

A New Structural Coverage Criterion for Dynamic Detection of Program Invariants *

Neelam Gupta
Department of Computer Science
The University of Arizona
Tucson, AZ 85721
ngupta@cs.arizona.edu

Zachary V. Heidepriem
Department of Computer Science
The University of Arizona
Tucson, AZ 85721
zachary@cs.arizona.edu

Abstract

Dynamic detection of program invariants is emerging as an important research area with many challenging problems. Generating suitable test cases that support accurate detection of program invariants is crucial to the dynamic approach for invariant detection. In this paper, we propose a new structural coverage criterion called Invariant-coverage criterion for dynamic detection of program invariants. We also show how the invariant-coverage criterion can be used to improve the accuracy of dynamically detected invariants. We first used the Daikon tool to report likely program invariants using the branch coverage and all definition-use pair coverage test suites for several programs. We then generated invariant-coverage suites for these likely invariants. When Daikon was run with the invariant-coverage suites, several spurious invariants reported earlier by the branch coverage and definition-use pair coverage test suites were removed from the reported invariants. Our approach also produced more meaningful invariants than randomly generated test suites.

Keywords - Test data generation, path testing, program invariants, dynamic analysis, execution traces.

1 Introduction

Invariants play an important role in all stages of software development. Dynamic detection of program invariants is an emerging research area with many challenging problems [2, 3]. As with any dynamic approach, the accuracy of dynamically discovered invariants critically depends upon the test

suite used for detection of invariants. One parameter of the test suite that is loosely related to the accuracy of dynamic detection of invariants is the size of the test suite. However, not all large test suites are equally effective in accurate detection of invariants due to the varying degree of structural coverage obtained. Thus, generation of test inputs that support accurate detection of program invariants at runtime is an important research problem.

In our prior work [7] we show that test suites satisfying commonly used structural coverage criteria, such as branch coverage and all definition-use pair coverage, may not be good enough for accurate detection of program invariants. These test suites are generated for executing all branches and all definition-use pairs of the program. However, they may not provide good coverage for detecting whether a given program property is true at runtime. We illustrate this with an example code segment taken from [7] and shown in Figure 1.

Let the expression $(a*b == c*d)$ in line 14 in Figure 1 represent a likely invariant property to be monitored during every execution of this code segment. Let us consider a test suite

$$T_1 : \{(x = 5, y = 2), (x = -5, y = -1)\}.$$

Executing the code segment in Figure 1 with the test cases in T_1 executes every statement and every branch outcome of the two predicates $P1$ and $P2$ in the code segment. Also note that every *definition-use* pair in this code segment is exercised by T_1 . The property tested in line 14 holds true for this test suite T_1 . But it is easy to see that this property does not hold for the test case $(x = 5, y = -1)$. This simple example illustrates that branch coverage (each branch outcome is evaluated at least once by some test case) and even all definition-use pair coverage (every definition-use pair is exercised at least once

*Supported by National Science Foundation grant EIA-9806525 to the University of Arizona

by some test case) are not strong enough criteria for the test suite to be adequate for dynamic invariant detection.

```
0:  int funcEx(int x, y)
1:  {
P1:  if (x > 0)
2:      a=3;
3:      c=6;
4:  else
5:      a=3;
6:      c=9;
7:  endif
P2:  if (y > 0)
8:      b=4;
9:      d=2;
10: else
11:     b=3;
12:     d=1;
13: endif
14: /* Monitored Property: (a*b == c*d) */
15:  :
16: }
```

Figure 1. An example code segment

The above example provides insight into the limitations of using existing coverage based test suites for detecting invariant properties at different points in the programs. These test suites are designed for *coverage of specific structural elements (such as statements, branches, definition-use pairs etc.) of the program and may not contain test cases that are specifically helpful in verification of properties being monitored for invariant discovery.*

In the above example, we need test cases *exercising all possible combinations of branch outcomes by which the program execution can reach the critical point* where the property of interest is being monitored. But in general, the number of paths reaching the critical point may be unbounded due to the presence of loops. So the crucial problem is *how to identify the important paths reaching the critical point* so that executing the program with test inputs for these paths gives higher confidence in the value of the property being monitored during execution.

