Selecting, Refining, and Evaluating Predicates for Program Analysis

Nii Dodoo Lee Lin Michael D. Ernst

Technical Report MIT-LCS-TR-914
July 21, 2003

MIT Lab for Computer Science
200 Technology Square
Cambridge, MA 02139 USA
{dodoo,leelin,mernst}@lcs.mit.edu

Abstract

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

Trying every possible predicate for conditional properties is computationally infeasible and yields too many undesirable properties. This paper compares four policies for selecting predicates: procedure return analysis, code conditionals, clustering, and random selection. It also shows how to improve predicates via iterated analysis. An experimental evaluation demonstrates that the techniques improve performance on two tasks: statically proving the absence of run-time errors with a theorem-prover, and separating faulty from correct executions of erroneous programs.

1 Introduction

The goal of program analysis is to determine facts about a program. The facts are presented to a user, depended on by a transformation, or used to aid another analysis. The properties frequently take the form of logical formulae that are true at a particular program point or points.

The usefulness of a program analysis depends on what properties it can report. 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 program analysis to include implications of the form $a \Rightarrow b$. Disjunctions such as $a \vee b$ are a special case of implications, since $(a \Rightarrow b) \equiv (\neg a \vee b)$. Our implementation and experimental evaluation are for a specific dynamic program analysis that, given program executions, produces likely invariants as output. The base analysis reports properties such as preconditions, postconditions, and object invariants that are unconditionally true over a test suite. (Section 2.3 describes the technique.)

A conditional property is one whose consequent is not universally true, but is true when the predicate is true. (Equivalently, the consequent is false only when the predicate is false.) For instance, the local invariant over a node n of a sorted binary tree, $(n.left.value \leq n.value) \wedge (n.right.value \geq n.value)$, is true unless one of *n*, *n.left*, or *n.right* is null. Conditional properties are particularly important in recursive data structures, where different properties typically hold in the base case and the recursive case. The predicates are also useful in other domains. For instance, it can be challenging to select predicates for predicate abstraction [BMMR01]. A related context is determining whether to discard information at join points in an abstract interpretation such as a dataflow analysis.

Extending an analysis to check implications is trivial. However, it is infeasible for a dynamic analysis to check $a \Rightarrow b$ for all properties a and b that the base analysis can produce. One reason is runtime cost: the change squares the number of potential properties 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 by a factor of 100. This is acceptable only if the number of true positives is also increased by a factor of 100, which is unlikely. False positives include properties that are true over the inputs but are not true in general. In the context of interaction with humans, false positives also include true properties that are not useful for the user's current task.

Since it is infeasible to check $a \Rightarrow b$ for every a and b, the program 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. We use *splitting conditions* to partition the data under analysis, and then combine separate analysis results to create implications or conditional properties. Splitting conditions limit the predicates that are considered, but predicates that are not splitting conditions may still appear. We also present a technique that leverages the base analysis to refine imprecise predicates via an iterated analysis.

This paper presents four policies (detailed in Section 3) for selecting predicates for implications: procedure return analysis; code conditionals; clustering; and random selection. The last two, which performed best in our experimental evaluation, are dynamic analyses that examine program executions rather than program text; the second one is static; and the first is a hybrid. Dynamic analyses can produce information (predicates) about program behavior that is not apparent from the program text — for instance, general alias analysis remains beyond the state of the art, but runtime behavior is easy to observe. Also, the internal structure of the source code does not effect the dynamic policies. It also enables them to work on programs for which source code is not available, so long as the underlying program analysis does not require source code.

We evaluated the four policies in two different ways. First, we compared the accuracy of the produced properties, where accuracy is measured by a program verification task (Section 4.1); the policies produced implications that reduced human effort by 40%. Second, we determined how well each of the policy choices exposes errors (Section 4.2); 12% of the implications directly reflected differences due to faulty behavior, even without foreknowledge of the faults.

The remainder of this paper is organized as follows. Section 2 proposes mechanisms for detecting and refining implications. Section 3 describes the four policies that determine which implications will be computed, and Section 4 evaluates them. Section 5 discusses related work, and Section 6 recaps our contributions.

2 Detecting implications

Figure 1 shows the mechanism for creation of implications. Rather than directly testing specific $a \Rightarrow b$ invariants, the analysis splits the input data into two mutually exhaustive parts based on a client-supplied predicate, which we call a *splitting condition*. The splitting conditions are not necessarily the same as the implication predicates (see Section 2.1). This paper focuses on automating the selection of splitting conditions, which are analogous to the predicates of predicate abstraction.

After the data is split into two parts, the base program analysis is performed to detect (non-implication) properties in each subset of the data. Finally, implications are generated from the separately-computed properties, if possible.

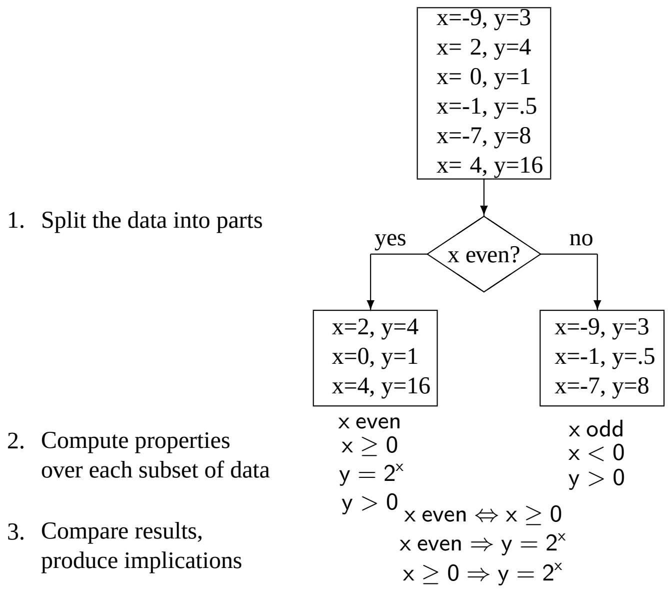
x=-9, y=3
x= 2, y=4
x= 0, y=1
x=-1, y=.5
x=-7, y=8
x= 4, y=16
1. Split the data into parts
yes
x even?
no
x=2, y=4
x=0, y=1
x=4, y=16
x=-9, y=3
x=-1, y=.5
x=-7, y=8
2. Compute properties over each subset of data
x even
$x \geq 0$
$y = 2^x$
$y > 0$
x odd
$x < 0$
$y > 0$
3. Compare results, produce implications
x even $\Leftrightarrow x \geq 0$
x even $\Rightarrow y = 2^x$
$x \geq 0 \Rightarrow y = 2^x$

