Selecting Predicates for Implications in Program Analysis

Nii Dodoo Alan Donovan Lee Lin Michael D. Ernst

MIT Lab for Computer Science
200 Technology Square
Cambridge, MA 02139 USA
{dodoo,adonovan,leelin,mernst}@lcs.mit.edu
http://pag.lcs.mit.edu/daikon/

Abstract

This research proposes and evaluates techniques for selecting predicates for conditional program properties — that is, implications such as $p \Rightarrow q$ whose consequent is true only when the predicate is true. Conditional properties are prevalent in recursive data structures, which behave differently in their base and recursive cases, and in many other situations. The experimental context of the research is dynamic detection of likely program invariants, but the ideas should also be applicable to other domains.

It is computationally infeasible to try every possible predicate for conditional properties, so we compare procedure return analysis, static analysis, clustering, random selection, and context-sensitive analysis for selecting predicates.

Even a simple static analysis is fairly effective, presumably because many of the important properties of a program are tested or expressed by programmers. However, important properties are implicit in the program's code or execution. We give examples of important properties discovered by each of the other analyses. We experimentally evaluate the techniques on two tasks: statically proving the absence of runtime errors with a theorem-prover, and detecting errors by separating erroneous from correct executions. We show that the techniques improve performance on both tasks, and we evaluate their costs.

1. Introduction

The goal of program analysis is to determine facts about a program. The facts are not collected for their own sake, but are presented to a user, depended on by a transformation, or used to aid another analysis. The properties typically take the form of predicates that are true at a particular program point or points. Such predicates are called invariants, and a program specification is a collection of invariants.

The usefulness of a program analysis depends on what properties it can report. There is a tradeoff between expressiveness and efficiency: increasing the grammar of a program analysis's output tends to make the results more useful but the analysis more costly. The cost arises from checking more potential properties or from new mechanisms required in order to detect new classes of properties. A major challenge is increasing the grammar of a program analysis without making the analysis unreasonably more expensive and without degrading the quality of the output, when measured by (human or machine) users of the output.

This paper investigates techniques for expanding the output grammar of a specific dynamic program analysis that, given program executions, produces likely specifications — sets of invariants — as output. (Section 2.1 describes the technique.) The base analysis reports properties such as preconditions, postconditions, and representation invariants that are unconditionally true over a test suite. This research adds implications of the form $a \Rightarrow b$. Disjunctions such as $a \vee b$ are a special case of implications, since $(a \Rightarrow b) \equiv (\neg a \vee b)$.

A conditional invariant is an invariant whose consequent is true only when the predicate is true. For instance, the local invariant over the node n of a heap, $(n.left.value \leq n.value)$ && $(n.right.value \leq n.value)$, is true only if n, $n.left$ and $n.right$ are non-null. Conditional invariants are particularly important in recursive data structures, where different properties typically hold in the base case and the recursive case.

Extending a dynamic analysis to check implications is trivial, but it is infeasible to check $a \Rightarrow b$ for all invariants a and b that the base analysis can produce. One reason is runtime cost: the analysis checks each invariant over entire program executions, and the proposed change squares the number of potential invariants that must be checked. A more serious objection concerns output accuracy. Checking (say) 100 times as many properties is likely to increase the number of false positives — properties that are reported but are not true or are not useful for the user's current task — by a factor of 100. This is acceptable only if if the number of true positives is also increased by a factor of 100, which is unlikely.

Since it is infeasible to check $a \Rightarrow b$ for every a and b, the dynamic analysis must restrict the implications that it checks. We propose to do so by restricting what properties are used for the predicate a, while permitting b to range over all properties reportable by the analysis.

This paper considers five techniques (detailed in Section 3) for selecting predicates for implications: procedure return analysis; static analysis; clustering; random selection; and context-sensitive analysis. All but the second are dynamic analyses that examine program executions rather than program text. This enables them to produce information (predicates) about program behavior that is not apparent from the program text. It also enables them to work on programs for which source code is not available, such as libraries or other system components not under the direct control of the software engineer running the tool.

We evaluated the five techniques in three different ways. First, we compared the accuracy of the produced invariants, where accuracy is measured by a program verification task

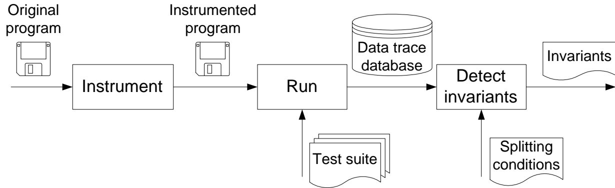

Figure 1: Architecture of the Daikon tool for dynamic invariant detection. The "splitting conditions" input is optional and enables detection of implications of the form "$a \Rightarrow b$". The focus of this paper is the selection of the splitting conditions.

(Section 4.1). Second, we determined how well each of the techniques may lead programmers to the errors underlying faulty behavior (Section 4.2). Third, we performed ad-hoc evaluation by reading the results to see what program properties they revealed that were unexpected and valuable even to a programmer familiar with the code (Section 4.3).

Before discussing the techniques and their evaluation, the paper provides background on the dynamic technique for detecting program invariants (Section 2.1) and details the mechanism for detecting implications (Section 2.2). The rest of the paper then focuses on policy decisions about which implications will be detected. Section 3 describes the five policies in detail, and Section 4 evaluates them. Finally, Section 5 discusses related work, and Section 6 concludes.

2. Background

2.1 Dynamic invariant detection

This section briefly describes dynamic detection of program invariants and the Daikon implementation. Full details appear elsewhere [Ern00, ECGN01].

Dynamic invariant detection discovers likely invariants from program executions by instrumenting the target program to trace the variables of interest, running the instrumented program over a test suite, and inferring invariants over the instrumented values (Figure 1). The inference step tests a set of possible invariants against the values captured from the instrumented variables; those invariants that are tested to a sufficient degree without falsification are reported to the programmer. As with other dynamic approaches such as testing and profiling, the accuracy of the inferred invariants depends in part on the quality and completeness of the test cases. The Daikon invariant detector is language independent, and currently includes instrumenters for C, Java, and IOA [GLV97].

Daikon detects invariants at specific program points such as procedure entries and exits; each program point is treated independently. The invariant detector is provided with a variable trace that contains, for each execution of a program point, the values of all variables in scope at that point. Each of a set of possible invariants is tested against various combinations of one, two, or three traced variables.

