Property Differencing for Incremental Checking

Guowei Yang[1], Sarfraz Khurshid[2], Suzette Person[3], Neha Rungta[4]

[1]Department of Computer Science, Texas State University, San Marcos, TX 78666, USA
[2]Department of Electrical and Computer Engineering, University of Texas, Austin, TX 78712, USA
[3]NASA Langley Research Center, Hampton, VA 23681, USA
[4]NASA Ames Research Center, Moffett Field, CA 94035, USA

[1]gyang@txstate.edu, [2]khurshid@utexas.edu, [3]suzette.person@nasa.gov, [4]neha.s.rungta@nasa.gov

ABSTRACT

This paper introduces *iProperty*, a novel approach that facilitates *incremental* checking of programs based on a property *differencing* technique. Specifically, iProperty aims to reduce the cost of checking properties as they are initially developed and as they co-evolve with the program. The key novelty of iProperty is to compute the *differences* between the new and old versions of expected *properties* to reduce the number and size of the properties that need to be checked during the initial development of the properties. Furthermore, property differencing is used in synergy with program behavior differencing techniques to optimize common regression scenarios, such as detecting regression errors or checking feature additions for conformance to new expected properties. Experimental results in the context of symbolic execution of Java programs annotated with properties written as assertions show the effectiveness of iProperty in utilizing change information to enable more efficient checking.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*assertion checkers*; D.2.5 [**Software Engineering**]: Testing and Debugging—*symbolic execution, testing tools*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*assertions, invariants*

General Terms

Verification, Algorithms

Keywords

Incremental symbolic execution, assertions, change-impact analysis, Symbolic PathFinder, Daikon

ICSE'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2756-5/14/05...$15.00
http://dx.doi.org/10.1145/2568225.2568319

1. INTRODUCTION

Annotating functional correctness properties of code, e.g., using assertions [17] or executable contracts, such as those supported by the Java Modeling Language [37] or Eiffel [39], offers a number of benefits in automated conformance checking of program behaviors to expected properties to support bug finding [18, 22, 29, 30, 38, 57]. Effectively utilizing such properties in practice is complicated by two basic factors. One, it requires the properties to be written and maintained meticulously, so they correctly reflect the expected behaviors of the code, even as it evolves. Two, it requires efficient and cost-effective techniques to check the actual behaviors of the code with respect to the given properties.

We believe an approach that integrates automated program analyses with the development and maintenance of code annotated with properties allows us to address both factors by making it possible for developers to efficiently check and correct the properties and the code they write. Our focus in this paper is on reducing the computational cost of such analyses, specifically in the context of *change* – to properties or to code – due to bugs fixes or feature additions (i.e., addition of new functionality).

In this work, we introduce *iProperty*, a novel approach to compute *differences* between *properties* of related programs in a manner that facilitates more efficient incremental checking of conformance of programs to properties. The key novelty of iProperty is to compute *logical* differences between old and new versions of properties that undergo change and facilitate the two development scenarios that form our focus: (1) writing properties correctly for an existing program, where the developer writes partial correctness properties, checks them, and then iteratively refines and re-checks them, and (2) checking changes to code and properties as they co-evolve to, for example, check feature additions for conformance to new expected properties or to check bug fixes for regression errors.

In a program annotated with properties that was checked previously but has since undergone change, iProperty utilizes the results of the previous analysis and information about the change to the properties to reduce the number of properties that need to be checked. Our insight is that if a property Φ about program p holds at a control point l in p and Φ implies another property Φ', Φ' also holds at l in p. This simple observation forms the basis of our algorithm for computing property differences, which checks for implications between corresponding old and new properties to minimize the set of properties that must be checked after a property change is made and hence reduce the overall cost of checking.

To improve incremental property checking, we leverage the distinction between changes to properties, i.e., assertions, and changes to the code that implements the program functionality. iProperty, uses the property differences in conjunction with code-based change impact analysis techniques to *direct* checking onto *relevant* parts of code and properties, thereby removing much of the redundancy otherwise present in re-applying the checking technique. iProperty uses the property differencing algorithm together with a previously developed behavioral differencing technique [45, 50].

iProperty supports incremental checking of properties written as assertions or that can be mechanically transformed into assertions.The `assert` statements in the source code are de-sugared into `if-then` statements in the object-code, which allows the use of traditional code change impact analyses, e.g., to compute behavioral differences [11, 45, 50, 51]. Change impact analysis techniques do not distinguish between code that implements functionality and code that represents expected behaviors and thus, are unable to characterize whether an assertion violation represents a regression error, incorrect update to the property, or non-conformance between addition of new features and properties.

To evaluate the efficacy of iProperty, we apply it in the context of symbolic execution of Java programs annotated with assertions to check for assertion violations. We use subject programs from recent work on incremental symbolic execution, which focused on changes to the *code* [45, 60]. To annotate these subjects with assertions and to simulate assertion changes, we (a) write some assertions manually, and (b) synthesize some assertions automatically using the Daikon tool [19]. While our application of Daikon for assertion synthesis was inspired by Daikon's spirit of discovering likely invariants, our specific context of symbolic execution enabled us to systematically validate the output generated by Daikon and make its output more precise, thereby laying the foundation of a future approach for more accurate discovery of likely invariants [61]. This paper makes the following contributions:

∘ **Property differencing.** We introduce the idea of incremental checking driven by property differencing.

∘ **iProperty.** We present algorithms to compute logical differences between old and new properties and utilize checking results from the previous analysis

∘ **Assertion-driven incremental checking.** Assertion differences are combined with a change impact analysis technique [45] to detect regression errors and check conformance of new functionality to new expected properties.

∘ **Experimental evaluation.** The experimental results show that iProperty can (1) provide a reduction in the number of SMT solver calls and total analysis time required for symbolic execution of Java programs compared to the conventional approach of using the Symbolic PathFinder tool [42, 43], and (2) enable verification of properties that could not be verified in a non-incremental manner.

2. MOTIVATING EXAMPLE

This section presents a motivating example to illustrate iProperty and how it is applied in the context of symbolic execution [14,16,24,34]. Symbolic execution uses symbolic values in lieu of concrete values for program inputs and builds a path condition along feasible execution paths. A path condition consists of a set of constraints over the symbolic input values and constants. Satisfiability of the constraints is checked using off-the-shelf SMT solvers. In this work, we use Symbolic PathFinder (SPF) [42,43], an open-source tool for symbolic execution of Java programs built on top of the Java PathFinder (JPF) model checker [57].

```
1 int median(int x, int y, int z) {
2   int m = z;
3   if (y < z) {
4     if (x < y)
5       m = y;
6     else if (x < z)
7       m = x;
8   }
9   else {
10    if (x > y)
11      m = y;
12    else if (x > z)
13      m = x;
14  }
15
16  assert x == y || x == z ? m == x : true; //#1
17  assert y == z ? m == y : true;           //#2
18  assert x <= z && z <= y ? m == z : true; //#3
19  assert y <= z && z <= x ? m == z : true; //#4
20
21  return m;
22 }
```

Figure 1: Method to compute the middle value of three input numbers [31] and expected properties (Lines 16–19).

Figure 1 shows the Java implementation of `median`, which computes the middle value of its three integer inputs; the code for `median` appears in previous work [31], but without property annotations. We manually add partial correctness properties, specifically *post-conditions*, to this example in the form of assertions just before the method exit.

