# Dynamically Discovering Likely Program Invariants to Support Program Evolution


**Michael D. Ernst**[†], **Jake Cockrell**[†], **William G. Griswold**[‡], and **David Notkin**[†]

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



## 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 instrumenter and an inference engine that embody 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 a C program lacking explicit invariants, we discovered invariants that assisted a software evolution task.



**Keywords**

Program invariants, formal specification, software evolution, dynamic analysis, execution traces, logical inference, pattern recognition


## 1 INTRODUCTION

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

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

An alternative to expecting programmers to annotate code with invariants is to automatically infer invariants. 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 and profiling, 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 our inference 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 over randomly-generated inputs, our techniques discover those same program properties, plus some additional ones (Section 2). The second set of programs — 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 one of these programs assisted in understanding and changing it.

Section 5 presents performance measurements and discusses techniques for mitigating combinatorial blowups and otherwise improving runtime performance. 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 elements of an array (Figure 1). We transliterated this program to a dialect of Lisp enhanced with Gries-style control


Appears in *Proceedings of the 19th International Conference on Software Engineering (ICSE '99)*, Los Angeles, CA, May 19–21, 1999, pp. 213–224.

$$
\begin{array}{l}
i, s := 0, 0; \\
\textbf{do } i \neq n \rightarrow \\
\quad i, s := i + 1, s + b[i] \\
\textbf{od}
\end{array}
$$

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

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

constructs. Our instrumenter (Section 3) added code that writes variable values into a data trace file; this code was automatically inserted at the beginning of the program, at the loop head, and at the end of the program. We ran this program on 100 randomly generated arrays of length 7 to 13, in which each element was a random number in the range $-100$ to 100, inclusive. Figure 2 shows the output of our invariant detector given the data trace file.

The precondition (`BEGIN`) 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 (`END`) inferences include the basic invariant of Gries, $\mathsf{S} = \mathsf{sum}(\mathsf{B})$; Section 3 describes inference over functions such as sum. In addition, the engine discovered that `N` and `B` remain unchanged.

The inferred loop (`LOOP`) invariants 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. Inference of these bounds is controlled by our statistical rules for determining invariants and by the vagaries of the actual input data; more samples tend to give more confidence in the bounds. Section 3 discusses these and other phenomena related to the extra invariants, including negative invariants.

