Dynamically Discovering Likely Program Invariants to Support Program Evolution

Michael Ernst[†], **Jake Cockrell**[†], **William G. Griswold**[‡], and **David Notkin**[†]
University of Washington Technical Report UW-CSE-98-08-03
August 27, 1998

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

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

ABSTRACT
Explicitly stated program invariants can help programmers by identifying program properties that must be preserved when modifying code. In practice, however, these invariants are usually implicit. An alternative to expecting programmers to fully annotate code with invariants is to automatically infer invariants from the program itself. This research focuses on dynamic techniques for discovering invariants from execution traces.

This paper reports two results. First, it describes techniques for dynamically discovering invariants, along with an instrumentation and inference engine that embodies these techniques. Second, it reports on the application of the engine to two sets of target programs. In programs from Gries's work on program derivation, we rediscovered predefined invariants. In C programs lacking explicit invariants, we discovered invariants that assisted a software evolution task.

Keywords
Program invariants, formal specification, software evolution, dynamic extraction, logical inference, pattern recognition

1 INTRODUCTION

Invariants play a central role in program development. Representative uses include refining a specification into a correct program, static verification of invariants such as type declarations, and run-time checking of invariants encoded as `assert` statements.

Invariants play an equally critical role in software evolution. In particular, invariants can protect a programmer from making changes that inadvertently violate assumptions upon which the program's correct behavior depends. The near absence of explicit invariants in existing programs makes it all too easy for programmers to introduce errors while making changes.

An alternative to expecting programmers to annotate code with invariants is to automatically infer invariants. In this research, we focus on the dynamic discovery of invariants: we execute a program on a collection of inputs and extract variable values from which we then infer invariants. As with other dynamic approaches such as testing, the accuracy of the inferred invariants depends on the quality and completeness of the test cases; additional test cases might provide new data from which more accurate invariants can be inferred. This approach is complementary to static approaches which examine the program text but do not run the program.

This paper presents two related results stemming from our initial experiences with this approach. Our first result is a set of techniques, and an implementation, for discovering invariants from execution traces (Section 3).

Our second result is the application of the engine to two sets of target programs. The first set of programs, taken from *The Science of Programming* [Gri81], was derived from formal preconditions, postconditions, and loop invariants. Given runs of the program based on randomized inputs, our techniques derive those same program properties, plus some additional ones (Section 2). In contrast, the second set—eight C programs, originally from Siemens [HFGO94], and modified by Rothermel and Harrold [RH98]—is not annotated with invariants, nor is there any indication that invariants were used in their construction. Section 4 shows how numeric invariants dynamically inferred from these programs assist in understanding, and making changes to, them.

Section 5 presents performance measurements and discusses techniques for mitigating combinatorial blowups and otherwise improving runtime. Section 6 surveys related work, and Section 7 concludes.

2 REDISCOVERY OF INVARIANTS

To introduce our approach and illustrate the output of our tool, we present the invariants detected in a simple program taken from *The Science of Programming* [Gri81], a book that espouses deriving programs from specifications. Unlike typical programs, for which it may be difficult to determine the desired output of invariant detection, many of the book's programs include preconditions, postconditions, and loop invariants that embody important properties of the computation. Our invariant detector successfully reports all the formally-specified preconditions, postconditions, and loop invariants in chapters 14 and 15 of the book (chapter 14 is the first in which such programs appear).

As a simple example, consider a program that sums the

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

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

Figure 1: Gries program 15.1.1 [Gri81, p. 180], which sums the values in array b (of length n) into result variable s, and its formal specification. The statement $i, s := 0, 0$ is a parallel (simultaneous) assignment of the values on the right-hand side of the := to the variables on the left-hand side. The **do-od** form repeatedly evaluates the condition on the left-hand side of the $\rightarrow$ and, if it is true, evaluates the body on the right-hand side; the form terminates when the condition evaluates to false.

elements of an array (Figure 1). We transliterated this program to a dialect of Lisp enhanced with Gries-style control constructs. Our instrumenter (Section 3) added code, at the beginning of the program, at the loop head, and at the end of the program, that writes variable values into a data trace file. We ran this program on 100 randomly generated arrays of length 7 to 13, in which each element was a random number between -100 and 100, inclusive. Figure 2 shows the output of our invariant detector given the data trace file.

The precondition inferences record the relationship between `N` and the length of array `B` (which is crucial to the correctness of the program but omitted from the formal invariants), the range of values for `N` appearing in the test cases, and that the test case array elements were always at least -100.

The postcondition inferences include the basic invariant of Gries, $\mathsf{S} = \mathsf{sum(B)}$; Section 3 describes inference over functions such as sum. In addition, the engine discovered that `N` and `B` remain unchanged (*var*`_orig` represents *var*'s value at the start of execution).

The inferred loop invariants (based on 1107 executions of the loop) include those of Gries (since i is an integer, $\mathsf{i} \in [0..13]$ is shorthand for $\mathsf{i} \geq 0$ and $\mathsf{i} \leq 13$), along with several others. For instance, these additional invariants bound the maximum value of the array elements, in addition to the minimum value noted in the precondition invariants. The presence of these bounds is controlled by our statistical rules for determining invariants and by the vagaries of the actual data; more samples tend to give more confidence in the bounds. Section 3 discusses these and other phenomena related to the extra invariants.