Version 1. Assume the user writes four assertions to check expected properties. The two assertions on Lines 16–17 check `median` when at least two inputs are equal, i.e., $x = y \lor x = z \Rightarrow m = x$, and $y = z \Rightarrow m = y$. The two assertions on Lines 18–19 check `median` when the definition of `m` at Line 2 *reaches* the `return` statement, i.e., the program behaves correctly when z is the middle value; specifically, the user asserts $x \leq z \land z \leq y \Rightarrow m = z$, and $y \leq z \land z \leq x \Rightarrow m = z$.

Symbolically executing this method using SPF checks the given properties (Lines 16–19) and reports no assertion failures after 2 seconds. SPF explores 173 states and makes 172 calls to the underlying SMT solver.

Version 2: Property *modification* and *addition*. Having gained some confidence in the implementation, assume now the user decides to check the program after *editing* some assertions and *adding* new assertions that check more program behaviors. Specifically, the user (1) modifies the third and fourth assertions to use '<' instead of '<=', e.g., to reduce the overlap between the different sets of inputs each assertion checks, *and* (2) adds four new assertions to check cases previously unchecked. The following sequence of assertions replaces the assertions in Version 1 (Lines 16–19).

```
23 assert x == y || x == z ? m == x : true; //#1
24 assert y == z ? m == y : true;           //#2
25 assert x < z && z < y ? m == z : true;   //#3'
26 assert y < z && z < x ? m == z : true;   //#4'
```

```
27 assert x < y && y < z ? m == y : true;    //#5
28 assert z < y && y < x ? m == y : true;    //#6
29 assert y < x && x < z ? m == x : true;    //#7
30 assert z < x && x < y ? m == x : true;    //#8
```

Applying the conventional symbolic execution approach to check the `median` program we invoke SPF once which reports no assertion failures after 3 seconds; for this check, SPF explores 301 states and makes 300 SMT solver calls. In contrast, applying iProperty to check the `median` program with this modified sequence of assertions we invoke SPF *three* times. Two of these invocations focus on the modified assertions (Lines 25–26) and the third invocation focuses on the new assertions that were added (Lines 27–30). None of the invocations involve previously checked assertions that were not modified (Lines 23–24).

The first invocation *directly* checks whether the Version 1 assertion #3 (Line 18) *implies* the Version 2 assertion #3$'$ (Line 25) by checking if the following method causes an assertion failure:

```
void checkImplies(int x, int y, int z, int m) {
  if (x <= z && z <= y ? m == z : true)
    assert x < z && z < y ? m == z : true;
}
```

This SPF invocation takes less than 1 second and reports no assertion failure, i.e., the implication holds. Thus, the validity of Version 1 assertion #3 guarantees the validity of Version 2 assertion #3$'$ so it is not necessary to check it against the implementation of `median`. In this case, SPF explores 19 states and invokes the CVC3 solver 18 times.

The second invocation of SPF similarly checks that Version 1 assertion #4 (Line 19) implies Version 2 assertion #4$'$ (Line 26) and hence Version 2 assertion #4$'$ is also valid and does not need to be checked explicitly against the implementation of `median`. SPF takes less than 1 second, explores 19 states, and makes 18 CVC3 solver calls to perform the check.

The third invocation of SPF checks only the Version 2 assertions #5–#8 against the implementation of `median`. This invocation takes less than 1 second and reports no assertion failure. In this invocation, SPF explores 121 states and makes 120 calls to the CVC3 solver.

Overall, using iProperty to check the Version 2 properties makes three invocations of SPF taking a total of 2 seconds (versus 3 seconds in the conventional approach), requires SPF to explore 159 states (versus 301 states in conventional approach), and makes 156 solver calls to CVC3 (versus 300 calls in conventional approach). Thus, iProperty provides a 48% reduction in SMT solver calls compared to the conventional approach using SPF for this small example.

3. APPROACH

In this section, we present the details of the iProperty approach that computes property differences for more efficient incremental checking of evolving programs. iProperty operates on two program versions, p and p', leverages the results from checking p in order to compute the logical differences in evolving properties. iProperty then combines the property differences and a code change impact analysis to enable efficient incremental property checking of p'.

Given two related program versions p and p', where p is annotated with one or more properties, and the properties in p' are changed with respect to p; for example, properties are added, deleted, or modified in p'. Moreover, p' may contain changes to the code that result in functional differences, e.g., added, changed, or removed program behaviors. Assume p was previously checked for property failures and the results from checking p are available. Then the basic problem addressed by our technique is how to efficiently check the properties in p'. Our technique addresses this problem in two ways: 1) by defining a mechanism to compute property *differences* between p and p' that leverages the results from checking p to automatically re-write the properties in p', retaining only the necessary parts of the properties, and 2) by checking p' incrementally with respect to the impact of the code changes along with the property differences.

Section 3.1 describes the initial property checking. Section 3.2 describes the algorithmic details of computing differences in properties specified as `assert` statements in Java source code. Section 3.3 describes three checks that can be performed using the property differences and the results of a change impact analysis to efficiently check for regression errors and conformance of new features in the code to the additional properties.

3.1 Initial Property Checking

In this work, we analyze properties specified as Java `assert` statements. We assume the specified properties are side-effect free.

DEFINITION 3.1. *A is a set of assertions in a Java program p such that each $a \in A$ is an Java assertion "`assert` Φ" where Φ specifies one or more properties as a conjunction of clauses, $\Phi_a := \phi_0 \wedge \ldots \wedge \phi_m$ $(m \geq 0)$.*

Each *clause*, ϕ, in a conjunction Φ can be an arbitrarily complex boolean expression and may include one or more invocations of pure functions (side-effect free).

iProperty uses the results from checking the previous version of the properties in the property differencing algorithm presented in Section 3.2. The results from analyzing a program, p, with a given set of initial properties, A, are stored for re-use by iProperty. During the initial analysis of the program we record the *outcome* from checking a property—whether the property passed or failed. In the case a property fails, program analysis techniques such as symbolic execution generate a *counterexample* to show how the property failed. We first define two data structures to store the analysis results: **Outcome** and **Counterexample**.

DEFINITION 3.2. **Outcome** *is a map $Out_p : A \mapsto$ \{`Fail`, `Not Fail`, `Unknown`\} that returns one of three possible outcomes of checking an assertion a with respect to a program p (1) `Fail` indicates that a violation of a was detected, (2) `Not Fail` indicates no failure of a was found, and (3) `Unknown` indicates that the results of checking a are not known.*

DEFINITION 3.3. **Counterexample** *is a map $Cex_p : A \mapsto 2^{traces}$ that returns a set of traces where each trace represents a feasible execution path from the start of the program p to the violation of an assertion, a and $Out_p(a) :=$ `Fail`.*

The **Outcome** map for p is populated with the outcomes generated when checking the corresponding assertions in A against program p; while the **Counterexample** map is populated with assertions which have the `Fail` outcome.

DEFINITION 3.4. **Transform** *is a map `transform`: $A \mapsto 2^{clauses}$ that returns the set of clauses contained in the conjunction, $\Phi_a := \phi_0 \wedge \ldots \wedge \phi_m$, for the assertion $a \in A$.*

```
procedure PropertyDifferencing($A', Cex_p, Out_p$)
 1: $\forall a' \in A' \,.\, Cex_{p'}(a') := \emptyset$
 2: for each $a' \in A'$ do
 3:     if matchedAsserts($a'$) $\neq \bot$ then
 4:         $\langle outcome, a_{diff} \rangle :=$ AssertDiff($a, a'$)
 5:     else $\langle outcome, a_{diff} \rangle := \langle$Unknown, transform($a'$)$\rangle$
 6:     $Out_{p'}(a') := outcome$
 7:     $\Delta_{A',A}(a') := a_{diff}$
 8:     if $outcome ==$ Fail then
 9:         $Cex_{p'}(a') := Cex_p(a)$
10:
procedure AssertDiff($a', a$)
11: if $Out_p(a) ==$ Unknown then
12:     return $\langle$Unknown, transform($a'$)$\rangle$
13: if LogicalDifference($a', a$) $== \emptyset$ then
14:     return $\langle Out_p(a), \emptyset \rangle$
15: if $Out_p(a) ==$ Not Fail then
16:     return $\langle$Unknown, {LogicalDifference($a', a$)}$\rangle$
17: else if $Out_p(a) ==$ Fail then
18:     return $\langle$Unknown, transform($a'$)$\rangle$
```

Figure 2: The property differencing algorithm.