In this paper, we define a new criterion called the *Invariant-coverage* criterion for dynamic detection of program invariants. We present an approach based on this criterion to improve the accuracy of dynamically discovered likely program invariants. Our approach is to first use Daikon [2, 3] to do runtime analysis to report likely invariants with branch coverage and definition-use pair coverage test suites (referred to as traditional coverage test suites from here onwards) for the given program. We use these likely invariants to gen-

erate respective invariant-coverage suites for the program. These invariant-coverage suites contain test inputs for program paths specific to verification of those likely invariant properties that were reported earlier by the traditional coverage test suites. Then we rerun Daikon using invariant-coverage suites. Many of the spurious invariants reported earlier, when the traditional coverage test suites were used, are no longer reported as likely invariants by Daikon when the invariant-coverage suites are used. The subset of the invariants reported (from among those reported with the traditional coverage test suites) by the invariant-coverage test suites are more likely to be accurate than the original set of likely invariants reported with the traditional coverage test suites. We applied our approach for several programs to evaluate the effectiveness of invariant-coverage criteria. Our preliminary results show that our approach can significantly improve the accuracy of dynamically detected program invariants. The significant contributions of this paper are as follows:

- Development of *invariant-coverage criteria*
- A new approach to improve the accuracy of invariants detected at runtime
- Demonstration of the effectiveness of the invariant-coverage criteria and the potential of our approach in improving the accuracy of reported invariants

The organization of the paper is as follows. In the next section, we discuss the invariant-coverage and our approach in detail. In section 3, we present our experiments with using invariant-coverage criterion. We discuss the related work in section 4 and present the conclusions of our work in section 5.

2 Our Approach

A dynamic analysis tool called Daikon was developed in [3] for dynamically discovering likely program invariants. Daikon discovers invariants dynamically from program traces that capture the variable values at program points of interest. The user runs the target program over a test suite to create execution traces of the program. An invariant detector determines which properties hold over both explicit variables and other expressions. Variable and expressions for which these properties hold over the traces, and also satisfy other tests such as being statistically justified, not being over unrelated variables and not being implied by other

invariants, are reported as likely invariants. The set of the likely invariants reported by the tool critically depend upon the test cases used to perform dynamic analysis.

Our approach is to obtain initial guesses of likely invariant program properties by executing programs with their branch coverage and definition-use coverage suites and using Daikon to perform runtime analysis. Note that our approach can also be applied if we have an initial guess of likely program invariants from some other source such as program specifications.

Next, we generate program inputs with the specific goal of thoroughly covering these likely invariant properties. Specifically, for each likely invariant property, we generate a finite (yet comprehensive) set of definition-use chains that can effect the value of the invariant property at the point of interest. We then generate inputs for paths that execute all feasible combinations of these definition-use chains. These inputs thoroughly test the respective invariant property. We define an invariant-coverage suite for each likely invariant as consisting of inputs that execute these combinations of definition-use chains. We then run Daikon using these invariant-coverage suites. Since these invariant-coverage suites thoroughly test the initial guesses of likely invariant properties, many of the spurious likely invariants reported earlier by branch coverage and definition-use pair coverage suites are removed from the invariants reported with the invariant coverage suites.

The motivation for defining our invariant-coverage suites is as follows. The execution of paths traversing all feasible combinations of a finite set of definition-use chains, that can effect the value of an invariant property at a point in the program, is a stronger criterion than branch coverage or all definition-use pairs coverage criteria. The inputs in the traditional definition-use pair coverage suite may exercise only a few of these chains and hence report false positives. Note that generating definition-use chains for every variable at every point in the program is not practical. That will be a huge set of definition-use chains. However, we focus on the definition-use chains of the variables appearing only in the likely invariant properties at the points of interest. Therefore, we do not need to generate definition-use chains for every variable at every point in the program. Since the number of all definition-use chains that can effect the value of an expression at a point in a program can be unbounded, to limit the number of

definition-use chains to be considered, we define a set of longest simple definition-use chains for each variable in each invariant property. With this background for the development of our approach we now formally describe some terminology used in our approach.

We represent a *definition-use* pair of a variable v in a program P as $v(dst, ust)$, where dst is the statement number at which there is a *definition* of v and ust is the statement number at which there is a *use* of v such that there exists a definition-clear path for v from dst to ust in P . Next, we define our notion of a *simple definition-use chain* for a variable v at a point S in a program.

Definition: A **Simple definition-use chain** for a variable v at a point S in a program P is a chain of definition-use pairs, on which the value of v at S is directly or indirectly data dependent, such that no definition-use pair on the chain appears more than once except possibly the last definition-use pair which may appear at most twice on the chain.

We illustrate this with the ShellSort example program from [10] written in Java shown in Figure 2. For simplicity, we use the line numbers for the program shown in Figure 2 as the statement numbers when defining definition-use pairs. The *doNothing()* procedure invocations in the program in Figure 2 are dummy procedures that do nothing. They are inserted at the loop entry points so that Daikon tool can report program invariants at these points. Some simple definition-use chains for variable j used at line number 10 are given below.

```
chain1:j(9, 10) ← i(6, 9).
chain2:j(13, 10) ← incr(3, 13).
chain3:j(13, 10) ← incr(17, 13) ← incr(3, 17).
chain4:j(13, 10) ← incr(17, 13) ← incr(17, 17)
                ← incr(3, 17).
chain5:j(13, 10) ← incr(17, 13) ← incr(17, 17)
                ← incr(17, 17)||
```

Note that chain 5 above is terminated with `||` to indicate that the last definition-use pair in the chain is repeated. Next we define a longest simple definition-use chain.