## 3 INVARIANT DETECTION ENGINE

We detect 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 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.

```
15.1.1:::BEGIN  100 samples
  N = size(B)                        (7 values)
  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)           (7 values)
  B = B_orig                         (100 values)
  S = sum(B)                         (96 values)
  N in [7..13]                       (7 values)
  B                                  (100 values)
    All elements >= -100             (200 values)

15.1.1:::LOOP   1107 samples
  N = size(B)                        (7 values)
  S = sum(B[0..I-1])                 (96 values)
  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 inferred for Gries program 15.1.1 over 100 randomly generated input arrays. 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`, and *var*`_orig` represents *var*'s value at the start of execution. Invariants for elements of an array are listed indented under the array; in this example, no array has multiple elementwise invariants.

Our prototype instruments procedure entry and exit points and loop heads. At these points, it writes to a file the values of all variables in scope, including global variables, procedure arguments, local variables, and the procedure's return value. 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 or writing time; another approach would be to perform invariant checking online rather than writing variable values to a file.

For every instrumented program point, the trace file contains 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). A separate boolean variable tracks initialization state for each original program variable.

We have implemented instrumenters for programs written in Lisp and C/C++ (the C/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 variable or to an element of an array. We hand-annotated the C programs with the lengths of arrays or with the information that the arrays are null-terminated (as strings are). The C instrumenter uses this information to avoid walking off the ends of arrays. 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. It also 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

The invariant detector, when provided with the output of an instrumented program, 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 scalar: 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 (Section 2) and which we believed would be generally useful. The list does not include all the invariants that we think programmers will find useful. For instance, we do not yet follow arbitrary-length paths through recursive data structures. However, we successfully detected many invariants that occur 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 and do not require full-fledged theorem-proving. 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 of inexpensive checking, a common modulus (variable $b$ in $\mathsf{x} \equiv \mathsf{a} \pmod{\mathsf{b}}$) is the greatest common divisor of the differences among list elements.

*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; essentially, we are testing this assumption. Much of our tool can be viewed as statistical tests of hypothesized distributions for variable 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 (non)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, a limit is reported if the several values near the range's extrema all appear about as often as would be expected, or if the extremum appears much more often than would be expected (as if greater or lesser values have been clipped to that value).

In Figure 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.

For the purposes of this paper — in part, to demonstrate spurious negative invariants like those of Figure 2 — we set the probability limit to .01. For actual use we recommend a substantially smaller value: if the system checks millions of potential invariants, reporting thousands of false positives is

[1]The values over which inequalities are inferred in the loop head are 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.

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

15.1.1:::END    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)
  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 inferred 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.

likely to be unacceptable.

A sufficiently strong static analysis can reveal useful invariants that are universally true of a function, no matter how it is used. A whole-program analysis provides stronger properties (that is, properties that logically imply those true of the function in isolation) about the function's execution that depend on the context in which it is called. Our system reports yet stronger invariants that depend on the data sets over which the program was run.

The invariants of Figure 2 include several not noted by Gries. These extra invariants are not merely artifacts of our technique; rather, they provide valuable information about the data set, such as variable ranges. This information 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 variables explicitly passed to the engine, we need to compute relations over non-manifest quantities. For instance, if array `a` and integer `lasti` are both in scope, then `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: first and last elements, length
- from numeric array: sum, min, max
- from array and scalar: element at that index (`a[i]`), subarray up to, and subarray beyond, that index (e.g., `a[0..i-1]`)
- from function invocation: number of calls so far

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 the engine can determine that $\mathsf{i} < \mathsf{len(A)}$ without hardcoding a less-than comparison check for the case of a scalar and the length of an array. In this manner, the implementation can report compound relations that we did not necessarily anticipate.

Many possible derived variables are not of general interest. For example, we do not want to run a battery of tests on $x^y$ for every $x$ and $y$, 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, by halting derivation after a fixed number of iterations and by mechanisms described below.

### Staged derivation and inference

Both variable derivation and invariant inference can avoid unnecessary work by examining previously-computed invariants. Therefore, derived variables are not introduced until invariants have been computed over previously-existing variables, and derived variables are introduced in stages rather than all at once. For instance, for 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 in creating the derived variable `A[j]`. When a derived variable is only sometimes sensible, as when `j` is only sometimes a valid index to `A`, no further derivations are performed over `A[j]`. Likewise, `A[0..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.

## 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. To help determine whether and how derived invariants might help a programmer modify a program that contains no explicitly-stated invariants, we used invariants produced by our engine in evolving a program from the Siemens suite [HFGO94, RH98]. After describing the scenario we went through in modifying this program, we discuss some of the factors that make the use of invariants qualitatively different from some more traditional styles of gathering information about programs.

### The Task

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

```
...
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`.

expression language of `replace` includes Kleene-* closure but omits Kleene-+ closure, so we decided that this would be a useful extension.

**Performing the Change**

We statically studied the program's call structure and high-level definitions and found that it is composed of a pattern parser, a pattern compiler, and a matching engine. To avoid modifying the matching engine and to minimize changes to the parser, we decided to compile an input pattern of the form $\langle pat \rangle$`+` into the semantically equivalent $\langle pat \rangle \langle pat \rangle$`*`.

The initial changes were straightforward and were based on informal, static analyses. In particular, simple text searches helped us find how `'*'` was handled during parsing. We mimicked the constant `CLOSURE` of value `'*'` with the constant `PCLOSURE` of value `'+'`, and we made several simple changes, such as adding `PCLOSURE` to internal sets that represent special classes of characters (`in_set_2` and `in_pat_set`).

We then studied the use of `CLOSURE` in `makepat`, since we knew we would have to handle `PCLOSURE` analogously. The basic code in `makepat` (Figure 4) determines whether the next character in the input is `CLOSURE`; if so, it calls the "star closure" function, `stclose` (Figure 5) under most conditions (and the exceptions should not differ for plus closure). We duplicated this code sequence, modifying the copy to check for `PCLOSURE` and to call a new function, `plclose`. Our initial body for `plclose` was a copy of the body of `stclose`.

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

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`:[2]

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

The third invariant implies that the loop body may not be executed (if $\mathsf{lastj} = *\mathsf{j}$, then `jp` is initialized to `lastj-1`), which was inconsistent with our initial belief.