For scalar variables x, y, and z, and computed constants a, b, and c, some examples of checked invariants are: equality with a constant ($\mathsf{x} = \mathsf{a}$) or a small set of constants ($\mathsf{x} \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$), lying in a range ($\mathsf{a} \leq \mathsf{x} \leq \mathsf{b}$), non-zero, modulus ($\mathsf{x} \equiv \mathsf{a} \pmod{\mathsf{b}}$), linear relationships ($\mathsf{z} = \mathsf{ax} + \mathsf{by} + \mathsf{c}$), ordering ($\mathsf{x} \leq \mathsf{y}$), and functions ($\mathsf{y} = \mathsf{fn}(\mathsf{x})$). Invariants involving a sequence variable (such as an array or linked list) include minimum and maximum sequence values, lexicographical ordering, element ordering, invariants holding for all elements in the sequence, or membership ($\mathsf{x} \in \mathsf{y}$). Given two sequences, some example checked invariants are elementwise linear relationship, lexicographic comparison, and subsequence relationship.

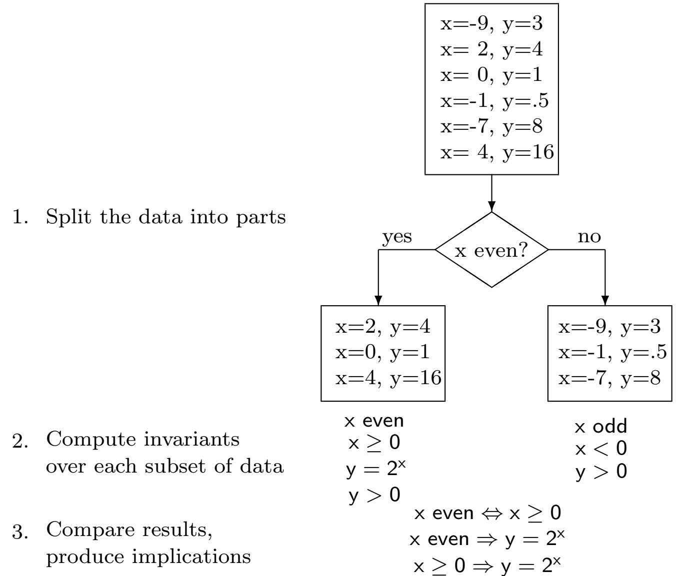

Figure 2: Mechanism for dynamic detection of implications.

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

An invariant is reported only if there is adequate statistical evidence for it. In particular, if there are an inadequate number of observations, observed patterns may be mere coincidence. Consequently, for each detected invariant, Daikon computes the probability that such a property would appear by chance in a random set of samples. The property is reported only if its probability is smaller than a user-defined confidence parameter [ECGN00].

The Daikon invariant detector is available from `http://pag.lcs.mit.edu/daikon/`.

2.2 Detecting implications

Figure 2 shows the mechanism for dynamic detection of implications [EGKN99, Ern00]. Rather than directly testing specific $a \Rightarrow b$ invariants, the invariant detector is provided with *splitting conditions* and then generates implications that may or may not reference the splitting conditions

The invariant detector first splits the input data (variable values) into parts based on a splitting condition; selection of the splitting condition is the main problem in detection

```
{Create implications from elements of S1 to elements of
S2. S1 and S2 are sets of invariants.}
CREATE-IMPLICATIONS(S1, S2)
for all s1 ∈ S1 do
  if ∃s2 ∈ S2 such that s1 ⇒ ¬s2 and s2 ⇒ ¬s1 then
    {s1 and s2 are mutually exclusive}
    for all s′ ∈ (S1 − S2) do
      output "s1 ⇒ s′"
    end for
  end if
end for
```

Figure 3: Pseudocode for creation of implications from invariants over mutually exclusive subsets of the data.

Invariants		Implications		Simplified
T_1	T_2	$a \Rightarrow b$	$\neg a \Rightarrow \neg b$	$a \Leftrightarrow b$
a	$\neg a$	$a \Rightarrow d$	$\neg a \Rightarrow f$	$a \Rightarrow d$
b	$\neg b$	$a \Rightarrow e$	$\neg b \Rightarrow \neg a$	$a \Rightarrow e$
c	c	$b \Rightarrow a$	$\neg b \Rightarrow f$	$\neg a \Rightarrow f$
d	f	$b \Rightarrow d$		
e		$b \Rightarrow e$		

Figure 4: Creation of implications from invariants over subsets of the data. The left portion of the figure shows invariants over two subsets T_1 and T_2. The middle portion shows all implications that are output by two calls to the CREATE-IMPLICATIONS routine of Figure 3 ; c appears unconditionally, so does not appear in any implication. The right portion shows the implications after logical simplification.

of implications, and is the focus of this paper. After the data is split into parts, ordinary invariant detection is performed to detect (non-implication) invariants in each subset of the data. Finally, the separately-detected invariants are combined into implications, if possible. If the splitting condition is poorly chosen, or if no implications hold over the data, then the same invariants are detected over each subset of the data, and no implications can be reported.

Figure 3 gives pseudocode for the third step of Figure 2, creation of implications from invariants over subsets of the data. The CREATE-IMPLICATIONS routine is run twice, swapping the arguments, and then the results are simplified according to the standard rules of Boolean logic. Figure 4 gives a concrete example of the algorithm's behavior.

Each mutually exclusive invariant implies everything else true for its own subset of the data. (This is true only if the subsets are mutually exhaustive; for instance, given three data subsets that induce invariant sets $\{a, b\}$, $\{\neg a, \neg b\}$, $\{a, \neg b\}$, it is not valid to examine only the first two subsets of the data and to conclude that $a \Rightarrow b$.) It is not necessary to use any universally true property as the consequent of an implication, since the universal property appears unconditionally. In other words, if c is universally true, there is no sense outputting "$a \Rightarrow c$".

The implication predicates need not be splitting conditions, but can be any invariants detected in the subsets of the data. Many of the resulting implications would not be created if predicates were limited to pre-specified splitting conditions; for example, $\mathsf{x} \geq 0 \Rightarrow \mathsf{y} = 2^{\mathsf{x}}$ in Figure 2. Daikon is also able to refine splitting conditions to simpler or more exact predicates (see Figure 6). In practice, the splitting condition does appear over one subset of the data, and its negation appears over the other subset. However, there are three reasons that the splitting condition (or its negation) might not be detected in a subset of the data.

The first reason is that the splitting condition is inexpressible in Daikon's grammar. It would be easy to extend the algorithm so that it uses the splitting condition as an implication predicate in such cases. However, those inexpressible invariants are likely to be beyond the capabilities of some other tools such as static checkers (see Section 4.1).