3.2 Property Differencing (No Code Changes)

The key idea our approach is to automatically re-write assertions in p', based on the logical differences between assertions in p and p'. The logical differences are combined with the information gathered from the previous analysis of p, in order to reduce the number of properties that need to be checked in p'. In this section we assume that there are no code changes between programs p and p', i.e., they are *syntactically the same* and *semantically equivalent* when assertions are turned off. In Section 3.3 we take into account code changes.

At the heart of computing property differences is the notion of logical difference between two properties. In this work we define logical difference between two Java assertions as follows:

Definition 3.5. **LogicalDifference** *is a function that takes as input two assertions a' and a, and returns the set of clauses in a' where $\{\phi' \in$ transform$(a') \mid \Phi_a \not\Rightarrow \phi'\}$.*

The logical difference function returns the set of clauses in a' that are not *subsumed*, i.e., logically implied, by the clauses in a. Moreover, if the logical difference function returns the empty set, it follows that $\Phi_a \Rightarrow \Phi_{a'}$, i.e., all of the clauses in the conjunction of a' are subsumed by those in a; indicating there is no functional change going from a to a'.

Lemma 3.6. *Assume checking p with a at location l shows that assertion a holds. Let $\delta =$ LogicalDifference(a', a). Then, checking p with conjunction $\bigwedge_{i=0}^{|\delta|} \delta_i$ at control-point l is equivalent to checking p with assertion a' at l, i.e., the set of executions of p that violate a' at l equals the set of executions of p that violate $\bigwedge_{i=0}^{|\delta|} \delta_i$ at l.*

Lemma 3.7. *If checking p with a at location l results in a violation of a and $a' \Rightarrow a$, then a' is also violated at l in p.*

The `PropertyDifferencing` algorithm is shown in Figure 2. The inputs to the algorithm are the set of all assertion statements in p' (A'), and the **Outcome** and **Counterexample** maps from checking p, (Out_p) and (Cex_p) respectively. As the `PropertyDifferencing` algorithm iterates over each assertion in p', it computes the logical difference between matching assertions in p and p' and uses the results from analyzing p to update i) the outcome map of p', $Out_{p'}$, ii) the counter-example map, $Cex_{p'}$, and iii) the map of assertions differences, $\Delta_{A',A}$, that contains for each assertion in p', the set of clauses in a' that need to be checked against p'. The entries in $Out_{p'}$ and $Cex_{p'}$ are initialized to `Unknown` and $\emptyset$ respectively for each assertion at line 1 during the initialization of the `PropertyDifferencing` algorithm.

The algorithm checks if a' is a new assertion added to p' by calling the function `matchedAsserts` at line 3 in Figure 2. This function takes as input an assert statement in p' and returns the corresponding assert statement in p; when no corresponding assert statement is found, it returns $\bot$. Assert statements can be matched using various techniques such as an AST matching algorithm [20] or an IDE monitor that tracks the changes to the code. When no matching assert statement in present in p, then the analysis concludes that a' is a new assertion added to p'. In this case, the outcome for a' is set to `Unknown` (lines 5 and 6) and the set of clauses in a' is added to $\Delta_{A',A}$ in order to be checked against p' (line 7). When the `matchedAsserts` function returns a matching assert statement, a, at line 3, the `PropertyDifferencing` algorithm computes the differences between a and a' by invoking the `AssertDiff` function.

The `AssertDiff` procedure shown at lines 11–18 in Figure 2 computes and returns the outcome and the logical difference of a' with respect to the matched assertion in p. When the outcome of the matched assertion, a, is `Unknown` with respect to p then a' is treated as a new assertion by setting its outcome to `Unknown` (lines $11-12$) and adding it to the set of assertions to be checked in p' (lines $6-7$).

When there is no logical difference between assertions a and a', and the outcome of a is either `Fail` or `Not Fail`, then the outcome of a is returned as the outcome of a' at lines $13-14$ in Figure 2. There are two cases to consider when, however, there is a logical difference between assertions a and a'. First consider the `Not Fail` case at lines 15 and 16. In the absence of code changes, the logical difference between the assertions is added to the difference map $\Delta_{A',A}$ to be checked for failures, and its outcome is set to `Unknown`. Next, consider the `Fail` case at lines 17 and 18. Since there is a difference between the two assertions we cannot reason about whether the new assertion a' will fail or not. Hence, the outcome for a' is set to `Unknown` and the set of clauses in a' is added to $\Delta_{A',A}$.

We also use previously generated counterexamples to potentially reduce the number of properties that need to be checked using an an expensive analysis technique such as symbolic execution. Often a program execution that is successful in eliciting a violation of one property may be able to detect a violation of another property as well. In order to leverage this, we concretely execute inputs of the program that lead to the counter-examples in Cex_p and check for all assertions, a'_f, such that $Out_{p'}(a'_f) :=$ `Unknown` if a violation of the assertion is detected. For these additional failing assertions their corresponding entries in the (a) outcome map are marked as `Fail`, (b) difference map are marked as empty, i.e., $\Delta_{A',A}(a'_f) := \emptyset$, and (c) counterexample map are updated with the error trace.

The algorithm in Figure 2 computes the map of assertion differences, $\Delta_{A',A}$. For every assertion a', the map contains a set of clauses (conjunction of clauses) that we need to check with respect to program p'. Each entry in the map $\Delta_{A',A}$ is mapped into assert statement **assert** Φ at the location of a' in p'. The clauses set is re-written as a Java expression. We analyze program p' with re-written assertions using symbolic execution for more efficient checking of the properties.

3.3 Property Checking (with Code Changes)

In this section, we present our incremental property checking approach in the presence of *code changes.* Recall that earlier, while computing the property differences we assumed there are no code changes between p and p'—only property changes. Our goal is to partition the incremental property checking in a manner that allows more efficient detection of regression errors introduced during bug fixes and checking conformance of feature additions to new properties.

The incremental property checking algorithm combines in a novel way results computed by the property differencing algorithm in Section 3.2 and our previous work on Directed Incremental Symbolic Execution (DiSE) [45, 50] that generates the set of program behaviors impacted by the code changes. The combination of the property differences with DiSE allows us to partition the incremental analysis into three different checks: (i) **Check I** analyzes the changed or impacted program behaviors with respect to the assertion differences; (ii) **Check II** analyzes the changed or impacted behaviors with respect to the original (preserved) assertions; and (iii) **Check III** analyzes the unchanged program behaviors with respect to the assertion differences. The three checks can be performed independent of each other. The partitioning also provides insight into whether the property violation was caused by the property differences (incorrect update of properties), code changes (regression errors), or a combination of property differences and code changes (new feature additions do not conform to the new properties).

