Efficient Incremental Algorithms for Dynamic Detection of Likely Invariants

Jeff H. Perkins   Michael D. Ernst
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
32 Vassar Street, Cambridge MA USA
jhp@csail.mit.edu, mernst@csail.mit.edu

Abstract

Dynamic detection of likely invariants is a program analysis that generalizes over observed values to hypothesize program properties. The reported program properties are a set of likely invariants over the program, also known as an operational abstraction. Operational abstractions are useful in testing, verification, bug detection, refactoring, comparing behavior, and many other tasks.

Previous techniques for dynamic invariant detection scale poorly or report too few properties. Incremental algorithms are attractive because they process each observed value only once and thus scale well with data sizes. Previous incremental algorithms only checked and reported a small number of properties. This paper takes steps toward correcting this problem. The paper presents two new incremental algorithms for invariant detection and compares them analytically and experimentally to two existing algorithms. Furthermore, the paper presents four optimizations and shows how to implement them in the context of incremental algorithms. The result is more scalable invariant detection that does not sacrifice functionality.

Categories and Subject Descriptors: F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Invariants*

General Terms: Algorithms, Performance

Keywords: dynamic invariant detection, incremental algorithm, batch algorithm, reversing optimizations

1. Introduction

This paper presents and evaluates algorithms and optimizations for obtaining an operational abstraction — a formal description of properties that held on a series of program runs and can be expected to hold on future runs. The task of generating an operational abstraction is also known as dynamic detection of likely invariants, or dynamic invariant detection.

Dynamic invariant detection is an important and practical problem. Operational abstractions have been used in verifying safety properties [35, 30, 31], automating theorem-proving [27, 28], identifying refactoring opportunities [19], predicate abstraction [8, 9], generating test cases [38, 39, 13, 14], selecting and prioritizing test cases [16], explaining test failures [12], predicting incompatibilities in component upgrades [24, 25], error detection [34, 15, 33, 23, 4], error isolation [37, 21], and choosing modalities [22], among other tasks. Dynamic invariant detection has been independently implemented by several research groups, and related tools that also produce formal descriptions of run-time behavior have seen wide use (see Section 10).

While dynamic invariant detection is valuable, implementing it efficiently is challenging. A naive implementation is straightforward but fails to scale to problems of substantial size. The key parameters that control runtime are the runtime of the subject program (a longer-running program produces more data to be analyzed), number of variables and fields examined, size of each variable (arrays are more expensive to test than integers or booleans), number of program points at which invariant detection is performed (for instance, after every instruction versus only at entry and exit points of one component), and the grammar of invariants checked (checking all invariants that relate any three variables is more expensive than only considering those that relate two variables). The size of the program is *not* relevant, except insofar as it might affect the measures above.

Previous invariant detection algorithms have controlled runtime and space by limiting one or more of the above factors (see Section 10). As a result, the results have been limited in their expressiveness, or they have been applicable to fewer programs than would be desired, or both. Our goal is to remove some of these limitations, enabling invariant detection to be applied to more programs and to produce more detailed results.

We address this goal by providing two new algorithms for invariant detection (the *top-down incremental* algorithm and the *bottom-up incremental* algorithm) and by analytically and experimentally comparing these algorithms to one another and to two previously known algorithms (the *simple incremental* algorithm and the *multi-pass batch* algorithm). We describe three optimizations (equivalence sets, constants, and suppressions) previously implemented only in the multi-pass batch algorithm and show how to extend them to the incremental algorithms, and we present a new optimization (hierarchy). All of these optimizations take advantage of the fact that when certain properties hold, other properties are guaranteed to hold and need not be checked. We experimentally evaluate the four optimizations and discuss the complications arising from integrating the optimizations with the algorithms and with each other.

This paper is organized as follows. Section 2 provides background about dynamic invariant detection and explains the factors that affect its runtime. Section 3 gives a simple incremental algorithm for invariant detection, and Section 4 notes four types of redundancy in its output. The following sections give algorithms that exploit those redundancies in different ways: in a multi-pass algorithm (Section 5), a bottom-up incremental algorithm (Section 6), and a top-down incremental algorithm (Section 7), along with op-

SIGSOFT'04/FSE-12, Oct. 31–Nov. 6, 2004, Newport Beach, CA, USA.

timizations for the incremental algorithms (Section 8). Section 9 gives results of our experiments with the various algorithms and optimizations. Section 10 discusses related work, and Section 11 concludes.

2. Background: dynamic invariant detection

Dynamic detection of likely invariants [10] discovers likely invariants from program executions by instrumenting the target program (in source or binary form) to trace the variables of interest, running the instrumented program over a set of test cases, and postulating and checking invariants over values that the program computes. The essential idea is to use a generate-and-check algorithm to test a set of possible invariants against the observed values of the instrumented variables. (Each set of observed values at a program point is called a "sample".) The invariant detector reports those properties that are tested to a sufficient degree without falsification. The output includes properties such as "at entry to procedure `foo`, $myList$ is sorted", "at exit from procedure `bar`, $return \geq myVar$" (where $return$ stands for the return value), and "for all `Link` objects, $this.next.prev = this$". As with other dynamic approaches such as profiling, the accuracy of the results depends in part on the quality and completeness of the test cases. Even modest test suites produce good results in practice [31, 30], and techniques exist for creating good test suites for invariant detection [16, 14, 39]. In the remainder of this paper, for brevity we use "invariant" to mean "likely invariant", unless otherwise noted.

Which invariants are reported depends on which properties are checked — more specifically, the grammar of properties that are expressible by the invariant detector, the variables over which the properties are checked, and the program points at which the properties are checked. We discuss each of these factors. Section 10 discusses the choices that various implementations make for these factors. The reported properties also depend on filtering performed after checking, such as statistical tests (to reduce false positives) [10] and elimination of redundant properties [11].

Program points. Invariants can be checked at arbitrary locations in a program. Two examples are procedure entries and exits, resulting in invariants that correspond to preconditions and postconditions. It can be useful to compute invariants at each procedure exit (i.e., `return` statement) and also to compute an *aggregate* exit point (as viewed by a client) by generalizing over the individual exit points. Object or class invariants are also computed at an aggregate program point (*object point*), by generalizing over all objects that are observed at entry to and exit from public methods of a class, that are passed into or returned from methods of other classes, or that are stored in object fields.

Grammar of properties. An invariant detector may check just a few types of properties (such as equalities and inequalities among variables), or it may check a larger variety. Let x,y,z be variables, and let a,b,c be constants. Properties that might be checked include being constant ($x = a$), non-zero ($x \neq 0$), being in a range ($a \leq x \leq b$), linear relationships ($y = ax + b$), ordering ($x \leq y$), functions from a library ($x = \mathrm{fn}(y)$), containment ($x \in y$), sortedness (x is sorted), and many others.

