Feedback-Driven Dynamic Invariant Discovery

Lingming Zhang
University of Texas, USA
zhanglm@utexas.edu

Guowei Yang
Texas State University, USA
gyang@txstate.edu

Neha Rungta
NASA Ames Center, USA
neha.s.rungta@nasa.gov

Suzette Person
NASA Langley Center, USA
suzette.person@nasa.gov

Sarfraz Khurshid
University of Texas, USA
khurshid@utexas.edu

ABSTRACT

Program invariants can help software developers identify program properties that must be preserved as the software evolves, however, formulating correct invariants can be challenging. In this work, we introduce *iDiscovery*, a technique which leverages symbolic execution to improve the quality of dynamically discovered invariants computed by Daikon. Candidate invariants generated by Daikon are synthesized into assertions and instrumented onto the program. The instrumented code is executed symbolically to generate new test cases that are fed back to Daikon to help further refine the set of candidate invariants. This feedback loop is executed until a fix-point is reached. To mitigate the cost of symbolic execution, we present optimizations to prune the symbolic state space and to reduce the complexity of the generated path conditions. We also leverage recent advances in constraint solution reuse techniques to avoid computing results for the same constraints across iterations. Experimental results show that iDiscovery converges to a set of higher quality invariants compared to the initial set of candidate invariants in a small number of iterations.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Verification, Experimentation

Keywords

Invariant generation, Symbolic execution, Model checking

1. INTRODUCTION

While researchers have long recognized the value of documenting properties of code, e.g., as behavioral specifications [20, 24], in the development and maintenance of correct software systems, writing specifications for non-trivial functional correctness properties is quite challenging and remains a largely manual process. Consequently, it is rare to find software systems with accompanying specifications that are up-to-date. This creates challenges not only for reasoning about correctness, but it also hinders the development of effective testing and analysis techniques that leverage specifications. An example of the latter—in our recent work on developing an incremental analysis to support software evolution [36], we were not able to find any existing artifacts that included multiple versions of expected properties corresponding to the different versions of code as it evolved.

Techniques to automatically extract properties of code, say to create *likely* specifications, provide users with an initial set of specifications from which additional, or more precise, specifications can be derived. Such techniques have been investigated by a number of researchers over the last several decades using a variety of static [5, 8, 22, 32] and dynamic [9, 12] techniques. While some of these techniques are efficient and applicable to real-world programs, the generated properties may not accurately characterize the program's behavior [28, 31]: (1) they may contain many incorrect or imprecise properties; and (2) they may miss many valid properties. The focus of this paper is on extracting higher quality properties by addressing both of these issues.

Dynamic invariant discovery is a well-known approach for generating invariants. It was pioneered over a decade ago by Daikon [12] and provided the foundation for a number of subsequent techniques [1, 3, 23]. Dynamic invariant discovery has a simple and practical basis: execute a program against a given test suite, monitor the executions at control-points of interest to generate program traces, and check the program states collected in the traces to validate a set of invariants to see which ones hold and compute candidate program invariants [12]. The number of the invariants inferred by these tools and their correctness is often highly dependent on the quality of the test suites used [12, 28]. We are unaware of any current work on how to construct a test-suite that would enable dynamic invariant inference techniques to generate better program invariants.

In this work, we introduce *iDiscovery*, a technique that employs a feedback loop [35] between *symbolic execution* [6, 19] and dynamic invariant discovery to infer more accurate and complete invariants until a fix-point is reached. Our embodiment of iDiscovery uses Daikon as the dynamic invariant inference technique. In each iteration, iDiscovery transforms candidate invariants inferred by Daikon into assertions that

ISSTA'14, July 21–25, 2014, San Jose, CA, USA
Copyright 2014 ACM 978-1-4503-2645-2/14/07...$15.00
http://dx.doi.org/10.1145/2610384.2610389

are instrumented in the program. The instrumented program is analyzed with symbolic execution to generate additional tests to augment the initial test suite provided to Daikon. The key intuition behind iDiscovery is that the constraints generated on the synthesized assertions provide additional test inputs that can refute incorrect/imprecise invariants or expose new invariants. Therefore, when the new inputs are used to augment the previous test suite, dynamic invariant discovery will be based on a richer set of program executions enabling discovery of higher quality invariants.

A key element in the performance cost of iDiscovery is the high cost of symbolic execution. To mitigate the cost of symbolic execution, iDiscovery provides two optimizations: *assertion separation* and *violation restriction*. Assertion separation leverages the underlying search engines' non-deterministic choice and backtracking features to focus symbolic execution on checking one assertion at a time. This generates fewer path conditions that are overall less complex and in turn reduces the constraint solving time. Violation restriction uses the state of the underlying execution engine to generate at most one violation of each assertion, since one test that reveals the violation is sufficient to form a counter-example. Both of these configurations can be applied in tandem to amortize the cost of iDiscovery. Moreover, iDiscovery's feedback loop, which repeatedly uses symbolic execution, involves iteratively checking the *same* code against *different* sets of properties. Thus, our approach offers a substantial opportunity to memoise and re-use the results of symbolic execution from previous iterations to further reduce its cost [4, 33, 37]. Our iDiscovery tool uses the Green library [33] for the Symbolic PathFinder [26] to re-use constraint solving results, which together with assertion separation and violation restriction, minimize the overall symbolic execution cost for improving dynamic invariant discovery.

We make the following contributions:

- **Dynamic invariant discovery and symbolic execution**. We combine Daikon and symbolic execution to generate test inputs that allow Daikon to discover higher quality invariants.
- **iDiscovery**. We present our core technique iDiscovery, which embodies our idea to dynamically discover invariants for Java programs.
- **Optimizations**. We present two optimizations: (a) assertion separation and (b) violation restriction that enhance the efficiency of iDiscovery.
- **Evaluation**. We present an experimental evaluation using a suite of Java programs that have been used in previous work on invariant discovery and symbolic execution. The experimental results show that iDiscovery converges to a set of higher quality invariants than the initial set of invariants computed by Daikon in a small number of iterations.