3.3.1 Directed Incremental Symbolic Execution

We first present a brief overview of the DiSE methodology. Given two related program versions p and p', (1) DiSE constructs the *change set*, C, which contains the set of program statements that are syntactically different between p and p', (2) DiSE then uses the change set C as the slicing criterion to statically compute the *impact set*, I, the set of program statements that may be impacted by the changes in p', and (3) Symbolic execution of p' is then directed along the program locations in the impact set, I, in order to generate the program behaviors impacted by the changes in C.

```
1 int a, b;
2 public int test(int x, int y) {
3   if (x > 0) a = x+1;
4   else a = x-1;
5
6   if (y == 0) b = y+1;
7   else b = y+2; // change
8 }
```

The simple program shown above contains two global variables a and b, and two input parameters x and y. Updates to the global variables are made based on the values of the input variables. Suppose the assignment $b := y + 2$ is changed to $b := y - 1$ (line 7). The program statement impacted by this change is the conditional statement ($y == 0$). The execution of the changed statement is control dependent on the false branch of the conditional statement. DiSE uses this impact information to direct symbolic execution. There are four possible paths through the program defined by these path constraints (a) $x > 0 \wedge y == 0$, (b) $x > 0 \wedge y \neq 0$, (c) $x \leq 0 \wedge y == 0$, and (d) $x \leq 0 \wedge y \neq 0$. DiSE, however generates only two paths out of the four. In its default search strategy DiSE will explore paths (a) and (b), and then prune the other paths since all the impacted program behaviors (operations on y and b) have been explored in paths (a) and (b). DiSE has been shown to be an efficient technique for characterizing evolving program behaviors. [5, 45, 50].

```
 1: Input : Assertion free $p \mapsto p_f$; Assertions $A$
 2:         Assertion free $p' \mapsto p'_f$; Assertions $A'$
 3:
 4: $C'_p$ := SyntaticDiff($p_f, p'_f$)
 5: $I'_p$ := ImpactAnalyis($C'_p, p'_f$)
 6:
 7: /* Check I: Impacted behaviors & assertion diffs */
 8: DirectedSymbolicExecution(($p'_f \circ \Delta_{A',A}$), $I_{p'}$)
 9:
10: /* Check II: Impacted behaviors & original assertions */
11: OldAssertions := $\emptyset$
12: for each $a \in A \wedge$ matchedAsserts($a$) $\neq \perp$ do
13:     OldAssertions := OldAssertions $\cup \{a\}$
14: DirectedSymbolicExecution(($p'_f \circ$ OldAssertions), $I_{p'}$)
15:
16: /* Check III: Unchanged behaviors & assertion diffs */
17: $U_{p'}$ := Statements($p'_f$) $\setminus I_{p'}$
18: DirectedSymbolicExecution(($p'_f \circ \Delta_{A',A}$), $U_{p'}$)
```

Figure 3: Three cases for incremental property checking.

3.3.2 Three Types of Property Checks

The three types of property checks in our incremental analysis are shown in Figure 3: (Check I) analyze the impacted program behaviors with respect to the property differences, (Check II) analyze the impacted program behaviors with respect to the preserved properties, and (Check III) analyze unchanged program behaviors with respect to the property differences. Programs p_f and p'_f are assertion-free program versions of p and p' respectively, A and $A_{p'}$ are the sets of assertions in p and p' respectively, and $\Delta_{A',A}$ is the set of assertion differences computed by the algorithm in Figure 2.

We leverage the conservative approximation of impacted behaviors computed by DiSE in the checks in Figure 3. At lines 4 and 5 in Figure 3, the change and impact sets ($C_{p'}$ and $I_{p'}$) are computed from the assertion-free program versions p_f and p'_f. This generates the set of program statements in the code that may be impacted by changes to *only* the code and not to the assertions. The `ImpactAnalysis` computes the set of program statements within the program slice using the change set as a slicing criterion. A forward as well as a backward slice is computed in order to account for all program statements that may impact the program behavior. Control and data dependence information is used by the impact analysis to construct the program slice. This is a conservative analysis—any program statement that may be dependent on a program statement in the change set is included in the impact set. The forward slice computes the flow of impact from the change set to the other statements while the backward slice accounts for control and data flow from program statements to those in the change set.

A violation of an assertion during Check I indicates that a new feature addition or modification does not conform to the new properties. In Check I, we check the assertion differences with respect to the changed behaviors in p'_f, shown at line 8 in Figure 3. A safe approximation of the impacted behaviors is generated by the symbolic execution performed by DiSE (the `DirectedSymbolicExecution` procedure). Symbolic execution is directed along the program statements in the impact set ($I_{p'}$) to generate the set of impacted behaviors in p'. The assertion differences, $\Delta_{A',A}$, which have been rewritten as assertions in p' are checked on-the-fly as the search is directed along the impacted program executions by `DirectedSymbolicExecution`.

A violation of an assertion during Check II indicates a regression error. In Check II, we analyze the original (preserved) assertions with respect to the changed behaviors at lines 10 through 13. The set *OldAssertions* is initialized as empty at line 10. We add to this set all assertions in A that have a matched assertion in p'. This check ensures that assertions removed in A' are not added to the *OldAssertions* set. We check the assertions in the *OldAssertions* set with respect to the changed behaviors in p'. Check II serves as a regression metric where the analysis ensures that changes to the code do not violate the preserved properties.

A violation of an assertion during Check III indicates a potential incorrect update to the property. In Check III, we analyze the assertion differences with respect to the behaviors that are unchanged between p and p'. In order to explore the preserved behaviors we generate the set of program statements that are not impacted by the code changes. At line 17 in Figure 3 the set, $U_{p'}$, contains any program statement that is *not* in the impact set ($I_{p'}$). Now we direct symbolic execution along program statements in $U_{p'}$ to generate the set of unchanged behaviors. We check the assertion differences with respect to these unchanged behaviors.

4. EVALUATION

We empirically evaluate the effectiveness of our technique for incremental property differencing and checking. Our evaluation addresses the following research questions:

∘ **RQ1**: How does the efficiency of checking assertion differences compare with checking the full assertions using symbolic execution?

∘ **RQ2**: How does the cost of checking an implication between an old assertion and a clause in a new assertion compare with checking the clause against the program?

∘ **RQ3**: How does the performance of the incremental property checking, in the presence of code changes, compare with full symbolic execution in detecting (a) regression errors and (b) conformance of new features to additional properties?

For the first two research questions, we restrict the program differences to assertion changes only; for the third research question we analyze programs with changes to both the code and the assertions.

4.1 Tool Support

We implement our approach using Symbolic PathFinder (SPF) [42,43], the symbolic execution extension to the Java PathFinder (JPF) model checker [57] for analyzing Java programs. SPF uses lazy initialization [33] to handle symbolic inputs of complex (user-defined) types. We use CVC3 [8] as the backend solver for all subjects except Apollo which contains non-linear constraints. We use the CORAL [53] solver for Apollo. We use the DiSE extension to JPF to perform incremental property checking [45,50].

Checking assertions with SPF When checking assertions with SPF, we use an interactive approach, stopping at the *first* violation of an assertion. We then remove the failing assertion, and recheck the program until the next assertion fails or SPF completes, e.g., reaches a depth bound. SPF supports finding multiple assertion violations in a single invocation, however, this approach would also find *all* possible ways of violating *each* assertion and result in unnecessary overhead, particularly when our goal is simply to determine for each assertion, if there exists *any* feasible execution that violates it.