Figure 1. Mechanism for creation of implications. In the figure, the analysis is a dynamic one that operates over program traces. Figure 2 gives the details of the third step.

```
// $S_1$ and $S_2$ are sets of properties resulting from
// analysis of partitions of the data.
procedure CREATE-IMPLICATIONS($S_1$, $S_2$)
  for all $p_1 \in S_1$ do
    if $\exists p_2 \in S_2$ such that $p_1 \Rightarrow \neg p_2$ and $p_2 \Rightarrow \neg p_1$ then
      // $p_1$ and $p_2$ are mutually exclusive
      for all $p' \in (S_1 - S_2 - \{p_1\})$ do
        output "$p_1 \Rightarrow p'$"
```

Figure 2. Pseudocode for creation of implications from properties over partitions of the data. (In our experiments, the underlying data is be partitioned into two sets of executions; other analyses might partition paths or other code artifacts.) Figure 3 shows an example of the procedure's input and output.

If the splitting condition is poorly chosen, or if no implications hold over the data, then the same properties are computed over each subset of the data, and no implications are reported.

Figure 2 gives pseudocode for creation of implications from properties over subsets of the data, which is the third step of Figure 1. The CREATE-IMPLICATIONS routine is run twice, swapping the arguments, and then the results are simplified according to the rules of Boolean logic. Figures 1 and 3 give concrete examples of the algorithm's behavior.

Each mutually exclusive property implies everything else true for its own subset of the data. (This is true only because the two subsets are mutually exhaustive. For instance, given a mechanism that generates three data subsets inducing property sets $\{a, b\}$, $\{\neg a, \neg b\}$, $\{a, \neg b\}$, it is

Properties		Implications		Simplified
S_1	S_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 3. Creation of implications from properties over subsets of the data. The left portion of the figure shows S_1 and S_2, sets of properties over two subsets of the data; these subsets resulted from some splitting condition, which is not shown. The middle portion shows all implications that are output by two calls to the CREATE-IMPLICATIONS routine of Figure 2; c appears unconditionally, so does not appear in any implication. The right portion shows the implications after logical simplification.

not valid to examine only the first two subsets of the data and to conclude that $a \Rightarrow b$.) The algorithm does not report self-implications or 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 "$a \Rightarrow c$" in addition to "c".

2.1 Splitting conditions and predicates

The left-hand-sides of implications resulting from the above procedure may differ from the splitting conditions used to create the subsets of the data. Some splitting conditions may not be left-hand-sides, and some non-splitting-conditions may become left-hand-sides.

Any properties detected in the subsets of the data — not just splitting conditions — may appear as implication predicates; for example, $x \geq 0 \Rightarrow y = 2^x$ in Figure 1. This is advantageous when the splitting condition is not reported (for instance, is not expressible in the underlying analysis); it permits strengthening or refining the splitting condition into a simpler or more exact predicate; and it enables reporting more implications than if predicates were limited to pre-specified splitting conditions.

In practice, the splitting condition does appear as a left-hand-side, because it is guaranteed to be true of one subset of the data (and likewise for its negation). However, there are three reasons that the splitting condition (or its negation) might not be reported in a subset of the data.

1. The splitting condition may be inexpressible in the analysis tool's output grammar. For example, the Daikon invariant detector (Section 2.3), which we used in our experimental evaluation, allows as a splitting condition any Java boolean expression, including program-specific declarations and calls to libraries. The Daikon implementation permits inexpressible splitting conditions to be passed through as implication predicates. However, those inexpressible invariants are usually beyond the capabilities of other tools such as static checkers (see Section 4.1), so we omit them in our experiments.
2. A stronger condition may be detected; the weaker, implied property need not be reported.
3. The splitting condition may not be statistically justified. A dynamic or stochastic analysis may use statistical tests to avoid overfitting based on too little input data. This can occur, for example, if one of the subsets is very small. Such statistical suppression is relatively infrequent, prevents many false positives, and rarely reduces the usefulness of the result [ECGN00].

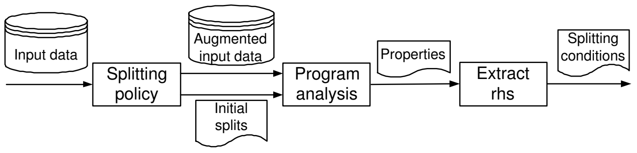

Figure 4. Producing splitting conditions via iterated analysis. The final splitting conditions make no mention of augmentations, if any, to the data traces. The final splitting conditions are used as shown in Figure 6.

2.2 Refining splitting conditions

A splitting policy may propose a good but imperfect partition of the data that does not precisely match true differences of behavior. Or, a splitting policy may use a (possibly costly) external analysis or other information that is not typically available in the program trace or is hard for people to understand. We propose a two-pass process that performs program analysis twice — the first time to produce a refined set of splitting conditions, and the second time to output implications — to correct both problems. The refined splitting conditions are the right-hand-sides (consequents) of the implications discovered in the first pass; see Figure 4. This approach has two benefits.