2. MOTIVATING EXAMPLE

We demonstrate our feedback-driven technique, iDiscovery, for inferring and refining program invariants using the example in Fig. 1. In Fig. 1 we first present snippets of code from the Wheel Brake System (WBS) [29] and the tests used to generate the initial invariants. We then show the artifacts produced by our technique: 1) the assertions generated based on the Daikon invariants, 2) the new tests generated by symbolic execution at each iteration of our technique, and 3) the updated assertions based on the program invariants generated by Daikon at each iteration of iDiscovery. The assertions and tests shown in Fig. 1 are a subset of the assertions and tests generated by iDiscovery for the WBS

```
public class WBS {
  ...
  BSCU_SystemModeSelCmd_rlt_PRE = 0;
  Nor_Pressure = 0;

  public void update(int PedalPos, boolean AutoBrake,
      boolean Skid){
    ....
    rlt_PRE2 = 100;
    ...
    if (PedalPos == 0)
        BSCU_Command_PedalCommand_Switch1 = 0;
    else if (PedalPos == 1)
        BSCU_Command_PedalCommand_Switch1 = 1;
    else if (PedalPos == 2)
        BSCU_Command_PedalCommand_Switch1 = 2;
    else if (PedalPos == 3)
        BSCU_Command_PedalCommand_Switch1 = 3;
    else if (PedalPos == 4)
        BSCU_Command_PedalCommand_Switch1 = 4;
    else BSCU_Command_PedalCommand_Switch1 = 0;
    ...
```

```
/** Subset of the initial test inputs **/
PedalPos = 0, AutoBrake = false, Skid = false;
PedalPos = 1, AutoBrake = false, Skid = false;
PedalPos = 2, AutoBrake = false, Skid = false;
PedalPos = 3, AutoBrake = false, Skid = false;
PedalPos = 4, AutoBrake = false, Skid = false;
PedalPos = 5, AutoBrake = false, Skid = false;
```

```
/** Assertions generated from initial invariants **/
A1: assert(this.rlt_PRE2 > PedalPos);
A2: assert(PedalPos >= 0);
A3: assert(this.BSCU_SystemModeSelCmd_rlt_PRE <=
     PedalPos);
A4: assert(this.rlt_PRE2 == 100);
A5: assert(this.BSCU_SystemModeSelCmd_rlt_PRE ==
     this.Nor_Pressure);
A6: assert(this.BSCU_SystemModeSelCmd_rlt_PRE ==
     this.BSCU_rlt_PRE1);
```

```
/** New tests generated in the first iteration **/
PedalPos = 100, AutoBrake = false, Skid = false;
PedalPos = -1000, AutoBrake = false, Skid = false;
```

```
/** Assertions updated in the first refinement **/
A4: assert(this.rlt_PRE2 == 100);
A5: assert(this.BSCU_SystemModeSelCmd_rlt_PRE ==
     this.Nor_Pressure);
A6: assert(this.BSCU_SystemModeSelCmd_rlt_PRE ==
     this.BSCU_rlt_PRE1);
A7: assert(this.rlt_PRE2 >= PedalPos);
```

```
/** New test generated in the second iteration **/
PedalPos = 101, AutoBrake = false, Skid = false;
```

```
/** Assertion updated in the second refinement **/
A4: assert(this.rlt_PRE2 == 100);
A5: assert(this.BSCU_SystemModeSelCmd_rlt_PRE ==
     this.Nor_Pressure);
A6: assert(this.BSCU_SystemModeSelCmd_rlt_PRE ==
     this.BSCU_rlt_PRE1);
```

Figure 1: Parts of the WBS example, the test cases generated by symbolic execution at each step, and the assertions generated from the Daikon invariants.

program. The complete results for the WBS example are discussed in Section 5.

The `WBS` program contains an `update` method with three input variables: `PedalPos`, `AutoBrake`, and `Skid`. These input variables are treated as symbolic values while all other fields in the systems are initialized with concrete values. An initial set of test inputs is generated for the `update` method using symbolic execution. A subset of these tests is shown in Fig. 1; in these tests there are six possible values for the input parameter `PedalPos`: 0, 1, 2, 3, 4, 5 while the `AutoBrake` and `Skid` variables are set to *false*. The six input values of `PedalPos` cover all of the branches shown in the `update` method in Fig. 1. Daikon monitors the concrete execution traces of the WBS program that are generated by the initial test inputs to infer the candidate invariants related to the `PedalPos` input variable and other fields in the `WBS` program. These invariants are transformed into Java assertions $A1$ through $A6$ as shown in Fig. 1.

Symbolic execution is performed on the original program instrumented with assertions $A1$-$A6$. Symbolic execution of the object code (i.e. Java bytecode) corresponding to the assertions creates additional constraints on the path conditions of the program. The SMT solver now generates new test inputs for the program by solving these path conditions. For example, the constraint `PedalPos` < 0 is generated by symbolic execution along the *false* branch of the conditional branch for assertion $A2$: `PedalPos` ≥ 0. The SMT solver provides a solution of -1000 for `PedalPos` based on this new constraint; this in turn results in a new test input. Similarly, the constraints `this.rlt_PRE2` $== 100 \wedge$ `this.rlt_PRE2` $\leq$ `PedalPos` are generated by symbolic execution along the true branch of assertion $A4$ and the false branch of assertion $A1$. The SMT solver generates a solution 100 for `PedalPos` to satisfy these constraints, which is used to create another new test input value.

The values of -1000 and 100 are provided as test inputs for the `PedalPos` input variable while the `AutoBrake` and `Skid` are set to *false*. Using the test traces generated by these *new* test inputs, Daikon refines the set of candidate invariants. Note that the test traces are generated based the original program *without* the assertions generated from invariants. Using information from the new traces, Daikon deletes (does not generate) assertions $A1$, $A2$, and $A3$; retains $A4$, $A5$, $A6$; and adds $A7$.

In the second iteration of iDiscovery, the invariants are again transformed into Java assertions and instrumented on the original program. Symbolic execution generates new constraints, `this.rlt_PRE2` $== 100 \wedge$ `this.rlt_PRE2` $<$ `PedalPos`, for assertions $A4$ (true branch) and $A7$ (false branch). These constraints cause the SMT solver to compute a solution of 101 for the `PedalPos` variable. A new test input is created with this value for `PedalPos`, and `AutoBrake` and `Skid` are set to *false*. Daikon then refines the candidate invariants using the program trace for the new input. In the second iteration, Daikon deletes assertion $A7$ and does not add any new assertions. For the code-snippet shown in Fig. 1 this represents a fix-point. For this example, iDiscovery successfully refutes 3 incorrect/imprecise invariants computed by Daikon (50% of the initial invariants).

3. APPROACH

In this section, we present the details of our iDiscovery approach. Given a program and an initial test suite, iDiscovery

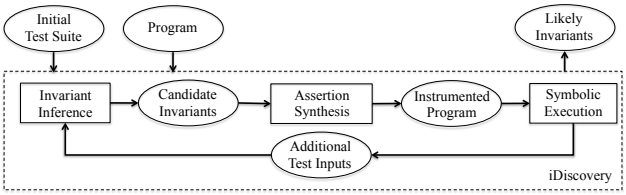


Figure 2: iDiscovery Overview.

automatically and iteratively applies two techniques: 1) dynamic invariant inference, e.g., Daikon [12], and 2) symbolic execution [6, 19], to infer program invariants for Java programs. An overview of the iDiscovery technique is shown in Figure 2. iDiscovery transforms the program invariants inferred by Daikon into Java expressions within `assert` statements. All of the paths in the original program code and the synthesized assertions are explored using symbolic execution in an attempt to generate additional test inputs. For every assertion, constraints encoding the true and the false branches of the assertions are added to the path condition computed by symbolic execution. The resulting path conditions are solved using an off-the-shelf decision procedure, in an attempt to generate *additional* test inputs. The additional test inputs, together with the initial test suite, are then provided to Daikon in the next iteration of iDiscovery. Note that iDiscovery makes no modifications to either the invariant inference technique or the symbolic execution algorithm; each is treated as a black-box by iDiscovery.

During each iteration of iDiscovery, the set of inferred invariants is refined until a fix-point is reached, e.g., the inferred invariants are unchanged with respect to the previous iteration of the algorithm. An invariant is refuted by Daikon when a counter example is found. Therefore, theoretically, all the invariants refuted in each iteration of iDiscovery are either incorrect or imprecise, because the symbolic execution engine of iDiscovery is able to generate counter examples for those invariants. New invariants can also be augmented in each iteration of iDiscovery because more evidence support can be found by the additional tests generated by iDiscovery. Note that similar to other dynamic invariant discovery techniques [9, 12, 16], the invariants discovered by iDiscovery may be unsound or incomplete (due to the limitations of Daikon). However, iDiscovery can generate valuable tests to guide Daikon to refute incorrect or imprecise invariants, or augment un-observed invariants.