The second reason that the splitting condition might not be detected in a subset of the data is that a stronger condition is detected by Daikon. This is not a concern, because the weaker condition is also detected and used for creation of implications. When Daikon performs output, it suppresses implied invariants, which do not add any new information but only clutter the output. Thus, if s is stronger than w (that is, s implies w), then Daikon will check both s and w but, if both are true, will only report s; similarly, it will check both $s \Rightarrow a$ and $w \Rightarrow a$ but, if both are true, will only report $w \Rightarrow a$.

The third reason is that the splitting condition is expressible and true, but is not statistically justified and so is suppressed from Daikon's output. This can occur, for example, if one of the subsets is very small; however, such suppression is infrequent and is usually advantageous. Eliminating statistically unjustified invariants tends to greatly reduce the size of Daikon's output and typically does not reduce the usefulness of the result.

3. Techniques for selecting predicates

We have reduced the problem of detecting predicates to that of selecting splitting conditions. This section outlines the techniques for detecting splitting conditions that we experimentally evaluated: procedure return analysis, simple static analysis, clustering, random selection, and context-sensitive analysis. Section 3.6 proposes additional techniques.

3.1 Procedure return analysis

Two simple splitting conditions based on dynamic checks of procedure returns are built into Daikon. (We disabled them in some of our experiments, in order to assess their efficacy.)

The first splits data based on the return site. If a procedure has multiple `return` statements, then it is likely that they evidence different behaviors: one may be a normal case and the other may be an exceptional case, a fast-path computation, a base case, or different in some other manner. Splitting based on the return site captures such differences.

The second splits based on boolean return values, separating cases for which a procedure returns true from those for which it returns false.

3.2 Static analysis

Daikon uses a simple static analysis for selecting splitting conditions. Each boolean condition used in the program (for instance, as the test of a `if`, `while`, or `for` statement) is used as a splitting condition. Additionally, the body of each pure boolean member function (of the same class as the one being analyzed) is used. (The functions must be side-effect-free and non-exception-throwing; allocations are

permitted so long as the new objects do not escape.) A more sophisticated static analysis (say, a dataflow analysis or abstract interpretation) is clearly possible, but we have not yet felt the need for one.

The rationale for this approach is that if the programmer considered a condition worth testing, then it is likely to be relevant to the problem. Furthermore, if a test can affect the implementation, then that condition may also affect the specification or externally visible behavior. That is not always the case; for instance, `for` loops conditions may be over local variables that cannot be sensibly used as splitting conditions at the specification level.

This is an opportune time to note that Daikon permits splitting conditions to be associated with a single program point (such as a procedure entry or exit) or to be used indiscriminately, at every possible program point. In our experiments, we use the latter option. When splitting conditions concern global variables, this permits a splitting condition produced for one program point to be used throughout. When splitting conditions concern procedure parameters, they typically are not valid at other program points, because there are no variables with the given names at the other program point. However, occasionally programmers consistently use specific parameter names for specific purposes, and we have seen quite a few cases in which a splitting condition produced for a specific program point was very advantageously used elsewhere. For instance, a condition might always be relevant to the program's state, but might only be statically checked in the one routine that checks it explicitly. Similar situations arise when a splitting condition is produced only at certain places by the other splitting strategies.

3.3 Clustering

Cluster analysis, or clustering, is a multivariate analysis technique that creates groups, or clusters, of self-similar datapoints. Clustering aims to partition datapoints into clusters that are internally homogeneous (members of the same group are similar to one another) and externally heterogeneous (members of different groups are different from one another). Clustering is a widely used technique in artificial intelligence, data mining, and information retrieval, and has been applied to problems in many other domains (see Section 5).

Splitting conditions are intended to split data traces into parts that contain similar conditions (the consequents of the implications) and that are different from other parts (the predicates of the implications). Clustering has exactly the same goals, so we reasoned that clustering might be effective for detecting splitting conditions. Furthermore, because clustering is a purely dynamic technique — it examines the run-time variable values and separates them into groups — it may discover distinctions in the data that would be difficult or impossible to detect via more traditional means.

Figure 5 demonstrates the three steps for producing splitting conditions via cluster analysis. First, clustering is performed on the data trace file. Each program point is clustered individually. Each datapoint is a tuple of variable values, one value for each variable in scope at the program point. Once each tuple of variable values has been assigned a cluster (numbered from 1 to k), the data trace file is rewritten to add a new variable, `cluster`, to each sample. Section 3.3.2 describes the clustering algorithm in greater detail.

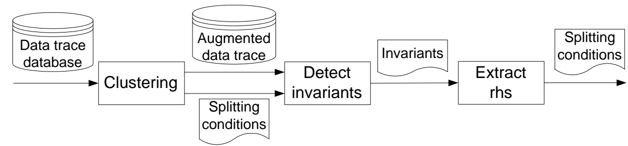

Figure 5: Producing splitting conditions via cluster analysis. The augmented data traces contain a new variable, `cluster`, at each program point, and the intermediate splitting conditions test that variable. The final step removes all mention of the cluster variables, resulting in splitting conditions for the original data trace file.

The second step is to run Daikon over the augmented data trace file. This recursive invocation of Daikon itself relies on detection of conditional invariants. The splitting conditions supplied to the recursive Daikon invocation are of the form "`cluster` $= c$", for each possible value c, $1 \leq c \leq k$, for the `cluster` variable. Daikon produces all the output it ordinarily would have, plus additional implications arising from the splitting conditions.

The third step extracts the consequent from each implication whose predicate mentions the `cluster` variable. These consequents, which mention variables from the original data trace file, are properties that were true in one (but not all) of the clusters. They are output as splitting conditions for the original, unmodified data trace file.

3.3.1 *Refining splitting conditions*

The procedure above employs invariant detection to produce a set of splitting conditions that can then be employed in a final invocation of invariant detection. An alternative one-pass technique might add the `cluster` variable to a trace file, perform invariant detection over that trace file, and filter references to the `cluster` variable out of the resulting invariants. The two-pass process that performs invariant detection twice is preferable to the one-pass technique for two reasons.

First, the two-pass procedure produces a set of splitting conditions that can be used on the original data traces. They are human-readable, easing inspection and editing, and they can be reused during other invariant detection steps.

Second, performing invariant detection helps to refine the cluster information. Clustering is inherently statistical and inexact; it may not partition the data exactly as desired. However, so long as at least one cluster induces one of the desired properties, the extra invariant detection step can leverage this into the desired splitting condition. If the original clustering produces the desired grouping, then the additional step does no harm.