Definition: A **longest simple definition-use chain** for a variable v at a point S in a program P is defined as a simple definition-use chain to which no more definition-use pairs can be added without violating the property of it being a simple chain.

For example, the simple chains 1, 2, 3 and 4 above

```

0: public class :ShellSort
1: { public static void sort(int[] a)
2:   { int n = a.length;
3:     int incr = n / 2;
4:     while (incr >= 1){
5:       doNothing1(incr);
6:       for (int i = incr; i < n; i++){
7:         doNothing2(a[i], i);
8:         int temp = a[i];
9:         int j = i;
10:        while (j >= incr && temp < a[j - incr]){
11:          doNothing3(temp, j, i, incr, a[j], a[j - incr]);
12:          a[j] = a[j - incr];
13:          j -= incr;
14:        }
15:        a[j] = temp;
16:      }
17:      incr /= 2;
18:    }
19:  }

```

Figure 2. ShellSort example program

are longest chains since no more definition-use pairs can be added to these chains. However, simple chain 5 is also a longest chain although more definition-use pairs can be added to this chain to make it longer. Note that if we add a definition-use pair to this chain, it would no longer satisfy the property of being a simple chain. For example, chain 5 can be made longer as shown below:

chain5': $j(13, 10) \leftarrow incr(17, 13) \leftarrow incr(17, 17)$
 $\leftarrow incr(17, 17) \leftarrow incr(3, 17)$

But now chain5' is does not satisfy the property of a simple chain since the definition-use pair $incr(17,17)$ appears more than once on chain5' and it is not the last definition-use pair on the chain.

Now we define our invariant-coverage criteria that we use for generating inputs to support accurate detection of program invariants at runtime. Let $Inv(v_1, v_2, \dots, v_n)$ be an expression in program variables v_1, v_2, \dots, v_n that represents a likely invariant property at a point S in a program P . Let $Sdu(v_i, S, P)$ represent the set of all longest simple definition-use chains for the variable v_i at point S in program P .

Definition: An **Invariant-coverage** suite $T(Inv, S, P)$, for an expression Inv representing a likely invariant property at point S in a program P , is the set of inputs that execute the paths through all the combinations of the longest *simple definition-use chains* of all the variables in Inv .

Let us generate invariant-coverage suite for the property monitored at line 14 in Figure 1. This invariant property has 4 variables a, b, c, d . There are

two definition use chains of each variable reaching line 14 as follows: $a(2, 14); a(5, 14); b(8, 14); b(11, 14); c(3, 14); c(6, 14); d(9, 14)$ and $d(12, 14)$. The set of paths that exercise all feasible combinations of these definition-use chains are the four paths in this code segment corresponding to all the combinations of the branch outcomes of $P1$ and $P2$. As another example, after running Daikon using a branch coverage test suite for the program in Figure 2, one of the likely invariants reported at line 10 in the program is " $j == \text{OneOf}[1,2,4]$ ". Since this invariant has only one variable, namely j , we just need to compute the set $Sdu(j, 10, P)$ which is precisely the set of 5 chains (chain1,...,chain5) given above. The invariant-coverage suite for this invariant is the set T of inputs such that each chain in $Sdu(j, 10, P)$ is executed by some input in T .

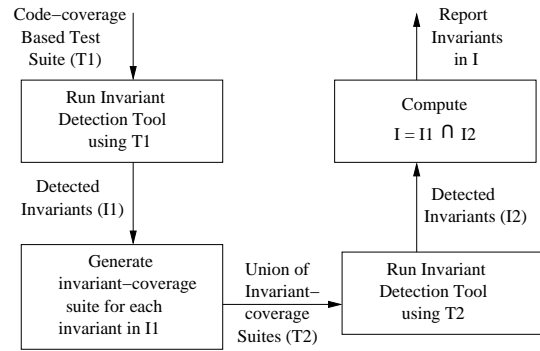


Figure 3. Our Approach to support accurate detection of Program Invariants at Runtime.