Formally, the overall iDiscovery algorithm is shown in Algorithm. 1. The inputs to the algorithm are the program under analysis $\mathcal{P}$, and an initial test suite $\mathcal{T}_{init}$. The set of inferred invariants is initialized to $\emptyset$ at line 2. The set of execution traces used by the invariant inference algorithm, $\textsc{InvarInfer}(\Gamma)$, is computed by invoking function $\textsc{TraceExec}(\mathcal{P}, \mathcal{T}_{init})$ at line 3. Recall that iDiscovery treats the invariant inference algorithm as a black box, providing only the program and the tests as input. During each iteration of the loop starting at line 4, the iDiscovery algorithm first invokes the invariant inference algorithm to generate a set of candidate invariants. If the set of inferred invariants is equivalent to the previous set of inferred invariants, iDiscovery has reached a fix-point and exits the loop at line 8. If the set of candidate invariants is changed, then at line 9, $\mathcal{P}$ is instrumented with code to check the invariants, e.g., with `assert` statements, and at line 10 symbolic execution of the instrumented version of $\mathcal{P}$ is performed. The test

Algorithm 1: iDiscovery Algorithm

```
   Input: Program P, test suite T_init
   Output: Invariants I
1  begin
       // Initialize the invariant set as empty
2      I ← ∅
       // Record the execution trace for the original
          test suite
3      Γ ← TraceExec(P, T_init)
4      while true do
5          I_old ← I
           // Infer invariants based on the test
              execution traces
6          I ← InvarInfer(Γ)
           // Terminate the algorithm if the fix-point is
              reached
7          if I = I_old then
8              break
           // Annotate program under test with property
              checks
9          P' ← PropertyInstr(P, I)
           // Use symbolic execution to generate
              additional test inputs
10         T_sym ← SymbcExec(P')
           // Expand the test traces with the generated
              tests
11         Γ ← TraceExec(P, T_sym) ∪ Γ
12     return I // Return the final set of inferred
          invariants
```

inputs computed by symbolic execution are used to augment the set of execution traces used by the invariant inference algorithm in the next iteration of the loop. iDiscovery includes test inputs for all paths explored by symbolic execution (non-violating and counter-examples). When a fix-point is reached, the loop terminates and the inferred invariants are returned at line 12.

3.1 Complexity Analysis

Symbolic execution is a powerful analysis technique; however, scalability is an issue because of the large number of execution paths generated during the symbolic analysis. This is known as the path explosion problem, and is further exacerbated in iDiscovery by the addition of the `assert` statements to the code. Symbolic execution generates a set of path conditions, Φ, for the program under test that does not contain any synthesized assertions from invariants. During the symbolic execution in iDiscovery, a path condition, $\phi \in \Phi$, for the original program is further expanded by adding constraints from the synthesized assertions.

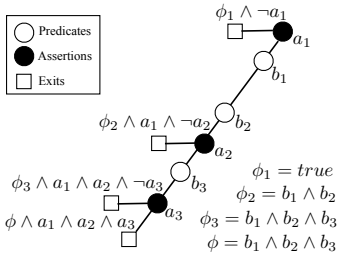
Suppose iDiscovery instruments m synthesized assertions in the locations specified by Daikon (e.g., beginning of program methods, or at the end of program methods). For the ease of presentation, assume the synthesized assertions are not contained within a loop, then for each path condition, $\phi \in \Phi$, there can be at most $m+1$ potential path conditions generated in iDiscovery (when ϕ reaches all the m assertions). Note that iDiscovery is applicable even when synthesized assertions are contained in loops; in that case, the total number of expanded paths for ϕ can be $\Sigma_{1 \leq i \leq m} |\gamma_\phi(a_i)| + 1$, where $|\gamma_\phi(a_i)| \geq 1$ denotes the number of instance assertions unrolled from loops for a_i in path ϕ ($|\gamma_\phi(a_i)| = 1$ for assertions not in loops). The set of path conditions containing constraints from ϕ and the synthesized assertions is described as $\{\phi_k \wedge (\bigwedge_{1 \leq j \leq k-1} a_j) \wedge \neg a_k | 1 \leq k \leq m+1\}$; where each a_i for $1 \leq i \leq m$ is the predicate constraint extracted from the corresponding `assert` statement, each ϕ_i for $1 \leq i \leq m$ is a prefix of the path condition ϕ that reaches a_i, and $\phi_{m+1} = \phi \wedge a_{m+1} = false$ so that the corresponding expanded path specifies the path that satisfies all assertions when $k = m+1$. To illustrate, Fig. 3(a) shows an execution path annotated with the path condition ϕ as well as the synthesized assertions. In Fig. 3, white nodes represent constraints in the original program, the black nodes denote the synthesized assertions, and the square nodes denote the program exit points. The path conditions generated due to the synthesized assertions are annotated at the corresponding exit nodes. In this way, the original program path ϕ is expanded into 3+1=4 different paths after instrumented with 3 assertions. Note that ϕ_1 is *true* because a_1 is instrumented at the beginning of the program. In total, symbolic execution of the program instrumented with synthesized assertions can generate up to $|\Phi| * (m+1)$ path conditions.

3.2 Cost Reduction

The path explosion problem in iDiscovery can be divided into two dimensions: (1) checking assertions along each original program path can be expensive when there are many assertions because the constraints accumulate over all the assertions, (2) the same assertions are repeatedly checked by different program paths. We discuss two optimizations that allow us to mitigate the path explosion problem in the above two dimensions respectively: (1) *assertion separation* where we direct symbolic execution to check one instrumented assertion in isolation at a time for each original program path, and (b) *violation restriction* to cause symbolic execution to generate at most one violation of each instrumented assertion across different program paths. Figure 3(b) shows the set of expanded paths explored for the original path ϕ during symbolic execution in iDiscovery without assertion separation in the second column and with assertion separation in the third column. As shown in the second column, each path condition contains constraints that represent the true branches of assertions (satisfied) with a constraint from the false branch of an assertion (refuted). The reason is that symbolic execution will treat assertions equally with branches. By simply checking constraints in the third column we can reduce the cost of symbolic execution. In addition, symbolic execution in the default configuration of iDiscovery attempts to find all possible violations of an assertion, since it can appear in multiple paths, each of which will try to violate the assertion. In order to mitigate this cost we configure iDiscovery to generate test inputs for only the first violation of an assertion. In the violation restriction configuration, iDiscovery keeps track of assertions that have been violated since the start of symbolic execution, and prunes paths leading to these violated assertions.

