# Extended Program Invariants: Applications in Testing and Fault Localization

Mohammad Amin Alipour, and Alex Groce
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, OR 97330
{alipour,alex}@eecs.oregonstate.edu

## ABSTRACT

Invariants are powerful tools for program analysis and reasoning. Several tools and techniques have been developed to infer invariants of a program. Given a test suite for a program, an invariant detection tool (IDT) extracts (potential) invariants from the program execution on test cases of the test suite. The resultant invariants contain relations only over variables and constants that are visible to the IDT. IDTs are usually unable to extract invariants about execution features like taken branches, since programs usually do not have state variables for such features. Thus, the IDT has no information about such features in order to infer relations between them. We speculate that invariants about execution features are useful for understanding test suites; we call these invariants, *extended invariants.*

In this paper, we discuss potential applications of extended invariants in understanding of test suites, and fault localization. We illustrate the usefulness of extended invariants with some small examples that use basic block count as the execution feature in extended invariants. We believe extended invariants provide useful information about execution of programs that can be utilized in program analysis and testing.

## 1. INTRODUCTION

Invariants are powerful tools for understanding and analysis of programs. Invariants state existing relations between variables of a program in different stages of the program execution. Such relations can be used to reason about properties of the program. In other words, they try to reflect the effects of different parts of the program on the program state, while abstracting away concrete computation steps of the program. For example, a loop invariant abstracts statements in a loop by relations between data values in an iteration to data values in the next iteration. Having such invariants can greatly reduce the efforts needed to reason about programs.

Inferring invariants from programs is hard. In fact, all efforts in verification of a program $p$ are spent to infer the invariant "given an input, $p$ produces correct output". Several techniques have been devised to infer invariants. These techniques are broadly classified in two categories: static techniques, and dynamic techniques.

Static invariant extraction techniques attempt to infer invariants from the source code. Since such inferences usually involve conservative approximations of program behavior, they do not scale to large code, but if such techniques finds an invariant it is guaranteed to hold. These techniques often use common static analysis frameworks like abstract interpretation and constraint analysis to extract invariant. For example, Cousot and Halbwachs uses abstract interpretation to find linear restraints between program variables [5], or Kovacs and Voronkov use theorem provers to infer loop invariants [14].

Dynamic invariant extraction techniques summarize common properties that are held true in multiple program runs. These invariants are often called dynamic invariants or potential invariants. Approximating behaviors of programs as finite-state automata [2, 7, 21], and extracting potential contracts [6, 9] are some of techniques that exploit information of executions.

Invariants facilitate analyzing a program by abstracting the program execution, but many tasks in software testing and dynamic analysis need to know what the program executes. For example, in software testing, we want to make sure that every part of a program is sufficiently exercised.

Unfortunately, an IDT cannot identify possible invariants on what a program executes, because related data is not visible to them. To address this, we suggest making information about the program execution available *inside* the program execution. To this end, first, some features that can characterize the execution are selected. Then, the program is modified such that it computes those features alongside its own computations. Now the IDT is able to observe the characteristics of an execution during execution of the program. Thus, it is able to summarize such characteristics. New invariants in the modified program characterize the execution by relating execution features of different parts of the program to each other. We call the new invariants in the modified program extended invariants. In the rest of this paper, we use basic block counts as a feature of a program execution to illustrate the idea.

Extended invariants can serve as powerful tools for understanding test suites and programs. They can be used to compare two executions of a program by comparing the characteristic invariants of them. Figure 1 shows a buggy

*WODA* '12, July 15, 2012, Minneapolis, MN, USA
Copyright 2012 ACM 978-1-4503-1455-8/12/07 ...$10.00.

```
#define SIZE 64
int s = 0;
int stack[SIZE];
int top(){
  return stack[s];
  }
void push(int i){
  stack[i++];
}
void pop(){
  if(s > 0)
    s--;
}
```

**Figure 1: Stack source code.**

```
Test Case 1     Test Case 2
-----------     -----------
push(3);        push(3);
top();          pop();
pop();          top();
                push();
```