First, the two-pass process produces a set of splitting conditions in terms of program quantities. They are human-readable, easing inspection and editing, and they can be reused during other analysis steps.

Second, the first program analysis pass helps to refine the initial splitting conditions. Statistical or inexact techniques may not partition the data exactly as desired. However, as long as at least one subset induces one of the desired properties, the first program analysis pass can leverage this into the desired splitting condition. If the original splitting condition produces the desired grouping, then the additional pass does no harm.

As a simple example, consider Figure 5. Suppose that the $\triangle$ points have properties $numSides = 3$ and $x < 0$, and

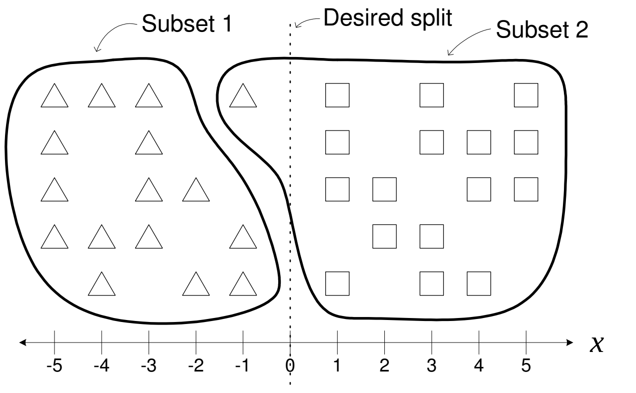

Figure 5. Refinement of initial splits via an extra step of program analysis. The initial subsets approximate, but do not exactly match, the natural division in the data (between triangles and squares, at $x = 0$). An extra program analysis step produces the desired splitting condition.

the □ points have properties $numSides = 4$ and $x > 0$. The initial splitting condition (displayed as two subsets) nearly, but not exactly, matches the true separation between behaviors. The first pass, using the subsets as the splitting condition, would produce "$subset = 1 \Rightarrow numSides = 3$" and "$subset = 1 \Rightarrow x < 0$". The refined splitting conditions are $numSides = 3$ and $x < 0$. The second program analysis pass yields the desired properties: "$(x > 0) \Rightarrow (numSides = 4)$" and "$x < 0 \Rightarrow (numSides = 3)$". The clustering policy (Section 3.3) uses this two-pass strategy directly, and the random selection policy (Section 3.4) relies on a similar refinement of an imperfect initial data subsetting.

2.3 Background: Dynamic invariant detection

This section briefly describes dynamic detection of likely program invariants [Ern00, ECGN01], which we use in our experimental evaluation of predicate selection. The techniques of this paper should be applicable to other static and dynamic program analyses as well. The experiments in this paper use the Daikon implementation, which reports representation invariants and procedure preconditions and postconditions. The techniques of this paper for producing implications are publicly available in the Daikon distribution (`http://pag.lcs.mit.edu/daikon`) and have been successfully used for several years.

We only briefly explain dynamic detection of likely invariants — enough to appreciate the experiments — but full details may be found elsewhere [ECGN00, Ern00, ECGN01].

Dynamic invariant detection discovers likely invariants from program executions by instrumenting[1] 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 6).

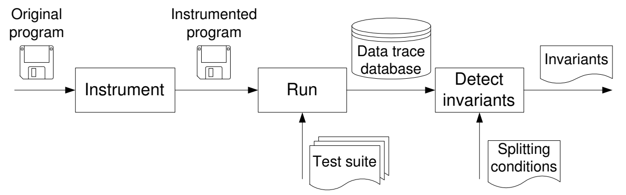

Figure 6. Architecture of the Daikon tool for dynamic invariant detection. The "splitting conditions" input is optional and enables detection of implications of the form "$a \Rightarrow b$" (see Section 2). This paper proposes and evaluates techniques for selecting splitting conditions.

The inference step creates many potential invariants (essentially, all instantiations of a set of templates), then tests each one against the variable values captured at the instrumentation points. 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.

The invariant templates include about three dozen properties over scalar variables (e.g., $\mathsf{x} \leq \mathsf{y}$, $\mathsf{z} = \mathsf{ax} + \mathsf{by} + \mathsf{c}$) and collections (e.g., mylist is sorted, $\mathsf{x} \in \mathsf{myset}$). Statistical and other techniques further improve the system's performance. 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.

3 Policies for selecting predicates

We have reduced the problem of detecting predicates to that of selecting (approximate) splitting conditions. This section describes four policies for detecting splitting conditions that we experimentally evaluated: procedure return analysis, code conditionals, clustering, and random selection.

3.1 Procedure return analysis

This section describes a simple splitting policy based on two dynamic checks of procedure returns. The first check

[1]The instrumentation may be over source code, over object code, or performed by a run-time system. For example, of the six Java front ends for Daikon of which we are aware, two fall into each category. The experiments reported in this paper used source code instrumentation, and required no modification or enhancement of the instrumenter.

splits data based on the return site. If a procedure has multiple `return` statements, then it is likely that they exhibit 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. The second check splits data based on boolean return values, separating cases for which a procedure returns true from those for which it returns false.

3.2 Static analysis for code conditionals

The code conditional policy is a simple static analysis that selects each boolean condition used in the program (as the test of a `if`, `while`, or `for` statement, or as the body of a pure boolean member function) as a splitting condition.

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 domain. Furthermore, if a test can affect the implementation, then that condition may also affect the externally visible behavior.