[2] For this scenario, our system extracted invariants at the beginning and end of all procedures in the program, using as input 100 randomly selected test cases from those provided with the Siemens suite.

```
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`.

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

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

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

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

This suggests that the program compiles literals by prefixing them with the character `c` and puts Kleene-* expressions into prefix form. (A coauthor who was not

```
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.

performing the change independently discovered this fact through careful study of the program text.) The negative indexing and assignment of `*` into position `lastj` moves the closed-over pattern rightward in the array to make room for the prefix `*`. For a call to `plclose` the result for the above test case should be `cacb*cb`, which would match one or more instances of character `b` rather than zero or more. This is a simple copy of the previous pattern, rather than a rightward shift, 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 is 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. As expected, most invariants remained unchanged, while some differing invariants verified our program modifications. 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`

We also investigated several invariants discovered for function `makepat`. In determining when `stclose` is called — to learn more about when our new `plclose` will be 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 indicated 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` (i.e., `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 at the exit of `makepat` returned several cases in which `j` is 1 and `lj` is 100, which contradicted our expectations and prevented us from introducing bugs based on a flawed understanding of the code.

Another inferred invariant was $\mathsf{number\_of\_calls(in\_set\_2)} = \mathsf{number\_of\_calls(stclose)}$. 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`. Rather than helping us modify the program, these invariants instead suggest a need to run more test cases to expose more of `replace`'s special-case behavior and produce more accurate invariants.

### Discussion

While the use of dynamically detected invariants was convenient and effective, everything we learned about the `replace` program could have been detected via a combination of careful reading of the code, additional static analyses (including lexical searches), and selected program instrumentation. Adding inferred invariants provides several qualitative benefits that do not accrue from using only these other approaches.

First, inferred invariants are a succinct abstraction of a mass of data contained in the trace database. The programmer is provided with information — in terms of program variables at well-defined program points — that captures properties that hold across all runs represented in the trace database. Although these invariants may not be complete (interesting properties may be missed) and some may even be falsified by additional executions, they provided substantial insight that would be difficult for a programmer to extract manually from the database or from the program using traditional means.

Second, queries against the trace database can help programmers delve deeper when unexpected invariants appear or when expected invariants do not appear. For example, our expectations regarding the preconditions for `stclose` were contradicted by the inferred invariants, and the necessary information to clarify our 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.

Third, queries against the database can also be used to build intuition about the source of an invariant. In the scenario, for example, these data helped us to determine the format of the `pat` array and the conditions under which the loop in `stclose` is not executed.

Fourth, inferred 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 such as, "`*j` 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 higher-level invariant.

Finally, invariants provide a beneficial degree of serendipity. Scanning the invariants reveals facts that programmers would not have otherwise noticed and almost surely would not have thought to check. This ability to draw human attention to suspicious but otherwise overlooked aspects of the code is a powerful advantage of our approach. Programmers who know exactly what they are seeking or are attempting to verify a specific invariant may not gain as much leverage from our techniques.

No technique can make it possible to evolve systems that were previously intractable to change. But our initial, limited experience with inferred invariants shows promise in simplifying evolution tasks both by conveying additional information to the programmer and also by providing the trace database as a resource for obtaining other pertinent information.

## 5 SCALABILITY

We ran several simple experiments to determine the costs of invariant inference and the stability of the reported invariants as the test suite increases in size. Based largely on the results of these experiments, we also suggest ways to accelerate inference, improve scalability, and manage the reporting of invariants.

### Performance Measurements

To gain insight on scalability-related issues, we performed several measurements of invariant computation over the Siemens `replace` program. Our goal was to identify quantitative, observable factors that a user can control to manage the time and space requirements of the invariant engine. In particular, we measured the influence of the number of test cases (program runs) and the number of variables in scope at an instrumented program point. Because each instrumented program point is processed independently, program size affects invariant detection time only insofar as larger programs afford more instrumentation points.

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

We instrumented and ran `replace` on subsets of the 5542 test cases supplied with the program, including runs over 500, 1000, 1500, 2000, 2500, and 3000 randomly-chosen test inputs, where each set is a subset of the next larger one. We also ran over all the test cases, but our prototype implementation ran out of memory, exceeding 180MB, for one program point over 3500 inputs and for another program point over 4500 inputs. We could save substantial space by using a different data representation or by not storing every tuple of values (including every distinct array value) encountered by the program, for instance by only retaining certain witnesses and counterexamples, for use by our query tool, to properties that we check.