3.3 Discussion

The two optimizations presented in this work reduce the number of path conditions generated in iDiscovery, however, they may also cause iDiscovery to generate fewer additional test inputs. The optimizations enable the analysis of artifacts that are too large to be analyzed with the standard iDiscovery algorithm. Therefore, in Section 5, we present a detailed study to evaluate the optimizations for both effectiveness and efficiency on various subject systems. Further-



Step	No Separation	Separation
1	$\phi_1 \wedge \neg a_1$	$\phi_1 \wedge \neg a_1$
2	$\phi_2 \wedge a_1 \wedge \neg a_2$	$\phi_2 \wedge \neg a_2$
3	$\phi_3 \wedge a_1 \wedge a_2 \wedge \neg a_3$	$\phi_3 \wedge \neg a_3$
...	...	...
m	$\phi_m \wedge a_1 \wedge a_2 \wedge ... \wedge a_{m-1} \wedge \neg a_m$	$\phi_m \wedge \neg a_m$
$m+1$	$\phi \wedge a_1 \wedge a_2 \wedge ... \wedge a_{m-1} \wedge a_m$	ϕ

Figure 3: (a) Example path expanded with assertions (b) Assertion checking

more, various techniques have been developed to reduce the cost of symbolic execution by reusing constraint solutions, e.g., [33, 37]. Given the iterative nature of the iDiscovery algorithm, it is an ideal candidate for constraint re-use solutions. Hence, in this work we leverage an existing constraint solution reuse technique, Green [33] to further mitigate the cost of symbolic execution in iDiscovery.

iDiscovery is guaranteed to terminate if the underlying dynamic invariant inference technique satisfies two key properties: (a) the invariant inference is deterministic, i.e., the same invariants are generated for a given test suite each time and (b) a potential invariant, violated by at least one test in the given suite, is *not* generated as a candidate invariant. In each iteration of iDiscovery, the set of invariants inferred may increase or decrease. A refuted invariant, however, is never generated again by Daikon [11]. Furthermore, there is an upper bound to the possible candidate invariants that Daikon can generate for a given program. These elements lead us to believe that iDiscovery when used with Daikon will terminate for most programs (Also confirmed by our experimental evaluation in Section 5).

4. DETAILED OPTIMIZATION

To evaluate our technique we implement iDiscovery using Daikon to dynamically infer program invariants and Symbolic PathFinder (SPF) to perform symbolic execution. iDiscovery takes advantage of SPF's support for non-deterministic choices within its underlying engine (Java PathFinder—JPF) and leverages the Green extension for reusing constraint solutions. We use CVC3 and CORAL as the underlying solvers to check satisfiability of path conditions and generate solutions. The main code for coordinating and combining Daikon and SPF is implemented using bash scripts. The instrumentation of Daikon invariants into the program under test is implemented using the `Eclipse JDT` framework.

To implement the assertion separation optimization, iDiscovery automatically generates a wrapper for each assertion as shown in Figure 4(a). Each synthesized assertion, a_i, is guarded by `Verify.getBoolean()` which is a modeling primitive in JPF that non-deterministically generates both *true* and *false* choices during symbolic execution. When `Verify.getBoolean()` returns *false*, assertion a_i is *not* checked and the search continues executing the instruction that follows the `if` statement. When `Verify.getBoolean()` returns *true*, assertion a_i is checked. The `Verify.ignoreIf(true)` is another modeling primitive in JPF, which forces the search to backtrack. Here it is used to force the search to backtrack after checking a_i. The combination of these two modeling primitives enable iDiscovery to check each assertion separately as shown in the `Separation` column of Figure 3(b). Consider the example in Figure 4(b), where the dashed nodes represent the new choices from `Verify.getBoolean()` that control whether to check an assertion. The symbolic execution engine generates and explores non-deterministic choices. The `Verify.ignore(true)` statement forces symbolic execution to backtrack to the previous choice point when the corresponding assertion has been checked. The pruned parts of the search tree are enclosed in dashed lines for each assertion. The final set of paths explored by symbolic execution are annotated with the path conditions at the corresponding exit nodes. Note that the modeling primitives are not considered as path conditions and the final path conditions generated are exactly the same as the ones shown in the `Separation` column of Figure 3(b).

To restrict the number of times an assertion is violated to at most once, iDiscovery also uses modeling primitives in JPF to wrap assertions and record the assertion checking information. For each `assert` statement, iDiscovery automatically creates a wrapper as shown below:

```
if(Verify.getBoolean()){ //non-deterministic boolean
     choice
  if (Verify.getCounter(i) < 1 && !(a_i)) {
    Verify.incrementCounter(i);
    throw new AssertionError(''ith assertion failed'');
  }
  Verify.ignoreIf(true); //backtrack
}
```

The code, `Verify.getCounter(i)`, controls whether assertion a_i is checked ($1 \leq i \leq m$ and m is the total number of synthesized assertions). When an assertion is violated on some execution path, the counter for that assertion is increased by 1, indicating the assertion has been violated during the search and does not need to be checked again. In the best case scenario, if the first path violates all the assertions then the pruning can reduce the number of assertion checks from $|\Phi| * (m+1)$ to $|\Phi| + m$. The reason is that only the first path that reaches the assertions will be explored $m+1$ times for all m assertions, while the other $|\Phi| - 1$ paths will only be checked once each. In the worst case scenario, if none of the paths lead to an assertion violation then no paths will be pruned.

5. EVALUATION

5.1 Artifacts

We evaluate iDiscovery on four Java artifacts. *1. Traffic Anti-Collision Avoidance System (TCAS)* is an aircraft collision avoidance system consisting of one class and approximately 150 lines of code. We use three randomly selected versions of TCAS from the Software Infrastructure Repository (SIR) [30]. We inline all functions in TCAS into the `TCAS.alt_sep_test` method, and apply iDiscovery on that method. *2. Wheel Brake System (WBS)* is a synchronous

```
//non-deterministic boolean
     choice
if(Verify.getBoolean()){
  //assertion checking
  assert(a_i);
  //backtrack
  Verify.ignoreIf(true);
}
```

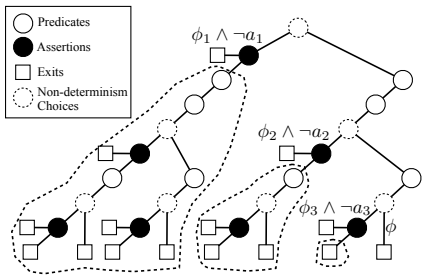


Figure 4: (a) Assertion separation wrapper (b) Assertion separation backtrack tree

reactive component derived from the WBS case example found in ARP 4761 [29]. It consists of a single class and 231 lines of code. We inline all functions in WBS into the `WBS.update` method, and apply iDiscovery on that method. *3. Altitude Switch (ASW)* is a synchronous reactive component from the avionics domain. The ASW consists of a single class and 610 lines of code. We apply iDiscovery on the `asw.Main0` method, which implements the main functionality in ASW. *4. The Apollo Lunar Autopilot (Apollo)* is a model created by an engineer from the Apollo Lunar Module digital autopilot team. We apply iDiscovery on the `rjc.MainSymbolic` method, which is the main method that invokes all other methods in Apollo. Apollo contains 2.6 KLOC in 54 classes.

5.2 Experimental Setup

5.2.1 Independent Variables

The independent variables in our study are as follows:

IV1: Different iDiscovery Optimizations. We evaluate four possible configurations of iDiscovery: (a) iDiscovery with no optimizations, (b) iDiscovery with only assertion separation (Optimization-1), (c) iDiscovery with only violation restriction (Optimization-2), (d) iDiscovery with both optimizations.

IV2: Different Solver Configurations. As iDiscovery applies symbolic execution to similar programs, i.e., the same code with different synthesized assertions, at each iteration, the solver's opportunities for constraint solution reuse may affect the efficiency of iDiscovery significantly. We evaluate iDiscovery with and without the Green solver interface [33] to measure the benefits of constraint reuse.