The Daikon implementation permits splitting conditions to be associated with a single program point (such as a procedure entry or exit) or to be used at all program points that contain variables of the same name and type. Our experiments use the latter option. For instance, a condition might always be relevant to the program's state, but might only be statically checked in one routine. Other splitting policies also benefit from such cross-fertilization of program points.

3.3 Clustering

Cluster analysis, or clustering [JMF99], 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). Data splitting shares these same goals.

As described in Section 2.2 and illustrated in Figure 4, the clustering policy uses a two-pass algorithm to refine its inherently approximate results.

Clustering operates on points in an n-dimensional space. Each point is a single program point execution (such as a procedure entry), and each dimension represents a scalar variable in scope at that program point. We applied clustering to each program point individually. 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 hash codes or floating-point values) do not swamp smaller differences in other attributes (such as booleans).

The experiments reported in this paper use x-means clustering [PM00], which automatically selects an appropriate number of clusters. We repeated the experiments with k-means and hierarchical clustering techniques; the results differed little [Dod02].

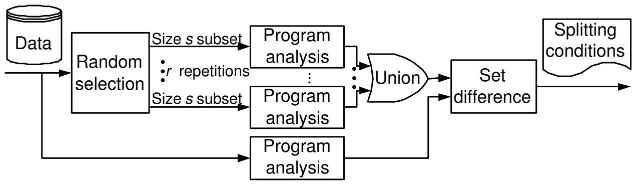

Figure 7. The randomized algorithm for choosing splitting conditions. The technique outputs each invariant that is detected over a randomly-chosen subset of the data, but is not detected over the whole data. The "detect invariants" steps are non-conditional invariant detection.

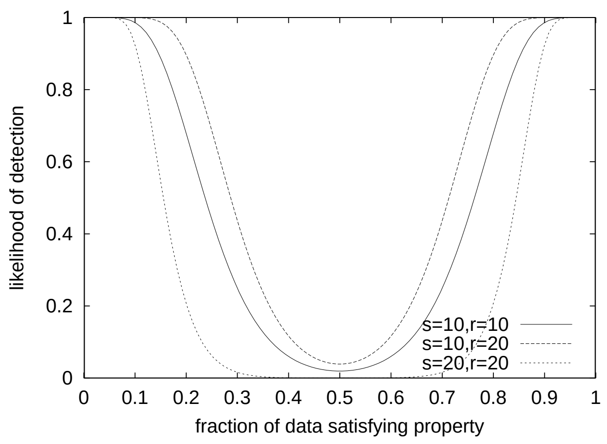

Figure 8. Likelihood of finding an arbitrary split via random selection. s is the size of each randomly-chosen subset, and r is the number of such subsets.

3.4 Random selection

Figure 7 shows a randomized analysis for selecting splitting conditions. First, select r different subsets of the data, each of size s, and perform program analysis over each subset. Then, use any property 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 perfectly separate the data. Suppose that some property holds in a fraction $f < 1$ of the data. It will never be detected by non-conditional program analysis. 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 program analysis).

Figure 8 shows how likely a property is to be detected by this technique, for several values of s and r. The property holds in all s elements of some subset with probability $p_1 = f^s$. Thus, the property is detected with probability p_1 on each trial for a given subset of size s. The negation holds with probability $p_2 = (1-f)^s$. The property or its negation holds on at least one of the subsets with probability $1-(1-p_1)^r + 1-(1-p_2)^r$, which is graphed in Figure 8.

The random selection technique is most effective for unbalanced data; 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 errors that are most difficult to identify, reproduce, and track down manifest themselves only rarely.

The likelihood of detecting a (conditional) property can be improved by increasing r or by reducing s. The danger of increasing r is that work linearly increases with r. The danger of reducing s is that the smaller the subset, the more likely that any resulting properties overfit the small sample. (By contrast, increasing s makes it less likely that a property holds over an entire size-s subset.) We chose $s = 10$ and $r = 20$ for our experiments; informal experimentation and graphs such as Figure 8 had suggested that these values were reasonable. We did not attempt to optimize them, so other values may perform even better, or may be better in specific domains.

An example illustrates the efficacy of this technique. Our first experiment with random splitting applied it to the well-known water jug problem. 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 you fill, empty, and pour water from one jug to the other in order to leave exactly 4 gallons in one of the jugs? We hoped 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 non-obvious property before using random splitting.

4 Evaluation

We evaluated Section 3's four policies for selecting splitting conditions — and thus, for computing implications — in two different ways. The first experimental evaluation measured the accuracy of program analysis results for a program verification task (Section 4.1). The second experimental evaluation measured how well the implications indicated faulty behavior (Section 4.2).

Both tasks are instances of information retrieval [Sal68, vR79], so we compute the standard *precision* and *recall* measures. Suppose that we have a goal set of results and a reported set of results; then the matching set is the intersection of the goal and reported sets. Precision, a measure of correctness, is defined as $\frac{|\text{matching}|}{|\text{reported}|}$. Recall, a measure of completeness, is defined as $\frac{|\text{matching}|}{|\text{goal}|}$. Both measures are always between 0 and 1, inclusive.

Implications that are not in the matching set are still correct statements about the program's conditional behavior; these properties simply do not relate to the goal set. Each task induces a different goal set, to which different properties are relevant, so some imprecision is inevitable.

4.1 Static checking

The static checking experiment measures how much a programmer's task in verifying a program is eased by the availability of conditional properties. Adding implications reduced human effort by 40% on average, and in some cases eliminated it entirely.

The programmer task is to change a given (automatically generated) set of program properties so that it is self-consistent and guarantees lack of null pointer dereferences, array bounds overruns, and type cast errors. The amount of change is a measure of human effort.