4.1.1 *Property Differencing: Checking Implications*

To check if the property Φ_a in assertion a implies a clause, ϕ', (RQ2) we (1) create a symbolic driver method with body "`if (`Φ_a`) assert` ϕ'`;`" where Φ_a is the conjunction of clauses in a, and the formal parameters are based on the free variables of Φ_a and ϕ'; and (2) symbolically execute the driver. If symbolic execution does not lead to an assertion violation, the implication holds; otherwise, it may not hold. Conceptually, symbolic execution of the driver checks the formula "$\forall\ v_0, \ldots, v_k \mid \Phi_a \Rightarrow \phi'$" where $v_0, \ldots, v_k$ are the free variables in Φ_a and ϕ'. If this formula holds, then the clauses in a imply c' since the formula checks the implication for all possible values for $v_0, \ldots, v_k$ and only a subset of these values can reach the control point of a.

If Φ_a or ϕ' invoke other method(s), symbolic execution dynamically traces through the bodies of the invoked methods. However, the cost of implication checking then depends on the number of (bounded) paths in the methods invoked by assertions. To optimize implication checking, we over-approximate the outcome of the calls to boolean methods by introducing non-deterministic boolean variables to conservatively model the results of method invocation [6]. Specifically, (1) for each unique boolean method invocation `m(...)` (based on method name and arguments) in "`if (`Φ_a`) assert` ϕ'`;`", declare a fresh boolean variable and initialize it using a non-deterministic choice (i.e., `Verify.getBoolean()` in SPF); (2) insert these variable declarations before the implication check; and (3) replace all occurrences of boolean method invocations in the implication check by the corresponding boolean variables. Thus, we conservatively capture the behavior of the boolean methods invoked: if no assertion violation is found in the generated program, then the original implication holds. Recall we assume the assertions are without side-effects.

4.2 Artifacts

In our evaluation, we use four artifacts with manually created assertions and three artifacts with dynamically inferred invariants that we transform into Java `assert` statements.

4.2.1 *Triangle Classification Artifact*

The artifacts with manually created assertions include a Java version of the classic triangle classification program by Ammann and Offutt [2] annotated with assertions. The triangle classification program `trityp` takes as inputs three numbers and outputs whether they represent sides of a valid *scalene*, *isosceles*, or *equilateral* triangle, or an *invalid* triangle. The classification logic of the `trityp` program is implemented by the `Triang` method using 53 lines of Java code [1] which seem deceptively simple, but are non-trivial to reason about. The assertions for this artifact were developed following an evolutionary process.

[1]In the textbook by Ammann and Offutt (Figures 3.2–3.3, Lines 29–81, [2])

Table 1: Data structure subjects.

Subject	*#Versions*	*Invariants*
SortedList	5	Acyclicity First element is smallest Last element is largest List is sorted List has unique elements (if unique inserts)
HeapArray	3	Root is largest Max-heap property Reverse-sorted array is max-heap
BST	4	Acyclicity Left-most leaf is smallest Right-most leaf is largest Binary search constraints

Assertions for `trityp`. We consider three versions of assertions for `trityp` (based on the actual experiences of one of the authors). Let A, B, and C be the three input numbers and T represent the output, where $T = 1$ means "scalene", $T = 2$ means "isosceles", $T = 3$ means "equilateral", and $T = 4$ means "invalid".

Version V1. The initial set of the `trityp` assertions includes the following assertions:

```
/*A0*/assert T != 0;
/*A1*/assert A <= 0 || B <= 0 || C <= 0 ? T==4 : true;
/*A2*/assert A+B >= C ? T==4 : true;
/*A3*/assert A+C >= B ? T==4 : true;
/*A4*/assert B+C >= A ? T==4 : true;
/*A5*/assert A==B && B==C ? T==3 : true;
/*A6*/assert A==B && C < A+B ? T==2 : true;
/*A7*/assert A==C && B < A+C ? T==2 : true;
/*A8*/assert B==C && A < B+C ? T==2 : true;
/*A9*/assert A != B && A != C && B != C &&
        A+B > C && A+C > B && B+C > A ? T==1 : true;
```

Version V2. Analysis of the `trityp` example revealed that assertions $A2$–$A8$ are invalid in the initial version and are therefore edited to:

```
/*A2'*/assert A+B <= C ? T==4 : true;
/*A3'*/assert A+C <= B ? T==4 : true;
/*A4'*/assert B+C <= A ? T==4 : true;
/*A5'*/assert A==B && B==C && A>0 ? T==3 : true;
/*A6'*/assert A==B && C < A+B && C > 0 ? T==2 : true;
/*A7'*/assert A==C && B < A+C && B > 0 ? T==2 : true;
/*A8'*/assert B==C && A < B+C && A > 0 ? T==2 : true;
```

V2 of the assertions thus includes the following 10 assertions in order: $A0$, $A1$, $A2'$–$A8'$, $A9$.

Version 3. After checking version V2, assertions $A6'$–$A8'$ in V2 are still invalid and edited to:

```
/*A6"*/assert A==B && A != C && C < A+B && C > 0 ?
                T==2 : true;
/*A7"*/assert A==C && A != B && B < A+C && B > 0 ?
                T==2 : true;
/*A8"*/assert B==C && B != A && A < B+C && A > 0 ?
                T==2 : true;
```

V3 of the assertions thus includes the following ten assertions in order: $A0$, $A1$, $A2'$–$A5'$, $A6''$–$A8''$, $A9$.

4.2.2 Data Structure Artifacts

Three artifacts with manually created assertions are data structure subjects with assertions based on structural representation invariants. Some of the data structure invariants were developed in previous work [13]; we adapt and augment them in this work to create multiple assertion versions.

Assertions for Data Structure Artifacts. Table 1 summarizes the data structure artifacts. For each artifact, we create a symbolic execution *driver*, which is common across all versions: "$T\ o = new\ T(); o.add(x_1); ...; o.add(x_5);$" for symbolic variables $x_1, \dots, x_5$ and data structure type T. Next, we create k versions, where k is the number of invariants for the subject. Let the invariants be $I_1, \dots, I_k$ and versions be $V_1, \dots, V_k$. Each version consists of the symbolic execution driver followed by a sequence of `assert` statements, where V_1 contains "$assert\ I_1;$", and version V_i $(1 < i \leq k)$ contains "$assert\ I_1; ...; assert\ I_i$". Thus, each new version adds the corresponding new assertion.

Table 2: Results for checking the `tritype` assertions.

Version	*CheckFull*			*CheckDiff*				
	#SMT	*#States*	*T[s]*	*#SMT*	*Red.*	*#States*	*T[s]*	*#Conc.*
V1	812	821	18	-	-	-	-	-
V2	763	767	16	32	95.8%	33	1	6
V3	786	787	13	212	73.0%	213	3	1

4.2.3 Artifacts with Synthesized Assertions

We also use three subject programs, WBS, TCAS, and Apollo with mechanically synthesized assertions. These artifacts were used in two recent papers [45, 60] on incremental symbolic execution. The focus of these papers was on changes to code in general and these artifacts did not contain assertions. To synthesize assertions for our experiments, we use the Daikon tool for invariant discovery [19]. Specifically, we apply Daikon on each subject to discover *relevant valid* invariants, which we transform to assertions. Our specific focus is the method *pre-conditions* and *post-conditions* generated by Daikon in Java.

Daikon requires a test suite to execute the program under analysis and detect its likely invariants. Among the artifacts, only TCAS had a test suite available in the Software Infrastructure Repository [52]. We used this test suite containing 1608 tests for TCAS. For WBS and Apollo, we wrote a random test generator to create a test suite with 1000 tests for each artifact. Our goal is to use Daikon to generate relevant valid invariants. However, dynamic invariant discovery can generate invariants which are irrelevant or invalid [46]. To filter such invariants, we remove any invariant that does not mention some method input or the return value, or is invalidated by symbolic execution.