An invariant detector may permit users to add domain-specific properties to be checked. An invariant detector may also report conditional properties or implications, such as $left \neq null \Rightarrow left.value \leq this.value$. Checking a larger number of properties makes it more likely that the output will contain the facts that are needed by a human or a tool; however, it also increases the runtime of the invariant detector and the likelihood of false positives.

Grammar of variables. The properties that an invariant detector can express must be instantiated over particular variables or other values; for example, $x \in y$ is instantiated over two variables, of which the second must be a collection. In addition to a procedure's parameters and return value, it is useful to detect invariants over additional variables (for instance, global variables or pre-state values), and also over values (called "derived variables") that are not manifest in the program. For example, if array `a` and integer `lasti` are in scope, then properties over `a[lasti]` may be of interest, even though it is not a variable and may not even appear in the program. At a procedure exit, including the original values of variables permits reporting side effects and input-output relationships. Given an object `a`, its fields (such as `a.f`) and their fields (such as `a.f.g`) provide useful additional information. The results of calls to side-effect-free methods can also be used as derived variables.

Examining structure elements results in *missing* variable values. For example, if variable `a` is null, then `a.f` is nonsensical. A subtler example results when fields are dereferenced to depth (say) 1, and class A contains field `x` of type integer and field `y` of type B, which contains field `z` of type integer. At uses of `b` of type B, both `b` itself (typically represented as an address or hash code) and field `b.z` are available. However, at uses of `a` of type A, only `a` and its fields `a.x` and `a.y` are available; the field `a.y.z` is beyond the field dereference depth and so is not available even if `a.y` is non-null. It requires special care to report correct results when one or more variables of an invariant may be missing; the invariant is not falsified but cannot be relied on in certain other situations.

Experimental Evaluation. This paper's experimental evaluation uses the Daikon implementation, which is publicly available from `http://pag.csail.mit.edu/daikon`. Daikon implements all the features listed in this section, including ternary derived variables and invariants, field dereferencing to a user-specified depth, user-specified invariants, conditional invariants, and statistical tests. A full list of its 31 derived variables (12 of which are enabled by default) and 161 invariants (152 enabled) appears in the Daikon user manual, available from its website. Daikon operates on C, Java, and Perl code, and on various other data formats.

3. Simple incremental algorithm

This section gives an *incremental*, or single-pass, invariant detection algorithm: it discards each sample after processing it, so the storage space requirements do not grow with the number of samples. An incremental algorithm can also run online (that is, simultaneously with the target program), eliminating the need to store trace files to disk.

The algorithm is as follows.

1. Initially, assume all properties in the grammar to be true. Instantiate a candidate invariant for each property and combination of variables. For example, if the grammar of properties is "*odd*" and "=", and the variables are x, y, and z, then instantiate $odd(x)$, $odd(y)$, $odd(z)$, $x = y$, $x = z$, and $y = z$.
2. For each sample, check each candidate invariant associated with the same program point as the sample and discard any that are contradicted by the sample. For example, the sample $\langle 3, 4, 3 \rangle$ eliminates the invariants $odd(y)$, $x = y$, and $y = z$ from the above list.
3. Report the invariants that remain after processing all sam-

V_P	number of variables/values in scope at a program point
$V_D(v)$	number of derived variables obtained from v original variables
V	total variables at a program point $= V_P + V_D(V_P)$
L	execution length: number of samples for a program point
I	number of possible invariants at a program point $= G(V)$
RI	number of reported invariants at a program point
P	program size: number of program points
$G(v)$	grammar: number of invariant templates, given v variables

Figure 1: Variables used in the running time and space analyses.

ples, and after applying post-processing filtering.

Each program point is processed independently.

The algorithm uses space only to store candidate invariants. The initial and maximum space usage is $S = O(P \cdot I) = O(P \cdot G(V))$. See Figure 1 for definitions of the variables. Suppose there are 12 types of derived variable that can involve up to 3 other variables (for example, the subarray `a[i..j]`), there are 152 types of invariant that can involve up to 3 variables (for example, $ax + by + cz = d$), 100 program points are instrumented, and there are 300 reachable variables and fields in scope at each program point. Then $P \cdot I \approx 10^{29}$, which is prohibitive. A static analysis can determine that not all variables can sensibly be compared to all others [32, 11], but the number of invariants is still a high-order polynomial in the number of variables (here, $I = O(v^9)$). On the other hand, if the grammar of invariants is very small, then the simple incremental algorithm's space usage is reasonable.

The worst-case runtime requires checking all invariants for each sample: $T = O(P \cdot I \cdot L)$. In practice, most invariants are false, and most false invariants are falsified quickly (after $O(1)$ samples), so the common-case runtime is $T = O(P \cdot I + P \cdot RI \cdot L)$, where the first summand is for falsified properties and the second summand is for never-falsified invariants, which are checked for all samples.

The simple incremental algorithm has been implemented by a number of research groups, and every invariant detector of which we are aware (except Daikon) is based on it.

4. Optimization opportunities: redundant properties

The simple incremental algorithm of Section 3 checks and reports more invariants than necessary. This section gives four examples of redundancy in the output. Optimizations based on three of these (equal variables, constants, and suppression) yielded significant performance enhancements in the multi-pass batch algorithm [11].

4.1 Equal variables

If two or more variables are always equal, then any invariant that is true for one of the variables is true for each of the variables. For example, if $x = y$ then for any invariant f, $f(x)$ implies $f(y)$.

In this context, the $x = y$ condition requires that whenever one variable is missing, so is the other. In the above example, if x was missing when y was not, then $f(x)$ does not necessarily imply $f(y)$ since the value for y when x was missing may have invalidated the invariant.

4.2 Dynamically constant variables

A dynamically constant variable is one that has the same value at each observed sample. The invariant $x = a$ (for constant a) makes any other invariant over (only) x redundant. For example, $x = 5$ implies $odd(x)$ and $x \geq 5$. Likewise for combinations of variables: $x = 5$ and $y = 6$ implies both $x < y$ and $x = y - 1$.

Missing values affect what can be concluded from a group of constants: there must exist at least one sample in which no constant in the group is missing.

4.3 Variable hierarchy

Some variable values contribute to invariants at multiple program points. For example, values observed at (public) method exits affect not only method postconditions but also object invariants. For two program points A and B, if all samples for B also appear at A, then any invariant true at A is necessarily true at B and is redundant at B. We have formalized this relationship as a partial order $\sqsubseteq_D$ in which lesser elements receive a subset of the samples received by a higher element, and higher elements contain a subset of the invariants that are true at a lower element [29, 26].

Because different variables can appear at different program points, the partial order is better understood as being over variables (which are organized into program points) rather than over program points.