**Figure 2: test cases for the stack implementation in Figure 1.**

implementation of a stack. This stack has two bugs: function `top` does not check if the stack empty, and `push` does not check if the stack is full before adding a new element to it. Consider test cases in Figure 2 for the stack. Test case 1 and test case 2 have similar blocks and branch coverage. It can observed that the number of `push` operations in test case 2 exceeds the number of `pop` operations in the test cases 1. Comparing branch and block coverage does not reveal this difference. However, if we transform the program to include the basic block count, as in Figure 3, invariants derived by dynamic IDTs reflect these difference. Dynamic invariants for test case 1 include extended invariant `btop = bpush = b1pop = b2pop`, and for test case 2 includes extended invariants `btop = b1pop = b2pop`, and `bpush - p2pop = 1`. Comparing the extended invariants of different test cases helps to understand how a test case contributes in examining different aspects of a program. These invariants provide more information than coverage data. Moreover, they can be used to guide testing efforts.

In the rest of this paper, in Section 2, we present a simple transformation to add computation of basic block counts of a program into the program. This transformation includes the required information for extended invariant to the program. In Section 3, we present some possible applications of extended invariants in software testing. In Section 4, we describe possible application of extended invariants as a new type of spectra for fault localization. Finally, Section 5 concludes the paper.

## 2. PROGRAM TRANSFORMATIONS FOR EXTENDED INVARIANTS

In this section, we outline transformation to include computation of basic block count in a program. Basic block count shows how many times a basic block has been executed in a program run. The proposed transformation in this section can be easily adapted for other execution aspects like branch coverage/count.

Algorithm 1 outlines steps for transformation of a program $P$ to include computation of basic blocks counts in the program. The algorithm first identifies set $BB$ of basic

```
#define SIZE 64
int s = 0;
int stack[SIZE];
int btop, bpush, b1pop, b2pop;
int top(){
  btop ++; // block count calculation
  return stack[s];
  }
void push(int i){
  bpush ++;//block count calculation
  stack[i++];
}
void pop(){
  b1pop ++;//block count calculation
  if(s > 0){
    b2pop ++;//block count calculation
    s--;
    }
}
```

**Figure 3: Transformed Stack source code which includes block count information.**

block s in $P$. For each basic block $b_i$,$1 \leq i \leq n$ a variable $gb_i$ is defined and added to global variables of the program. This variable captures total number of executions of $b_i$ in the *entire* execution of program. Moreover, there might be some relations within basic blocks of a function $f$ internal to *individual* executions of $f$. Thus, the algorithm adds new variables local to the function to capture the number of times a basic block is executed in a single invocation of $f$. Since IDTs usually infer invariants at before entry point and after exit point of functions, the algorithm adds the corresponding variables to arguments of function, thus it makes IDTs to process them. Suppose $f$ has $k$ basic blocks; to capture their relations, the algorithm adds $k$ new variables $lb_j$ to arguments of $f$. Statement $lb_j = lb_j + 1$ is added to each basic block $j$, $1 \leq j \leq k$ to compute number of times block $j$ is executed in a single invocation of $f$. Algorithm 1 changes each call-site to $f$ to invoke $f$ with new fresh values of $lb_i$ by reference.

**Algorithm 1** Transformation to compute basic block count.

```
Input: Program $P$, and $BB = b_1, ..., b_n$ set of basic blocks in $P$
 1: for all basic block $b_i$ do
 2:   add an integer variable $gb_i$ to global variables.
 3:   add statement $gb_i = gb_i + 1$ to $b_i$
 4: end for
 5: for all function $f(a_1, ..., a_m)$ except main in $P$ do
 6:   $LBB$ =Set $\{b'_1, ..., b'_k\} \subset BB$ of basic blocks in $f$
 7:   change the $f(a_1, ..., a_m)$ signature to $f(a_1, ..., a_m, lb_1, ..., lb_k)$.
 8:   for all basic block $b'_i$ do
 9:     add statement $lb_i = lb_i + 1$ to $b'_i$
10:   end for
11:   for all call sites of $f$ in $P$ do
12:     extend the $f$ function call to include a fresh integer for each $lb_i$.
13:   end for
14: end for
```

Figure 4 shows the result of transformation of an implementation of Quick-Sort. `g_bb_count` array stores the basic block counters during an execution of a program. Similarly, `l_bb_count` array stores the function specific basic block counters.