4.3 Results and Analysis

4.3.1 RQ1: Checking Assertion Differences

We check the `trityp` program with respect to each of the three assertion versions comparing checking assertion differences with checking the full assertions using symbolic execution. The results are shown in Table 2. For each artifact and its versions, we show the results for checking full assertions (CheckFull) and the results for checking assertions differences that are computed by our property differencing algorithm (CheckDiff). *#SMT* is the number of SMT calls made, *#States* is the number of states explored during symbolic execution, *T[s]* is the wall clock time in seconds for symbolic execution, and *Red.* is the reduction in SMT calls achieved by checking only the assertion differences. The *#Conc.* column indicates the number of concrete executions used to check the counterexamples from the previous version. The results in Table 2 show the cumulative results for all invocations of SPF. For the `trityp` artifact, our property differencing algorithm reduces the number of assertions that need to be checked by symbolic execution; consequently it reduces the SMT calls by 95.8% for V2 and 73.0% for V3.

Table 3: Subjects with mechanically synthesized assertions using Daikon.

Subject	*Method*	*Daikon application*		*Version generation*	
		#Tests	*#Invariants*	*#Versions*	*#Invariants in each version*
WBS	`update`	1000	4	4	*V1*: 1, *V2*: 2, *V3*: 3, *V4*: 4
TCAS	`alt_sep_test`	1608	35	5	*V1*: 15, *V2*: 20, *V3*: 25, *V4*: 30, *V5*: 35
	`Non_Crossing_Biased_Climb`	1608	53	5	*V1*: 10, *V2*: 20, *V3*: 30, *V4*: 40, *V5*: 53
	`Non_Crossing_Biased_Descend`	1608	56	5	*V1*: 15, *V2*: 25, *V3*: 35, *V4*: 45, *V5*: 56
Apollo	`Reaction_Jet_Control0.Main1`	1000	5	5	*V1*: 1, *V2*: 2, *V3*: 3, *V4*: 4, *V5*: 5

Table 4: Results for checking data structure assertions.

Subject	*Version*	*CheckFull*			*CheckDiff*			
		#SMT	*#States*	*T[s]*	*#SMT*	*Red.*	*#States*	*T[s]*
SortedList	*V1*	238	239	6	-	-	-	-
	V2	1438	1439	30	1438	0%	1439	24
	V3	4630	4631	101	3430	25.9%	3431	59
	V4	8958	8959	209	1198	86.6%	1199	21
	V5	10316	10317	204	652	93.6%	653	12
HeapArray	*V1*	1492	1493	28	-	-	-	-
	V2	3510	3511	73	2340	33.3%	2341	44
	V3	4292	4293	99	1104	74.2%	1105	22
BST	*V1*	2162	2163	64	-	-	-	-
	V2	9364	9365	225	9364	0%	9365	223
	V3	16566	16567	392	9364	43.4%	9365	221
	V4	36256	36257	836	21852	39.7%	21853	451

In Table 4 we present the results of checking full assertions and checking assertion differences on the data structure artifacts. Versions $V1$ and $V2$ of `SortedList` and `BinarySearchTree` (BST) artifacts contain properties that assert acyclicity. These assertions are checked using lazy initialization that concretizes complex data structures and does not require any constraint solving during symbolic execution. Currently our reductions are targeted for programs and properties that *do use* constraint solving; hence the number of states generated and the number of SMT calls in the CheckFull and CheckDiff columns for version $V2$ of `SortedList` and `BinarySearchTree` (BST) artifacts are the same. Checking assertion differences in other versions requires fewer SMT calls than checking all assertions and costs less in time and states visited. Checking assertion differences in the `SortedList` example provides up to 93.6% reduction in the number of SMT calls compared to checking full assertions. In the `HeapArray` artifact, we observe up to 74.2% reduction in the number of SMT calls compared to checking full assertions. Whereas in the `BinarySearchTree` example, checking assertion differences provides up to 43.4% reduction in the number of SMT calls compared to checking full assertions. Overall, experiments with `trityp`, `SortedList`, `HeapArray`, and `BinarySearchTree` show that checking assertion differences based on the property differencing algorithm outperforms checking full assertions in terms of the number of SMT solver calls, number of states explored, and the time taken for symbolic execution.

Table 3 summarizes the artifacts with mechanically synthesized assertions and lists the methods we consider, the number of tests used to run Daikon, the relevant valid invariants reported by Daikon, the number of assertion versions we create using those invariants, and the number of assertions in each version. We create four and five versions of the WBS and Apollo artifacts respectively, using the relevant and valid invariants generated by Daikon. The first version consists of one randomly selected invariant, and each subsequent version contains all the invariants from the previous version and one additional invariant also selected at random. Daikon's invariant generation is particularly effective for TCAS [19], so we are able to create versions with larger numbers of relevant valid invariants. We create five versions for each chosen method in TCAS, where the first version consists of randomly selected invariants, and each subsequent version contains a number of additional invariants also selected at random.

Table 5: Results for checking WBS, TCAS, and Apollo.

Subject	*Version*	*CheckFull*			*CheckDiff*			
		#SMT	*#States*	*T[s]*	*#SMT*	*Red.*	*#States*	*T[s]*
WBS	*V1*	72	73	1	-	-	-	-
	V2	84	85	1	72	14.2%	73	1
	V3	96	97	1	72	25%	73	1
	V4	108	109	1	72	33.3%	73	1
TCAS$_1$	*V1*	3182	3183	155	-	-	-	-
	V2	4270	4271	210	1822	57.3%	1823	89
	V3	4814	4815	222	1278	73.4%	1279	62
	V4	5494	5495	249	1414	74.2%	1415	68
	V5	6582	6583	323	1822	72.3%	1823	89
TCAS$_2$	*V1*	2054	2055	129	-	-	-	-
	V2	3014	3015	193	2054	31.8%	2055	136
	V3	3974	3975	252	2054	48.3%	2055	125
	V4	5510	5511	348	2630	52.5%	2631	164
	V5	7046	7047	457	2630	62.6%	2631	159
TCAS$_3$	*V1*	7686	7687	483	-	-	-	-
	V2	8934	8935	555	6918	22.5%	6919	426
	V3	9894	9895	621	6630	32.9%	6631	407
	V4	10854	10855	680	6630	38.9%	6631	387
	V5	11910	11911	724	6726	43.5%	6727	409
Apollo	*V1*	1014	1016	590	-	-	-	-
	V2	1557	1559	946	1020	34.4%	1022	544
	V3	1728	1730	1100	591	65.7%	593	266
	V4	7330	7332	4100	6937	5.3%	6939	3382
	V5	TO	TO	>28800	29729	N/A	29731	15100

In all the versions, each assertion contains a single clause that corresponds to one relevant valid invariant. We use information about how the different versions are constructed in order to manually mark the changed and added assertions between consecutive versions. However, a Unix diff could be used to compute the changes, and it only takes a few milliseconds to run Unix diff for any two consecutive versions.

Table 5 presents the results of checking full assertions and checking assertion differences on WBS, TCAS, and Apollo. TCAS$_1$ refers to first TCAS method listed in Table 3, i.e, `TCAS.alt_sep_test`, TCAS$_2$ refers the second, etc. Checking full assertions in V5 of Apollo times out (TO) after 8 hours, whereas, checking assertion differences based on property differencing successfully completes for all artifacts. Moreover, checking assertion differences outperforms checking full assertions in terms of states visited and time taken for symbolic execution, and provides between 5.3% and 74.2% reduction in SMT calls across the different versions. While these results show the benefit of using iProperty for more efficient checking of these subjects, the underlying cost of symbolic execution still is a considerable factor in the scalability of the overall incremental approach.