IV3: Different Initial Test Suites. The initial test suite provided to iDiscovery may affect the invariants computed by iDiscovery. We investigate the impact of different initial test suites on iDiscovery: (a) an initial test suite generated by full symbolic execution, (b) an initial test suite with the same size as the test suite generated by symbolic execution but generated using a random test generator, (c) initial test suites of varying sizes (e.g., 5%, 10%, 15%, and 20%) of randomly generated tests.

We set a time-out limit of 20 hours for each of our experimental settings. All of the experiments were performed on a Dell machine with an Intel i7 Quad-Core processor, 8G RAM, and Ubuntu 12.04 64-bit version OS.

5.2.2 Dependent Variables

DV1: Effectiveness. For each configuration of iDiscovery, we collect the number of refuted invariants (indicating iDiscovery's effectiveness in falsifying incorrect or imprecise invariants), and added invariants (indicating iDiscovery's effectiveness at augmenting new invariants).

DV2: Efficiency. For each configuration of iDiscovery, we collect the time costs for test trace collection, invariant inference, and test augmentation time using symbolic execution at each iteration.

5.3 Results and Analysis

Table 1 shows the detailed experimental results for the iDiscovery configuration without optimizations using an initial test suite generated by symbolic execution. The first column lists the artifact names. The second column lists the iDiscovery iteration. For each iteration, column 3 lists the number of invariants generated by Daikon, and columns 4 and 5 list the number of deleted and augmented invariants respectively, with respect to the original set of invariants generated by Daikon in the first iteration. Columns 6 and 7 list the execution time costs (in seconds) for computing the execution traces and the invariant inference time respectively, for each iteration. Columns 8 to 12 list the number of test cases generated by symbolic execution and the cost of symbolic execution to analyze the code instrumented with the invariants generated in each iteration: the number of solvers calls, the number of backtracked states, the maximum memory consumption, and the symbolic execution time[1]. We only show the iterations up to the point where iDiscovery reaches a fix-point. Similarly, Table 2 shows the detailed results for iDiscovery using both optimizations (assertion separation and violation restriction) with the initial test suite generated by symbolic execution.

The results in Table 1 show that for the TCAS and WBS artifacts, iDiscovery is able to refine invariants effectively without any optimizations. For the ASW and Apollo artifacts, however, iDiscovery times out after 20 hours. For the first version of TCAS, iDiscovery successfully falsified 71.50% of the original invariants (148 out of 207), indicating that the additional test cases generated by symbolic execution helped Daikon falsify invariants. Theoretically, Daikon falsifies invariants only when it has counter-example(s), indicating that all falsified invariants are correctly deleted. In addition, the final set of augmented invariants are quite small in number (e.g., only 2 for Apollo, and 0 for all other subjects). The reason is that the initial test suites generated by symbolic execution already give enough evidence support for the possible candidate invariants, making iDiscovery mainly refute invariants in this case (also confirmed by our later study on different initial test suites).

iDiscovery reaches a fix-point in a relatively small number of iterations. In the subjects where iDiscovery termi-

[1]The symbolic execution time for WBS is sometimes 0 because the time is too small to be captured by the SPF timer.

Table 1: Experimental results for iDiscovery without optimizations

Subjects	Iter.	Invariants			Invariant Cost		Symbolic Execution Cost				
		Num	Del	New	Tracing	Inference	Tests	SCalls	States	Memory	Time
TCAS-v1	1	207	0	0	40s	11s	183	10066	10067	129MB	250s
	2	79	132	4	109s	13s	193	2894	2895	129MB	44s
	3	63	144	0	115s	15s	74	2342	2343	118MB	28s
	4	59	148	0	44s	16s	72	2090	2091	118MB	22s
TCAS-v2	1	149	0	0	100s	13s	445	13944	13945	126MB	348s
	2	78	77	6	265s	17s	299	5604	5605	238MB	93s
	3	63	86	0	178s	20s	167	5404	5405	128MB	64s
	4	59	90	0	99s	22s	165	4752	4753	127MB	50s
TCAS-v3	1	149	0	0	218s	16s	483	14804	14805	128MB	356s
	2	78	77	6	285s	20s	321	5830	5831	128MB	87s
	3	63	86	0	189s	23s	184	5750	5751	127MB	63s
	4	59	90	0	109s	25s	182	5022	5023	126MB	46s
WBS	1	15	0	0	11s	2s	26	100	101	118MB	0s
	2	12	4	1	13s	3s	25	48	49	118MB	0s
	3	11	4	0	12s	3s	24	46	47	118MB	0s
ASW	1	55	0	0	300s	10s	–	–	–	–	TimeOut
Apollo	1	269	0	0	171s	59s	1677	26714	26716	2130MB	6058s
	2	245	26	2	3554s	560s	2756	43984	43986	2120MB	10869s
	3	215	56	2	5850s	1371s	2262	34570	34572	2181MB	8083s
	4	187	84	2	4807s	2036s	–	–	–	–	TimeOut

Table 2: Experimental results for iDiscovery with both optimizations

Subjects	Iter.	Invariants			Invariant Cost		Symbolic Execution Cost				
		Num	Del	New	Tracing	Inference	Tests	SCalls	States	Memory	Time
TCAS-v1	1	207	0	0	40s	11s	178	3538	20835	121MB	45s
	2	59	148	0	106s	12s	72	2072	8353	120MB	24s
TCAS-v2	1	208	0	0	100s	13s	279	7644	50143	121MB	93s
	2	59	149	0	166s	14s	172	4656	20137	119MB	51s
TCAS-v3	1	149	0	0	218s	16s	447	11562	82323	120MB	129s
	2	59	90	0	265s	20s	372	8940	42821	119MB	91s
WBS	1	15	0	0	12s	2s	28	94	539	118MB	1s
	2	12	4	1	14s	3s	25	48	441	118MB	0s
	3	11	4	0	12s	3s	24	46	437	118MB	0s
ASW	1	55	0	0	301s	10s	531	4809	44154	2080MB	843s
	2	42	13	0	316s	15s	504	4580	36857	2091MB	802s
Apollo	1	269	0	0	159s	56s	82	704	802	1105MB	34s
	2	213	58	2	161s	79s	35	303	401	1718MB	17s
	3	185	86	2	68s	88s	14	470	716	763MB	143s
	4	175	96	2	27s	91s	0	348	1394	898MB	135s

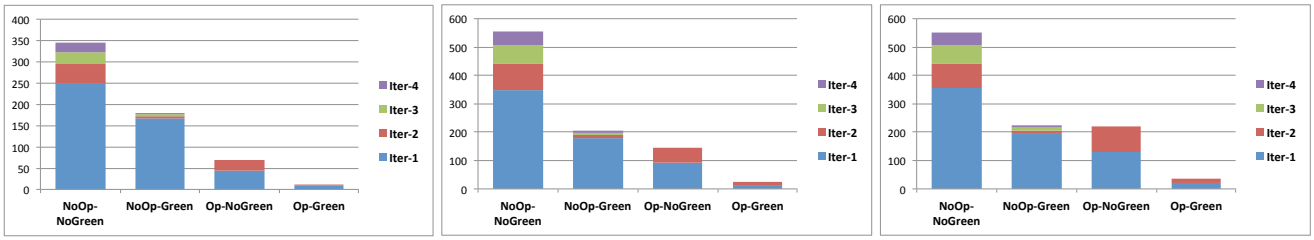
nates, it takes iDiscovery at most four iterations to reach the fix-point. Since iDiscovery without optimizations does not finish within the 20-hour limit on ASW and Apollo, optimizations to mitigate the path explosion problem are needed.