The steps of our approach are shown in Figure 3. First we generate a branch coverage or an all definition-use pair coverage test suite $T1$ of the given program if it is not already available. Next we run Daikon using the test suites generated in step 1 to obtain a set $I1$ of likely program invariants. Then we generate *invariant-coverage* suites for the set of likely invariants reported in $I1$. Let $T2$ be the union of the test cases in all the invariant coverage suites. We then run Daikon using the test cases in $T2$ to generate a set $I2$ of likely program invariants. Finally, we compute the invariants in intersection $I = I1 \cap I2$. The intersection of invariants is performed literally, i.e., an invariant in I appears exactly in identical form in each of $I1$ and $I2$. The invariants reported in the set I are more likely to be accurate than the original set $I1$ of invariants reported by the structural coverage of the program.

3 Experiments

We applied our approach to remove spurious invariants reported by branch coverage and definition-use coverage test suites for several programs. We also used randomly generated test suites of the same size as that of the invariant-coverage suites and 5 times the size of invariant-coverage suites and compared their effectiveness in generating program invariants with the test suites satisfying other coverage criteria.

Daikon reports invariants at the entry and exit of a procedure. The invariants are reported on the procedure parameters and arguments of the *return* statement. For generating program invariants at other program points such as loop entry, a call to a dummy procedure (which does nothing) with the variables of interest as parameters was inserted in the program. At present, this is the only method provided by Daikon for detection of loop invariants. All variables used or defined in a loop (not including variables that were used only in the nested loops) were examined for loop invariants by passing them as parameters to these procedure invocations at the loop entry. We chose not to examine inner loops so that we could focus only on those variables that were being used or defined in the current loop. Thus, all variables were examined only in the context(s) in which they are used. Additionally, those variables must be defined at the loop entrance, so that the code may be compiled. If an array was defined but not initialized, we assumed all elements to be initialized to 0.

Now we discuss the programs selected in our experiments. The basis for selection of these programs was to cover a variety of program features as explained below. We also discuss how we generated various test suites for each program. All these programs were written in Java.

3.1 Programs

Boyer-Moore Matching: The Boyer-Moore (BM) algorithm finds the index of the leftmost substring of the text string matching the pattern string, or returns -1 if there is no match. We used the source code for this program from [4]. We selected this method because it has multiple nested control structures inside of a while loop, including a return statement. For this reason we believed this program would allow us to generate many possible definition-use chains. Because the method had three return statements, we required three test

cases to provide full branch coverage. The remaining branches could be covered by one of those cases because we utilized multiple iterations of the outermost loop. We also generated an all definition-use pair coverage test suite. After generating all the def-use pairs for the program, we generated the required path of execution for each pair. We then manually generated a test suite that executed each of these paths.

For invariant-coverage suites corresponding to the invariants reported by the the branch coverage, we found a total of 113 statically possible definition-use chains, 40 of which turned out to be feasible. Similarly, for invariant-coverage suites corresponding to the invariants reported by the definition-use coverage, there were 46 feasible chains out of a total of 113 statically possible definition-use chains. We eliminated the infeasible chains by visual inspection. Next, we created paths corresponding to each chain. We manually tried test cases until we found a set of cases that executed each required path.

Levenshtein Distance: This method, taken from [11] finds the Levenshtein distance between two strings. This is the number character modifications needed to change one string into another. This method has nested for-loops that fills in a two-dimensional array. This is another type of program that we believe will allow us to generate many possible definition-use chains for the invariant-coverage suites. The traditional structural coverage test-suites were generated manually. For this program, the branch coverage test-suite also provided definition-use coverage.

For the invariant-coverage suites we found a total of 189 possible definition-use chains, 81 of which turned out to be feasible. To generate the cases for these chains, we needed to look at the semantics of the program. This involved understanding how the matrix was filled in, and what the dependencies were for each cell. This allowed us to understand which chains were feasible so that we could generate an execution path for each chain. We then created corresponding test cases manually.

Matrix Adjoint: This program, taken from [12], calculates the adjoint of a matrix. We chose this program because in contrast to the previous program that manipulated strings, this program was mathematical in nature and still contained interesting control structures that could yield a large number of def-use chains. We manually generated test cases for each traditional structural coverage test suite. For this program, we were able to pro-

vide definition-use and branch coverage with a single test suite which contained a single test case. From the invariants reported by this test case, we constructed 70 corresponding definition-use chains for the invariant-coverage suites, all of which were feasible. We then selected corresponding paths and generated test cases manually.

Ordered Array: This array class supports five operations: insert, delete, find, display, and getSize. It is interesting to us because it contains an instance variable that is accessible at all program points. The source code of this program was taken from [15]. We generated the traditional structural coverage test suites by hand. For the invariant-coverage suites corresponding to the invariants reported by the branch coverage suite, we found a total of 109 possible definition-use chains, all of which turned out to be feasible. Similarly, there were 118 possible chains for the invariants reported by the definition-use coverage suite, 116 of which turned out to be feasible chains. We manually created test cases for each of those chains.