3 INVARIANT DETECTION ENGINE

Our basic approach for detecting invariants from program executions consists of instrumenting the source program to trace the variables of interest, running the

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

15.1.1:::END    100 samples
  N = I = N_orig = size(B)
  B = B_orig
  S = sum(B)
  N in [7..13]                          (7 values)
  B                                     (100 values)
    All elements >= -100                (200 values)

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

Figure 2: Invariants for Gries program 15.1.1. Inferred invariants are shown for the beginning (precondition) and end (postcondition) of the program, as well as the loop head (the loop invariant). `B[-1]` is shorthand for `B[size(B)-1]`, the last element of array `B`.

instrumented program over a set of test cases, and inferring invariants over both the instrumented variables and derived variables that are not manifest in the original program.

Instrumentation

The goal of instrumentation is to capture the values of variables, so that patterns can be detected among those values. The two primary decisions are selecting the program points at which to insert instrumentation and selecting the variables to examine at those points.

Our prototype instruments procedure entry and exit points and loop heads. At these points, it records the values of all variables in scope, including global variables, procedure arguments, local variables, and the procedure's return value. Each variable's value is written out to a file, along with the name and type of the variable. Instrumenting is much faster than compilation. For the relatively small, compute-bound programs we have examined so far, the instrumented code can be slowed down by more than an order of magnitude, because the programs become I/O-bound. We have not yet optimized trace file size; another approach would be

to perform invariant checking online rather than writing variable values to a file.

For every instrumented program point, the output is a list of sets of values, one value per instrumented variable. For instance, if procedure `p` has two formal parameters, is in the scope of three global variables, and is called twelve times, then when computing a precondition for `p` the invariant engine would be presented a list of twelve elements, each element being a set of five variable values (one for each visible variable). We also track uninitialized variables by maintaining a separate boolean variable tracking initialization state for each original program variable.

We have implemented instrumenters for programs written in Lisp and C/C++ (the C instrumenter currently does not instrument loop heads). Instrumentation is conceptually simple, but requires care in practice. It can be difficult to determine the size of (the valid data of) an array passed to a C procedure, or even whether a pointer refers to a single element or to an element of an array. We hand-annotated the C programs with the lengths of array, or with the information that the arrays are null-terminated, as for strings. (A static or dynamic analysis may be able to determine many of these types for C programs, and many other languages make this information manifest at compile time or run time.) The C instrumenter uses this information to avoid walking off the ends of arrays, and it outputs values both as pointer addresses and as contents (single elements or entire arrays), to permit both pointer comparisons and comparisons over the underlying values.

Test suite
Invariant discovery requires use of a test suite, which is also necessary for tasks like testing, debugging, and profiling. A single test suite may not be ideal for all tasks. Some test suites are crafted to be as small as possible while still achieving complete code coverage. Invariant detection requires repeated execution of each instrumentation point, because no statistically valid inferences can be made about the distribution of values based on just a few samples. We have obtained good results so far by using pre-existing test suites; for an example, see Section 4.

Inferring invariants
Provided with the output of an instrumented program, the invariant detector lists the invariants detected at each instrumented program point. These invariants may involve a single variable (a constraint that holds over its values) or multiple variables (a relationship among the values of the variables). Our system checks for the following invariants, among others (x, y, and z are variables, and a, b, and c are computed constants):

- any variable: constant value or small number of values
- numeric variable: range ($\mathsf{a} \leq \mathsf{x} \leq \mathsf{b}$), modulus ($\mathsf{x} \equiv \mathsf{a} \pmod{\mathsf{b}}$), nonmodulus ($\mathsf{x} \not\equiv \mathsf{a} \pmod{\mathsf{b}}$)
- multiple numbers: linear relationship (such as $\mathsf{x} = \mathsf{ay} + \mathsf{bz} + \mathsf{c}$), functions (including all those in the standard library, such as $\mathsf{x} = \mathsf{abs(y)}$ or $\mathsf{x} = \mathsf{max(y, z)}$), comparisons ($\mathsf{x} < \mathsf{y}$, $\mathsf{x} \geq \mathsf{y}$, $\mathsf{x} = \mathsf{y}$), invariants over $x + y$ and $x - y$
- sequence: sortedness, invariants over all elements (e.g., every element < 100)
- multiple sequences: subsequence relationship, lexicographic comparison
- sequence and another variable: membership

We produced this list incrementally, starting with invariants that seemed basic and natural, then adding invariants we found helpful in analyzing the Gries programs (Sections 2) and which we believed would be generally applicable. The list is surely not exhaustive; for instance, we do not yet follow arbitrary-length paths through recursive data structures. However, we successfully detected many invariants that occurred in the Siemens suite (Section 4).