Figure 2 illustrates the variable hierarchy for two simple Java classes. Three of the ways that the partial order relates variables are:

Enter – Exit: At an exit point, we use the notation x for the final value of variable x, and $orig(x)$ for the initial value of x (on entry to the procedure). Variables at procedure entry points are $\sqsupseteq_D$ the corresponding original variables at the procedure exit point.

Object – Method: Variables on object points are $\sqsupseteq_D$ the corresponding variables at each method entry and exit point.

Object – Client: Variables on object points are $\sqsupseteq_D$ the corresponding variables at each client of the class.

All of the relations are automatically determined from the program source.

4.4 Suppression of weaker invariants

An invariant is *suppressed* if it is logically implied by some set of other invariants. (The previous three examples of redundancy are special cases of this one that give rise to specific optimization opportunities.) For example, $x > y$ implies $x \geq y$, and $0 < x < y$ and $z = 0$ imply $x \text{ div } y = z$.

As with equal variables and dynamic constants, missing values affect what can be suppressed. In the second example above, there must be at least one sample where x, y, and z are all present in order for $x \text{ div } y = z$ to be true.

5. Multi-pass batch algorithm

The opportunities identified in Section 4 give powerful ways to identify logically redundant properties. Whenever certain antecedent properties hold, other properties are redundant and need not be created, checked, or reported. However, the simple incremental algorithm cannot take advantage of the redundancy. The antecedent properties cannot be relied upon until invariant detection is complete, because properties may be falsified at any time.

The multi-pass algorithm [11] addresses this issue by performing invariant detection in multiple passes. Early passes check simple invariants, and later passes check more complex invariants (and create derived variables) only if necessary.

Our implementation uses 5 passes. This number represents a compromise between exposing optimization opportunities and reducing the number of passes.

1. **Unary constant.** This pass determines whether each variable is constant and whether it can be missing. Subsequent passes ignore constant variables.

```
public class A {
  public B b;
  public void put(B arg);
}

public class B {
  public int x;
  public int get();
}
```

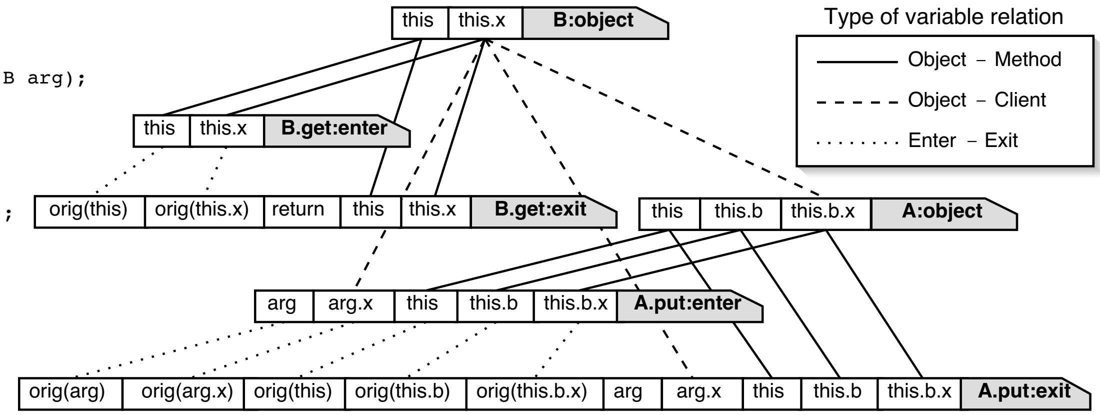

Figure 2: Variable hierarchy for two simple Java classes. Shaded areas name the program point, while unshaded boxes represent variables at that program point. Lines show the partial ordering $\sqsubseteq_D$ described in Section 4.3. Lesser elements appear lower in the figure; for instance, `orig(arg)` $\sqsubseteq_D$ `arg` in the lower left corner. Note that the `this` variable is not connected to any variable by an object–client relation. A variable such as `arg` (a parameter to method `put`) can be null, but `this` can never be null.

2. **Binary equality.** This pass checks equality for each pair of non-constant variables. For each set of equal variables, a *leader* variable is (arbitrarily) chosen to represent the set. Subsequent passes process only the leaders.
3. **All other unary.**
4. **All other binary.**
5. **Ternary.**

The suppression of weaker invariants optimization is built into the passes. Before creating any invariant, invariant-specific code checks whether the invariant is implied by any existing invariants. As just one example, if pass 4 discovered that $y = x$ and $x \geq z$, then pass 5 need not check the ternary invariant $y = \max(x, z)$.

Note that per pass 1, invariants are not checked between constants and variables. This is a powerful optimization, but it has undesirable side effects: some interesting invariants may be omitted as a result. Previous experiments demonstrated that this was necessary to achieve acceptable performance [10]. The version of the constant variables optimization for the incremental algorithms (Section 8.2) does not have this flaw.

Our implementation of the multi-pass algorithm does not currently implement hierarchy-based optimizations. Doing so is possible, but would make the algorithm more complicated and would require either postprocessing or increasing the number of passes by at least an additive factor of the depth of the hierarchy. Sections 6 and 7 describe the hierarchy optimization in incremental algorithms.

The multi-pass algorithm processes program points one by one, performing 5 passes over the trace data for each program point. As program points are processed, the samples associated with those program points can be discarded. With this optimization, the multi-pass algorithm's worst-case space cost is $S = O(P \cdot \max(VL, RI)$. The first argument is for storing the traces and represents the total space usage at the start of the algorithm, and the second is for storing the invariants and represents space usage at the end of the algorithm. In practice, the first argument dominates. Interning the trace data can reduce its storage costs.

The worst-case runtime is $T = O(P \cdot I \cdot L \cdot |\text{passes}|)$, but many invariants are quickly falsified, and even more are never even checked, leading to common-case runtime of $T = O(P \cdot I + P \cdot RI \cdot L \cdot |\text{passes}|)$.

A downside of the multi-pass approach is the need to store the trace data so it can be processed multiple times. Even modest traces can occupy gigabytes of disk space and memory, limiting the ability of this algorithm to work with non-trivial programs and making it inappropriate for online use. Reading files multiple times is another possibility, however, file I/O is a substantial cost in our implementation, so it stores the traces in memory, performing run-length encoding and interning identical samples to save space.

As an alternative to storing the trace data, one could run the target program multiple times, once for each pass of the multi-pass algorithm. This is not acceptable (or is prohibitively difficult) if the program performs side effects, depends on aspects of its environment that may change (including random number generators, memory addresses, hash codes, or thread scheduling), or uses expensive resources, including human attention. Many programs, such as operating systems and web servers, never terminate. A user may also wish to switch from gathering invariants to checking invariants on the fly [15].

6. Bottom-up incremental algorithm

An incremental algorithm can handle arbitrarily long executions of the target program, because it processes each sample exactly once, then discards it. However, the optimizations become both more complicated and more costly: the invariant detector must undo an optimization if the antecedents it depends on are ever falsified.