The experiment uses the ESC/Java static checker [FLL$^+$02] to verify lack of runtime errors. ESC/Java issues warnings about potential run-time errors and about annotations that cannot be verified. Like the Houdini annotation assistant [FL01], Daikon can automatically insert its output into programs in the form of ESC/Java annotations, which are similar in flavor to `assert` statements.

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. People find eliminating undesirable annotations easy but adding new ones hard (see Section 4.1.1). Therefore, the complement of recall — the number of missing properties — is the best measure of how much work a human would have to perform in order to verify the lack of run-time errors in the code.

We analyzed the Java programs listed in Figure 9. `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. Each program verification attempt included client code (not counted in the size measures of Figure 9) to ensure that the verified properties also satisfied their intended specification. For each program and each set of initial annotations, we chose a goal set by hand [NE02a]. There is no unique verifiable

	Program size		No implications		Return		Static		Cluster		Random	
Program	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	0.73	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	0.83	0.89
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	0.79	0.95
Total	4886	2451	0.91	0.93	0.91	0.95	0.89	0.96	0.80	0.96	0.80	0.96
Missing			0.09	**0.07**	0.09	**0.05**	0.11	**0.04**	0.20	**0.04**	0.20	**0.04**

Figure 9. Invariants detected by Daikon and verified by ESC/Java, using four policies for selecting splitting conditions (or using none). "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. "Missing" indicates the overall missing precision or recall. The most important measure is the missing recall (in bold): it is the most accurate measure of human effort, since it indicates how much humans must add to the reported set. By comparison, removing elements from the set — measured by the complement of precision — is an easy task.

set of annotations, so we chose a verifiable set that we believed to be closest to the initial annotations (that is, the analysis output). This gives (an upper bound on) the minimum amount of work a programmer must do, and is a good approximation of the actual work a programmer would do.

Figure 9 gives the experimental results. Return value analysis produced the fewest implications, followed by code conditionals, clustering, and random splitting.

As indicated in the "No implications" column of Figure 9, even when supplied with no splitting conditions, dynamic invariant detection performs well at this task. (All results were obtained without adding or removing invariants or otherwise tuning Daikon to the particular set of programs or to the ESC/Java checker; however, the results suggest that Daikon is well-matched to ESC/Java's strengths.) 91% of the reported properties are verifiable; the other 9% are true, but their verification either depends on missing properties or is beyond the capabilities of ESC/Java. Furthermore, 93% of all properties necessary for verification are already present; for 4 of the programs, no properties at all need to be added. Therefore, there is little room for improvement. Nonetheless, adding implications reduced the fraction of missing properties by 40% on average (from .07 to .04), and by up to 100%.

As a specific example, consider the `QueueAr` program [Wei99]. The data structure is an array-based Java implementation of a queue. Its fields are:

```
Object[] theArray;
int front;         // index of head element
int back;          // index of tail element
int currentSize;   // number of valid elements
```

After calling enough enqueue commands on a `QueueAr` object, the back pointer wraps around to the front of the array. The front pointer does the same after enough dequeue commands.

In a user study [NE02b], no users succeeded in writing correct `QueueAr` object invariants in one hour, despite being given a textbook's description of the implementation, complete with figures [Wei99]. The most troublesome annotations for users were implications, suggesting that machine help is appropriate for suggesting them.

As indicated in Figure 9, invariant detection without implications found 71% of the necessary annotations. The missing annotations included the following, among other similar properties. (For brevity, let $\mathit{size} = \texttt{currentSize}$ and $\mathit{len} = \texttt{theArray.length}$.)

Properties when the queue is empty

For example,

$$((\mathit{size} = 0) \wedge (\texttt{front} > \texttt{back})) \Rightarrow (\texttt{back} = \texttt{front}{-}1))$$

$$(\mathit{size} = 0) \Rightarrow \underset{0 \le i < \mathit{len}}{\forall i} (\texttt{theArray}[i] = \texttt{null})$$

The actual ESC/Java annotation inserted by Daikon in the second case is as follows; for brevity, we will generally present logical formulae instead.

```
/*@ invariant
   (currentSize == 0)
   ==> (\forall int i;
        (0 <= i && i <= theArray.length-1)
        ==> (theArray[i] == null));       */
```

Properties when the concrete rep is wrapped

For example,

$$((\mathit{size} > 0) \ \wedge \ (\texttt{front} \le \texttt{back}))$$

$$\Rightarrow (size = \texttt{back} - \texttt{front} + 1)$$

$$((size > 0) \ \wedge \ (\texttt{front} > \texttt{back}) \\ \Rightarrow (size = len + \texttt{back} - \texttt{front} + 1)$$

Properties over valid/invalid elements These refer to the array locations logically between the `front` and `back` indices, or between the `back` and `front` indices. For example,

$$((size > 0) \ \wedge \ (\texttt{front} \leq \texttt{back})) \\ \Rightarrow \underset{0 \leq i < \texttt{front}}{\forall i} (\texttt{theArray}[i] = \texttt{null})$$

$$((size > 0) \ \wedge \ (\texttt{front} \leq \texttt{back})) \\ \Rightarrow \underset{\texttt{back} < i < len}{\forall i} (\texttt{theArray}[i] = \texttt{null})$$

The random and clustering policies found properties from all three categories. By contrast, the code conditionals policy only found the properties in the first category. See Section 4.3.

4.1.1 Recall vs. precision

Adding implications to the set of properties presented to the static verifier ESC/Java improved the recall of the properties (as measured against a verifiable set) but reduced precision. The drop in precision is due to over-fitting: partitioning the data leads to more false positives in each subset, particularly since the test suites were quite small [NE02a].