For each variable or tuple of variables, each potential invariant is tested. As soon as an invariant is determined not to hold, it is not checked for the remainder of the values taken on by the variable(s). Thus, the cost of computing invariants tends to be proportional to the number of invariants discovered (see also Section 5). The invariants listed above are inexpensive to test. For example, the linear relationship $\mathsf{x} = \mathsf{ay} + \mathsf{bz} + \mathsf{c}$ with unknown coefficients a, b, and c and variables x, y, and z has three degrees of freedom. Consequently, three tuples of values for x, y, and z are sufficient to infer the possible coefficients. As another example, a common modulus (variable b in $\mathsf{x} \equiv \mathsf{a} \pmod{\mathsf{b}}$) is the gcd of the differences among list elements.

Negative invariants
Negative invariants are relationships that might be expected to occur but were never observed in the input. We compute the probability that such a property would not appear in a random input; if this probability is sufficiently small, then the property is reported as possibly non-coincidental. For example, if the reported values for variable x fit in a range of size r that includes 0, the probability that a single instance of x is not 0 is $1 - \frac{1}{r}$. (We make the simplifying assumption of a uniform distribution of values.) Given v reported values, the probability that x is never 0 is $(1-\frac{1}{r})^v$; if this is less than a user-defined confidence level, then the negative invariant $\mathsf{x} \neq 0$ is reported; $\mathsf{x} \neq \mathsf{y}$ and modulus tests are analogous.

Ranges for numeric variables (such as $\mathsf{c} \in [32..126]$ or $\mathsf{x} > 0$) are also not reported unless they appear to be non-coincidental. In particular, if the several values near the range's extrema all appear about as often as would be expected, or if the extremum appears much

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

15.1.1:::END    100 samples
  B = B_orig
  N = I = N_orig = size(B)
  S = sum(B)
  N >= 0                              (24 values)

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

Figure 3: Invariants for Gries program 15.1.1 over an input set whose array lengths and element values were chosen from exponential rather than uniform distributions, as in Figure 2.

more often than would be expected (as if greater or lesser values have been clipped to that value), then the limit is reported.

In Figure 2, negative invariants are reported for the loop head, but not for the beginning or end of the procedure, where the 100 samples were insufficient to support any inequality inferences.[1] Similarly, the elements of array `B` were bounded from above and below at the loop head, but only from below (as being at least -100) at procedure entry and exit. The random distribution of array elements happened to support only one boundedness inference for 100 samples; on another run over a similarly small set of test cases, only the upper bound, neither bound, or both bounds might be inferred.[2]

In Figure 2, the invariants beyond those of Gries, rather than being artifacts of our technique, provide valuable information about the data set. (This can help validate a test suite or indicate the contexts in which a function or other computation is used.) Figure 3 shows the result of running our system on a different set of 100 arrays; the output is almost precisely the Gries invariants.

Derived variables

In addition to manifest values explicitly passed to the engine, we need to compute relations over non-manifest values. For instance, if array `a` and integer `lasti` are both in scope, then the value of `a[lasti]` may be of interest, even if that expression does not appear in the program text.

Therefore, we add certain "derived variables" (actually expressions) to the list of variables given to the engine as input. These derived variables include the following:

- from any array: length, first and last elements
- from a numeric array: sum, min, and max
- from array and scalar: element at that index (`a[i]`), subarray up to or subarray beyond that index (e.g., `a[0..i-1]`)

Derived variables are treated just like other variables by the inference engine.

Derived variables permit the engine to infer invariants that are not hard-coded into its list. For instance, if `len(A)` is derived from array `A`, then we can determine that $\mathsf{i} < \mathsf{len(A)}$ without hardcoding a less-than comparison check for the case of a scalar and an array. The implementation maintains short lists of simple invariants and variable derivations, and can thus report compound relations that we did not necessarily anticipate.

Many possible derived values are not of general interest. For example, we don't want to run a battery of tests on x^y for every x and y, much less compute $ax+b$ for every variable x and constant a and b. Moreover, each new variable introduces costs of checking invariants over it. We also take care not to introduce arbitrarily many new variables when deriving variables from derived variables.

Staged derivation and inference

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

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

[1] The values for which inequalities are inferred in the loop head are actually the same as the values at procedure entry and exit. However, the loop head is executed more times. We plan to enhance the implementation so that loop iterations do not incorrectly add support for values unchanged by the loop.

[2] This paper uses .01 as the probability limit. A smaller value is probably more practical but is poorer for illustrative purposes, as it eliminates the negative invariants in Figure 2.

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

Figure 4: Function `makepat`'s use of constant `CLOSURE` in Siemens program `replace`.

4 USE OF INVARIANTS

The techniques described in the previous section are sufficient for rediscovering the known invariants for the Gries programs discussed in Section 2. This section demonstrates that derived invariants can help a programmer to evolve a program that contains no explicitly stated invariants. In particular, we used invariants produced by our implementation in evolving a program from the Siemens suite [HFGO94, RH98].

The Task

The Siemens `replace` program, 563 lines of C, takes a regular expression and a replacement string as command-line arguments, writing an input stream to an output stream while replacing any substring matched by the regular expression with the replacement string.

