants shown for unconstrained, and `Other_Capability` is the canonical variable in this case).

Although this data set is quite small, the reductions in the size of the invariant sets on these two programs, especially for binary invariants, is significant and promising. Our qualitative analysis focused on `replace`, both because we are more familiar with it and also because there are fewer invariants to consider manually. Every one of the invariants that the Lackwit approach did not compute looks irrelevant for most likely tasks.

6 POLYMORPHISM ELIMINATION

Polymorphism permits functions and containers to manipulate objects regardless of their runtime types; it 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 values that would not be sensible for arbitrary objects.

Specific design decisions in Daikon prevent it from directly finding invariants over polymorphic variables; this reduces the relevance of the reported invariants by omitting potentially useful properties. The key design decision is that the data trace format is declared *a priori* (at instrumentation time) and read before invariant detection begins. Compared to permitting arbitrary new variables (and values of arbitrary types) to appear in data traces, this decision decreases the complexity and improves the performance and error-checking of the system.

A simple two-pass technique permits Daikon to infer 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 inference engine. The effect of this technique is to add to the output of our system invariants over quantities that would not be accessible otherwise.

To assess our technique, we ran it on a number of Java programs from a data structures text [Wei99]. The data structures include polymorphic linked lists, trees, and more. 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.) The first pass of our system 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 universal invariant in all routines of the `LinkedList` class:

```
header.~next~.element.value
    intra-sequence ordering: <=
```

The "`intra-sequence ordering: <=`" annotation indicates that the list is sorted in non-decreasing order. The name of the variable requires some explanation. Field `header` of `LinkedList` is a `ListNode`. Its `next` field is also of type `ListNode`. For all such recursive fields, the front end outputs the reachable nodes. The notation `header.~next~` is the collection reachable through `next` fields. When an array or other collection has a field element applied, that indicates the array composed of that field of each element of the original collection. Thus, `header.~next~.element` is the elements reachable from `header`. These have declared type `Object` and are supplied to the system as hashcodes (which are good only for determining object equality), but a previous pass found them to be of runtime type `MyObject`, which has field `value` of type `int`. The list of those slots of the elements of the reachable `ListNode`s is non-decreasing.

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

7 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. For instance, given $0 < x, y < 10$, if there are only three (x, y) pairs, then the invariant $x \neq y$ should not be reported. If there are 10,000 such pairs, but x was never equal to y, then the relationship

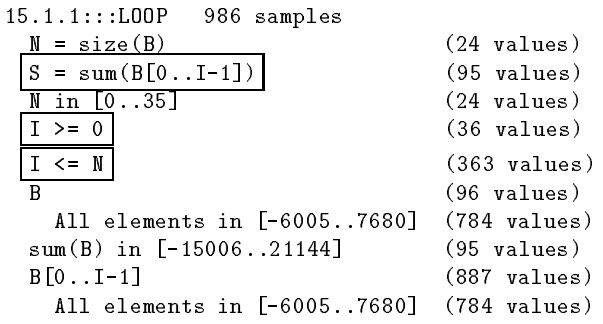

Figure 8: Invariants inferred for the loop head of Gries's Program 15.1.1 [Gri81], with every sample contributing to invariant confidence. Compared to Figure 2, which used the "assignment" rule for determining when a sample contributes to confidence, the last three invariants (which appear in at the loop head but not elsewhere, even though B does not change during the program's execution) are extraneous. The invariants for the procedure entry and exit are unchanged from Figure 2.

is likely to be more than a coincidence.

When a program point is visited multiple times between variable assignments, those variable values can 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. Procedure invocations and other sorts of control flow cause similar problems.

If the dynamic execution path to the 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.

In this section we compare five strategies for determining whether a particular sample of values should increase confidence in an invariant.

Always. Every sample contributes to confidence. While this strategy is trivial to implement (and was the first thing we tried), we quickly found it to be unacceptable. As a concrete example, compare Figure 8, which shows the invariants for the Gries program using this rule, to Figure 2, which uses the "assignment" rule described below. Figure 8 contains three extra invariants at the loop head because of undue confidence.

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.

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" case above.

Except for the "assignment" rule, all the rules are easily implemented by preprocessing data trace files. For the "assignment" rule, Daikon tracks variable assignments with a boolean bit vector indexed by program point. When a variable is assigned, all its bits are set to true ("assigned since last visit to program point"), and the instrumentation at a program point clears all 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 looping problem, 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, because the parameters are assigned outside the function's program points. Indeed, we found a number of undesirable invariants because of this limitation, which is corrected in our Java instrumenter.

As a performance optimization, Daikon can use modification information not only to produce more accurate confidence measures, but also to skip entire samples for an invariant. In particular, if a sample for an invariant is unmodified, the sample can be simply discarded since the sample must have the same values as on the previous visit to the program point and hence the expected relationship must (still) hold.

We compared the rules listed above to assess their relative benefits. For each of the Gries and Siemens programs, we repeated invariant inference 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 553 differing invariants) as either relevant or

	All	Value	Random	Random $\propto$
Added	131	37	163	43
relevant	18	11	24	17
irrelevant	113	26	139	26
Removed	0	78	74	27
relevant	0	10	1	3
irrelevant	0	68	73	24

Figure 9: Number of differing invariants reported 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. The penultimate rule selects which samples add confidence randomly, with probability $\frac{1}{2}$. The last rule selects which samples add confidence randomly, with the probability selected so that the total number of significant samples is the same as for the baseline.