ShellSort: This program sorts an input array of int values and was taken from [16]. For invariants reported using the branch coverage suite, the test case generation for invariant-coverage suites was done semi-automatically. We generated the chains for each invariant by hand. Because this program was more complex we were unable to easily create corresponding test cases for those chains. We instead generated large sets of input data. This data was a combination of larger random arrays and sets of all orderings of small arrays with a specified size. We instrumented the program so that it could record the chains for a specified variable v at a specified point P in the program. The instrumented program stored the chains executed for each input. We continued to add inputs to the suite until we found a set of inputs that executed all the chains we had generated by hand.

With the definition-use coverage suite, Daikon reported 20 likely invariants of which 7 involved two or more variables. Our technique for generating the invariant-coverage suite requires, that for invariants with multiple variables, we generate the combinations of the chains of all the variables used in each invariant. We generated these combinations automatically after generating the chains for each variable by hand. This resulted in a set of over 3500 chains. Many of the resulting chains will turn out to be infeasible since they are not syntactically allowed by the program structure. In order for us to show that we have created all the neces-

sary test cases, we need to eliminate those infeasible chains. A parser can be instrumented to automatically eliminate these syntactically infeasible chains. However, there were too many chains for us to inspect manually for feasibility.

After generating the above test suites, we used these test suites and the Daikon tool to report the likely program invariants. We also generated random test suites for the above programs. Random suites of size equal to the size of invariant-coverage suites and of size 5 times the size of invariant-coverage suites were constructed. This was to see how the invariants reported by random suites were effected by the size of the test suite. In the next section, we discuss the results of our experiments.

3.2 Results and Discussion

The number of likely invariants reported by Daikon for each program for different test suites is shown in Table 1. There are two rows for each program. The rows with *(Br)* and *(Du)* appended at the end of the program name correspond respectively to conventional branch coverage and all definition-use pair coverage test suites. The columns labeled by Tc represent the number of test cases and the columns labeled by Inv correspond to the number of invariants detected by the respective test suite. The columns under the heading *C-Coverage* correspond to the branch and all definition-use pairs coverage suites used to report invariants. The columns under the heading *Invariant-coverage* correspond to the number of test cases used and the number of invariants reported when the invariant-coverage suites were used with Daikon. Finally, columns under the heading *Random Coverage* are the number of test cases used and the number of invariants reported with randomly generated test suites. The columns labeled with the intersection of two sets of invariants report the number of invariants that were identical in both the sets.

Boyer-Moore Matching: In this program, although the number of test cases in the all definition-use pairs suite were only one more than the number of test cases in the branch coverage suite, 37 invariants were reported by branch coverage suite whereas only 16 invariants were reported by the all definition-use pairs suites. However, upon inspection we found that most of the likely invariants reported by the branch coverage suite were indeed not true invariants. These invariants are the ones that give the value(s) of variables explicitly;

Program	C-Coverage		Invariant-coverage			Random Coverage					
	$Tc1$	$Inv1$	$Tc2$	$Inv2$	$Inv1 \cap Inv2$	$Tc3$	$Inv3$	$Inv1 \cap Inv3$	$Tc4$	$Inv4$	$Inv1 \cap Inv4$
BMMatch(<i>Br</i>)	3	37	6	27	22	6	9	9	30	12	5
BMMatch(<i>Du</i>)	4	16	4	14	9	4	8	4	20	12	5
L-Distance(<i>Br</i>)	3	37	18	34	20	18	32	13	90	28	11
L-Distance(<i>Du</i>)	3	37	18	34	20	-	-	-	-	-	-
MatrixAdj(<i>Br</i>)	1	61	3	62	6	3	82	7	15	97	8
MatrixAdj(<i>Du</i>)	1	61	3	62	6	-	-	-	-	-	-
OrdArray(<i>Br</i>)	10	89	113	82	27	113	36	14	565	36	16
OrdArray(<i>Du</i>)	71	98	248	102	37	248	36	14	1240	36	14
ShellSort(<i>Br</i>)	3	9	876	15	1	876	29	1	4380	29	1

Table 1. Number of Test Cases (Tc) and number of likely Invariants (Inv) reported by Daikon for Conventional Coverage test suites, Invariant-coverage suites and Randomly generated test suites.

e.g. `str == "hello"` or `"a == OneOf[1,2,3]"` and were tied to actual input data used rather than invariant properties of the programs. The all definition-use pairs suite detected more meaningful invariants since the above spurious invariants were not reported due to increased coverage of the program. Note that by meaningful invariants we mean invariants that are not linked to specific input values used and are abstractions over multiple inputs.