This section presents our new bottom-up incremental algorithm; Section 7 then proposes a top-down incremental algorithm. The two algorithms differ primarily in how they address the variable hierarchy optimization; the other optimizations are similar for the two incremental algorithms and are given in Section 8.

Any invariant that is true at a higher level in the variable hierarchy is also true at all lower levels. (For instance, every object invariant is a precondition and postcondition of every public method.) The key idea of the hierarchy optimizations is that when a particular property is true at multiple program points, the invariant should be checked at only one level of the hierarchy. In the bottom-up implementation, invariants are checked at the lowest possible level, and in the top-down implementation, they are checked at the highest possible level.

The bottom-up algorithm processes samples only at the leaves of the variable hierarchy; all aggregate program points are handled by a postpass. After all of the samples have been processed, the invariants at each non-leaf are built by merging the invariants at its children. An invariant is created at the parent iff an invariant of the same type, over the same variables, exists at each child. If the invariant does not exist at some child, it must have been falsified at that child and should thus be falsified at the parent.

Each invariant has its own merging routine that operates in a manner similar to its run-time processing. This is straightforward for *stateless*, or *sample-independent*, invariants that express only one fact about their variables and whose internal state does not change as a result of processing a sample. For example, greater-than ($x > y$) and product ($x = y \cdot z$) are stateless invariants.

Sample-dependent invariants are those whose equation includes a constant, such as $x \geq 42$ and $y = 2x + 1$. It is not reasonable to instantiate and check every possible $x \geq c$ or $y = ax + b$ invariant; instead, one object stands for all such invariants over given variables, and the constants are computed to fit the observed samples. At run time, processing a sample may change the meaning of the invariant — for instance, by adding a linearly independent point that permits the constants to be computed or by weakening the invariant, such as by changing $x \geq 42$ to $x \geq 10$ when a sample with value 10 is observed.

Merging sample-dependent invariants requires invariant-specific processing. For example, if one child contains the invariant $x \leq 15$ and the other child contains the invariant $x \leq 22$, the merged invariant at the parent is $x \leq 22$.

One complication when populating parent program points is the need to properly relate variables at multiple program points, as illustrated in Figure 2. Variables of the same name need not correspond, and the correspondence can cross class boundaries. The optimizations of Section 8 further complicate this correspondence. When merging, the parent's set of equivalence sets of variables (Section 8.1) is the intersection of the sets of equivalence sets at each child. Two variables are equal at the parent iff they are equal at each child. Invariant merging must also merge the information used by the statistical tests mentioned in Section 2, so that the statistics can be computed as if the parent program point had processed the samples itself.

In the bottom up approach, each sample is processed exactly once (at its leaf). Non-leaf program points are only considered after all processing is complete. The processing time for parents is dependent on the (small) number of invariants found at the children and not the (large) number of samples.

7. Top-down incremental algorithm

The bottom-up algorithm instantiates a given invariant — say, $x < y$ — at every leaf program point that contains both x and y. By contrast, the top-down algorithm aims to save more space, at the cost of additional runtime, by instantiating each invariant only at the highest point(s) in the hierarchy at which it is true. All invariants are initially created at the top of the partial order. When a sample is read from a trace, it is processed at every program point from the top of the partial order down to the leaf program point to which the sample belongs. If an invariant is falsified at a program point, then the invariant is immediately removed from that program point and is added to all its children, before the sample is processed at the child.[1] As a result, each invariant appears at most once on any path starting at the top of the hierarchy. The set of invariants for a program point consists of all invariants that appear at it or at a higher program point. For an example, see Figure 3.

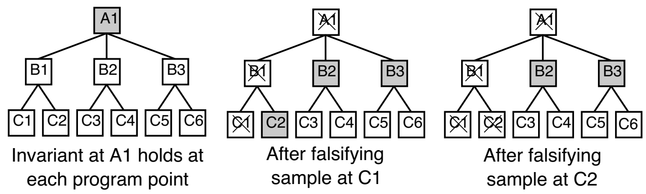

Figure 3: Example of top-down hierarchy optimization. The left diagram shows an invariant (point A1, marked with the gray background) at the top of the hierarchy: it holds at every program point. The center diagram shows the hierarchy after a falsifying sample is received at point C1 (and processed in order at A1, B1, and C1). The right diagram shows the hierarchy after a falsifying sample is received at point C2.

In order to be applied at multiple program points, the sample must be transformed to include the correct values in the correct order. This is required because different variables appear at different program points and a non-leaf program point may see samples originating at multiple leaves. (The bottom-up algorithm does this transformation only once, during postprocessing. It also only transforms invariants and not samples.)

The top-down algorithm also makes the incremental optimizations (Section 8) more complex. The invariants that hold at a program point can be physically located at that program point or at any higher location(s) in the partial order. In order to search for an invariant the optimizations traverse the hierarchy, and when copying an invariant to a (possibly different) program point they place each copy at the target program point only if it does not appear anywhere higher than the target.

Unfortunately, sample-dependent invariants (which tend to be the most numerous in practice) must be exempted from the top-down hierarchy optimization and instantiated everywhere in the partial order. Processing of a sample-dependent invariant must observe every sample for a program point, without omitting any. For example, suppose that in Figure 3 the invariant was $x = c$ (that is, x is equal to some constant c), and the order of samples was:

Prog. point	value
C1	5
C2	5
C1	4

The first sample would set the equation as $x = 5$. The third sample would falsify the invariant. Following the algorithm outlined above, the $x = 5$ invariant would be copied down to each of its children as shown in the center diagram of Figure 3. This, however, would be incorrect since program points B2 and B3 have received no samples and should instead indicate that c is not yet bound in $x = c$.

These and other problems with achieving correctness led us to modify our implementation of the top-down algorithm. Rather than instantiating invariants only at the highest point in the hierarchy, they are instantiated at each level in the tree. Those that are lower in the tree are suppressed (see Section 8.3) by those higher in the tree. This gains the time advantages made possible by utilizing the variable hierarchy (each invariant is only checked at one level of the hierarchy), but not the space advantages (invariants are duplicated). In other words, although the top-down algorithm is intuitively appealing and promises space savings by instantiating invariants at fewer locations in the hierarchy, it did not achieve its goals.

[1] Our implementation accounts for the fact that the partial order forms a dag, not a tree.

Before	After
(v w) (x y z)	(v w) (x y) (z)
$f(v)$	$f(v)$
$f(x)$	$f(x)$ $f(z)$
$g(v,v)$	$g(v,v)$
$g(v,x)$	$g(v,x)$ $g(v,z)$
$g(x,v)$	$g(x,v)$ $g(z,v)$
$g(x,x)$	$g(x,x)$ $g(x,z)$ $g(z,x)$ $g(z,z)$

Figure 4: Copying of invariants due to splitting of equality sets. Equality sets and invariants are shown before and after processing a sample in which $v = w$, $x = y$, $y \neq z$.