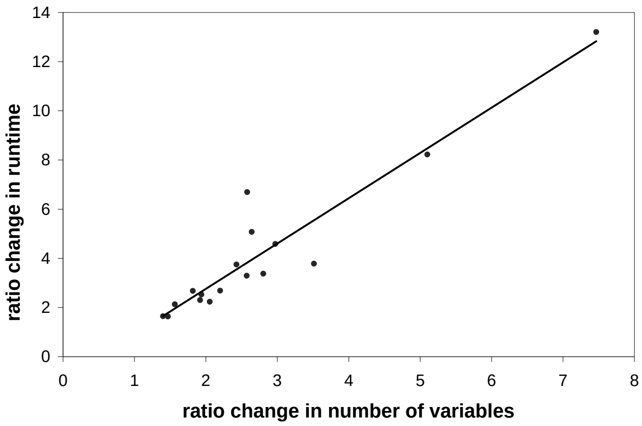


Figure 7: Change in invariant detection runtime versus change in number of variables. A least-squares trend line highlights the relationship; its $R^2$ value is over .89, indicating good fit. Each data point compares inference over two different sets of variables at a single instrumentation point, for invariant inference over 1000 program runs. (For 3000 test cases, the graph is similar, also with $R^2 = .89$.) If one run has $v_1$ variables and a runtime of $t_1$, and the other has $v_2$ variables and a runtime of $t_2$, the x axis measures $\frac{v_2}{v_1}$ and the y axis measures $\frac{t_2}{t_1}$. Doubling the number of variables tends to increase runtime by a factor of 2.5, while increasing the number of variables fivefold increases runtime by eight times.

Our system infers invariants over an average of 71 variables (6 original, 65 derived; 52 scalars, 19 sequences) per instrumentation point in `replace`. On average, 1000 test cases produce 10,120 samples per instrumentation point, and our system takes 220 seconds to infer the invariants for that point; for 3000 test cases there are 33,801 samples and processing takes 540 seconds.

*Number of instrumented variables*

The number of variables over which invariants are checked is the most important factor affecting 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. The number of derived variables is difficult to predict a priori because it depends on the values of other variables, as described in Section 3. On average, we found that about ten variables are derived for each original one; this number is remarkably insensitive to the relative numbers of scalars and arrays. In all of our statistics, we found that the number of scalars or sequences has no more (sometimes less) predictive power than the total number of variables.

Figure 7 plots growth in invariant detection time against growth in number of variables. It is difficult to make a fair comparison among program points with different sample sizes, value distributions, and inferred invariants. Therefore, 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. Our timing-related graphs omit three functions whose invariant detection runtimes were under one second, because runtime or measurement variations could produce

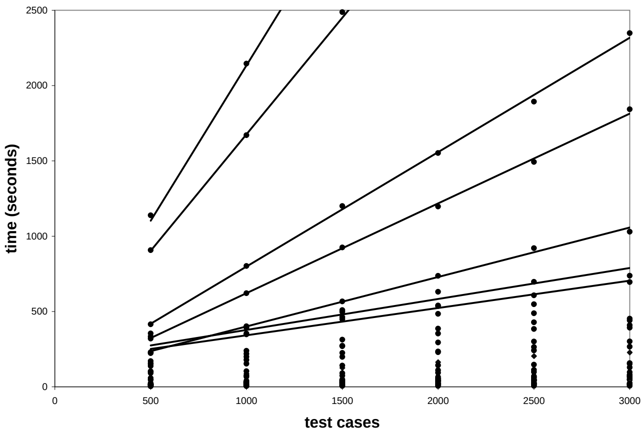


Figure 8: Invariant detection runtime as a function of number of test cases (program runs). Each program point is plotted individually, and lines are drawn through some data sets.