As an example, consider Figure 6. The clusters nearly, but not exactly, match the true separation between behaviors, so no non-cluster-related implications can be reported. However, the implication `cluster` $= 1 \Rightarrow x < 0$ is produced, and using $x < 0$ as a splitting condition produces the desired properties "$x < 0 \Leftrightarrow \triangle$" and "$x > 0 \Leftrightarrow \square$".

The random selection technique (Section 3.4) relies on similar refinement of an imperfect initial data subsetting.

3.3.2 *Clustering details*

Two common clustering techniques are k-means and hierarchical. We use the k-means technique, which starts with

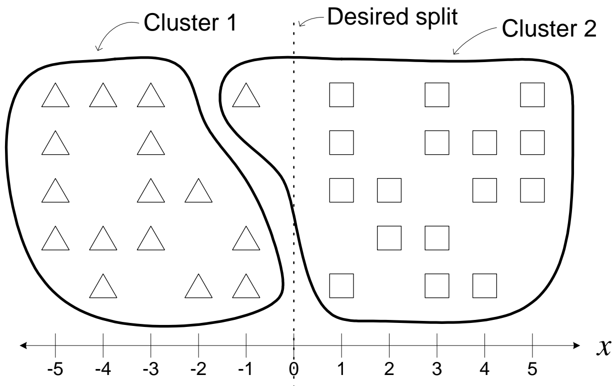

Figure 6: Refinement of cluster information via an extra step of invariant detection. The groups created by clustering approximate, but do not exactly match, the natural division in the data (between triangles and squares, at $x = 0$). An extra invariant detection step produces the desired property as a splitting condition.

k random clusters and then repeatedly assigns points to the nearest cluster and recomputes the cluster centroids, until the clusters and centroids stop changing. We also experimented with hierarchical clustering, which starts with one point per cluster, then repeatedly merges the two closest clusters into a single cluster. Hierarchical clustering is more computationally expensive and gave identical results in our experiments.

For each input, we ran k-means clustering 10 times and chose the best result, which has the minimum sum of distances from points to their cluster centroids. The repeated trials avoid returning a poor local minimum. We chose $k = 4$ because it gave good results in preliminary experiments; thus, the result contained four clusters. In our experiments, $k = 4$ tended to produce results like that of $k = 3$, except that one of the three clusters was split into two; likewise for $k = 2$. Therefore, there would have been little advantage to running the algorithm for $k = 2$ and $k = 3$ as well as for $k = 4$.

Clustering operates on points in an n-dimensional space. Each point is a single program point execution (such as a procedure entry) and the dimensions are values for scalar variables in scope at that program point. Before performing clustering, we normalized the data so that each dimension has a mean of 0 and a standard deviation of 1. This ensures that large differences in some attributes (such as hashcodes or floating-point values) do not swamp smaller differences in other attributes (such as booleans).

3.4 Random selection

Randomized algorithms [MR97] have seen fruitful application to many problem domains; they can be just as effective in finding splitting conditions to enable production of implications. Figure 7 how to select splitting conditions via randomized analysis. First, select r different subsets of the data, each of size s, and perform invariant detection over each subset. Then, use any invariant detected in one of the subsets, but not detected in the full data, as a splitting condition.

As with the clustering technique, the randomly-selected subsets of the data need not be perfect. Suppose that some

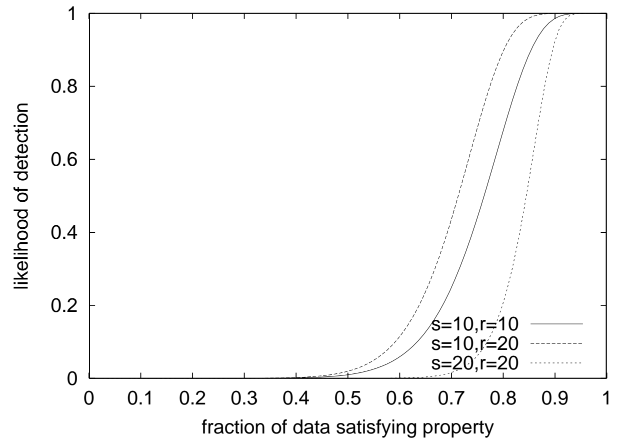

Figure 8: Likelihood of finding an arbitrary split via random selection. s is the size of each randomly-chosen subset, and r is the number of such subsets.

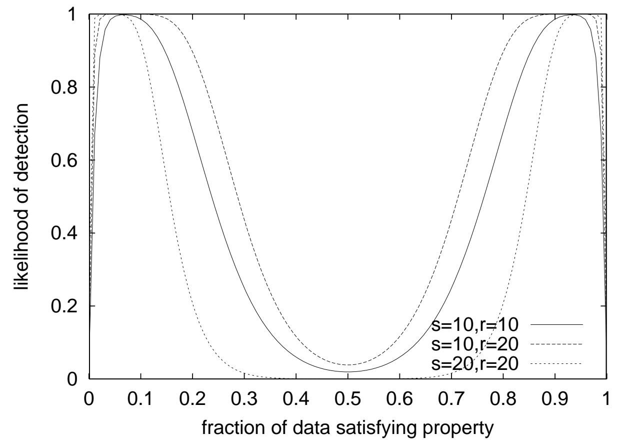

Figure 9: Likelihood of finding an arbitrary split via random selection, without comparison against invariants over the full data. s is the size of each randomly-chosen subset, and r is the number of such subsets.

property holds in a fraction $f < 1$ of the data. It will never be detected by ordinary (unconditional) invariant detection. However, if one of the randomly-selected subsets of the data happens to contain only datapoints where the property holds, then the condition will be detected (and re-detected when the splitting conditions are used in invariant detection).

Figure 8 shows how likely a property is to be detected by this technique, for for several values of s and r. The property holds in all s elements of some subset with with probability $p = f^s$. Thus, the property is detected with probability p on each trial. The property holds on at least one of the subsets with probability $1 - (1 - p)^r$, and this is the quantity graphed in Figure 8.