There are two reasons that reduced precision is not a problem in practice. First, recall is more important to people performing the program verification task. Users can easily recognize and eliminate undesirable properties [NE02b], but they have more trouble producing annotations from scratch — particularly implications, which tend to be the most difficult invariants for users to write. Therefore, adding properties may be worthwhile even if precision decreases. Two of the implications that decreased the clustering technique's precision on the `QueueAr` program are:

$$(size \geq 2) \Rightarrow \texttt{theArray}[\texttt{back} - 1] \neq \texttt{null} \\ (\texttt{front} > \texttt{back}) \Rightarrow (size \neq \texttt{back})$$

Humans with modest familiarity with the `QueueAr` implementation quickly skipped over these as candidates for aiding the ESC/Java verification.

There is a second reason that lowered precision due to implications is not a concern: a human can easily augment the test suite to improve the precision. We augmented three of the test suites, taking less than one hour for each (and less than the verification time). The invariant detector and static verifier output, which indicated which properties had been induced from too little data (such as the ones immediately above), made it obvious how to improve the test suites. Figure 10 shows the accuracy (with respect to the program verification task) of cluster-based splitting with both the original and the augmented test suites. Augmentation increased precision from .71 to .93 and, unexpectedly, increased recall from .96 to .98. The final recall is a substantial reduction to 2% missing annotations, down from 13% missing annotations when no implications were present: users need to add less than one sixth as many annotations. Thus, results in practice (with more reasonable test suites, or with modest human effort to improve poor ones) may be even better than indicated by Figure 9.

	No implications		Cluster		Augmented	
Program	Prec.	Recall	Prec.	Recall	Prec.	Recall
StackAr	1.00	0.90	0.78	1.00	1.00	1.00
QueueAr	0.92	0.71	0.62	0.89	0.91	0.93
RatNum	0.93	1.00	0.72	1.00	0.88	1.00
Total	0.95	0.87	0.71	0.96	0.93	0.98
Missing	0.05	**0.13**	0.29	**0.04**	0.07	**0.02**

Figure 10. Augmenting test suites to improve precision for static verification. The table layout is as in Figure 9.

4.2 Error detection

Our experiment with error detection evaluates implications based on a methodology for helping to locate program errors. Errors 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 (a deviation from desired behavior). Even if no fault occurs, the error affects the program's data structures or control flow. Our goal is to capture those differences and present them to a user. As also observed by other authors [DDLE02, HL02, RKS02, GV03, PRKR03], the differences may lead programmers to the underlying errors. This is true even if users do no initially know which of two behaviors is the correct one and which is erroneous: that distinction is easy to make.

There are two different scenarios in which our tool might help to locate an error [DDLE02].

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

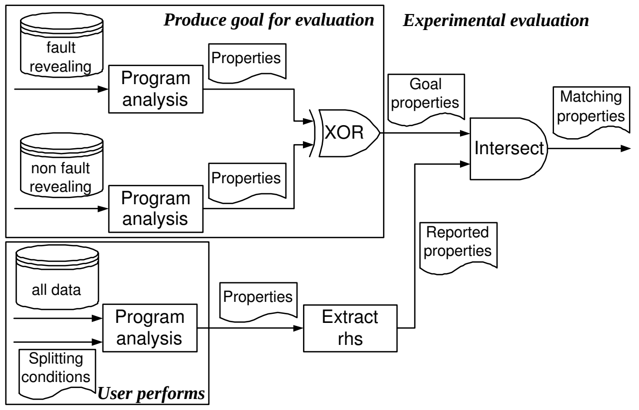

Figure 11. Evaluation of predicates for separating faulty and non-faulty runs. The goal set is the properties that discriminate between faulty and non-faulty runs. The reported set is the consequents of conjectured implications. The intersection between the goal and reported sets is the matching set.

test cases.

Scenario 2. The user knows errors exist but cannot reproduce and/or detect faults (and perhaps does not even know which test cases might yield faults). Alternately, the user does not know whether errors exist but wishes to be appraised of evidence of potential errors, for instance to further investigate critical code or code that has been flagged as suspicious by a human or another tool.

In scenario 2, we propose to present the user 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 implications will reveal the differing behavior, perhaps leading the user to the error. (Other implications will be irrelevant to any errors, even if they are true and useful for other tasks.) Anecdotal results support the speculation. In many cases, after examining the invariants but before looking at the code, we were able to correctly guess the errors. It might be interesting to test the speculation via a user study. Such a study is beyond the scope of this paper, and it is not germane to this section's main point of testing whether our technique is able to identify noteworthy differences in behavior. There are many potential uses for conditional properties other than debugging.

Our evaluation focuses on scenario 2, because existing solutions for it are less satisfactory than those for scenario 1.

Figure 11 diagrams our evaluation technique. The goal set of properties is the ones that would have been created in scenario 1, in which the data is partitioned based on whether a test case is fault-revealing. We simulate this by detecting properties individually on the fault-revealing and non-fault-revealing tests. The goal set contains all properties detected on one of those inputs but not on the other.

As a simple example, one `NFL` program contained a recursion error, causing faulty output whenever the input was not in the base case. The goal properties for this program distinguished between the base case and the recursive case (and the faulty program also had fewer invariants overall than correct ones did [ECGN01]). As a second example, one `replace` program failed to warn about illegal uses of the hyphen in regular expressions. On erroneous runs, properties over the replacement routine were different, including relationships between the pre- and post-values of indices into the pattern.