The regular expression language of `replace` includes Kleene-* closure but omits Kleene-+ closure, so we decided that this would be a useful extension. The basic plan for the enhancement was to introduce a new operator, `+`, and implement it by mimicking the implementation of `*`, except that at least one instance of the pattern must be matched.

Performing the Change

The program's call structure and high-level definitions reveal that it is composed of a pattern parser, pattern compiler, and matching engine. The compiled pattern is stored in a 100-element array named `pat`. Function `addstr` is often used to add characters to this array during pattern compilation. We decided to avoid modifying the matching engine and minimize changes to the parser by compiling an input pattern of the form $\langle pat \rangle$`+` as though it were the semantically equivalent $\langle pat \rangle \langle pat \rangle$`*`.

Simple text searches helped us determine what code needed to be extended and added. A constant called `CLOSURE` with value `'*'` is referenced in several places, so we introduced a constant `PCLOSURE` of value `'+'`. After several simple changes, we focused on the use of `CLOSURE` in `makepat`, which controls the invocation of `stclose` (Figure 4), whose name stands for "star closure."

This code in `makepat` determines whether the next character in the input is `CLOSURE`; if so, it calls `stclose` if

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

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

Figure 5: Function `stclose` in Siemens program `replace`.

some previous element in the input was not in the set specified by `in_set_2`. We duplicated this code in the if-then-else sequence, modified the copy to check against `PCLOSURE`, and then modified it to call a new `plclose` function. Since `in_set_2` was earlier modified to include `PCLOSURE`, no other changes to `makepat` should be required.

Since `plclose` should be similar to `stclose`, we copied function `stclose` (Figure 5) and renamed it to `plclose`. To decide what `plclose` should do, we studied `stclose`. We speculated that the uses of array `pat` in the loop manipulate the pattern that is the target of the closure operator. Since closure must always be applied to a pattern, we wanted to verify that the loop was indeed entered on every call to `stclose`. The loop's exit condition says the loop would not be entered if `*j` were equal to `lastj`, so we examined the invariants inferred for them on entry to `stclose`, finding the following:

$$*j \geq 2$$
$$lastj \geq 0$$
$$lastj \leq *j$$

The third invariant implies that there are cases when the loop is not entered.

To find the offending values of `*j` and `lastj`, we queried the trace database for those calls to `stclose` in which variable `junk` is never set, since these are the cases when the loop is not entered. (We wrote a tool which finds the tuples in the execution trace database that satisfy a given invariant.) The query returned several calls in which the value of `*j` is 101 or more, greater than the size of the array `pat`. We soon determined that the program contains an array bounds error which is triggered in some instances when the compiled pattern is too long. In the absence of this bug, the loop in `stclose` should always be entered when it is called, which increased our confidence that the loop is manipulating the pattern to which the closure operator is being applied. To in-

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

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

Figure 6: Function `plclose` in the extended `replace` program.

crease the precision of `stclose`'s invariants for the rest of the task, we recomputed the invariants without the test cases that caused the improper calls to `stclose`.

Returning to consideration of `stclose`'s manipulation of array `pat` (Figure 5), we observed that the loop index is being decremented, and `pat` is both read and written by `addstr`. Moreover, the closure character is inserted into the array at index `lastj`, which is not at the end of the compiled pattern. Looking at the invariants for `pat`, we found the string comparison $\mathsf{pat}_{\mathsf{orig}} \neq \mathsf{pat}$, which indicates that `pat` is always updated. To find out exactly what `stclose` does to `pat`, we queried the trace database for values of `pat` on entry to and exit from `stclose`. For example:

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

These revealed that the program compiles literals by prefixing them with the character `c` and puts Kleene-* expressions into prefix form. The negative indexing and assignment of `*` into position `lastj` moves the closed-over pattern rightward in the array to make room for the `*` itself in the prefix format. For a call to `plclose` the result for the above test case should be `cacb*cb`, which would match one or more instances of character `b` rather than zero or more. This is a simple copy of the previous pattern, rather than a rightward move, so the resulting implementation can be a bit simpler. After figuring out what `addstr` is doing with the address of the index passed in (it increments the index unless the array bound are exceeded), we converged on the version of `plclose` in Figure 6.

To check that the modified program does not violate invariants that are still expected to hold, we added test cases for Kleene-+ and recomputed the invariants for the modified program. By displaying the difference between the old and new invariants, we ascertained that the expected invariants hold. Comparing the invariants for `stclose` and `plclose`, we observed that whereas `stclose` has the invariant $*\mathsf{j} = *\mathsf{j}_{\mathsf{orig}} + 1$, `plclose` has the invariant $*\mathsf{j} \geq *\mathsf{j}_{\mathsf{orig}} + 2$. This difference is expected, since the compilation of Kleene-+ replicates the entire target pattern, which is two or more characters long in its compiled form.

Invariants for `makepat`

Several invariants discovered for `makepat` were also helpful. In determining when `stclose` is called, the invariants showed us that parameter `start` (tested in Figure 4) is always 0, and parameter `delim`, which controls the outer loop, is always the null character (character 0). These invariants helped reveal that `makepat` is used in more specialized contexts than we anticipated, saving us considerable effort in understanding its role in pattern compilation.