## 3. EXTENDED INVARIANTS AND TESTING

In this section, we discuss possible applications of extended invariants in testing. First we look at extended invariants to understand test suites and the diversity of behaviors that they explore. Then, we discuss potential use of extended invariants in random testing.

### 3.1 Extended Invariants to Measure Test Diversity

Software testing techniques attempt to explore as diverse as possible a range of program behaviors. They usually rely on code coverage criteria to measure the diversity of program behaviors explored by a test case/suite. Traditional code coverage criteria such as statement or branch coverage look at coverage of individual textual components of code, but they ignore possible associations between coverage of different areas of a program. Extended invariants seem to be useful to relate the coverage of different parts of a program.

Figure 4 depicts an implementation of quick sort that was transformed to include basic block count variables. Now, assume the following three inputs to the quicksort program:

```
int[] arrSorted = {1,2,3,4,5,6,7,8,9};
int[] arrReverseSorted = {9,8,7,6,5,4,3,2,1};
int[] arrShuffled= {3,2,3,4,2,6,7,1,9};
```

The corresponding extended behaviors follows.

```
l_bb_count[] one of {[1, 0, 0],[1, 0, 1]}
l_bb_count[] one of {[1, 0, 0],[1, 0, 1],[1, 1, 0]}
l_bb_count[] one of {[1, 0, 0],[1, 0, 1],[1, 1, 1]}
```

`arrShuffled` and `arrReverseSorted` have similar block coverage. It can be observed that each of the extended invariants represent different *coverage behavior* on inputs even though the tests have the same coverage. Thus, it can be justified that all three test cases are needed to provide a diverse test suite.

Test case selection techniques based on operational abstraction [12] or residual branch coverage [18] cannot distinguish these differences and may discard any of above test cases from the test suite. Pavlopoulou and Young propose residual branch coverage for test case selection [18]. In residual branch coverage, a test case is added to a test suite if it covers a new branch that is not covered by any of the already selected test cases. Harder et al. propose selection of test cases that either violate an operational abstraction of program (i.e. potential invariant) or add branch coverage [12]. It seems that extended invariants subsumes this idea. Moreover, extended invariants include patterns of coverage which seem to have been ignored in most efforts.

We believe extended invariants can serve as a metric to measure diversity in test suites. We also stipulate using

```
public class qs{
static int[] g_bb_count = new int[9];
static int partition(int arr[], int left, int right, int l_bb_count[]){
  g_bb_count[0] ++; l_bb_count[0] ++;
  int i = left, j = right;
  int tmp;
  int pivot = arr[left];
  while(i <= j){
    g_bb_count[1] ++; l_bb_count[1] ++;
    while(arr[i] <= j){
      g_bb_count[2] ++; l_bb_count[2] ++;
      i++;
    }
    while (arr[j] > pivot){
      g_bb_count[3] ++; l_bb_count[3] ++;
      j--;
    }
    if(i <= j){
      g_bb_count[4] ++; l_bb_count[4] ++;
      tmp = arr[i];
      arr[i] = arr[j];
      arr[j] = tmp;
      i++;
      j--;
    }
  }
  g_bb_count[5] ++; l_bb_count[5] ++;
  return i;
}
static void quikSort(int arr[], int left, int right, int[] l_bb_count){
  int index = partition(arr, left, right, new int[6]);
  g_bb_count[6] ++; l_bb_count[0] ++;
  if(left < index -1){
    g_bb_count[7] ++; l_bb_count[1] ++;
    quickSort(arr, left, index - 1, new int [3]);
    }
  if(index < right){
    g_bb_count[8] ++; l_bb_count[2] ++;
    quickSort(arr, index, right, new int[3]);
  }
  }
}
```

**Figure 4: Result of transformation of an implementation of quick sort.**

```
for (i = 0; i < SIZE; i++){
  int op = random(3);
  switch(op){
  case 0: top()
          break;
  case 1: pop();
          break;
  case 2: push();
          break;
  }
}
```

**Figure 5: Random test generator for stack.**

extended invariants for test case selection may work better than traditional approaches based on coverage or operational abstraction.