The reported properties are those resulting from running the program analysis, augmented by a set of splitting conditions produced by one of the policies in 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 eleven different sets of programs totaling over 137,000 lines of code. Each set of programs was written to the same specification. The `NFL`, `Contest`, and `Azot` programs came from the TopCoder programming competition website (`www.topcoder.com`). These programs were submitted by contestants; the website published actual submissions and test cases. The three programs 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 26 submissions at random that contained real errors made by contestants. The TopCoder test suites are relatively complete, because the contestants augmented them in an effort to disqualify their rivals. About half of the TopCoder programs contained one error, and the others contained multiple errors within the same function. The `RatPoly` and `CompostiteRoute` programs were written by students in an undergraduate class at MIT (6.170 Laboratory in Software Engineering). Students implemented a datatype for polynomials with rational coefficients and a set of algorithms for modeling geographical points and paths. We selected all student submissions that compiled successfully and failed at least one staff test case. These programs, too, contained real, naturally occurring errors. The students had a week to complete their assignment, unlike the TopCoder competitors who were under time pressure. Most of the student programs contained multiple, distinct errors in unrelated sections of the code. The remaining programs were supplied by researchers from Siemens [HFGO94, RH98] and are commonly used in testing research. Every faulty version of the Siemens programs has exactly one distinct error. The errors were seeded by humans, who chose them to be realistic. The `print_tokens` and `print_tokens2` programs (and their errors) are unrelated.

Figure 12 summarizes the results of the error detection experiment. The results indicate that regardless of the split-

		Program size			Return		Static		Cluster		Random	
Program	Source	Ver.	LOC	NCNB	Prec.	Recall	Prec.	Recall	Prec.	Recall	Prec.	Recall
NFL	TC	10	23	21	0.00	0.00	0.03	0.08	0.03	0.08	0.09	0.37
Contest	TC	10	21	17	0.00	0.00	0.19	0.40	0.11	0.23	0.15	0.21
Azot	TC	6	18	17	0.00	0.00	0.00	0.00	0.13	0.46	0.12	0.15
RatPoly	MIT	32	853	498	0.03	0.00	0.03	0.01	0.07	0.03	0.14	0.09
CompostiteRoute	MIT	67	883	319	0.22	0.09	0.22	0.09	0.21	0.47	0.21	0.45
print_tokens	S	7	703	452	0.00	0.00	0.03	0.13	0.04	0.22	0.04	0.34
print_tokens2	S	10	549	379	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
replace	S	32	506	456	0.24	0.31	0.14	0.28	0.10	0.42	0.15	0.37
schedule2	S	9	369	280	0.00	0.00	0.20	0.35	0.20	0.35	0.24	0.41
tcas	S	41	178	136	0.17	0.24	0.19	0.31	0.12	0.23	0.13	0.40
tot_info	S	23	556	334	0.00	0.00	0.12	0.27	0.09	0.40	0.03	0.09
Total		247	137015	75115	**0.06**	0.06	**0.10**	0.18	**0.10**	0.26	**0.12**	0.26

Figure 12. Detection of conditional behavior induced by program errors, compared for four splitting policies. (When no splitting is in effect, the precision and recall are always zero.) "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 matching to reported. "Recall" is the recall of the reported invariants, the ratio of matching to goal. In this experiment, precision (in bold) is the most important measure: it indicates how many of the reported implications indicate erroneous behavior. In cases where precision is 0.00, the experiment did report some implications, but none of them contained consequents in the goal set.

ting policy, the technique is effective. For this experiment, the precision measurements are the most important. (Low recall measures are not a concern. Typically an error induces many differences in behavior, and recognizing and understanding just one of them is sufficient.)

The precision measurements show that on average 6–12% of the reported properties indicate a difference in behavior between succeeding and failing runs. In other words, a programmer using our methodology with random sampling could expect to examine 8 reported implications before discovering one that indicates the difference between correct and erroneous behavior. (The other 7 implications may be useful for other tasks, but not for error detection. The Siemens programs and some MIT student programs contained only one or two faults in a program averaging 617 lines long, so there is a lot of conditional behavior in the programs that has nothing to do with program faults. In fact, it represents a significant success that precision is so high.)

4.3 Comparing policies

Our experiments show that our technique for selecting predicates for program analysis, along with the four policies for partitioning the data being analyzed, are successful. The resulting implications substantially eased the task of program verification and frequently pointed out behavior induced by errors.

It is natural to ask which of the splitting policies is best: which one should be used in preference to the others? Unfortunately, there is no clear winner: each approach is best in certain contexts or on certain programs, and overall the approaches are complementary. This is not surprising: programming tasks, and programs themselves, differ enough that no approach is a panacea. (In some cases, such as error detection in `print_tokens2`, no policy worked well, and future research is required.) We can draw some general conclusions, however.

On average, the random selection policy has a slight edge. This technique has no built-in biases regarding what sort of behavioral differences may exist, so it can do well regardless of whether those differences have some obvious underlying structure. Thanks to eliminating statistically unjustified properties (often resulting from data sets that are too small), the technique does not produce an excessive number of false positives. On the other hand, random selection does not work well when there is a relatively equal split between behaviors, and random selection cannot take advantage of structure when it is present (e.g., from source code).

Clustering is the second-best policy. It looks for structure in a vector space composed of values from the traces being analyzed. Fairly often, the structure in that space corresponded to real differences in behavior, either for verifiable behavior or due to errors. Clustering is less effective when the behavioral differences do not fall into such clusters in that vector space, however, and there can be problems with recognizing the correct number of clusters. It is interesting that the purely dynamic policies, which examine run-time values but ignore the implementation details of the source code, perform so well. This suggests that the structure of useful implications sometimes differs from the surface structure of the program itself; this observation could have implications for static and dynamic analysis.