An alternate technique would compare invariants detected over the subsets with one another rather than with invariants detected over the full data. This avoids the need for a potentially costly full run of Daikon. The analysis changes as follows. The property holds in either all or none of the s elements with probability $p = f^s + (1 - f)^s$. The property (or its negation) holds on at least one of the subsets

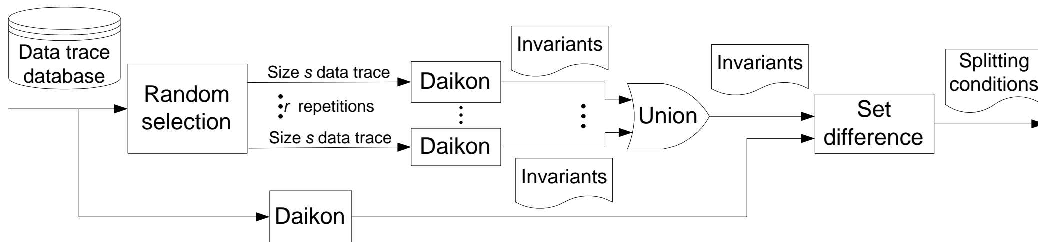

Figure 7: The randomized algorithm for choosing splitting conditions. The technique outputs each invariant that is detected over a randomly-chosen subset of the data, but is not detected over the whole data.

with probability $1 - (1-p)^r$. However, if it holds on all subsets, it is still not detected, so we adjust the value down, to $(1 - (1-p)^r)(1 - f^{rs} - (1-f)^{rs})$. Figure 9 graphs this quantity against f, for several values of s and r.

Figure 9 indicates that the alternative approach is superior for relatively uncommon properties ($.02 < f < .4$) but worse for rarely violated properties ($f > .98$). If both the property and its negation are expressible in Daikon's grammar, as is usually the case, then the original approach always dominates.

The likelihood of detecting a property can be improved by reducing s or by increasing r. The danger of reducing s is that the smaller the subset, the more likely that invariants are not statistically justified and the more likely that any resulting invariants overfit the small sample. The danger of increasing r is that work linearly increases with r. We chose $s = 10$ and $r = 20$ for our experiments.

Figures 8 and 9 indicate that the random selection technique is most effective for unbalanced data; when when f is near .5, it is likely that both an example and a counterexample appears in each subset of size s.

We believe that many interesting properties of data are at least moderately unbalanced. For example, the base case appears infrequently for data structures such as linked lists; unusual conditions or special-case code paths tend to be executed only occasionally; and the bugs that are most difficult to identify, reproduce, and track down manifest themselves only rarely.

An example illustrates the efficacy of this technique. Our first experiment with random splitting applied it to the water jug problem popularized by the film *Die Hard: With a Vengeance*. Given two water jugs, one holding (say) exactly 3 gallons and the other holding (say) exactly 5 gallons, and neither of which has any calibrations, how can one fill, empty, and pour water from one jug to the other in order to leave exactly 4 gallons in one of the jugs? We expected to obtain properties about the insolubility of the problem when the two jugs have sizes that are not relatively prime. In addition, we learned that minimal-length solutions have either one step (the goal size is the size of one of the jugs) or an even number of steps (odd-numbered steps fill or empty a jug, and even-numbered steps pour as much of a jug as possible into the other jug). We were not aware of this property before using random splitting.

3.5 Context-sensitive dynamic analysis

By default, invariant detection is context-insensitive: its output properties hold over all executions of a program point. However, many procedures are used in different ways by different clients. A context-sensitive analysis distinguishes among different paths to a program point.

As one example, the `addElement` method of a a polymorphic container class such as `java.util.Vector` might be called with a `String` argument from one call-site and an `Integer` argument from another. Ordinary invariant detection would report obj.type $\in$ {String, Integer}. However, it would be preferable to instead note that `Vector` is always used as a homogeneous container.

It is also common to pass fixed values at a given call-site, especially with functions of many arguments. Often, the fixed values (whether statically constant, or dynamically unchanging) passed to a function determine its behavior. For example, the Unix `lseek` system call's `whence` parameter determines the arithmetic applied to the `offset` parameter. Splitting by caller enables detection of invariants over the offset calculation, such as:

whence = SEEK_SET ⇒ position = offset
whence = SEEK_CUR ⇒ position = orig(position) + offset
whence = SEEK_END ⇒ position = length − offset

As another example, in the Berkeley-sockets `accept` procedure, parameters `addr` and `addrlen` are either both NULL or both non-NULL. If both types of call are made, no invariant will be detected, but splitting by caller enables detection of addr = NULL ⇔ addrlen = NULL.

We extended invariant detection to perform splitting based on calling context by adding, to the data traces for procedure entries and exits, variables that indicate the call-site. The instrumenter of Figure 1 transforms programs to add an additional parameter to all procedures, to assign a unique identifier to each call-site, and to pass that identifier in the call. The invariant detector is supplied with splitting conditions over the new call-site parameters, similarly to those used by the clustering technique (Figure 5). The splitting conditions can specify calls from a single call-site, all calls from a single procedure, or all calls from a single class; the latter two are achieved by making splitting conditions that are disjunctions of those for single call-sites. Finally, a post-processing step after invariant detection converts predicates over the call-site ids into statements about context, such as changing "caller = 3152145" into "called from method M".

3.6 Other policies

Many other techniques could be used to divide the data into parts likely to exhibit different behaviors; we plan to investigate additional ones in the future. Two of the candidates are the following.

A *special values* policy compares a variable to preselected values chosen statically (such as null, zero, or literals in the source code) or dynamically (such as commonly-occurring values, minima, or maxima).

A policy based on *exceptions to detected invariants* tracks variable values that violate potential invariants, rather than immediately discarding the falsified invariant. If the number of falsifying samples is moderate, those samples can be separately processed, resulting in a nearly-true invariant plus an invariant over the exceptions.

Additionally, programmers can select splitting conditions by hand or according to any other strategy. (The Daikon manual specifies a file format for splitting conditions, and that format was used in all of our experiments.) Programmers usually have intuitions about what properties are most important, are most likely to be of interest, or are most relevant to their particular task. Additionally, programmers can supply extra information, such as which test case group a particular run came from, in order to compare such characteristics.

4. Evaluation

We evaluated Section 3's five policies for selecting splitting conditions — and thus, for dynamic detection of implications — in three different ways. The first experimental evaluation measured the accuracy of the invariants in a program verification task (Section 4.1). The second experimental evaluation measured how well the implications indicated and explained faulty behavior (Section 4.2). The third evaluation was qualitative and assessed the likely usefulness of the resulting implications (Section 4.3).

We report results for the two experiments in terms of *precision* and *recall*, standard measures from information retrieval. Suppose that we have a goal set of answers and a reported set of answers produced by some technique; the correct set is the intersection of the goal and reported sets. Precision, a measure of correctness, is defined as $\frac{\text{correct}}{\text{reported}}$. Recall, a measure of completeness, is defined as $\frac{\text{correct}}{\text{goal}}$. Both measures are always between 0 and 1.