8. Optimizations for incremental algorithms

In the multi-pass algorithm, later passes can rely on properties that are known to be true across all samples. In contrast, properties that enable optimizations in the incremental algorithms are always subject to change. The algorithm must be able to undo any optimization. This may require creating previously suppressed invariants and putting them in the same state they would have been in, had they processed all the samples that have been seen so far.

8.1 Equal variables

The incremental algorithms can use equivalence sets of variables just as the batch algorithm does, but they must dynamically update the equivalence sets. Initially all variables are placed in a single equivalence set (modulo comparability, as noted in Section 3). Invariants are instantiated only over the leaders of sets.

During processing, if any elements of the equivalence set differ (in their value or in whether they are missing), the equivalence set is broken into multiple parts, and invariants over the original leader are copied to all the new leaders.

The copying proceeds as follows. For each invariant that mentions the leader of the old equivalence set, duplicate the invariant as many times as there are new equivalence sets. In each duplicate, replace the first instance of the old leader by a different one of the new leaders. If the original invariant mentioned the old leader more than once, then the duplicates still mention the old leader at least once and must be recursively processed. Each original invariant mentioning the old leader n times turns into $|newsets|^n$ duplicates. For an example, see Figure 4. This algorithm extends in a straightforward manner to multiple equivalence sets breaking up simultaneously.

Reflexive invariants that use a single variable more than once, such as $x \leq x$, are uninteresting and should never be reported; however, the equal variables optimization requires their existence during processing. Under the equal variables optimization, an invariant over a leader variable stands for multiple invariants over members of the equivalence set. On the left side of Figure 4, $f(v)$ stands for both $f(v)$ and $f(w)$, and $g(v, x)$ stands for six invariants. Reflexive invariants are required because an equivalence set might later break up. For example, $g(x, x)$ stands for $g(x, y)$, $g(x, z)$, and 7 other invariants. Without $g(x, x)$, the invariants $g(x, z)$ and $g(z, x)$ on the right side of the figure would not have been created by the duplication step. As a further optimization, reflexive invariants need not be created when there are fewer members of the equality set than variables in the invariant. Thus, in the example, it is not necessary to create $g(z, z)$.

8.2 Constant variables

The incremental algorithms dynamically maintain a set of constant variables. It is not necessary to instantiate an invariant if each of its variables is a constant. However, if any of an invariant's variables is non-constant (not a member of the constant set), the invariant is instantiated.

As each sample is processed, the new value for each constant variable is examined. If the new value is different or is missing, the variable is removed from the constant set, and invariants relating its values to the remaining constants are created.

A variable that is ever missing is not considered constant. If a variable is sometimes missing, it is not known what combinations of that variable with other variables have ever existed.

The incremental algorithms instantiate invariants between constants and non-constants for two reasons. First, they are interesting in their own right. Second, they are necessary if a constant later becomes a non-constant. When the last variable of a set becomes non-constant, it is too late to create an invariant over the set, because the previous values of the other variables have been lost. Instantiating and checking invariants over every set of variables containing at least one non-constant solves this problem.

8.3 Suppression of weaker invariants

Each type of invariant can specify a set of possible *suppressions*. Each suppression is a set of antecedent invariants that together imply the invariant. For example, $\{\{x{=}y,\ z{=}1\}, \{x{=}z,\ y{=}1\}\}$ is a possible set of suppressions for the "product" invariant $x = y \cdot z$. An invariant is suppressed if all of the antecedents in any suppression hold.

We discuss two implementations of the suppression mechanism. The first one instantiates every invariant as usual and then checks, for each invariant that might be suppressed, whether all of the antecedents exist for any suppression. If so, a data structure records that fact, and the suppressee invariant need not be checked at each sample. If any antecedent is falsified, then the suppressee becomes unsuppressed, unless some other suppression holds. Eliminating checking at each sample saves runtime, but the data structures that track the suppressions increase memory usage.

The second implementation avoids instantiating suppressed invariants. It maintains no state whatsoever to indicate whether a non-existent invariant has been removed because it was falsified, or was never instantiated because it is suppressed. This saves space, but at the cost of more complicated processing.

Rather than linking each instantiated invariant that acts as an antecedent to information indicating what instantiated invariants it helps to suppress, each invariant type has a list of the suppression types for which it may be an antecedent.

Whenever an invariant that might be an antecedent is falsified, every non-instantiated invariant that it might be suppressing is checked. If any suppression for the invariant holds before falsification of the antecedent, but no suppression holds after falsification of the antecedent, then the invariant must be instantiated. In other words, the invariant is un-suppressed (instantiated) as soon as the last suppression no longer holds.

For example, if potential antecedent $w = 1$ is falsified, the "product" invariant ($x = y \cdot z$) must be considered over every set of 3 variables that includes w, and every suppression of each such invariant must be checked. This is expensive, but it achieves the goal of requiring no storage for each suppressed invariant.

In our implementation, the top-down algorithm uses the solution that instantiates suppressed invariants and the bottom-up algorithm uses the solution that does not. It is more complex and expensive to search for antecedents in top-down (Section 7). The combination of this expense with the extra searches required in the non-instantiating solution is prohibitive.

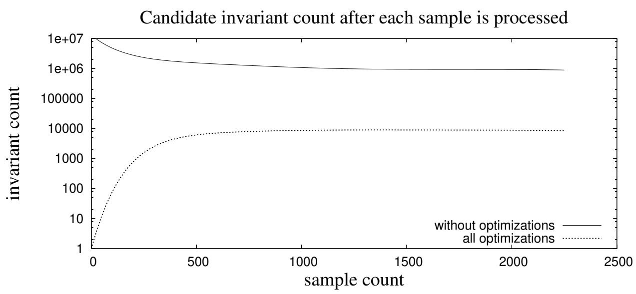

Figure 5: The number of candidate invariants as samples are processed, with and without optimizations by the bottom-up algorithm. The data is from a single program point in the flex lexical analyzer.

8.4 Discussion

The optimizations to the incremental algorithms are most effective while processing initial samples of a trace. Initially, all invariants are true; but most are redundant, and all variables are in the same equality set. The in-memory representation is relatively small. As samples are processed, antecedent properties are falsified (and equality sets break up) and the number of instantiated invariants grows. By contrast, without the optimizations, the simple incremental algorithm initially creates all invariants, but the number only decreases thereafter. Figure 5 shows how the number of invariants grows without and with optimizations. The figure also highlights the effectiveness of the optimizations. The maximum number of invariants with optimizations is 100 times less than the minimum number of invariants without the optimizations.

9. Experiments

In order to evaluate the algorithms and optimizations described in this paper, we ran experiments to measure, in terms of time and space, the effects of algorithm, optimizations, program size, and trace size. This paper presents experiments over two rather different target programs: the flex lexical analyzer (a C program that is part of the standard Linux distribution) and the utility libraries of Daikon itself (written in Java).