The code conditional policy performs only marginally worse than the previous two. For the program verification task, it dominated the other policies, except on `QueueAr`, where it did very poorly. We speculate that the code conditional policy is well-matched to ESC/Java because ESC/Java verifies the program source code, and it often needs to prove conditions having to do with code paths through conditionals. Furthermore, whereas errors can be complicated and subtle, ESC/Java is (intentionally) limited in its capabilities and in its grammar, and is matched to the sort of code that (some) programmers write in practice. The poor showing on `QueueAr` is due at least in part to the fact that comparisons over variables `front` and `back` were crucial to verification, but those variables never appeared in the same procedure, much less in the same expression. In other cases, the code conditionals policy was hampered by the fact that ESC/Java does not permit method calls in annotations. Our code conditionals policy partially worked around this by inlining the bodies of all one-line methods (i.e., of the form "`return` *expression*"), with parameters appropriately substituted, in each conditional expression.

The procedure return policy performed worse than we anticipated. Like the code conditionals policy, it is highly dependent on the particular implementation chosen by the programmer. In some cases, this worked to its advantage; for instance, it outperformed all other policies at error detection for the `replace` program.

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 use related to our clustering splitting policy, 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 outlier regions is effective at detecting failures.

Comparing behavior to look for differences has long been applied by working programmers; however, both this research and some related research has found new ways to apply those ideas to the domain of error detection.

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] used dynamic invariant detection in conjunction with a checking engine and showed that the techniques are effective at bug detection. Related ideas regarding comparison of properties, but applied in a static context, were evaluated by Engler et al [ECH$^+$01], who detected numerous bugs in operating system code by exploiting the same underlying idea: 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.

Groce and Visser [GV03] use dynamic invariant detection to determine the essence of counterexamples: given a set of counterexamples, they report the properties that are true over all of them. These properties (or those that are true over only succeeding runs) abstract away from the specific details of individual counterexamples. (This is scenario 1 of Section 4.2.)

6 Conclusion

This paper proposes a technique for improving the quality of program analysis. The improvement uses *splitting conditions* to partition data under analysis and then to create implications or conditional properties. It is computationally infeasible to consider every possible implication. Splitting conditions limit the predicates that are considered, but predicates that are not splitting conditions may still appear. Concretely, the experimental results show the benefits for dynamic detection of likely program invariants.

The paper proposes four splitting (data partitioning) policies that can be used in conjunction with the implication technique: return value analysis, simple static analysis to obtain code conditionals, clustering, and random selection. No policy dominates any other, but on average the latter two perform best. We provided preliminary explanations of this behavior.

The paper introduces a two-pass program analysis technique that refines inexact, statistical, or inexpressible splitting conditions into ones that can be applied to arbitrary runs and understood by humans. This technique leverages a base program analysis to produce more useful predicates and implications.

Two separate experimental evaluations confirm the efficacy of our techniques. First, they improve performance on a program verification task: they reduce the number of missing properties, which must be devised by a human, by 40%. Second, we proposed a methodology for detecting differences in behavior between faulty and non-faulty program runs, even when the user has not identified which runs are faulty and which runs are not. Most conditional behavior in a program results from other aspects of program execution than errors, but 12% of reported properties directly reflect errors.

References

[BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN 2001 Conference* *on Programming Language Design and Implementation*, pages 203–213, Snowbird, Utah, June 20–22, 2001.

[DDLE02] Nii Dodoo, Alan Donovan, Lee Lin, and Michael D. Ernst. Selecting predicates for implications in program analysis, March 16, 2002. Draft. `http://pag.lcs.mit.edu/~mernst/pubs/invariants-implications.ps`.

[DLP01] William Dickinson, David Leon, and Andy Podgurski. Pursuing failure: the distribution of program failures in a profile space. In *Proceedings of the 8th European Software Engineering Conference and the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 246–255, Vienna, Austria, September 12–14, 2001.

[Dod02] Nii Dodoo. Selecting predicates for conditional invariant detection using cluster analysis. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, September 2002.

[ECGN00] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *ICSE 2000, Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, Limerick, Ireland, June 7–9, 2000.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001. A previous version appeared in *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 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, October 21–24, 2001.

[Ern00] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[FL01] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *International Symposium of Formal Methods Europe 2001: Formal Methods for Increasing Software Productivity*, pages 500–517, Berlin, Germany, March 2001.

[FLL$^+$02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245, Berlin, Germany, June 17–19, 2002.

[GV03] Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In *10th International SPIN Workshop on Model Checking of Software*, pages 121–135, Portland, Oregon, May 9–10, 2003.

[HFGO94] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, pages 191–200, Sorrento, Italy, May 16–21, 1994.

[HL02] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE'02, Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, Orlando, Florida, May 22–24, 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.

[NE02a] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 232–242, Rome, Italy, July 22–24, 2002.

[NE02b] Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking: An empirical evaluation. In *Proceedings of the ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE 2002)*, pages 11–20, Charleston, SC, November 20–22, 2002.

[PM00] Dan Pelleg and Andrew Moore. X-means: Extending K-means with efficient estimation of the number of clusters. In *Proc. 17th International Conf. on Machine Learning*, pages 727–734, 2000.

[PMM$^+$99] Andy Podgurski, Wassim Masri, Yolanda McCleese, Francis G. Wolff, and Charles Yang. Estimation of software reliability by stratified sampling. *ACM Transactions on Software Engineering and Methodology*, 8(3):263–283, July 1999.

[PRKR03] Brock Pytlik, Manos Renieris, Shriram Krishnamurthi, and Steven P. Reiss. Automated fault localization using potential invariants. In *AADEBUG'2003, Fifth International Workshop on Automated and Algorithmic Debugging*, Ghent, Belgium, September8–10, 2003.

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

[RKS02] Orna Raz, Philip Koopman, and Mary Shaw. Semantic anomaly detection in online data sources. In *ICSE'02, Proceedings of the 24th International Conference on Software Engineering*, pages 302–312, Orlando, Florida, May 22–24, 2002.

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

[vR79] C. J. van Rijsbergen. *Information Retrieval*. Butterworths, London, second edition, 1979.

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