When we used the invariant-coverage suites corresponding to the invariants detected by the branch coverage suite, the number of invariants reported was reduced from 37 to 27. Of these 27, 5 were "new" invariants and 22 were the same as those reported by the branch coverage suite. Although the invariant-coverage suite removed 15 spurious invariants reported by the branch coverage suite, of the 22 in common, many are still linked to the test suite because some of the chains for the invariant-coverage suite were also exercised by the test cases in the branch coverage suite.

The invariant-coverage suite corresponding to the invariants reported by the all definition-use pairs suite reduced the number of invariants from 16 to 14. Of these 14, 5 were "new" invariants and 9 were reported by the def-use coverage suite. The 5 "new" invariants are unverified invariants since invariant-coverage suites for these invariants were not constructed. However, the invariant-coverage suite removed 7 false invariants from the 16 invariants reported by the all definition-use pairs suite. The remaining 9 invariants which were reported by both the invariant-coverage suite and the all definition-use pairs suite appeared to be accurate and least linked to either test suite. These results lead us to conclude that the set of invariants, obtained by taking the intersection of the invari-

ants reported by the all definition-use pairs coverage suite and the invariants reported by the corresponding invariant-coverage suite, are more likely to be accurate.

Levenshtein Distance: For this program the branch coverage suite also exercised all definition-use pairs in the program. Therefore, the branch coverage and all definition-use pairs coverage suite were identical for this program and had 3 test cases. Daikon reported 37 likely invariants with this test suite. When Daikon was run with the corresponding invariant-coverage suite, 17 false invariants out of these 37 likely invariants were removed from the set of reported likely invariants. When the 20 invariants that belonged to the intersection of the invariants reported by the traditional structural coverage suites and the invariant-coverage suites were inspected, about half of these 20 appear to be linked to the test inputs used. The reason for this is that some of the test-cases in the traditional structural coverage suites were re-used in the invariant-coverage suite. Because of this fact, the intersection of the suites carried over some of the false invariants reported by the traditional structural coverage suites. If different input were used for the common paths in the traditional structural coverage suite and the invariant-coverage suite, then these false invariants will also be removed from the intersection of the invariants reported by conventional coverage and invariant-coverage suites.

Matrix Adjoint: In this program the conventional branch coverage and all definition-use pair coverage suites were identical and contained 1 test case. A total of 61 likely invariants were reported by the conventional coverage suites. The invariant-coverage suite eliminated 55 false invariants from

these 61 likely invariants that Daikon reported with the conventional coverage suites. Thus, there were only 6 invariants (all of which were meaningful) that belonged to the intersection of invariants reported by the conventional coverage suites and the invariant-coverage suites. This shows the effectiveness of invariant-coverage suites in removing the spurious invariants reported by traditional structural coverage suites.

Ordered Array: The branch coverage suite required only 10 cases and Daikon reported 89 invariants with this test suite. The all definition-use pairs coverage suite with 71 test cases reported only 9 more invariants than the branch coverage suite. However, as was observed in the case of Boyer-Moore Matching algorithm, the invariants reported by the all definition-use pairs coverage suite contained more meaningful invariants than the branch coverage suite.

Let us now compare the invariants reported by the branch coverage suite and the invariants reported by the corresponding invariant-coverage suite. Daikon detected 89 invariants using the branch coverage suite, while it detected 82 using the corresponding invariant-coverage suite. The invariant-coverage suite removed 62 false invariants from the invariants reported by the branch coverage suit. There were 27 invariants in the intersection of the two suites. Because the suites were mutually distinct, these invariants were not linked directly to either suite and were more likely to be accurate invariants as compared to other invariants reported by the branch coverage suite. Thus, the intersection does provide us with a number of useful invariants.

Now we compare the invariants reported by the conventional all definition-use pairs suite and the invariants reported by corresponding invariant-coverage suite. The invariant-coverage suite removed 61 false invariants from those reported by the all definition-use pair coverage suite. There were 37 invariants in the intersection of the invariants reported by these two suites out of which 27 are meaningful. This indicates to us that the intersection of the suites provides a decent number of meaningful invariants.

ShellSort: The branch coverage suite for this program contained 3 test cases. Using this suite, Daikon detected 9 invariants. However, only 1 of these was a meaningful invariant. The invariant-coverage suite generated from these 9 had 876 cases, which were generated semi-automatically as described in the previous section. Daikon de-

TECTED 15 invariants (all meaningful) using this suite, none of which were directly linked to the input data. The one valid invariant found using the branch coverage suite was also detected in the invariant-coverage suite. We conclude from this example that the branch coverage criteria is not strong enough to extract a complete set of meaningful invariants, but the invariant coverage suite associated with the coverage suite may still produce good invariants.