The results in Table 2 demonstrate that the iDiscovery optimizations are able to reduce the symbolic execution cost dramatically. It takes more than one hour to complete symbolic execution on Apollo instrumented with assertions in the first iteration of iDiscovery without optimizations, while symbolic execution is completed in 34 seconds when iDiscovery is used with optimizations, indicating a speed-up of more than 100X. Note that the number of symbolic states may increase when using optimizations because of the additional non-determinism choice points added by SPF at the `Verify` modeling primitives. For example, for TCAS-v3, the first iteration of symbolic execution without optimizations explores 14805 states, while the first iteration of symbolic execution with optimizations explores 82323 states. However, the symbolic execution time is much less for the latter, because a large number of the symbolic states are pruned by the iDiscovery optimizations. The optimizations not only reduce the total symbolic execution time at each iteration of iDiscovery, but also reduce the number of iterations for some subjects as seen in Table 2. For all three versions of TCAS, iDiscovery with no optimizations requires four iterations to reach a fix-point, while iDiscovery with optimizations requires only two iterations to reach the same fix-point. The reason is that the first optimization (assertion separation) greatly simplifies path constraint complexity and can solve more constraints within the solver time limits for each constraint, thus leaving less solvable constraints for following iterations, and reducing the number of iterations.

Different iDiscovery Optimizations. To better understand the effects of the optimizations, we evaluate iDiscovery with each optimization independently of the other on all of the artifacts. The assertion separation optimization is denoted as Optimization-1, and the violation restriction optimization is denoted as Optimization-2. The invariants generated on the last iteration of Optimization-1, Optimization-2, iDiscovery with no optimizations and iDiscovery with both optimizations are the same. Thus, in Table 3, we show only the symbolic execution cost for each of the iDiscovery configurations to study the impact of each individual optimization. In the table, Columns 1 and 2 list the corresponding artifacts and iterations; Columns 3 to 5, 6-8, 9-11, and 12-14 list the symbolic execution cost for each iteration of iDiscovery with no optimizations, with only Optimization-

Table 3: Symbolic execution costs for iDiscovery using different optimizations

Subjects	Iter.	No Optimizations			Optimization-1			Optimization-2			Both Optimizations		
		SCalls	States	Time	SCalls	States	Time	SCalls	States	Time	SCalls	States	Time
TCAS-v1	1	10066	10067	250s	12044	29341	201s	1454	1455	25s	3538	20835	45s
	2	2894	2895	44s	2312	8593	25s	2934	2935	40s	2072	8353	24s
	3	2342	2343	28s	–	–	–	2202	2203	25s	–	–	–
	4	2090	2091	22s	–	–	–	1950	1951	21s	–	–	–
TCAS-v2	1	13944	13945	348s	23866	56561	406s	18814	18815	380s	7644	50143	93s
	2	5604	5605	93s	5298	20779	59s	6860	6861	84s	4656	20137	51s
	3	5404	5405	64s	–	–	–	5184	5185	54s	–	–	–
	4	4752	4753	50s	–	–	–	4532	4533	43s	–	–	–
TCAS-v3	1	14804	14805	356s	53116	123877	859s	24316	24317	508s	11562	82323	129s
	2	5830	5831	87s	11068	44949	109s	7720	7721	99s	8940	42821	91s
	3	5750	5751	63s	–	–	–	5470	5471	54s	–	–	–
	4	5022	5023	46s	–	–	–	4742	4743	41s	–	–	–
WBS	1	100	101	0s	100	545	0s	100	101	0s	94	539	1s
	2	48	49	0s	48	441	0s	48	49	0s	48	441	0s
	3	46	47	0s	46	437	0s	46	47	0s	46	437	0s
ASW	1	–	–	TimeOut	60861	100206	11970s	112322	112323	32028s	4809	44154	843s
	2	–	–	–	4580	36857	770s	4571	4572	773s	4580	36857	802s
Apollo	1	26714	26716	6058s	704	802	33s	12259	12261	8550s	704	802	34s
	2	43984	43986	10869s	303	401	16s	17201	17203	14771s	303	401	17s
	3	34570	34572	8083s	470	716	143s	16221	16223	14066s	470	716	143s
	4	–	–	TimeOut	348	1394	136s	14548	14550	12884s	348	1394	135s



(a) TCAS-v1 (b) TCAS-v2 (c) TCAS-v3

Figure 5: Symbolic execution costs (in seconds) for iDiscovery with/without Green

1, with only Optimization-2, and with both optimizations, respectively. Based on the results, we make the following observations. First, Optimization-1 can reduce the symbolic execution time as well as the number of iterations. To illustrate, for TCAS-v1, iDiscovery without optimizations requires four iterations to finalize the invariants with a total symbolic execution time of 5 minutes 44 seconds, while Optimization-1 requires only two iterations and 3 minutes 46 seconds. Second, Optimization-2 reduces the symbolic execution time without reducing the number of iterations. The reason is that Optimization-2 only restricts the number of violations for each assertion, but does not separate the interaction between different assertions. Third, the combination of the two optimizations yields much better results than either of the optimizations independently. To illustrate, at the end of the first iteration of ASW, the symbolic execution time is more than three hours for Optimization-1 and more than eight hours for Optimization-2 (the time for iDiscovery with no optimization is more than 20 hours). However, symbolic execution for iDiscovery with both optimizations finishes within 15 minutes, indicating a speed-up of more than 14X and 37X over Optimization-1 and Optimization-2, respectively. These results emphasize the importance of reducing the complexity of assertions in each program path (addressed by Optimization-1) and reducing the number of times an assertion is checked across different program paths (addressed by Optimization-2).

Different Solver Configurations. We also study how state-of-the-art solver techniques based on constraint re-use can benefit iDiscovery. Figure 5 shows the symbolic execution time for each iteration of iDiscovery with and without the Green solver interface, which reuses constraint solution histories from previously observed executions. Note that we reset the Green solver at the beginning of each experiment on each artifact. The reductions can be even more dramatic if we allow for constraint re-use across different programs. Each sub-figure in Figure 5 represents the results for one subject. In each figure, each stacked column denotes the total symbolic execution time (in seconds) for each of the four configurations (i.e., no optimizations with no green, no optimizations with green, optimizations with no green, and optimizations with green). In each stacked column, the four different colors show the time cost distribution for different iterations. Note that we do not show Green results for WBS because its symbolic execution cost is too small to investigate (0s for the majority cases), and we do not show Green results for ASW and Apollo because the current version of Green does not support the CORAL solver which is required for solving invariant constraints of these artifacts. From the results in Figure 5, we make the following observations. First, the use of the Green solver extension can reduce the cost of symbolic execution independent of the optimizations. Second, using the Green extension together with the optimizations can yield even better results. To illustrate, consider the first iteration of TCAS-v1; Green is able to reduce the symbolic execution cost of iDiscovery without the optimizations from 4 minutes and 10 seconds to 2 minutes and 46 seconds. However, when optimizations are also used, symbolic execution takes just 8 seconds, which is much better than applying them separately. Thus, we believe that the

combination of a Green-like solver extension and our optimizations can make future applications of iDiscovery even more tractable.