## 3.2 Extended Invariants in Random Testing

Systematic test techniques exploit some information about the program under test (SUT) to divide the input space into partitions, and then they pick samples from the partitions to test the program. Essentially partitions are regions with different failure rates [3]. Thus, success of systematic test techniques relies on (1) appropriate partitioning of data which represents situations that software will face in real world, and (2) choosing good samples from equivalence classes to represent partitions and reveal more diverse behavior of software. Both of these factors require precise information about the SUT. In other words, if the partitioning is based on imprecise information, the effectiveness of systematic testing to reveal errors decreases substantially.

On the other hand, random testing techniques pick inputs randomly. They have shown to be effective to reveal bugs in important complex programs [8,17]. Random testing is well-suited when there is a lack of information about the input space. The effectiveness of random testing highly depends on the configuration of random inputs. Thus, it is important to monitor the random testing process and identify when its continuation does not benefit testing anymore. At such a point, it can be effective to switch to more expensive test techniques such as (dynamic) symbolic execution, or change the configuration of the random tester.

Recall stack example in Section 1 (Figure 3). If maximum size of the array is 64, at least 65 consecutive `push` operations are required to manifest a failure. Figure 5 shows a random tester for the stack data structure. This random tester generates tests of length `SIZE` which fails to detect the error. Suppose we change the `for` loop to `for(i = 0; i < SIZE + 1; i++)`. Now the bug *might* be revealed with probability of $\frac{1}{3^{65}}$! If the bug was not found after a while, extended invariants over test cases could be used to summarize the properties of coverage in the program. They reveal that `bpush - b1pop() < SIZE`. Knowing this fact about test cases, a tester can remove `pop` from the configuration to increase the likelihood of stack overflow.

We think that extended invariants can be used to guide random testing. Traditional test case selection techniques (discussed in Section 3.1) can be used to decide when to stop random testing, but as discussed they are not as precise as extended invariants, moreover, they do not provide clues about the test suite to guide the random testing. Groce et. al propose a different approach in random testing [10]. Instead of using a single configuration that includes all test features, they propose to create several different configurations which each include a random subset of features. Their approach also does not suggest a condition to stop random testing, or to guide it.

## 4. EXTENDED INVARIANTS AND FAULT LOCALIZATION

Fault localization is a part of software debugging that focuses on finding the location of faults in the program. Several techniques have been proposed for fault localizations. Spectrum-based fault localization techniques contrast code coverage (e.g. statement coverage [13], block coverage [20]) in failing runs and passing runs to find suspicious statements. Liblit et. al propose cooperative bug isolation [15,16] which uses a statistical framework to find suspicious predicates in programs that tend to appear (be held true) more in failing executions than passing executions. Some techniques compare dynamic slices of failing executions and passing execution to find the fault, e.g. [1,11]. Cleve and Zeller have used cause transitions to isolate suspicious statements in the program [4].

Pytlik et al. attempted to use invariants for fault localization [19]. They contrast dynamic invariants in failing traces with passing traces. They applied their technique to the *Tcas* and *print_tokens* Siemens subject programs and failed to find meaningful results. However, when we used extended invariants instead of traditional invariants on a version of `Tcas`, we were been able to spot a difference between extended invariants of passing executions and failing executions that corresponds to the location of fault. Therefore, it seems that extended invariants can be useful for fault localization.

## 5. CONCLUSION

In this paper we introduced the notion of extended invariants. They state relationships between coverage of different parts of programs together or between execution and program variables. We speculated about the potential use of extended invariants in software testing and fault localization. The idea needs to be examined thoroughly for effectiveness.

Exploiting extended invariants suffers from common drawbacks of dynamic invariant detection techniques: performance and irrelevant predicates. The current invariant detectors are slow for large programs due to heavy profiling of the program execution, and exhaustive search for identifying relationships between all variables. Moreover, they return a lot of *uninteresting* invariants.