Table 6: Implication checking results using symbolic execution. "*Exp*$_1$" is the first experiment. "*Exp*$_2$" is the second experiment.

Subject	Check implication "$\phi_a \Rightarrow \phi_{a'}$"									*Check* "`assert` $\phi_{a'}$;" in code								
	#SMT			*#States*			*T[s]*			*#SMT*			*#States*			*T[s]*		
	min	*max*	*ave*	*min*	*max*	*ave*	*min*	*max*	*ave*	*min*	*max*	*ave*	*min*	*max*	*ave*	*min*	*max*	*ave*
Exp$_1$*: WBS*	4	6	4	5	7	5	<1	<1	<1	72	72	72	73	73	73	1	1	1
Exp$_2$*: WBS*	4	4	4	5	5	5	<1	<1	<1	72	72	72	73	73	73	1	1	1
Exp$_1$*: TCAS*$_1$	2	2	2	3	3	3	<1	<1	<1	870	1278	951	871	1279	952	41	60	45
Exp$_2$*: TCAS*$_1$	4	8	4	5	9	5	<1	<1	<1	870	870	870	871	871	871	42	44	42
Exp$_1$*: TCAS*$_2$	1	2	1	3	4	3	<1	<1	<1	1190	1190	1190	1191	1191	1191	67	72	69
Exp$_2$*: TCAS*$_2$	0	4	2	5	7	5	<1	<1	<1	1190	1478	1305	1191	1479	1306	68	97	78
Exp$_1$*: TCAS*$_3$	2	2	2	3	3	3	<1	<1	<1	5766	5766	5766	5767	5767	5767	336	356	344
Exp$_2$*: TCAS*$_3$	4	4	4	5	5	5	<1	<1	<1	5766	5766	5766	5767	5767	5767	333	339	335
Exp$_1$*: Apollo*	0	0	0	6	6	6	<1	<1	<1	591	29729	7858	593	29731	7860	270	15100	3987
Exp$_2$*: Apollo*	0	0	0	7	7	7	<1	<1	<1	591	29729	7858	593	29731	7860	270	15100	3987

Table 7: Results for checking regression errors.

Version	*SPF*			*DiSE*			
	#SMT	*#States*	*T[s]*	*#SMT*	*Red.*	*#States*	*T[s]*
WBS'	15	16	1	15	0%	16	1
TCAS$_1$*'*	3238	3239	138	3238	0%	3239	163
TCAS$_2$*'*	2198	2199	155	107	95.1%	108	7
TCAS$_3$*'*	7958	7959	506	158	98.0%	159	10
Apollo'	158	159	456	4	97.5%	5	4

Table 8: Results for checking conformance between code change and property change.

Version	*SPF*			*DiSE*			
	#SMT	*#States*	*T[s]*	*#SMT*	*Red.*	*#States*	*T[s]*
WBS'	15	16	1	15	0%	16	2
TCAS$_1$*'*	1878	1879	94	1878	0%	1879	93
TCAS$_2$*'*	2198	2199	83	107	95.1%	108	7
TCAS$_3$*'*	7062	7063	418	142	98.0%	143	10
Apollo'	158	159	254	4	97.5%	5	2

4.3.2 RQ2: Implication Checking

To address *RQ2* we analyzed assertion *pairs*, (a, a') where assertion a is in program p and its matching assertion a' is in program p'. In our experiments Φ_a and $\Phi_{a'}$ each contain a single clause ϕ_a and $\phi_{a'}$ respectively. Hence, for each assertion pair we symbolically execute "`if (`ϕ_a`) assert` $\phi_{a'}$;" to check if the implication holds as described in Section 3.2. We compare the cost of checking the implication with the cost of checking "`assert` $\phi_{a'}$;" in the relevant method body at the corresponding exit point.

We conduct two different experiments, each with five assertion pairs. In the first experiment, we create assertion pairs by selecting an invariant (ϕ_a) at random from the set of relevant valid invariants. In the second experiment, we create pairs by applying mechanical transformations [1]; for invariant ϕ_a selected at random: (1) if ϕ_a contains the '`>`' comparison operator, we create $\phi_{a'}$ by mutating '`>`' to '`>=`' to form assertion pair $\langle a, a' \rangle$; (2) if ϕ_a contains the '`<`' comparison operator, we create $\phi_{a'}$ by mutating '`<`' to '`<=`' to form assertion pair $\langle a, a' \rangle$; (3) if ϕ_a contains a numeric expression of the form "*l op r*" we generate $\phi_{a'}$ by adding a constant $c \in \{-100, \ldots, 100\}$, selected at random, to both sides of the expression in ϕ_a to form assertion pair $\langle a, a' \rangle$; and (4) if ϕ_a invokes a Daikon library method, we select an invariant $\phi_{a'}$ ($\neq \phi_a$) at random to create assertion pair $\langle (\phi_a \wedge \phi_{a'}) \vee (\phi_a \wedge \neg\phi_{a'}), a \rangle$.

Table 6 shows experimental results for implication checking for the two experiments. All implications for the first experiment were false (as expected since implication is unlikely to hold between two invariants selected at random); all implications for the second experiment were true (as expected given the design of the transformations). Thus, the two experiments together cover both possible outcomes for implication checking. Some invariants, including all of the Apollo invariants, invoke Daikon library methods, e.g., `daikon.Quant.memberOf(this.NumeratorTerms_5557, arg1)`, and therefore, the resulting implication check is handled by our implication check optimization which uses non-deterministic choice to model boolean method invocations (Section 4.1.1). Thus, some cells have 0 SMT calls – SPF state exploration does not require SMT if the implication only consists of boolean methods and boolean operators.

The results show that the number of SMT calls for the implication check $\phi_a \Rightarrow \phi_{a'}$ is at most 8.3%, 1%, and 0% for WBS, TCAS, and Apollo respectively, of the cost to check "`assert` $\phi_{a'}$;" in the code. Overall, the cost of checking an implication between an assertion in p and a new clause in the assertion a' is substantially less than the cost of checking the clause against p'.

4.3.3 RQ3: Incremental Assertion Checking

To address RQ3 we analyze programs that contain changes to both code and properties. We compare the performance of incremental property checking with full symbolic execution in (a) detecting regression errors, and (b) checking conformance of new features to additional properties. We create the version pairs by manually making changes to the code that were annotated with relevant and valid Daikon invariants; we then also make changes to the properties by adding additional Daikon invariants. We choose to make manual changes to the code in order to simulate potential regression errors and non-conformance between code and properties. We create one version for each subject listed in Table 5.

To compare the performance in finding regression errors we perform DiSE and full symbolic execution as implemented in SPF on the program p' annotated with the *OldAssertions* as defined in Check II Figure 3 of Section 3.3. An assertion violation in p' is indicative of a regression error. Table 7 shows the experimental results for this experiment. DiSE and SPF make the same number of SMT calls for WBS' and TCAS$_1$'. For TCAS$_1$', all the paths in the code are marked impacted by the code changes, thus DiSE explores all the paths; for WBS', both SPF and DiSE detect an assertion violation. While in other versions, DiSE achieves substantial reduction, for example, there is 97.5% reduction in the number of SMT calls in Apollo'. The same pattern of results is seen in the the number of states explored by symbolic execution in SPF and DiSE and also total wall clock time taken in seconds. Due to the overhead of static analysis performed by DiSE, for TCAS$_1$', DiSE takes a small amount of additional analysis time compared to symbolic execution in SPF; however, even when reductions are not possible, the overhead is very small.