	All	Value	Random	Random $\propto$
Added	63	32	122	31
relevant	6	6	7	6
irrelevant	57	26	115	25
Removed	0	67	71	19
relevant	0	5	1	1
irrelevant	0	26	70	18

Figure 10: Number of differing invariants reported when using various rules for determining whether a sample increases confidence. This table repeats the data of Figure 9, but omits the Gries programs, which have (intentionally) minimal test suites.

irrelevant. An invariant was considered 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 were already familiar with these programs ourselves.)

Figure 9 presents the results of this analysis. Each rule for whether a sample adds confidence was compared to the "assignment" rule, which we took as our baseline. The differences were approximately 10% as large as the full invariants (5747 invariants over more than 200 program points; see the "reported" line of Figure 4). As expected, weighting each sample equally performed significantly worse than the baseline: equal weighting added many irrelevant invariants and only a few relevant ones, while not removing any of the previous output. The other rules did sometimes suppress invariants produced by the "assignment" rule, and they also outperformed equal weighting. Surprisingly, random assignment performed competitively with the "assignment" rule. Switching to random assignment removed quite a few irrelevant invariants, though it added even more. The final column cannot be computed without first computing the "assignment" rule; it is provided primarily for comparison.

The Gries programs skew these results; Figure 10 repeats the data, omitting the Gries programs. We (intentionally) supplied the Gries programs with very small test suites, to determine the effect of such test suites on invariant inference. (We already know that in most cases, increasing test suite size greatly improves the accuracy of invariant detection, by providing counterexamples to undesirable invariants and providing increased confidence in desirable ones.) In the previous data, 45 of the 70 added relevant invariants came from the Gries programs, as did 7 or the 14 removed relevant data. This difference indicates that the "assignment" rule performs relatively better on programs with rich test suites than on programs with a paucity of tests.

8 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. Our approach to assessing our techniques differs from the use of recall and precision in a key way: our "corpus" of invariant properties on a given program is infinite, which precludes us from even theoretically determining the set of true matches.

Lackwit. Our approach uses Lackwit's analysis to reduce the cost and improve the quality of a dynamic analysis. In contrast, O'Callahan and Jackson developed Lackwit as a static technique to support reverse or re-engineering [OJ97]. The analysis *per se* 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.

Although there may be some overlap in the aspects of programs that a programmer could find using Lackwit and using Daikon, the way in which they each use Lackwit's analysis is fundamentally different. 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. Our use of statistics to report invariants is quite different from conventional uses of statistical significance. In particular, we use statistical methods to decide whether a specific invariant should be reported to the user. That is, given an unfalsified invariant, we use a statistical approach to decide if we have sufficient confidence that is is non-random. Since the actual distribution of the data is not known, the absolute value of this confidence cannot be relied upon, but it still provides a useful filter. Additionally, reporting an invariant does *not* imply that the invariant is relevant (as defined above).

9 CONCLUSION

Dynamically inferring program invariants expands a programmer's ability to gather information pertinent to a given software evolution task. 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 are practically limited in terms of the programming languages and sizes of programs to which they can be applied; 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.

In this paper, we described how to improve both the performance of the underlying inference engine — which is critical to scaling the approach — and also the relevance of the reported invariants — which is a critical step in gaining users for the tools. In particular, we described and measured four specific techniques for improving performance, relevance, or both. First, we eliminated implied invariants, which makes the performance of the engine tractable and removes accurate invariants that add no additional information to the output. Second, we explored type-based techniques for reducing the set of variables that the engine compares during invariant discovery. By using the Lackwit type system, we can reduce the computational cost and remove a set of invariants that are highly likely to be irrelevant. Third, we used a two-pass technique that allows us to find invariants over structures that could in principle hold a collection of polymorphic types, but that in practice have instances of only a single type. This increases the performance costs but adds invariants that are likely to be relevant. Fourth, we explored a set of mechanisms for determining when the Daikon engine decides that the value of a variable has been modified; this goes to the heart of our statistical decision procedure that determines when we have enough evidence to report an invariant.

We continue to work on a set of other dimensions related to dynamic invariant inference. Our most significant progress has been in the area of pointer-based data structures, where we can now extract invariants such as p = NULL or p.left ∈ mytree; elsewhere we report on how we discover this class of disjunctive properties over pointers and fields in pointer-based data structures [EGKN99]. We are also taking initial steps towards a richer user interface for Daikon, which would support some graphical displays (when appropriate) and would allow the programmer far more control over what variables and program points are monitored, as well as much more direct control over which reported invariants are viewed as relevant (a complementary approach to the one discussed in this paper, as mentioned above). We intend to push in these and other areas of related research until we develop dynamic invariant inference technology into one that is generally useful.

The Daikon tool is available for download from the first author's WWW page.

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. Support for the research comes in part from Ernst's IBM Graduate Fellowship and from gifts from Edison Design Group, Microsoft Corporation, and Toshiba Corporation. Griswold is on leave at Xerox PARC.

REFERENCES

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

[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 *ICSE*, pages 191–200, May 1994.

[OJ97] Robert O'Callahan and Daniel Jackson. Lackwit: A program understanding tool based on type inference. In *ICSE*, pages 338–348, May 1997.

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

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

[Wei99] Mark Allen Weiss. *Data structures and algorithm analysis in Java.* Addison Wesley Longman, 1999.