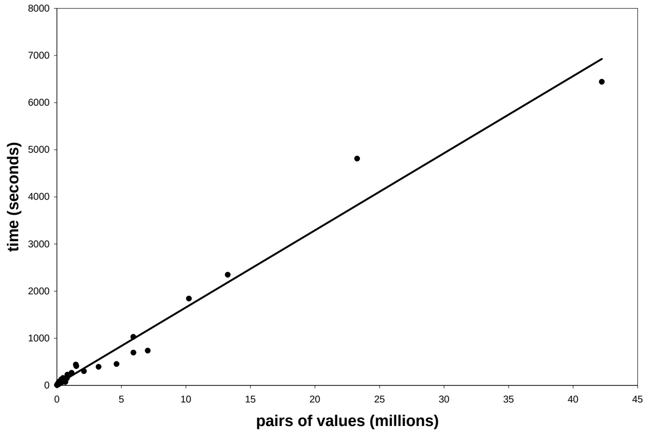


Figure 10: Number of pairs of values is the best predictor of runtime ($R^2$ = .94), but is itself not predictable from (though correlated with) number of test inputs and number of variables.

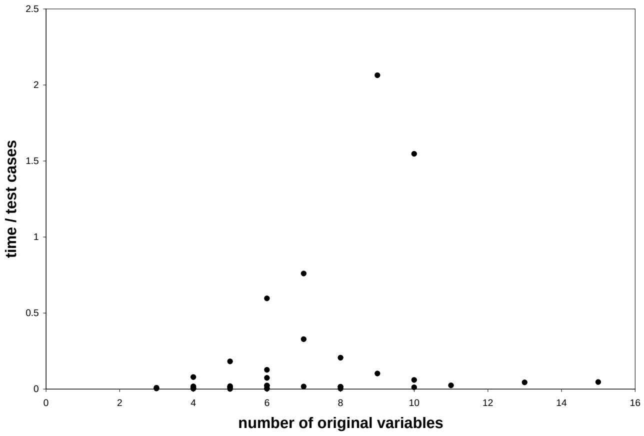


Figure 9: Growth of runtime with test suite size, plotted against number of source variables at a program point. The y axis plots slopes of lines in Figure 8.

inaccurate results. The other absolute runtimes vary from 4.5 to 2100 seconds, while the number of variables ranges from 14 to 230.

Figure 7 suggests that invariant detection time grows quadratically with the number of variables over which invariants are checked. The number of possible binary invariants (relationships over two variables) is also quadratic in the number of variables at a program point. We found that binary invariants dominate unary ones in number and cost of computation.

*Test suite size*

Test suite size has a somewhat less pronounced effect on invariant detection runtime. Figure 8 plots growth in time against growth in number of test cases (program runs) for each program point. Most of these relationships are strongly linear: nine have $R^2$ above .99, nine others have $R^2$ above .9, and five more have $R^2$ above .85. The remaining twelve relationships have runtime anomalies of varying severity; usually the data points fall on a line, with a single exception. Although the timings are reproducible, we have not yet isolated a cause for these departures from linearity.

Although runtime is (for the most part) linearly related to test suite size, the slopes of these relationships vary considerably, as can be seen by the divergent lines of Figure 8. Figure 9 plots these slopes against the total number of original variables (the variables in scope at the program point). When the slopes are plotted against total (original and derived) variables, or against variables of scalar or sequence type, the plot looks very similar: there is no obvious correlation that predicts the slopes, and thus the growth of runtime with test suite size.

The best independent predictor for runtime is the number of pairs of values encountered by the invariant engine; Figure 10 plots that linear relationship. Unfortunately, the number of pairs of values cannot be predicted from the number of test cases provided to the instrumented program, which the user can directly control. Unsurprisingly, the number of samples (number of times a particular program point is executed) is linearly related to test suite size (number of program runs). The number of distinct values is also well-correlated with the number of samples.[3] Finally, the number of pairs of values is correlated with the number of values, but apparently not with the ratio of scalar to sequence variables.

### Invariant stability

To determine whether the computed invariants tend to stabilize as test suite size increases, we compared, pairwise, the invariants detected on `replace` for different numbers of test cases. Figures 11 and 12 chart the number of identical, missing, and different invariants detected between two sets of test cases, where the smaller is a subset of the larger. Missing invariants are invariants that were detected in one of the test suites but not in the other. We also separate the differences into potentially interesting ones and probably uninteresting ones.

Typical uninteresting invariant differences are the following,

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

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

Figure 11: Invariant differences versus 3000 test cases.

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

Figure 12: Invariant differences versus 2500 test cases.

computed at the exit of function `putsub` when comparing a test suite of size 1000 to one of size 3000:

```
s1 in [0..98]              (99 values)
s1 >= 0                    (96 values)

i in [0..99]               (73 values)
i in [0..92]               (76 values)
```

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

We classify all other differences as potentially interesting; for example, when comparing a test suite of size 2000 to one of size 3000, the following difference is reported at the exit of `dodash`:

```
*j >= 2                    (105 values)
*j = 0 (mod 2)             (117 values)
```

Such differences may be worthy of further study to determine their relevance. In some cases, missing invariants may also merit closer examination.