We believe extended invariants can help in understanding test suites and they deserve more investigation. Therefore, in the future, we would like to explore capabilities of extended invariants in test suite minimization and fault localization.

## 6. REFERENCES

[1] Agrawal, H., De Millo, R., and Spafford, E. An execution-backtracking approach to debugging. *Software, IEEE 8*, 3 (may 1991), 21 –26.

[2] Ammons, G., Bodík, R., and Larus, J. R. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2002), POPL '02, ACM, pp. 4–16.

[3] Boland, P. J., Singh, H., and Cukic, B. Comparing partition and random testing via majorization and schur functions. *IEEE Trans. Softw. Eng. 29* (January 2003), 88–94.

[4] Cleve, H., and Zeller, A. Locating causes of program failures. In *Proceedings of the 27th international conference on Software engineering* (New York, NY, USA, 2005), ICSE '05, ACM, pp. 342–351.

[5] Cousot, P., and Halbwachs, N. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (New York, NY, USA, 1978), POPL '78, ACM, pp. 84–96.

[6] Csallner, C., and Smaragdakis, Y. Dynamically discovering likely interface invariants. In *Proceedings of the 28th international conference on Software engineering* (New York, NY, USA, 2006), ICSE '06, ACM, pp. 861–864.

[7] Dallmeier, V., Lindig, C., Wasylkowski, A., and Zeller, A. Mining object behavior with adabu. In *Proceedings of the 2006 international workshop on Dynamic systems analysis* (New York, NY, USA, 2006), WODA '06, ACM, pp. 17–24.

[8] Duran, J. W., and Ntafos, S. C. An evaluation of random testing. *IEEE Trans. Software Eng. 10*, 4 (1984), 438–444.

[9] Ernst, M., Cockrell, J., Griswold, W., and Notkin, D. Dynamically discovering likely program invariants to support program evolution. *Software Engineering, IEEE Transactions on 27*, 2 (feb 2001), 99 –123.

[10] Groce, A., Zhang, C., Eide, E., and Regehr, Y. C. J. Swarm testing. In *ISSTA* (2012). (In submission).

[11] Gupta, N., He, H., Zhang, X., and Gupta, R. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering* (New York, NY, USA, 2005), ASE '05, ACM, pp. 263–272.

[12] Harder, M., Mellen, J., and Ernst, M. Improving test suites via operational abstraction. In *Software Engineering, 2003. Proceedings. 25th International Conference on* (may 2003), pp. 60 – 71.

[13] Jones, J. A., Harrold, M. J., and Stasko, J. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering* (New York, NY, USA, 2002), ICSE '02, ACM, pp. 467–477.

[14] Kovacs, L., and Voronkov, A. Finding loop invariants for programs over arrays using a theorem prover. In *Fundamental Approaches to Software Engineering*, M. Chechik and M. Wirsing, Eds., vol. 5503 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2009, pp. 470–485. 10.1007/978-3-642-00593-0.

[15] Liblit, B., Aiken, A., Zheng, A. X., and Jordan, M. I. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation* (New York, NY, USA, 2003), PLDI '03, ACM, pp. 141–154.

[16] Liblit, B., Naik, M., Zheng, A. X., Aiken, A., and Jordan, M. I. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2005), PLDI '05, ACM, pp. 15–26.

[17] Miller, B. P., Cooksey, G., and Moore, F. An empirical study of the robustness of macos applications using random testing. In *Proceedings of the 1st international workshop on Random testing* (New York, NY, USA, 2006), RT '06, ACM, pp. 46–54.

[18] Pavlopoulou, C., and Young, M. Residual test coverage monitoring. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on* (may 1999), pp. 277 –284.

[19] Pytlik, B., Renieris, M., Krishnamurthi, S., and Reiss, S. Automated fault localization using potential invariants. In *AADEBUG* (2003).

[20] Renieres, M., and Reiss, S. Fault localization with nearest neighbor queries. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on* (oct. 2003), pp. 30 – 39.

[21] Whaley, J., Martin, M. C., and Lam, M. S. Automatic extraction of object-oriented component interfaces. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis* (New York, NY, USA, 2002), ISSTA '02, ACM, pp. 218–228.