4.1 Static checking

Our experiment with static checking used the ESC/Java static checker [Det96, DLNS98, LN98] to verify invariants detected by the Daikon invariant detector. ESC/Java statically detects null dereference errors, array bounds errors, and type cast errors. Programmers must write program annotations, many of which are similar in flavor to `assert` statements. (Daikon can automatically insert its output into programs in the form of ESC/Java annotations.) ESC/Java issues warnings about annotations that cannot be verified and about potential run-time errors.

Daikon's output may not be completely verifiable by ESC/Java. Verification may require removal of certain annotations that are not verifiable, either because they are not universally true or because they are beyond the checker's capabilities. Verification may also require addition of missing annotations, when those missing annotations are necessary for the correctness proof or for verification of other necessary annotations. We performed the removals and additions by hand in order to find the verifiable set of annotations that was closest to Daikon's output. (This task is far beyond the state of the art in program verification. Every other processing step performed for this research was automatic, with no human intervention required.) The number of changes to Daikon's output is a measure of its accuracy — in other words, how much work a human would have to perform in order to verify the lack of run-time errors in the code. When measuring precision and recall, the verifiable set is the goal set.

Use of splitting conditions adds implications to Daikon's output. Adding such invariants may increase recall by including a condition necessary for verification, but new invariants may also decrease precision by including invariants that are not true or are not verifiable. Both anecdotal evidence and controlled user experiments [NE02] have demonstrated that recall is more important than precision for users. Users can easily skip over or delete undesirable invariants but have more trouble producing annotations from scratch — particularly implications, which tend to be the most difficult invariants for users to write. Therefore, adding invariants may be worthwhile even if precision decreases.

We analyzed the programs listed in Figure 10. `DisjSets`, `StackAr`, and `QueueAr` come from a data structures textbook [Wei99]; `Vector` is part of the Java standard library; and the remaining programs are solutions to assignments in a programming course at MIT.

Figure 10 gives the experimental results. Over all the programs, the return value analysis produced the smallest number of implications, followed by static analysis, clustering, and random splitting respectively.

Previous user studies [NE02] had determined that the object invariants over `QueueAr` were particularly troublesome for users to state; furthermore, users were little troubled by imprecision (extraneous properties in Daikon's output). Clustering analysis was able to recover three of these twelve invariants, which more than counterbalanced its drop in precision. Random selection produced two, and static analysis produced one, of the twelve troublesome invariants that were missing from the output of the return value analysis. These invariants involved comparisons over variables that were never used together in the program text, and so were difficult for people or static analyses to directly relate. The dynamic techniques, which ignore program structure, were not hindered by the structure of the program text.

Despite these successes for the dynamic techniques, their overall recall was no better than that of the simple static analysis. However, their false positives (irrelevant splitting conditions or coincidental detected invariants) reduced their relative precision. This indicates that techniques to throttle their output may be necessary.

Static analysis produced relatively few valid splitting conditions, mostly because many of the conditions in the program called functions, used local variables, or otherwise were not valid predicates at procedure entries and exits.

One potential explanation of our results is that the total number of missing properties was relatively low even in the base case. That is, the return value analysis produced nearly all the invariants that ESC/Java required for verification. Therefore, there was little room for the other techniques to improve recall.

4.2 Error detection

Our experiment with error detection uses a novel methodology for helping to locate the semantic error that may underly program faults. The insight motivating the technique is that bugs induce different program behaviors; that is, a program behaves differently on erroneous runs than on correct runs. One such difference is that the erroneous run may exhibit a fault. Even if it does not, however, the program's data structures or control flow are affected by the error. Our

Program	Program size		None		Return		Static		Cluster		Random	
	LOC	NCNB	Prec.	Recall	Prec.	Recall	Prec.	Recall	Prec.	Recall	Prec.	Recall
FixedSizeSet	76	28	1.00	0.86	1.00	0.86	1.00	0.86	1.00	0.86	1.00	0.86
DisjSets	75	29	0.82	1.00	1.00	0.97	1.00	1.00	1.00	0.94	0.80	0.98
StackAr	114	50	1.00	0.90	1.00	1.00	0.95	1.00	0.78	1.00	0.95	1.00
QueueAr	116	56	0.92	0.71	0.98	0.78	0.89	0.84	0.62	0.89	0.77	0.91
Graph	180	99	0.80	1.00	0.80	1.00	0.80	1.00	0.80	1.00	0.80	1.00
GeoSegment	269	116	1.00	1.00	1.00	1.00	1.00	1.00	0.94	1.00	–	–
RatNum	276	139	0.93	1.00	0.91	1.00	1.00	1.00	0.72	1.00	0.50	1.00
StreetNumberSet	303	201	0.82	0.95	0.77	0.95	0.77	0.96	0.77	0.96	–	–
Vector	536	202	0.96	0.95	0.99	0.95	0.76	0.98	0.71	0.97	0.81	0.97
RatPoly	853	498	0.81	0.97	0.67	0.95	0.71	0.96	0.68	0.96	–	–
Total	4886	2451	0.91	0.93	0.91	0.94	0.89	0.96	0.80	0.96	0.80	0.96

Figure 10: Invariants detected by Daikon and verified by ESC/Java. "LOC" is the total lines of code. "NCNB" is the non-comment, non-blank lines of code. "Prec" is the precision of the reported invariants, the ratio of verifiable to verifiable plus unverifiable invariants. "Recall" is the recall of the reported invariants, the ratio of verifiable to verifiable plus missing.

goal is to capture those differences and present them to a user. The differences may lead programmers to the underlying errors, perhaps without even needing to understand the faults as deeply as would otherwise be required.

The methodology requires user cooperation. The user is presented with a set of automatically-generated implications that result from differing behaviors in the target program. We speculate that, if there are errors in the program, then some of the dynamically detected implications will reveal the errors. We have not yet experimentally verified this speculation. However, anecdotal results we observed while performing this study strongly support the supposition. In many cases, after examining the invariants but before looking at the code, we were able to correctly guess exactly what errors existed in the target program. If other programmers can be equally effective at interpreting the results, then they can benefit from this error localization technology.

There are two different scenarios in which a software engineer might use our tool to locate an error.