Different Initial Test Suites. In practice, it is possible for the test suites provided to iDiscovery to be generated using different test generation techniques, and for the test suites to vary in size and coverage. Thus, we also study the impact of test suites generated by different techniques and test suites of various sizes on the performance of iDiscovery. As we already studied the performance of iDiscovery when given an initial test suite generated by symbolic execution, we now analyze the performance of iDiscovery when given an initial test suite generated randomly. We randomly generated 5%, 10%, 15%, 20%, and 100% of the number of initial tests generated by symbolic execution as initial test suites. We then apply iDiscovery with optimizations on each initial test suite for each artifact. The final set of candidate invariants and the time cost for the various initial tests are quite similar. However, the number of added (augmented) invariants and refuted invariants differs. Therefore, we show the number of added invariants and refuted invariants for each artifact and initial suite combination in Figure 6. In Figure 6a, the horizontal axis shows the different initial test suites (*symbc* denotes the symbolically generated test suites, *rand* denotes the randomly generated test suites, and *x%-rand* denotes x% of the randomly generated test suites); the vertical axis shows the number of refuted invariants by iDiscovery for a given initial test suite; each line show the corresponding results for each artifact. Similarly, Figure 6b shows the number of added invariants by iDiscovery for each artifact and each type of initial test suite. To understand the differences in performance of iDiscovery for different initial test suites, we also show the instruction coverage, branch coverage, and line coverage for each initial test suite for each subject in Figure 7. For example, in Figure 7a, the horizontal axis shows the different initial test suites, while the vertical axis shows the instruction coverage for each suite on each artifact.

Based on the results, we highlight several interesting findings. First, the number of refuted invariants is low for most artifacts when initial test suites have extremely low or high coverage. The reason is that when the initial test suites have extremely low coverage, the number of invariants inferred by the first iteration of iDiscovery is limited by the small set of trace samples, making the number of candidate invariants to be refuted small; when the initial test suites have extremely high coverage, the number of invariants inferred during the first iteration is also limited, because the high-coverage test suites can already refute a number of incorrect or imprecise invariants. Second, for each subject, the number of augmented invariants consistently goes down as the initial test suite coverage goes up. The reason is that traditional dynamic invariant inference techniques, such as Daikon, first generate candidate invariants for the covered program behaviors, and then try to refute them based on additional tests. Therefore, when a test suite has high coverage, Daikon has likely already computed the largest possible set of invariants (with potentially many false-positives), so that iDiscovery can only help refute incorrect/imprecise invariants (but cannot augment correct/precise invariants).

5.4 Threats to Validity

Threats to construct validity. The main threat to construct validity for our study is the set of metrics used to evaluate the different iDiscovery configurations. To reduce this threat, we use the number of finally generated candidate invariants, augmented invariants, as well as refuted invariants to measure the effectiveness of iDiscovery; we also use test tracing time, invariant inference time, and symbolic execution time of each iteration of iDiscovery to measure the efficiency of iDiscovery. Our study still has a threat to construct validity: although all the refuted invariants by iDiscovery are proven to be incorrect or imprecise, we did not check the quality of the augmented invariants.

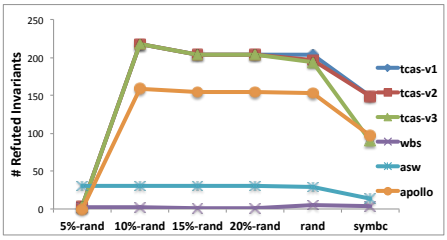
Threats to internal validity. The main threat to internal validity is the potential faults in the implementation of different configurations of iDiscovery or in our data analysis. To reduce this threat, the first author carefully reviewed all the code for iDiscovery and data analysis during the study.

Threats to external validity. The main threat to external validity is that our experimental results may not generalize to other contexts, including programs, initial test suites, invariant inference tools, and constraint solvers. Further reduction of threats to external validity requires more extensive evaluation on more invariant inference tools and constraint solvers, as well as more and larger subject programs.

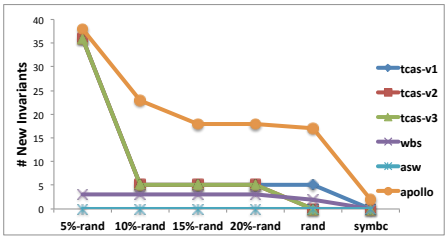
6. RELATED WORK

Automated techniques for discovering invariants can generally be categorized as either a static or a dynamic inference technique. Static inference techniques are typically sound, but in practice, suffer from several limitations including the undecidability of the underlying mathematical domains, and the high cost of modeling program states. Much of the static inference work [5,22,32] uses abstract interpretation or symbolic execution to approximate the fix-point of properties on semantics of program expressions. Some work constructs, derives, or strengthens specifications by using existing specifications [2,10,15,18,27,34]. While several projects [13,25] have used static verification tools, e.g., ESC/Java [14] to check the invariants, the results are not used as feedback to further improve invariant discovery as is done in this work.

Dynamic invariant detection techniques follow two main approaches. They either start with a set of candidate invariants and refine them using program trace information from a user-provided test suite [12,16], or they use the trace information to essentially build the invariants "from scratch" [9]. In this work, we use Daikon, which follows the first approach. Daikon first instantiates a set of initial candidate invariants based on a database of invariant templates, and then during execution of a test suite, it tracks the values of program variables and uses these values to refute invariants in the initial set [12]. Whereas, a tool such as DySy takes the second approach and builds the invariants by collecting symbolic path conditions while executing the user-supplied test suite [9]. DySy then summarizes the generated path conditions as invariants. Note that unlike symbolic execution techniques, DySy is not reliant on a constraint solver or a decision procedure to check feasibility of the invariants since it relies exclusively on concrete execution traces. A key difference between the two approaches is that techniques starting with candidate invariants, such as Daikon, may over-approximate the inferred invariants if the set of test traces does not provide enough information to refute the generated candidate invariants. On the contrary, techniques that build invariants "from scratch", such as DySy, can generate a more precise set

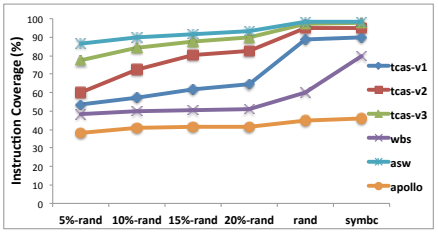


(a) Trends for Refuted Invariants

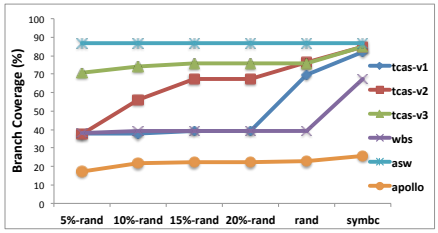


(b) Trends for Added Invariants

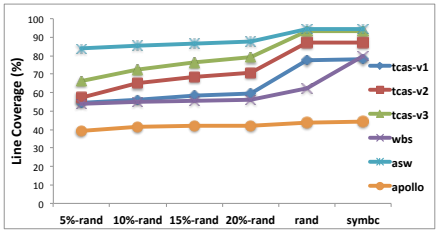
Figure 6: The trends for added and refuted invariants for different initial test suites



(a) Trends for Instruction Coverage



(b) Trends for Branch Coverage



(c) Trends for Line Coverage

Figure 7: The coverage trends for different initial test suites

of invariants, but the invariants are more similar to symbolic summaries than the invariants generated by Daikon.