3.2.1 Invariants Detected with Randomly Generated Suites

Now we present our analysis about the invariants reported by Daikon when randomly generated test suites are used to report likely invariants for the above programs. Our experience in using Daikon to report likely program invariants with randomly generated test suites is as follows. The number of likely invariants reported by Daikon initially increases with the increase in the size of a randomly generated test suite since more test cases provide more sample points to Daikon. This is what is observed in the Boyer-Moore matching program. In this program, increasing the size of the random suites increased the number of invariants detected from 8 to 12 and from 9 to 12. Similar behavior is observed in the Matrix Adjoint program for which increasing the size of the suite increased the number of invariants detected from 82 to 97. However, after reaching a certain number of invariants, the number of likely invariants reported by Daikon decreases with the increase in the size of a randomly generated test suite. This is because adding more test cases allows Daikon to eliminate more false invariants. This is what is observed in the Levenshtein Distance program. In this program, increasing the size of the suite decreased the number of invariants detected from 32 to 28. However, if the size of the random suites continues to increase, the number of invariants reported do not change with further increase in the size of the test suite. This is because additional test cases do not provide new coverage, i.e., new sample points to Daikon. Besides for each of the covered program points, the test suite already has enough samples for Daikon to detect invariants. This limiting behavior is observed in the case of OrdArray and ShellSort programs as shown in Table 1.

Program	Invariant-coverage	Random Coverage	
	$M(Inv1 \cap Inv2)$	$M(Inv1 \cap Inv3)$	$M(Inv1 \cap Inv4)$
BMMatch(<i>Br</i>)	8	5	4
BMMatch(<i>Du</i>)	9	3	4
L-Distance(<i>Br</i>)	7	6	6
L-Distance(<i>Du</i>)	7	-	-
MatrixAdj(<i>Br</i>)	6	6	8
MatrixAdj(<i>Du</i>)	6	-	-
OrdArray(<i>Br</i>)	18	10	12
OrdArray(<i>Du</i>)	27	12	12
ShellSort(<i>Br</i>)	1	1	1

Table 2. Results excluding uninteresting invariants. Uninteresting invariants are those invariants that give the value(s) of variables explicitly; e.g. `str == "hello"` or `"a == OneOf[1,2,3]"`.

3.2.2 Comparisons after Removal of Uninteresting Invariants

We also looked at the invariants in the intersections corresponding to the columns labeled $Inv1 \cap Inv2$, $Inv1 \cap Inv3$ and $Inv1 \cap Inv4$ in Table 1, after removing uninteresting invariants. Uninteresting invariants are those invariants that give the value(s) of variables explicitly from the input data used in the test cases; e.g. `str == "hello"` or `"a == OneOf[1,2,3]"`. These uninteresting likely invariants are tied to the input data used and would change (unless a path is executed only for a single input value) if different input was used for the same path. We call the remaining likely invariants as meaningful invariants.

The columns labeled with $M(Inv_i \cap Inv_j)$ for $i=1$ and $j=2,3,4$ in Table 2 show the number of meaningful invariants in respective columns labeled with $Inv_i \cap Inv_j$ for $i=1$ and $j=2,3,4$ in Table 1. It is seen from Table 2 that in almost all the cases (except Matrix Adjoint) the number of meaningful invariants in $Inv1 \cap Inv2$ were more than or equal to the number of meaningful invariants in $Inv1 \cap Inv3$ and $Inv1 \cap Inv4$. This indicates that in most cases, the random suites weren't generating enough sample points for Daikon to produce a larger subset of meaningful invariants reported by traditional coverage suites. This is because randomly generated test suites do not provide [3] adequate coverage of program points to Daikon. However, they do provide a large number of samples at frequently executed program points to Daikon to be able to infer meaningful invariants at those points. Thus, we can use the random suites to help verify likely invariants, but we cannot use them to eliminate any spurious invariants from those reported by the traditional coverage suites. However, our invariant-

coverage test suites are found to be effective in eliminating false invariants from the likely invariants reported by the traditional structural coverage suites. Besides, invariant-coverage suites corresponding to the likely invariants reported by the traditional structural coverage suites can provide a larger number of meaningful invariants than randomly generated test suites.

4 Related Work

In prior work [2, 3], randomly generated and grammar generated test suites have been used for invariant detection. Randomly generated test suites have poor coverage and are most effective at highly peculiar bugs [9]. In the experiments reported in [3], the randomly generated test suites failed to execute many portions of a program. The experiments using randomly generated test suites from a grammar describing valid inputs detected more invariants than completely randomly generated test suites. However, generating test cases using grammar rules is a black box approach to test case generation and in general can fail to cover a significant part of the implementation. Some pre-existing test suites were also used in [2, 3] for invariant detection. However, so far there is no characterization of the properties of a test suite that make it suitable for invariant detection.