The number of identical unary invariants grows modestly as the smaller test suite size increases. Identical binary invariants show a greater increase, particularly in the jump from 500 to 1000 test cases. Especially in comparisons with the 3000 case test set, there are some indications that the number of identical invariants is stabilizing, which might indicate asymptotically approaching the true set of invariants for a program. (We saw this property in the Gries programs in Section 2, where we found all the invariants Gries listed.)

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

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

Figures 11 and 12 paint somewhat different pictures of the sensitivity of invariants to the particular test cases over which the program is run. Only 2.5% of binary invariants detected for the 2000 or 2500 case test suites are not found identically in the other, and the number of invariants that differ is in the noise, though these are likely to be the most important differences. For comparisons against the 2500 test case suite, these numbers drop rapidly as the two test suites approach the same size. When the larger test suite has size 3000, more invariants are different or missing, and these numbers stabilize quickly. The 3000 case test suite appears somewhat anomalous: comparisons with other sizes (including larger test suites, for the functions over which we could compute invariants) show more similarity with the numbers reported for the 2500 case test suite. Our preliminary investigations have not revealed a precise cause for the larger differences between the 3000 case test suite and others, nor can we accurately predict the sizes of invariant differences; further investigation will be required in order to fully understand these phenomena.

When we examined invariant differences by hand, we discovered that many of them result from different values for pointers and uninitialized array elements. For example, the minimum value found in an array might be −128 in one set of runs and −120 in another, even though the array should contain only (nonnegative) characters. Other nonsensical values, such as the sum of the elements of a string, also appeared frequently in differing invariants. Important future directions of research will include reporting, or directing the user to, more relevant invariants and determining which invariant differences are significant and which can be safely ignored.

### 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]; our prototype does not use even the types present in the source code to prevent nonsensical comparisons.

**User Interface**

A large data set and large number of derived invariants can be overwhelming. We have already developed a tool that retrieves 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. Our invariant differencing tool can indicate how a program change has affected the computed invariants.

## 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]. This variety of ILP requires counterexamples (which are not available in our domain) and background knowledge, and the resulting relations typically misclassify portions of the training set. (Our approach characterizes the training set perfectly; either approach can misclassify additional data.) Other AI approaches like neural nets can 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 that generalizing these techniques, or applying them to subproblems of our task, can 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. Other researchers use event traces, which describe the sequence of events in a possibly concurrent system, to produce a finite state machine generating the trace [BG97, And98, CW98a, CW98b].

**Static Inference**

Static analyses operate on the program text, not on particular test runs, and are typically sound but conservative. As a result, properties they report are true for any program run, and theoretically they can detect all sound invariants if run to convergence [CC77]. (Static analyses will miss true but uncomputable properties and properties that depend on how the program is used, including properties of its inputs; dynamic techniques can detect both varieties.) Some program understanding tools have taken this abstract interpretation/dataflow approach [GC96, JH98]. In practice, 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 dynamically checking some such properties 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 checking it [Weg74, BLS96]; our goal is the discovery of such properties from a broad class of possible ones.

Variable types are a variety of formal specification and documentation. 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 [HHN92, GH96] propagates data structure shape descriptions through a program. 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, DB84, BBM97]. In the case of array bounds checking [SI77, Gup90, KW95, NL98, XP98], the desired property is obvious.

The Illustrating Compiler heuristically detects the abstract datatype implemented by a collection of concrete operations [HWF90], while ReForm semi-automatically transforms, by provably correct steps, a program into a specification [War96]. Other related work includes staging and binding-time analyses, which determine invariant or semi-invariant values for use in partial evaluation [JGS93].

## 7 CONCLUSIONS

This paper documents the feasibility and effectiveness of discovering program invariants based on execution traces. Our technique automatically detected all the stated invariants in a set of formally-specified programs, and the invariants detected in a real C program proved useful 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, the tool, and the overall approach. Moreover, we found that the use of dynamically inferred invariants qualitatively affected our programming, encouraging us to think in terms of invariants in situations that we might otherwise not have.