Flex contains 391 program points averaging 275 variables each, and the trace we used is 9.2 gigabytes with 232,000 samples. The trace was created by running flex over a number of sample scanners. The Daikon utilities contain 1,593 program points averaging 60 variables (with a maximum of 3672 variables), and the trace file is 11.5 gigabytes with 26 million samples. The trace file was created by running Daikon over a small example program. The simple incremental algorithm creates over 750 million candidate invariants for flex, and over 460 million for Daikon (ignoring all program points with more than 1500 variables, because we could not even instantiate them in 1.4 gigabytes of memory).

We ran two sets of experiments. The first evaluates the different algorithms against one another, with all optimizations enabled. The second evaluates the optimizations by comparing the effects of each optimization in isolation. For each run of the invariant detector, we measured run time (wall clock time) and maximum memory size. We used five identical 2.2 GHz Pentium 4 PCs with 1 Gbyte of memory, limiting the maximum Java heap to 750 Mbytes to eliminate thrashing. If results are not shown for some or all sizes on a graph, the experiment ran out of memory before completion.

For each set of experiments, we simulated programs of various sizes by considering fewer or more program points. (The file I/O operations read the entire file, but the invariant detector did not process samples at the ignored program points.) Using part of a single program rather than many distinct programs of different sizes avoids conflating program size with number or type of variables, patterns of data, or other factors. This permits direct comparison of results and also indicates performance when a user instruments part of a large program, which is a realistic scenario. We simulated shorter runs of the target program by reading only a fraction of the trace file, while processing all program points.

The invariants created by each algorithm, both with and without optimizations, are identical except for some relatively minor differences noted in the discussion of their implementation. Comparing outputs has increased our confidence in the implementations.

9.1 Algorithm Comparison

Figure 6 shows the time and space usage of the multi-pass, bottom-up, and top-down algorithms, graphed against target program run time (trace size). Figure 7 graphs time and space against target program size (number of program points). The simple incremental algorithm does not complete even at the minimum program size or minimum data trace file size.

The top-down algorithm runs out of memory when processing the full flex trace. The top-down algorithm requires more memory than the bottom-up one for two reasons. First, for correctness, it must instantiate invariants at every level in the variable hierarchy. Second, its data structures are significantly larger because it needs to relate the variables in the hierarchy as it is processing samples. The bottom up algorithm uses the hierarchy only when merging invariants. In the Daikon utilities, memory usage is similar, but the bottom up algorithm runs about twice as fast, more so with large program sizes and trace sizes. This is due to the greater overhead required for top-down processing.

Memory usage for bottom-up is independent of trace size and linearly related to program size (since each program point is processed independently). The multi-pass algorithm is similar in speed to the bottom-up algorithm, but multi-pass cannot process the full trace for flex or the Daikon utilities. Maximum memory use for multi-pass grew less quickly than we expected, probably because of the effectiveness of the interning optimization and the fact that the Java memory system works harder to save memory (for instance, garbage-collecting more frequently) as the limit is approached. This explains why some of the memory graphs, which measure maximum memory usage, are flat near the memory limit.

9.2 Optimization Comparison

When comparing the optimizations we used the bottom-up algorithm, which dominates the others and is able to process the full traces, because we wished to use a single, good baseline. Furthermore, the optimizations are an integral part of the multi-pass approach and are difficult to turn on and off. The results are shown in Figure 8.

The equality optimization is by far the most powerful. It is crucial because so many variables are equal to one another. For example, for a side-effect-free procedure, the post-state value of each parameter and global variable is equal to its pre-state value. Eliminating any one optimization causes Daikon to run out of memory on the flex example, so none of them is extraneous.

The optimization benefits for the Daikon utility libraries are less than those for flex. The utilities have far fewer variables per program point — a library has little state — and the most heavily exercised library procedures were very simple — `assert` represented almost 20% of all calls into the library. Thus, much of the processing time for the utilities was expended where the optimizations afforded little benefit.

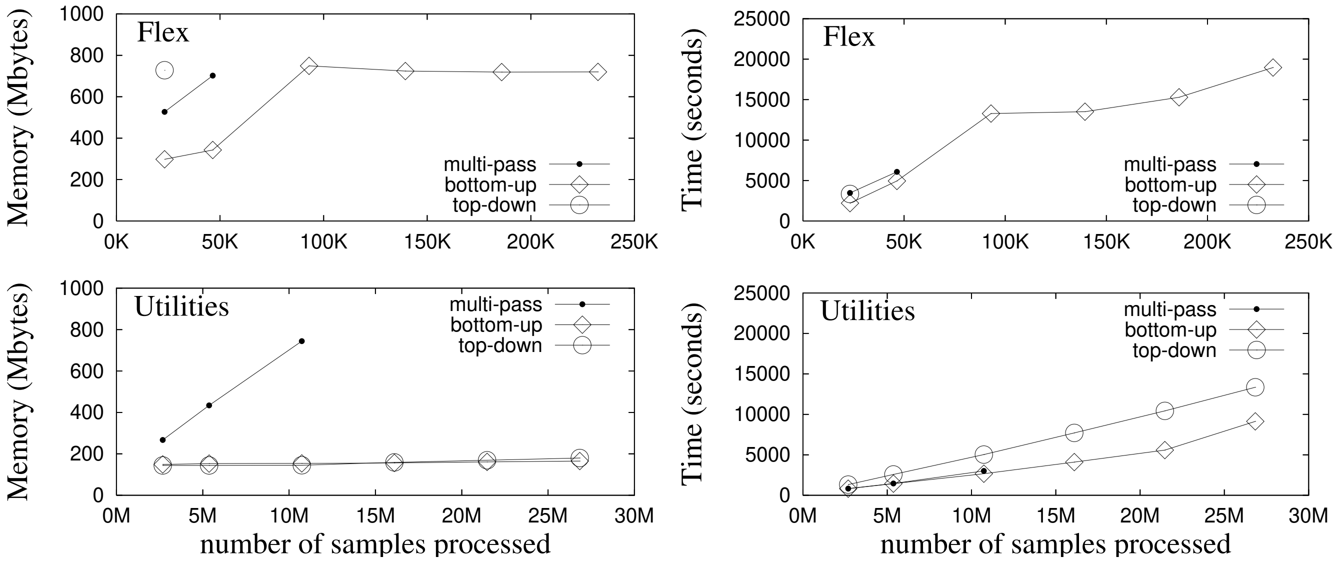

Figure 6: Comparison of algorithms over Flex and the Daikon utilities with respect to trace length. The multi-pass and top-down algorithms run out of memory before processing the full Flex trace. Multi-pass also runs out of memory before processing the full Daikon utilities trace. The simple incremental algorithm never completes even at the minimum trace length of 20,000 samples.