The number of invariants generated by dynamic invariant detection techniques may be quite large, and the quality of the generated candidate invariants for techniques such as Daikon [12] and DIDUCE [16], depends largely on the collection of invariants in its invariant repository. Recent studies have also shown that the test suite used to generate the program traces can affect the quantity and quality of the generated candidate invariants [12, 28]. Recently, Li et al. [21] presented a technique to help derive more relevant invariants and reduce noise. Their technique enhances the dynamic invariant detection algorithm by including additional ("secondary") constraints which relate classes of invariants. These constraints can also be inferred automatically through dynamic observations of program behavior. This work is orthogonal to iDiscovery and would be interesting to explore in conjunction with iDiscovery in future work.

The work most closely related to iDiscovery is that of Xie and Notkin [35]. Their approach enhances both tests and specifications using a feedback loop between dynamic invariant discovery using Daikon and test generation using Jtest [7]. This is similar to the feedback loop in iDiscovery, however, the approach in [35] uses the specifications computed by Daikon to guide the test generation process and the process includes a test selection step which involves manual inspection of the generated tests. The work presented here is fully automated and uses a (bounded) symbolic execution of the program instrumented with the candidate invariants to compute additional test inputs. Howar et al. [17] proposed an iterative learning algorithm, X-PSYCO, which combines dynamic analysis and symbolic component analysis to infer constraints on method parameters, i.e., invariants. Unlike the method-level pre- and post-conditions generated by Daikon, the results of X-PSYCO are represented using finite-state automata and specify a component's interface in terms of safe method sequences where the inferred constraints on method parameters serve as guards on the transitions in the automata.

Our recent work on iProperty [36] introduces an approach for *incremental* checking of conformance of code to specifications that evolve. Specifically, iProperty introduces a *property differencing* algorithm that computes logical differences between two sets of properties to allow checking code against a minimal set of properties taking into account the previous version of code and properties already checked as well as those checking results (e.g., any counterexamples that were previously found). Property differencing of iProperty can in principle benefit iDiscovery by further optimizing its iterative use of symbolic execution in enhancing the quality of invariants discovered. We plan to investigate the combination of iProperty and iDiscovery in the future.

7. CONCLUSION

This paper introduced iDiscovery, a novel technique that applies two well-known approaches – dynamic invariant discovery and symbolic execution – in synergy to discover higher quality invariants. The Daikon tool for invariant discovery and the Symbolic PathFinder (SPF) tool for symbolic execution provide the enabling technology. A feedback loop iteratively runs Daikon and SPF to discover more accurate and complete invariants until a fix-point is reached. Two optimizations, namely assertion separation and violation restriction, enhance the efficiency of iDiscovery by reducing the cost of symbolic execution. An experimental evaluation using a suite of Java subject programs shows that iDiscovery converges to a set of higher quality invariants than the initial set in a few iterations.

8. ACKNOWLEDGMENTS

This work was funded in part by the National Science Foundation (NSF Grant Nos. CCF-0845628, CCF-1319688, and CNS-0958231) and the Google Summer of Code 2013.

9. REFERENCES

[1] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *ACSAC*, pages 77–86, 2008.

[2] N. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theor. Comput. Sci.*, 173(1):49–87, Feb. 1997.

[3] V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric prediction of heap memory requirements. In *ISMM*, pages 141–150, 2008.

[4] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[5] B.-Y. E. Chang and K. R. M. Leino. Abstract interpretation with alien expressions and heap structures. In *VMCAI*, pages 147–163, 2005.

[6] L. A. Clarke. A program testing system. In *Proceedings of the 1976 annual conference*, ACM '76, pages 488–491, 1976.

[7] P. Corporation. Jtest manuals version 4.5 october 23 (2002). `http://www.parasoft.com/`, 2002.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[9] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *ICSE*, pages 281–290, 2008.

[10] D. D. Dunlop and V. R. Basili. A heuristic for deriving loop functions. *IEEE Trans. Softw. Eng.*, 10(3):275–285, May 1984.

[11] M. D. Ernst. *Dynamically discovering likely program invariants*. PhD thesis, University of Washington, 2000.

[12] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[13] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME*, pages 500–517, 2001.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245, 2002.

[15] S. M. German and B. Wegbreit. A synthesizer of inductive assertions. In *AFIPS*, pages 369–376, 1975.

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

[17] F. Howar, D. Giannakopoulou, and Z. Rakamarić. Hybrid learning: Interface generation through static, dynamic, and symbolic analysis. In *ISSTA*, pages 268–279, 2013.

[18] S. Katz and Z. Manna. Logical analysis of programs. *Commun. ACM*, 19(4):188–206, Apr. 1976.

[19] J. C. King. Symbolic execution and program testing. *CACM*, 19(7):385–394, 1976.

[20] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program.*, 55(1-3):185–208, Mar. 2005.

[21] K. Li, C. Reichenbach, Y. Smaragdakis, and M. Young. Second-order constraints in dynamic invariant inference. In *ESEC/FSE 2013*, pages 103–113, 2013.

[22] F. Logozzo. Automatic inference of class invariants. In *VMCAI*, pages 211–222, 2004.

[23] M. Z. Malik, A. Pervaiz, and S. Khurshid. Generating representation invariants of structurally complex data. In *TACAS*, pages 34–49, 2007.

[24] B. Meyer, J.-M. Nerson, and M. Matsuo. Eiffel: Object-oriented design for software engineering. In *ESEC*, pages 221–229, 1987.

[25] J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *RV*, 2001.

[26] C. S. Păsăreanu and N. Rungta. Symbolic Pathfinder: symbolic execution of Java bytecode. In *ASE*, pages 179–180, 2010.

[27] C. S. Pasareanu and W. Visser. Verification of Java programs using symbolic execution and invariant generation. In *SPIN*, pages 164–181, 2004.

[28] N. Polikarpova, I. Ciupa, and B. Meyer. A comparative study of programmer-written and automatically inferred contracts. In *ISSTA*, pages 93–104, 2009.

[29] SAE-ARP4761. *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. SAE International, December 1996.

[30] SIR. Software-artifact infrastructure repository: Home. `http://sir.unl.edu`, 2008.

[31] M. Staats, S. Hong, M. Kim, and G. Rothermel. Understanding user understanding: Determining correctness of generated program invariants. In *ISSTA*, pages 188–198, 2012.

[32] N. Tillmann, F. Chen, and W. Schulte. Discovering likely method specifications. In *ICFEM*, pages 717–736, 2006.

[33] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In *FSE*, page 58, 2012.

[34] B. Wegbreit. The synthesis of loop predicates. *Commun. ACM*, 17(2):102–113, Feb. 1974.

[35] T. Xie and D. Notkin. Mutually enhancing test generation and specification inference. In *In Proc. 3rd International Workshop on Formal Approaches to Testing of Software*, pages 60–69, 2003.

[36] G. Yang, S. Khurshid, S. Person, and N. Rungta. Property differencing for incremental checking. `http://cs.txstate.edu/~g_y10/publications/icse14-iproperty.pdf`, to appear in ICSE 2014.

[37] G. Yang, C. S. Păsăreanu, and S. Khurshid. Memoized symbolic execution. In *ISSTA*, pages 144–154, 2012.