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 = 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 more effective. For instance, dynamic invariants form a program spectrum, changes to which can indicate properties of a changed program or input. They may 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. Invariants could augment the low-level execution information used in profile-directed compilation with higher-level program properties, permitting better optimization for the common case. Detected invariants could also bootstrap or direct a (manual or automatic) correctness proof.

## ACKNOWLEDGMENTS

Many of our colleagues provided comments on our ideas; we are particularly grateful for the feedback of Craig Chambers, Oren Etzioni, Tessa Lau, David Madigan, and Jared Saia. Vibha Sazawal provided valuable assistance with statistical analysis. Gregg Rothermel shared his modified versions of the Siemens test programs. Greg Badros, Craig Chambers, Tessa Lau, Todd Millstein, Jon Nowitz, Steve Wolfman, and the anonymous referees improved this paper by critiquing a draft. Daniel Jackson, Vass Litvinov, George Necula, James Noble, and the referees suggested related work.

This work was supported by NSF grants CCR-9506779 and CCR-9508745, an IBM Cooperative Fellowship, and a gift from Edison Design Group [EDG95].

## REFERENCES

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

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

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

[BG97] Bernard Boigelot and Patrice Godefroid. Automatic synthesis of specifications from the dynamic observation of reactive programs. In *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 321–333, Twente, April 1997.

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

[CC77] Patrick M. Cousot and Radhia Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. In *Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages*, volume 12(8) of *ACM SIGPLAN Notices*, pages 1–12, Rochester, NY, August 1977.

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

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

[CW98b] Jonathan E. Cook and Alexander L. Wolf. Event-based detection of concurrency. In *Sixth International Symposium on the Foundations of Software Engineering (FSE-6)*, Orlando, FL, November 1998.

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

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

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

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

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

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

[GC96] Gerald C. Gannod and Betty H.C. Cheng. Strongest postcondition semantics as the formal basis for reverse engineering. *Journal of Automated Software Engineering*, 3(1/2):139–164, June 1996.

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

[Giv96] Robert Givan. Inferring program specifications in polynomial-time. In *Proceedings of the Third International Symposium on Static Analysis, SAS '96*, pages 205–219, Aachen, Germany, September 1996.

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

[Gup90] Rajiv Gupta. A fresh look at optimizing array bound checking. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 272–282, White Plains, NY, USA, June 1990.

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

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

[HHN92] Laurie J. Hendren, Joseph Hummel, and Alexandru Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 249–260, San Francisco, California, June 1992.

[HRWY98] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. An empirical investigation of program spectra. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '98)*, pages 83–90, Montreal, Canada, June 1998.

[HWF90] Robert Henry, Kenneth M. Whaley, and Bruce Forstall. The University of Washington Illustrating Compiler. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 223–246, June 1990.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[JH98] Ralph Jeffords and Constance Heitmeyer. Automatic generation of state invariants from requirements specifications. In *Proceedings of the ACM SIGSOFT '98 Symposium on the Foundations of Software Engineering*, pages 56–69, Orlando, Florida, November 3-5, 1998.

[JvH+98] Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, Ulrich Hensel, and Hendrik Tews. Reasoning about Java classes. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, Vancouver, BC, Canada, October 1998.

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

[KW95] Priyadarshan Kolte and Michael Wolfe. Elimination of redundant array subscript range checks. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 270–278, La Jolla, California, June 1995.

[LN98] K. Rustan M. Leino and Greg Nelson. An extended static checker for Modula-3. In *Compiler Construction: 7th International Conference, CC'98*, pages 302–305. Springer-Verlag, April 1998.

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

[NCOD97] Gleb Naumovich, Lori A. Clarke, Leon J. Osterweil, and Matthew B. Dwyer. Verification of concurrent software with FLAVERS. In *Proceedings of the 19th International Conference on Software Engineering*, pages 594–595, May 1997.

[NL98] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 333–344, Montreal, Canada, June 1998.

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

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

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

[RBDL97] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the Sixth European Software Engineering Conference and Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 97)*, pages 432–449, Zurich, Switzerland, September 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.

[SI77] Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 132–143, Los Angeles, CA, January 1977.

[van97] Guido van Rossum. *Python Reference Manual*, release 1.5 edition, December 1997.

[War96] Martin P. Ward. Program analysis by formal transformation. *The Computer Journal*, 39(7):598–618, 1996.

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

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, Canada, June 1998.