We had hypothesized that `lastj` and `lj` in `makepat` should both always be less than local `j` (`lastj` and `lj` refer to the last generated element of the compiled pattern, whereas `j` refers to the next place to append). Although the invariants for `makepat` confirmed this relation over `lastj` and `j`, no invariant between `lj` and `j` was reported. A query on the trace database for variables `j` and `lj` at the exit of `makepat` revealed several cases in which `j` is 1 and `lj` is 100, a complete surprise to us.

Another inferred invariant was number_of_calls(in_set_2) = number_of_calls(stclose). Since `in_set_2` is only called in the predicate controlling `stclose`'s invocation, the equal number of calls indicates that none of the test cases caused `in_set_2` to return `false`. (We used a randomly-selected set of 100 inputs from those supplied with the `replace` program.) This suggests that we needed to run more test cases to expose more of `replace`'s special-case behavior and produce more accurate invariants. Nevertheless, the inferred invariants were still helpful in the task.

Discussion

Although the invariants inferred by our prototype are simple, they provide substantial insight. For example, our expectations regarding the preconditions for `stclose` were contradicted by the inferred invariants, and the necessary intuition was provided by supporting data. This not only helped us discover a bug, but also helped establish the conditions under which our postulated invariant holds. This knowledge simplified our task because the need for special-case processing inside `plclose` was quickly proven unwarranted. On the other hand, dynamic invariants alone did not solve these problems. Static analyses identified the general architecture of the system and the code likely to require change.

The usefulness of elementary invariants in a software evolution task can be attributed to at least three factors. First, the programmer does not need a full specification, only insight on program properties that should not be violated by the proposed change. Second, the invariants provide a suitable basis for the programmer's own, more complex inferences. Because the inferred invariants concern observable entities in the program, the programmer can examine the program text or perform supporting static analyses to better understand the invariants' implications. For example, we might have liked to see an invariant that `*j` always refers to the next place to append a character into `pat`, but this is at best expensive to compute. However, the presence of several related invariants indicating that `*j` starts with a 0 value and is regularly incremented by 1 during the compilation of the pattern allowed us to ascertain its basic function and quickly determine the more higher-level invariant. Third, the invariants are a succinct abstraction of a mass of supporting data. These data can be accessed to provide the evidence needed to build intuition about the source of an invariant or lack thereof. These data helped us to determine under which conditions the loop in `stclose` is not executed and the format of the `pat` array.

5 SCALABILITY

We ran several simple experiments to identify the costs of invariant inference. These suggest ways to accelerate inference and manage the number of invariants reported.

Performance Measurements

We present three scalability-related measurements in turn: the effect of number of variables on invariant detection time, the effect of test suite size on invariant detection time, and the effect of test suite size on what invariants are detected. To investigate these phenomena, we instrumented and ran the Siemens program `replace` on 500 test cases randomly selected from the 5542 supplied with the program. We broke the test cases into 5 runs consisting of numbers 1–50, 1–100, 101–200, 1–200, and 1–500.

Our system infers invariants over an average of 59 (original and derived) variables per instrumentation point in `replace`. On average, 500 test cases produce 4775 samples per instrumentation point, and our system takes 207 seconds to infer the invariants for that point on a 200MHz UltraSPARC 2. Our prototype implementation uses the interpreted language Python [van97]. At one point we improved performance by nearly a factor of ten by inlining two one-line procedures, but we have not yet seriously optimized our implementation. In addition to local optimizations and algorithmic improvements, use of a compiled language such as C could improve performance by another order of magnitude or more.

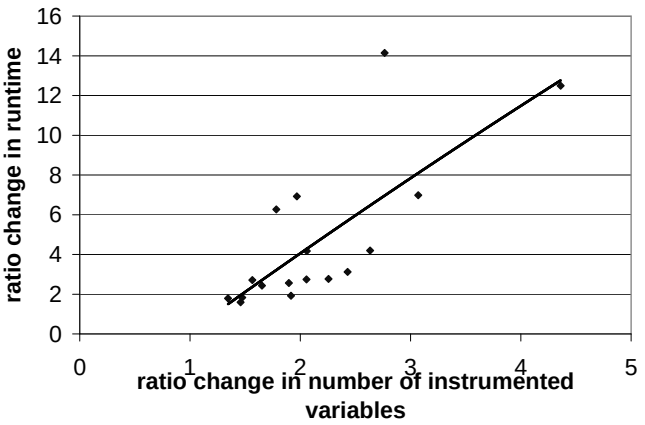

Figure 7: Change in invariant detection runtime versus change in number of variables. A least-squares trend line highlights the relationship. Each data point compares inference over two different sets of variables at a single instrumentation point, for a single test run of 100 test cases. If one run has v_1 variables and a runtime of t_1, and the other has v_2 variables and a runtime of t_2, the x axis measures $\frac{v_2}{v_1}$ and the y axis measures $\frac{t_2}{t_1}$. For example, doubling the number of variables tends to increase runtime fourfold.