An operational difference technique [8] for generating, augmenting, and minimizing test suites has been recently proposed. The technique is analogous to structural code coverage techniques, but it operates in the semantic domain of program properties rather than the syntactic domain of program text. Another recent related research is on test suite augmentation [17] for inferring program specifica-

tions. This approach is different from ours since test case selection is not based on structural coverage but on the violated specification.

Several approaches [1, 5, 6, 13, 14] are available for automated test data generation for structural testing of programs. Our technique for test data generation in [6] has been implemented for handling programs written in a subset of the C language. In the experiments presented in this paper, all the programs used were in the Java language. Since the programs were small, we generated the required test cases manually.

5 Conclusions and Future Work

In this paper, we have developed a new structural coverage criteria called invariant-coverage criteria for removing false invariants from those reported by traditional structural coverage suites. We present an approach to compute a subset of invariants that are more likely to be accurate invariants. We present experimental results for several programs to demonstrate the effectiveness of our approach. Our experiments show that the intersection of the invariants reported by traditional structural coverage suites with the invariants reported by invariant-coverage suites can remove many false positives reported by the traditional structural coverage suites.

In our future work we plan to experiment with iteratively applying our technique, to the set of likely invariants reported in each iteration, until there is no change in the set of invariants reported. We expect this iterative application of our approach to be helpful in accurately deriving most of the invariants of a program.

Acknowledgments

The authors would like to thank Michael D. Ernst, EECS Dept., Massachusetts Institute of Technology, for help with using the Daikon tool. The authors also thank Suzanne Westbrook, CS Dept., The University of Arizona, for providing feedback on the manuscript.

References

- [1] R.A. DeMillo and A.J. Offutt, "Constraint-based Automatic Test Data Generation," *IEEE Transactions on Software Engineering*, Vol. 17, No. 9, pages 900-910, September 1991.

- [2] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, Feb. 2001, pages 1-25.
- [3] M. D. Ernst. "Dynamically Discovering Likely Program Invariants," *Ph.D. dissertation*, University of Washington Department of Computer Science and Engineering, (Seattle, Washington), Aug. 2000.
- [4] M. T. Goodrich and R. Tamassia. Online supplement to the book "Data Structures and Algorithms in Java," <http://java.datastructures.net/source/ch11/Java/PatternMatching-BMmatch.html>.
- [5] A. Gotlieb, B. Botella, and M. Rueher, "Automatic Test Data Generation using Constraint Solving Techniques," *International Symposium on Software Testing and Analysis*, 1998.
- [6] N. Gupta, A. P. Mathur, and M. L. Soffa, "Automated Test Data Generation using An Iterative Relaxation Method" *ACM SIGSOFT Sixth International Symposium on Foundations of Software Engineering (FSE-6)*, pages 231-244, Orlando, Florida, November 1998.
- [7] N. Gupta, "Generating Test Data for Dynamically Discovering Likely Program Invariants", in the *Proceedings of ICSE 2003 Workshop on Dynamic Analysis (WODA 2003)*, Portland, Oregon, pages 21-24, May 2003. <http://www.cs.nmsu.edu/~jcook/woda2003/>
- [8] M. Harder, J. Mellen and M. D. Ernst. "Improving test suites via operational abstraction," in *Proceedings of the 25th International Conference on Software Engineering, (Portland, Oregon)*, May 6-8, 2003, pages 60-71.
- [9] D. Hamlet, "Random Testing," *Encyclopedia of Software Engg.*, 1994.
- [10] C. Horstmann, *Core Java*, Volume 1, Sun Microsystems, page 95-96.
- [11] <http://www.merriampark.com/ld.htm>
- [12] <http://www.mkaz.com/math/matrix.html>
- [13] B. Korel, "Automated Software Test Data Generation," *IEEE Transactions on Software Engineering*, Vol. 16, No. 8, pages 870-879, August 1990.
- [14] B. Korel, A Dynamic Approach of Test Data Generation. In *Conference on Software Maintenance*, pages 311-317, San Diego, CA, November 1990.
- [15] R. Lafore "Data Structures and Algorithms in Java", Waite Group Press, 1998, pages 47-48.
- [16] C. Horstmann, *Core Java*, Volume 1, pages 95-96, Sun Microsystems.
- [17] T. Xie and D. Notkin, "Exploiting Synergy Between Testing and Inferred Partial Specifications", in the *Proceedings of ICSE 2003 Workshop on Dynamic Analysis (WODA 2003)*, Portland, Oregon, pp. 17-20, May 2003.