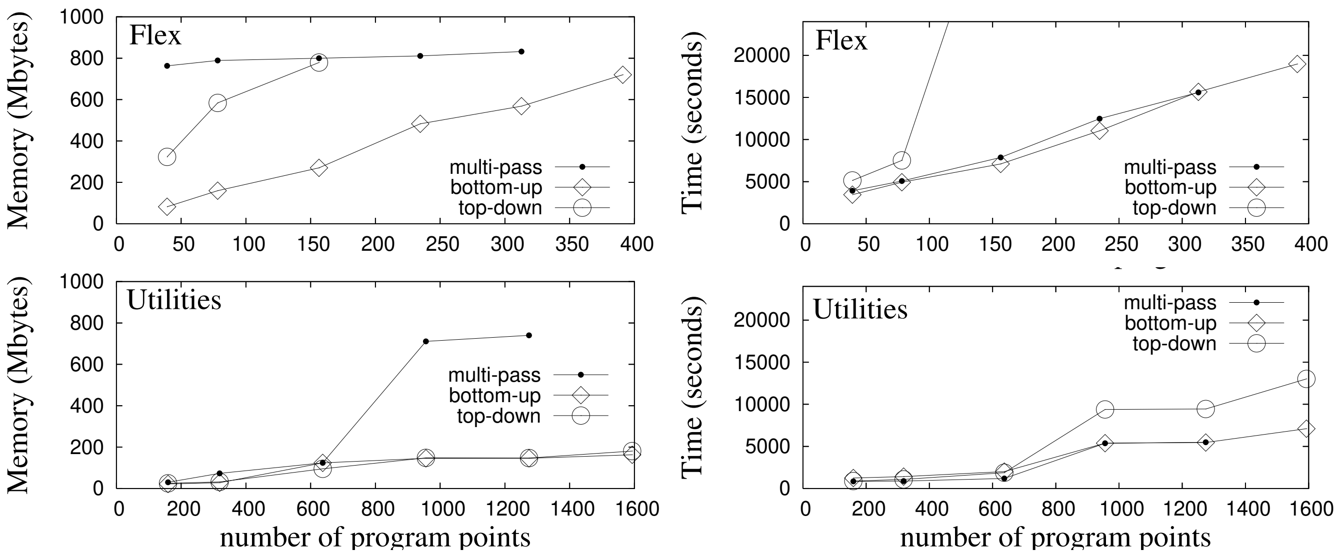

Figure 7: Comparison of algorithms over Flex and the Daikon utilities with respect to program size. The multi-pass and top-down algorithms run out of memory before processing all of the program points in Flex. Multi-pass also runs out of memory before processing all of the program points in the Daikon utilities. The simple incremental algorithm does not complete even for one program point.

10. Related work

Numerous researchers have adopted and adapted the ideas of dynamic detection of likely invariants. Section 2 discussed relevant features of the Daikon implementation. We now describe other implementations, based on published information (no implementations other than Daikon are publicly available). While the other implementations use the simple incremental algorithm, some of them run faster than Daikon, primarily because Daikon checks many more invariants (millions of times more, on the flex example). However, the incremental algorithms and optimizations should be applicable to other implementations.

The DIDUCE tool [15] checks one unary invariant on Java programs; at each program point (a field or array reference, or a procedure call), that invariant is checked for three values: a variable's current and previous values and their difference. The invariant indicates all previously seen values of every bit of the value. As an invariant is weakened (new values are observed in a given bit), a message is printed. A user can look for weakenings that are printed

	Program	Hierarchy	Equality	Constants	Suppression
Time	Flex	2.48	17.9	1.60	1.11
Space	Flex	1.86	6.70	1.81	1.68
Time	Utilities	1.20	1.72	1.16	1.00
Space	Utilities	1.16	1.56	1.26	1.00

Figure 8: Comparison of the optimizations, averaged over a variety of different program sizes and trace lengths. Reported numbers are the result without one optimization, divided by the result when all optimizations are enabled. For example, processing flex is 17.9 times slower and uses 6.7 times more memory without the equality optimization. For this figure, time and space spent reading the trace file (which is common to all approaches) are not included.

on erroneous runs or just before an error occurs, to find rare corner cases. The tool was used to help explain several known errors and to reveal two new errors.

The Carrot tool [33] checks 2 unary and 4 binary invariants. By re-using Daikon's instrumentation, it can handle any of the languages Daikon can, and can dereference fields. In an experiment comparing faulty and non-faulty program runs, the results did not indicate the problem, in contrast to other work with similar aims [15, 12].

Remote program sampling [21] is a light-weight mechanism that evaluates two properties (one unary and one binary, but instantiated for a linear rather than quadratic number of variable pairs) at assignments in C programs (plus the predicate at each branch), counting the number of times that each property is satisfied. The properties are checked probabilistically: on most executions of a program point, property checking is skipped. Any of the other implementations could be so extended, sacrificing soundness (over the test suite) but gaining performance. The results are processed statistically to indicate which properties are best correlated with faults and thus most likely to be indicative of faults. As with DIDUCE, relatively dense instrumentation means early warning, if one of the relatively simple properties indicates a bug. In an experiment, the tool rediscovered 7 known errors and found one new one.

Arnout's tool for extracting implicit contracts [2] aims to add preconditions. The analysis determines what conditions give rise to an exception being thrown, then adds the negation of that condition as a precondition.

Henkel and Diwan [17, 18] have built a tool that discovers algebraic specifications, which relate the meaning of sequences of code operations, such as "pop(push(x,stack))=stack". The tool generates many terms (test cases) from the signature of a Java class and proposes equations based on the results of the tests. The tool also proposes and tests generalizations.

The SPIN model checker has been extended to check whether two variables are related by $=, <, >, \leq$, or $\geq$ [35]. The output is a graph with variables at the nodes and edges labeled by the comparison relations.

Programming by demonstration and inductive logic programming aim to generate a program from a sequence of examples or other data [3, 7, 20]. The output is similar to that obtained from dynamic invariant detection, but aims to be complete rather than partial and so must be targeted to a smaller domain.

Several researchers have inferred, from program or system traces, finite state automata that represent the permitted transitions [6, 5, 1, 36]. Specifications written in the form of automata are complementary to the formula-based program properties that are generated by a dynamic invariant detector.

11. Conclusion

We have presented two new incremental invariant detection algorithms and compared them to two existing algorithms (one incremental and one batch). We have shown how to perform three previously described optimizations, and one new one, in the more challenging incremental context, which requires undoing optimizations if the antecedents they depend on become invalidated. (This aspect of the work can be viewed as a special-purpose automatic theorem prover optimized to efficient retraction of axioms.) We have implemented all the algorithms and optimizations in a single framework, permitting a direct experimental comparison.