The number of variables over which invariants are checked is the most important factor in invariant detection runtime. On average, each of the 20 functions in `replace` has 5 parameters (2 of them arrays and the others scalars), and 1 scalar local variable is in scope at the procedure exit. On average, about ten derived variables are derived for each original one, a number that is remarkably insensitive to the relative numbers of scalars and arrays. Figure 7 plots growth in invariant detection time against growth in number of variables. Each data point compares inference times for two sets of variables at a single instrumentation point. The instrumentation points are procedure exits; one set of variables is the global variables and initial argument values, while the other set adds final argument values, local variables, and the return value. Absolute runtimes vary from 5 to 519 seconds, while the number of variables ranges from 28 to 182.

Test suite size has a less pronounced effect on invariant detection runtime. Unsurprisingly, test suite size is linearly related to number of samples for each instrumentation point. The number of distinct variable values at each instrumentation point also follows an almost perfectly linear relationship to these measures, with about 1 new value per 20 samples.[3] Runtime is approximately linearly related to test suite size, number of samples, and number of values per variable. Fig-

[3] We expected fewer new values to appear in later runs. However, repeated array values are rare. Also, our smallest test suite of 50 inputs produced 600 samples per function on average, perhaps avoiding the high distinct variable values per sample ratio expected with few inputs.

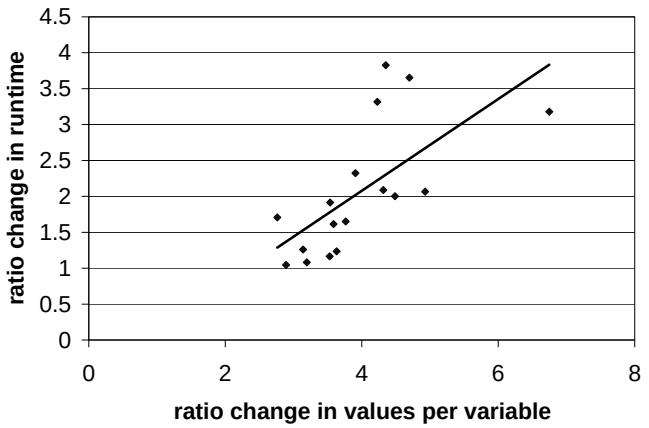

Figure 8: Change in invariant detection runtime versus change in number of values per variable. A least-squares trend line highlights the relationship. Each data point compares inference at one program point over two test runs. If one test runs has v_1 values per variable and a runtime of t_1, and the has with v_2 values per variable and a runtime of t_2, the x axis measures $\frac{v_2}{v_1}$ and the y axis measures $\frac{t_2}{t_1}$. For example, quadrupling the number of values per variable tends to double runtime.

ure 8 plots growth in runtime against growth in values per variable. Quadrupling the number of values per variable (or increasing the test suite size by a factor of 80) only doubles invariant detection runtime. Runtimes vary from 27 to 1721 seconds, while the number of values per variable ranges from 10 to 1798.

In Figures 7 and 8, we removed two data points corresponding to functions with tiny runtimes (times smaller than 1 second). In these cases, timing errors may produce inaccurate results.

Test suite size does not appear to dramatically affect invariant precision at the sizes we examined. Each of the runs described above produced about 6000 lines of output (the larger runs had slightly smaller output). Between two 100-test-case runs, 1500 lines of output differed; between a 100-test-case-run and the 500-test-case-run, 1300 lines differed; and between the 200-test-case-run and the 500-test-case-run, 900 lines differed.

Substantive differences in invariants, such as detecting $\mathsf{result} = \mathsf{i}$ in one case but not another, were rare—far fewer than one per procedure on average. Changes in a constant bound on a variable value, such as $\mathsf{outset}[0] \in [-1..113]$ vs. $\mathsf{outset}[0] \in [-1..126]$ or $*\mathsf{j} \in [1..103]$ vs. $*\mathsf{j} \geq 1$, accounted for 20–30% of the differences. Similar changes in an inferred comparison (say, $\geq$ vs. $>$) were also non-negligible. Differences in negative invariants accounted for 15–25% of the differences; of these, about two thirds appeared only in the invariants produced from larger test cases, and vice versa for the other third. Most of the positive invariants discovered in one procedure but not in another were between clearly incomparable or unrelated quantities (such as a comparison between an integer and an address, or between two elements of an array or of different arrays) or were artifacts of the particular test cases (such as adding $*\mathsf{i} \neq 5 \pmod{13}$ to $*\mathsf{i} \geq 0$).

Improvements to the Approach

There are potentially large numbers of program points to instrument, variables to examine at each point, and invariants to check over those variables. We have identified ways to mitigate this combinatorial blowup in instrumentation output size, inference time, and number of results. The techniques generally trade off the time spent on inference against the precision of the discovered invariants, possibly under programmer control.