1. The user knows errors are present, has a test suite, and knows which test cases are fault-revealing. Daikon can produce invariants using, as a splitting condition, whether a test case is fault-revealing. The resulting conditional invariants capture the differences between faulty and non-faulty runs and explicate what data structures or variable values underly the faulty behavior. Daikon's generalization over multiple faulty runs spares the user from being distracted by specifics of any one test case and from personally examining many test cases.

2. The user knows errors exist but does not know which test cases expose them; or, the user does not know whether errors exist but wishes to be appraised of evidence of potential errors. In this situation, the user can still be presented with a list of implications in the hopes that some of them are the same as would have been detected in the first scenario and so will lead the user to the error. Anecdotal evidence indicates the technique is effective in both scenarios.

Our evaluation focuses on scenario 2 because it is more challenging and interesting and because existing solutions for it are less satisfactory than those for scenario 1. Our approach is to detect conditional invariants and present them to the user. The user examines the conditional invariants, some of which may be irrelevant to any errors and some of which may indicate the underlying causes of faults. We already know that imprecision tends to be little hindrance, so long as desirable results are present and are not completely swamped by irrelevant output.

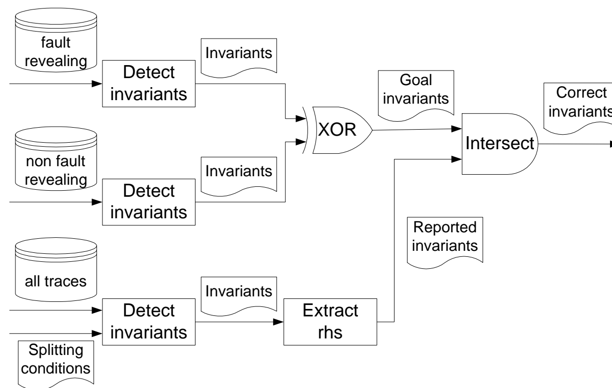

Figure 11: Error detection.

We evaluated our technique not via a user study, but analytically. Figure 11 diagrams our evaluation technique. The goal set of invariants is the ones that would have been created in the first scenario above, in which a user can split based on whether a test case is fault-revealing. We simulate this by running Daikon individually on the fault-revealing and non-fault-revealing tests. The goal is all invariants detected on one of those inputs but not on the other. The reported invariants are those resulting from running Daikon, augmented by a set of splitting conditions produced by one of the techniques of Section 3. Given the goal and reported sets, we compute precision and recall, as described at the beginning of Section 4.

We evaluated our technique over four different set of programs. Each set of programs was written to the same specification. Three of the sets came from the TopCoder programming competition website. These programs were submitted by contestants; the website publishes actual submissions and test cases after the match is complete. The three contests determined how football scores could be achieved, how elements could be distributed into bins, and how lines could be drawn to pass through certain points. We selected a total of 26 submissions which contained real errors made by contestants. The last set of programs were written by students in an undergraduate class at MIT (6.170 Laboratory in Software Engineering). We selected 20 assignments that implemented a datatype for polynomials with rational coefficients. These programs, too, contained real, naturally

Program	Program size			Return		Static		Cluster		Random	
	Ver.	LOC	NCNB	Prec.	Recall	Prec.	Recall	Prec.	Recall	Prec.	Recall
NFL	10	??	??	–	0.00	0.13	0.05	0.29	0.80	0.21	0.65
Contest	10	??	??	–	0.00	0.21	0.50	0.17	0.34	0.19	0.82
Azot	6	???	??	–	0.00	–	0.00	0.03	0.22	0.03	0.66
RatPoly	20	???	??	1.00	0.05	0.57	0.07	0.77	0.06	0.12	0.73
Total				1.00	0.01	0.30	0.15	0.32	0.36	0.14	0.72

Figure 12: Detection of invariants induced by program errors. "Ver" is the number of versions of the program. "LOC" is the average total lines of code in each version. "NCNB" is the average non-comment, non-blank lines of code. "Prec" is the precision of the reported invariants, the ratio of correct to reported. "Recall" is the recall of the reported invariants, the ratio of correct to goal.

occurring errors. The students had a week to complete their assignment, unlike the TopCoder competitors who were under time pressure. The TopCoder test suites tended to be more exhaustive and to better exercise boundary cases, because the contestants augmented them in an effort to disqualify their rivals.

[Note to the referees: the final version of this paper will also evaluate the popular Siemens programs, which contain manually seeded faults, and the space program, which contains real faults.]

Figure 12 summarizes the experimental results. The return value analysis rarely proposed any splitting conditions, so the "reported invariants" set was empty for all the contest problems. For the rational polynomial example, the return value analysis returned on average six fault-revealing invariants.

The static analysis strategy also reported relatively few invariants, compared to the other strategies.

Cluster analysis performed extraordinarily well on the NFL programs, but less well on the other programs. (For the RatPoly programs, clustering produced few additional invariants beyond the return value analysis.) This suggests that when the data can be naturally partitioned, clustering is very effective, but that it does less well when the data do not fall into easily separable groups. Clustering is likely to be a good complement to other techniques, though in some cases it produces no more than the return value analysis.

The random technique produced by far the largest set of splitting conditions and also the largest set of reported invariants — an order of magnitude more than the other approaches. For RatPoly, this translated to about two invariants per line of code, which is a large, but not unmanageable, number. Even though very many invariants were reported, about as many were fault-revealing as for the more parsimonious methods, so its precision remained acceptable. In other words, hundreds or thousands of invariants were reported, but so many of those were fault-revealing that a user need not examine many before being led to the cause of the error. In other words, the large amount of output should not be a hindrance. Furthermore, the many reported invariants permitted the random method to reveal more of the differences in behavior between erroneous and non-erroneous runs: in other words, more potential bugs were indicated by the output.

A frequent source of incorrect reported invariants was minor differences in results; for instance, the goal might state return ≥ 0 whereas one of the randomly selected subsets of the data induced return > 0, due to fewer samples.

4.3 Context-sensitive analysis

In contrast to the other techniques, which can report interesting properties even over small programs, the context-sensitive analysis of Section 3.5 requires a larger scope. For any benefit from this analysis, a method must have at least two callers, yet smaller test programs have typically fewer calls per method and less variation.

Therefore, we evaluated the context-sensitive analysis qualitatively rather than quantitatively, using `Rational` as a test case.

`Rational` is a library abstracting polynomials over the rational numbers. It consists of 2258 lines of code (977 lines of non-comment non-blank code) and defines classes: `RatNum`, rational numbers; `RatPoly`, polynomials; `RatTerm`, a coefficient and exponent; and `RatTermVec`, a vector of terms.