We conclude with a discussion of the merits of the various algorithms. The simple incremental algorithm is very easy to understand and to implement. At least 6 implemented invariant detectors are based on this algorithm. When the number of invariants being checked is very small — for example, because the implementation only considers a small number of variables or invariant types — this is clearly the best algorithm. However, it does not scale.

The multi-pass algorithm offers a convenient framework for optimizations, because no work ever need be undone. This greatly reduces both the computational complexity and the storage requirements of the optimizations. The storage savings are offset by the need to store trace data for re-processing (or to re-run the target program, which is often impractical), and one of the optimizations eliminates some desirable invariants from the output. The multi-pass algorithm is reasonable for moderate-sized datasets and large numbers of invariants, which are beyond the scope of the simple incremental algorithm. However, longer runs require an incremental algorithm.

The bottom-up incremental algorithm was the best performer in our experiments. No other algorithm was able to fully process the datasets. The optimizations control space usage by ensuring that only a modest number of invariants exist at any one time, even near the beginning of the run; it is a substantial accomplishment to be competitive with the performance of the multi-pass algorithm, which has been tuned over years of use. The incremental nature of the algorithm makes its runtime proportional to (and its space usage independent of) dataset size, and permits on-line invariant detection (concurrently with the target program, without storing any trace data). Much of the complexity of the algorithm is incurred only in the final postprocessing step.

The top-down incremental algorithm shares many characteristics with the bottom-up algorithm. However, the top-down algorithm is ineffective for sample-dependent invariants, has more complicated processing for each sample, and requires a more difficult search to determine whether an invariant is true.

Overall, our algorithms, implementations, and experiments suggest how invariant detection can simultaneously scale to both nontrivial numbers of invariants, and programs of nontrivial size. The incremental algorithms and optimizations are implemented in version 3 of the Daikon invariant detector, which is available at `http://pag.csail.mit.edu/daikon`.

Acknowledgments

Jeremy Nimmer and Toh Ne Win wrote a preliminary implementation of the top-down incremental algorithm, and discussions with them helped to develop many of the ideas. Comments from the anonymous referees helped us to improve the presentation of this paper. This research was funded by NSF grants CCR-0133580 and CCR-0234651, the Oxygen Project, the Deshpande Center for Technological Innovation, and gifts from NTT and Toshiba.

12. References

[1] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*, pages 4–16, Jan. 2002.

[2] K. Arnout and B. Meyer. Extracting implicit contracts from .NET components. `http://se.inf.ethz.ch/people/arnout/work_in_progress/contract_extraction.pdf`, Sept. 13, 2002.

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

[4] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE*, pages 480–490, May 2004.

[5] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM TOSEM*, 7(3):215–249, July 1998.

[6] J. E. Cook and A. L. Wolf. Event-based detection of concurrency. In *FSE*, pages 35–45, Nov. 1998.

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

[8] N. Dodoo, A. Donovan, L. Lin, and M. D. Ernst. Selecting predicates for implications in program analysis, Mar. 16, 2002. Draft. `http://pag.csail.mit.edu/~mernst/pubs/invariants-implications.ps`.

[9] N. Dodoo, L. Lin, and M. D. Ernst. Selecting, refining, and evaluating predicates for program analysis. Technical Report MIT-LCS-TR-914, MIT Lab for Computer Science, July 21, 2003.

[10] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):1–25, Feb. 2001.

[11] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE*, pages 449–458, June 2000.

[12] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN 2003*, pages 121–135, May 2003.

[13] N. Gupta. Generating test data for dynamically discovering likely program invariants. In *WODA 2003*, pages 21–24, May 2003.

[14] N. Gupta and Z. V. Heidepriem. A new structural coverage criterion for dynamic detection of program invariants. In *ASE 2003*, Oct. 2003.

[15] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, pages 291–301, May 2002.

[16] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *ICSE*, pages 60–71, May 2003.

[17] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *ECOOP*, pages 431–456, July 2003.

[18] J. Henkel and A. Diwan. A tool for writing and debugging algebraic specifications. In *ICSE*, pages 449–458, May 2004.

[19] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *ICSM*, pages 736–743, Nov. 2001.

[20] T. Lau, P. Domingos, and D. S. Weld. Learning programs from traces using version space algebra. In *K-CAP 2003: Second International Conference on Knowledge Capture*, Florida, USA, Oct. 23–26, 2003.

[21] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, pages 141–154, June 2003.

[22] L. Lin and M. D. Ernst. Improving adaptability via program steering. In *ISSTA*, pages 206–216, July 2004.

[23] L. Mariani and M. Pezzè. A technique for verifying component-based software. In *TACoS*, Mar. 2004.

[24] S. McCamant and M. D. Ernst. Predicting problems caused by component upgrades. In *ESEC/FSE*, pages 287–296, Sept. 2003.

[25] S. McCamant and M. D. Ernst. Early identification of incompatibilities in multi-component upgrades. In *ECOOP*, pages 440–464, June 2004.

[26] T. Ne Win. Theorem-proving distributed algorithms with dynamic analysis. Master's thesis, MIT Dept. of EECS, May 2003.

[27] T. Ne Win and M. D. Ernst. Verifying distributed algorithms via dynamic analysis and theorem proving. Technical Report 841, MIT Lab for Computer Science, May 25, 2002.

[28] T. Ne Win, M. D. Ernst, S. J. Garland, D. Kırlı, and N. Lynch. Using simulated execution in verifying distributed algorithms. *STTT*, 6(1):67–76, July 2004.

[29] J. W. Nimmer. Automatic generation and checking of program specifications. Technical Report 852, MIT Lab for Computer Science, June 10, 2002. Revision of author's Master's thesis.

[30] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *ISSTA*, pages 232–242, July 2002.

[31] J. W. Nimmer and M. D. Ernst. Invariant inference for static checking: An empirical evaluation. In *FSE*, pages 11–20, Nov. 2002.

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

[33] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss. Automated fault localization using potential invariants. In *AADEBUG*, Sept. 2003.

[34] O. Raz, P. Koopman, and M. Shaw. Semantic anomaly detection in online data sources. In *ICSE*, pages 302–312, May 2002.

[35] M. Vaziri and G. Holzmann. Automatic detection of invariants in Spin. In *SPIN 1998*, Nov. 1998.

[36] J. Whaley, M. Martin, and M. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA*, pages 218–228, July 2002.

[37] T. Xie and D. Notkin. Checking inside the black box: Regression fault exposure and localization based on value spectra differences. Technical Report UW-CSE-02-12-04, U. Wash. Dept. of Comp. Sci. & Eng., Seattle, WA, USA, Dec. 2002.

[38] T. Xie and D. Notkin. Exploiting synergy between testing and inferred partial specifications. In *WODA 2003*, pages 17–20, May 2003.

[39] T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In *ASE 2003*, pages 40–48, Oct. 2003.