The granularity of instrumentation affects the amount of data gathered and thus the time required to process it. Inferring loop invariants or relationships among local variables can require instrumentation at loop heads, at function calls, or elsewhere, whereas determining properties of global variables or other large-scale structures does not require so many instrumentation points; perhaps module entry and exit points would be sufficient. When only a part of the program is of interest, the whole program need not be instrumented; we often computed invariants over just a single procedure. The choice of variables instrumented at each program point also affects inference performance. When some are not of interest, they can be skipped, and variables that cannot have changed since the last instrumentation point may not need to be reexamined. Finally, supplying fewer test cases results in faster runtimes at the risk of less precise output.

The inference engine can be directly sped up by checking for fewer invariants; this is particularly useful when a programmer is focusing on part of the program and is not interested in certain kinds of properties (say, transcendental arithmetic functions). Derived variables can likewise be throttled to save time or increased to provide more extensive coverage. More complicated derived variables may be added for complex expressions that appear in the program text; derived variables or invariants may also involve functions defined in the program. Type analysis can indicate which variables are incomparable, even if they have the same type in the programming language [OJ97].

User Interface

A large data set and large number of derived invariants can be overwhelming. We have already developed a tool that retrieves from the data set the variable–value tuples that satisfy or falsify an invariant. There are also several ways to improve the presentation of invariants themselves.

To control the number of displayed invariants, a text editor could provide a list of invariants for the variable

under the cursor. Programmers could also be permitted to filter out classes of invariants (e.g., array relationships). Statically obvious invariants (such as $x = y + 1$ immediately after `x:=y+1`) could be filtered. Presenting invariants on demand naturally permits computing the invariants on demand, possibly avoiding delays for the computation of unneeded invariants. Users should be permitted to declaratively specify additional relations and derived variables for analysis.

Ordering the reported invariants according to category or predicted usefulness could also help a programmer find a relevant invariant more quickly. Since a programmer might like to know how a software change has affected the invariants, an invariant-differencing tool could help focus on invariants that have changed.

6 RELATED WORK

Dynamic Inference

The research most directly related to ours uses inductive logic programming (ILP) [Qui90, Coh94] to construct Horn clause loop invariants from variable values on particular loop executions [BG93]. ILP requires counterexamples (which are not available in our domain) and background knowledge, and the resulting relations typically misclassify 10% or more of the training set. (Our approach characterizes the training set perfectly; either approach may misclassify additional data.) Other AI approaches like neural nets may predict results but have little explicative power. Traditionally, machine learning attempts to learn a function over $n - 1$ variables producing the n^{th} or to classify examples into specified categories, neither of which is directly applicable to our problem [Mit97]. However, we believe applying these techniques to subproblems of our task will be fruitful.

Other dynamic analyses that examine program executions are used for software tasks from testing to debugging. Program spectra (specific aspects of program runs, such as event traces, code coverage, or outputs) [RBDL97, HRWY98] can reveal differences in inputs or program versions. Cook and Wolf use event traces, which describe the sequence of events in a possibly concurrent system, to produce a finite state machine generating the trace [CW98b, CW98a].

Static Inference

Static analyses operate on the program text, not particular test runs, and are typically sound but conservative. As a result, properties they report are true for any program run, but static analyses are limited by uncertainty about properties beyond their capabilities and by the high cost of modeling program states. For instance, accurate alias analysis is still beyond the state of the art, so many static checkers must give up in the face of pointer manipulation. The ease of checking some such properties at runtime makes static and dynamic techniques complementary.

Considerable research has addressed checking formal specifications [DC94, EGHT94, Det96, Eva96, NCOD97, LN98, JvH+98, Pfe92]; this work could be used to verify likely invariants discovered dynamically. Determining what property to check is considered harder than actually doing the checking [Weg74, BLS96]; our goal is the discovery of such properties from a broad class of possible ones.

Some formal proof systems generate intermediate assertions for help in proving a given goal formula by propagating known invariants forward or backward in the program [Weg74, GW75, KM76, BBM97]. Variable types are a variety of formal specification and documentation and whose checking can detect errors. Type inference extends partial type annotations to full ones; similarly, Givan [Giv96] extends specifications on the inputs of a procedure to its output, and ADDS propagates data structure shape descriptions through a program [HHN92, GH96]. In the case of array bounds checking [SI77, Gup90, KW95, XP98], the desired property is obvious.

Other related work includes the Illustrating Compiler, which heuristically detects the abstract datatype implemented by a collection of concrete operations [HWF90]. Staging and binding-time analyses determine invariant or semi-invariant values for use in partial evaluation [JGS93].

7 CONCLUSIONS

This paper documents the feasibility of discovering program invariants based on execution traces, as well as the usefulness of these invariants in a software evolution task. The techniques we have developed, along with the prototype implementation, are adequately fast when applied to programs of several hundred lines.

Acting as our own users was advantageous in the initial phases of this research. Working on evolution tasks with programs that we did not write gave us insights into the strengths and weaknesses of the techniques and the tool, as well as to the overall approach. Moreover, we found that the use of dynamically inferred invariants qualitatively affected our programming, encouraging us to actively think in terms of invariants in situations that we might otherwise not.