The invariants detected using context-sensitive analysis, and expressed here in OCL precondition/postcondition syntax, demonstrate defiances in the test suite and reveal properties of the implementation.

`RatNum.approx()`
 pre: caller = RatPoly.eval ⇒ this.denom = 1

During evaluation of a polynomial, the result is approximated by a floating-point value; however, `RatPoly.eval` only called `RatNum.approx` with integers, indicating that only fractionless polynomials were encountered while testing `eval` — an important omission from the test suite for such a procedure. This invariant did not show up earlier because a `RatNum` unit-test suite did exercise `approx` with a wide range of values.

`RatNum.div(RatNum arg)`
 pre: caller = RatPoly.divAndRem ⇒ arg.denom = 1
 pre: caller = RatPoly.divAndRem ⇒ arg.numer ≠ 0

`RatNum.mul(RatNum arg)`
 pre: caller = RatPoly.divAndRem ⇒ arg.numer ≥ 0
 pre: caller = RatPoly.divAndRem ⇒ this.denom ≥ 1
 post: caller = RatPoly.divAndRem ⇒ return.denom ≥ 1

These five invariants indicate further deficiencies in the test suite. Invariant #2 shows that zero-valued coefficients are never used by `div`, while invariants #4 and #5 show that `NaN` values, which are represented by a zero denominator, are never manipulated by `mul`.

Similarly, invariant #1 indicates that `div` is not called with rational-coefficient polynomials, and invariant #3 shows that `div` is never exercised with negative coefficients.

`RatNum.sub(RatNum arg)`
 pre: caller = RatNum.compareTo ⇒ arg.denom ≥ 1
 pre: caller = RatNum.compareTo ⇒ this.denom ≥ 1

This pair of invariants reveals that `compareTo` short-circuits (does not call `sub`) if its argument is `NaN`. .

`RatPoly.parse(String polyStr)`
 pre: caller = RatPoly.div ⇒ polyStr = "NaN"

This invariant indicates that `div` is implemented inefficiently, since it repeatedly calls the the pure (and relatively expensive) function `parse` in order to generate `NaN`-valued polynomials, instead of using a reference to a constant or caching the result.

`RatPoly.scaleCoeff(RatTermVec t, RatNum scalar)`

pre: caller = RatPoly.negate $\Rightarrow$ scalar.denom = 1
pre: caller = RatPoly.negate $\Rightarrow$ scalar.numer = -1

Finally, this pair of invariants reveal that `negate`, by invoking the general-purpose `scaleCoeff` with a value of -1, is being needlessly inefficient. This example is a candidate for partial specialization.

5. Related work

Clustering [JMF99] aims to partition data so as to reflect distinctions present in the underlying data. It is now widely used in software engineering as well as in other fields. As just one example of a related use, Podgurski et al [PMM+99] use clustering on execution profiles (similar to our data traces) to differentiate among operational executions. This can reduce the cost of testing. In related work, Dickinson et al [DLP01] use clustering to identify outliers; sampling those outlier region is effective at detecting failures.

The techniques for dynamic detection of implications and for error detection presented in this paper were previously proposed in a technical report and dissertation [EGKN99, Ern00]. Raz et al [RKS02] used the Daikon implementation (albeit without most of the implication techniques discussed in this paper) to detect anomalies in online data sources. Hangal and Lam [HL02] re-implemented the dynamic invariant detector (including improvements such as running online, but including some other restrictions in an engineering tradeoff) and showed that the techniques are effective at bug detection. The ideas were also implemented and evaluated by Engler et al [ECH+01], who detected numerous bugs in operating system code by exploiting the same underlying idea: that when behavior is inconsistent, then a bug is present, because one of the behaviors must be incorrect. An automated system can flag such inconsistencies even in the absence of a specification or other information that would indicate which of the behaviors is erroneous. Of course, none of the cited works can claim credit for the idea of comparing behavior to look for differences, which has long been applied by working programmers and others; but they have found effective ways to apply those ideas to the domain of error detection.

6. Conclusion

This paper suggests five methods for improving the quality of specifications that can be generated by automatic invariant detection.

Our experimental data confirm the value of these methods: all improve the recall of the reported invariants by several percent, with minimal loss in precision.

The data suggests that each technique represents a tradeoff between better recall (fewer missing invariants for a verifiable specification) and better precision (broadly speaking, lower "noise").

The static analysis and return methods give increased recall with little or no loss in precision, but, if recall is the dominant factor in user-acceptability, as we believe it is, then the use of the random and cluster methods are still of value despite their decreased precision.

The improved average recall, from around 93%–96%, represents approximately one third fewer missing invariants required to complete the specification by hand.

This paper also contributes an approach to error identification, even in the absence of test data evidencing the fault or even knowledge of the existence of the error.

In the future, we wish to further exploit invariants produced by context-sensitive splitting conditions, both to aid understanding of complex programs, and to improve the quality of generated specifications.

Acknowledgments

James Anderson, Yuriy Brun, Michael Harder, and Jeremy Nimmer gave significant help with the experimental evaluation. This research was supported in part by NSF grants CCR-9970985 and CCR-6891317.

References

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

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, December 18, 1998.

[DLP01] William Dickinson, David Leon, and Andy Podgurski. Pursuing failure: the distribution of program failures in a profile space. In *ESEC/FSE*, pages 246–255, Vienna, Austria, 2001.

[ECGN00] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *ICSE*, pages 449–458, June 2000.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):1–25, February 2001. A previous version appeared in *ICSE*, pages 213–224, Los Angeles, CA, USA, May 1999.

[ECH+01] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 57–72, Banff, Alberta, Canada, 2001.

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

[Ern00] Michael D. Ernst. *Dynamically discovering likely program invariants*. PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[GLV97] Stephen J. Garland, Nancy A. Lynch, and Mandana Vaziri. IOA: A language for specifying, programming, and validating distributed systems. Technical report, MIT Laboratory for Computer Science, 1997.

[HL02] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, May 2002.

[JMF99] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, September 1999.

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

[MR97] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1997.

[NE02] Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking: an empirical evaluation, March 2002.

[PMM+99] Andy Podgurski, Wassim Masri, Yolanda McCleese, Francis G. Wolff, and Charles Yang. Estimation of software reliability by stratified sampling. *ACM TOSEM*, 8(3):263–28883, July 1999.

[RKS02] Orna Raz, Philip Koopman, and Mary Shaw. Semantic anomaly detection in online data sources. In *ICSE*, May 2002.

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