With a variety of performance improvements, including user-directed indications of instrumentation points and variables of interest, the approach should be applicable to the evolution of larger systems. More sophisticated invariants will also be required; a few of the most critical are invariants over pointer-based data structures such as trees, predicated invariants (if *condition* then *invariant*), and disjunctions ($p = \text{NULL}$ or $*p > i$). We will need to assess the enhanced technology by having programmers apply it to larger, more complicated programming tasks.

Dynamically inferred invariants can be used in many

situations that statically-supplied invariants can, and in some cases the application of dynamic ones may be even more effective. For instance, dynamic invariants form a program spectrum, changes to which can indicate properties of a changed program or input. They can assist in test case generation and can also validate a test suite; invariants in the resulting program runs can indicate insufficient coverage of program values, even if every line is executed at least once. A nearly-true invariant may indicate a bug or special case that should be brought to the programmer's attention. Discovered invariants can be inserted into a program as `assert` statements to further test the invariant or to ensure that detected invariants are not later violated as code evolves. They can also double-check existing documentation or `assert` statements. Compilers can exploit invariants by optimizing for the common case, like profile-directed optimization but potentially at a higher level of abstraction. Detected invariants can bootstrap or direct a (manual or automatic) correctness proof.

ACKNOWLEDGMENTS

We are grateful for discussions of these ideas with Craig Chambers, Oren Etzioni, Tessa Lau, David Madigan, and Jared Saia. Gregg Rothermel shared his modified versions of the Siemens test programs. Greg Badros, Craig Chambers, Tessa Lau, Todd Millstein, and Jon Nowitz improved this paper by critiquing a draft. Daniel Jackson, Vass Litvinov, George Necula, and James Noble suggested related work.

REFERENCES

[BBM97] N. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, Feb. 1997.

[BG93] I. Bratko and M. Grobelnik. Inductive learning applied to program construction and verification. In J. Cuena, editor, *AIFIPP '92*, pp 169–182. North-Holland, 1993.

[BLS96] S. Bensalem, Y. Lakhnech, and H. Saidi. Powerful techniques for the automatic generation of invariants. In *CAV*, pp 323–335, 1996.

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

[CW98a] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, July 1998.

[CW98b] J. E. Cook and A. L. Wolf. Event-based detection of concurrency. In *FSE*, Orlando, FL, Nov. 1998.

[DC94] M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *FSE*, pp 62–75, Dec. 1994.

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

[EGHT94] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. LCLint: A tool for using specifications to check code. *FSE*, pp 87–97, Dec. 1994.

[Eva96] D. Evans. Static detection of dynamic memory errors. In *PLDI*, pp 44–53, May 1996.

[GH96] R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *POPL*, pp 1–15, Jan. 1996.

[Giv96] R. Givan. Inferring program specifications in polynomial-time. In *SAS*, pp 205–219, Sept. 1996.

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

[Gup90] R. Gupta. A fresh look at optimizing array bound checking. In *PLDI*, pp 272–282, June 1990.

[GW75] S. M. German and B. Wegbreit. A synthesizer of inductive assertions. *IEEE Transactions on Software Engineering*, 1(1):68–75, Mar. 1975.

[HFGO94] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE*, pp 191–200, May 1994.

[HHN92] L. J. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *PLDI*, pp 249–260, June 1992.

[HRWY98] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. In *PASTE '98*, pp 83–90, June 1998.

[HWF90] R. Henry, K. M. Whaley, and B. Forstall. The University of Washington Illustrating Compiler. In *PLDI*, pp 223–246, June 1990.

[JGS93] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[JvH+98] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about Java classes. In *OOPSLA*, Vancouver, BC, Canada, Oct. 1998.

[KM76] S. Katz and Z. Manna. Logical analysis of programs. *Communications of the ACM*, 19(4):188–206, Apr. 1976.

[KW95] P. Kolte and M. Wolfe. Elimination of redundant array subscript range checks. In *PLDI*, pp 270–278, June 1995.

[LN98] K. R. M. Leino and G. Nelson. An extended static checker for Modula-3. In *Compiler Construction '98*, pp 302–305. Springer-Verlag, Apr. 1998.

[Mit97] T. M. Mitchell. *Machine Learning.* WCB/McGraw-Hill, Boston, MA, 1997.

[NCOD97] G. Naumovich, L. A. Clarke, L. J. Osterweil, and M. B. Dwyer. Verification of concurrent software with FLAVERS. In *ICSE*, pp 594–595. Springer, May 1997.

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

[Pfe92] F. Pfenning. Dependent types in logic programming. In *Types in Logic Programming*, chapter 10, pp 285–311. MIT Press, Cambridge, MA, 1992.

[Qui90] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[RBDL97] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *ESEC/FSE*, pp 432–449, Sept. 1997.

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

[SI77] N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *POPL*, pp 132–143, Los Angeles, California, Jan. 1977.

[van97] G. van Rossum. *Python Reference Manual*, release 1.5 edition, Dec. 1997.

[Weg74] B. Wegbreit. The synthesis of loop predicates. *Communications of the ACM*, 17(2):102–112, Feb. 1974.

[XP98] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, pp 249–257, June 1998.