In order to check conformance between code changes and properties changes, we perform DiSE and full symbolic execution as implemented in SPF on p' with the annotated property differences $\Delta_{A',A}$ as described in Check I of Figure 3. We compare the cost of performing DiSE with the cost of performing symbolic execution in SPF in each experiment. Table 8 shows the results. In a manner similar to the results observed in Table 7, for some versions, e.g., WBS' and TCAS_1', DiSE and symbolic execution in SPF explore the same number of states and make the same number of SMT calls. For some versions, however, such TCAS_2', DiSE makes fewer SMT calls, explores fewer states, and takes less time compared to symbolic execution in SPF.

4.4 Threats to Validity

The primary threats to external validity in our study are the use of SPF and DiSE, the selection of artifacts and properties, and the changes made to create versions of the code and properties. Although our observations may not generalize to other artifacts, properties, and changes, we attempted to mitigate these threats by analyzing multiple artifacts, all of which have been used in previous studies of symbolic execution based techniques, with respect to properties (assertions) that were both manually developed and mechanically synthesized. Further evaluation of our technique on a broader range of program types, property specifications, and change types would address this threat.

The primary threats to internal validity are the potential faults in the implementation of our algorithms and in SPF and DiSE. We controlled for this threat by testing the tools and implementations of the algorithms on examples that could be manually verified. With respect to threats to construct validity, the metrics we selected to evaluate the cost of our algorithms are commonly used to measure the cost of symbolic execution based techniques.

5. RELATED WORK

Incremental program analysis techniques which leverage change impact analysis results [4, 5, 28, 44, 45, 48, 59] have been widely studied to help reduce the cost of program analysis by enabling incremental analysis of previously checked programs. Recent work has also focused on techniques which reuse verification results to help reduce the cost of program analysis [10, 56, 60] and reuse of reachability information through similarity checking of test goals represented as automata to reduce the cost of test input generation [9]. iProperty partitions the analysis of code and property specifications, leveraging the results of a change impact analysis and reusing the verification results from checking the previous version of the program against the property specifications to reduce the cost of property checking.

A number of recent techniques optimize symbolic execution for one program version using compositional analysis [23], abstraction [3], and partitioning [54]. The idea of optimizing symbolic execution in the context of code that undergoes change was introduced by DiSE [45], which first uses a static analysis as we describe and utilize in this paper. An alternative optimization of storing results of symbolic execution and re-using them is taken by Green [56] and Memoise [60]. Since property differencing likely reduces the number of paths to explore using symbolic execution, we expect it to offer a reduction in the exploration space for Memoise and Green, similar to DiSE.

Recent work explores program differencing in the context of assertions [32, 35, 36, 47]. Lahiri et al. present differential contract checking [36] which uses static analysis to determine possible inputs that will cause a contract to evaluate differently between two versions. Joshi et al. employ assertion checking to limit false alarms during static concurrency analysis of open systems [32]. In more recent work, Lahiri et al. present a differential assertion checking technique (DAC) for comparing relative correctness of the program [35]: *do the two programs versions perform consistent memory accesses?*, at a cost that is lower than absolute correctness.

We believe that core algorithms in iProperty can be extended to handle other lightweight specifications as well, e.g., JML [37], Eiffel [39], and Spec# [7]. These specification languages provide developers the ability to specify properties at a component, module, or method-level in a manner very similar to assertions.

There is a large body of the work on property-based slicing and property-aware testing and verification of programs [12, 15, 18, 21, 25, 26] as well as on change-impact analysis, e.g., for regression test selection [27, 41, 49]. The key difference between iProperty and previous work is our property differencing technique and its synergistic application with change-impact analysis that characterizes impacted behaviors to enable more efficient checking of code conformance to behavioral properties. Property differencing as a stand-alone problem has also been addressed by a number of previous projects, e.g., in the context of UML models [40, 58], and Object-Z [55]; however, previous work on property differencing has not considered using the differences to optimize checking of code.

6. CONCLUSION

This paper introduced *iProperty*, a novel approach to compute differences between properties of related programs in a manner that facilitates more efficient incremental checking of conformance of programs to properties. The key novelty of iProperty is to compute logical differences between old and new versions of properties that undergo change and facilitate two development scenarios that form our focus: (1) writing properties correctly for an existing program, and (2) checking changes to code and properties as they coevolve. We evaluate iProperty on Java programs with manually written and mechanically synthesized assertions. The experimental results show that iProperty reduces the number of SMT solver calls – a key metric for the cost of symbolic execution – in comparison with the conventional approach. We believe iProperty provides a promising approach to not only reduce the cost of checking properties but also to make it easier to formulate correct properties – manually or mechanically. While our focus in this paper was on symbolic execution, we believe property differencing can enable scalable incremental checking using other software analyses, such as model checking and static analysis.

7. ACKNOWLEDGMENTS

This work was funded in part by the National Science Foundation (NSF Grant Nos. CCF-0845628, CCF-1319688, and CNS-1239498).

8. REFERENCES

[1] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical report, Georgia Institute of Technology, Atlanta, GA, 1979.

[2] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[3] S. Anand, C. S. Pasareanu, and W. Visser. Symbolic execution with abstraction. *International Journal on Software Tools for Technology Transfer (STTT)*, 11:53–67, January 2009.

[4] T. Apiwattanapong, R. Santelices, P. K. Chittimalli, A. Orso, and M. J. Harrold. MATRIX: Maintenance–oriented testing requirements identifier and examiner. In *TAIC PART*, pages 137–146, 2006.

[5] J. Backes, S. Person, N. Rungta, and O. Tkachuk. Regression verification using impact summaries. In *SPIN '13*, 2013.

[6] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, pages 203–213, 2001.

[7] M. Barnett, K. R. M. Leino, and W. Schulte. The spec# programming system: an overview. In *CASSIS*, pages 49–69, Berlin, Heidelberg, 2005. Springer-Verlag.

[8] C. Barrett and C. Tinelli. CVC3. In *CAV*, pages 298–302, July 2007.

[9] D. Beyer, A. Holzer, M. Tautschnig, and H. Veith. Information reuse for multi-goal reachability analyses. In *ESOP*, pages 472–491, 2013.

[10] D. Beyer, S. Löwe, E. Novikov, A. Stahlbauer, and P. Wendler. Precision reuse for efficient regression verification. In *ESEC/FSE*, pages 389–399, 2013.

[11] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury. Partition-based regression verification. In *ICSE*, pages 302–311, 2013.

[12] R. H. Bordini, M. Fisher, M. Wooldridge, and W. Visser. Property-based slicing for agent verification. *J. Log. and Comput.*, 19(6):1385–1425, Dec. 2009.

[13] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ISSTA*, 2002.

[14] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[15] G. Canfora, A. Cimitile, and A. D. Lucia. Conditioned program slicing. *Information & Software Technology*, 40(11-12):595–607, 1998.

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

[17] L. A. Clarke and D. S. Rosenblum. A historical perspective on runtime assertion checking in software development. *SIGSOFT Software Engineering Notes*, 31(3), 2006.

[18] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. Bandera: a source-level interface for model checking java programs. In *ICSE*, pages 762–765, 2000.

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

[20] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gail. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[21] C. Fox, S. Danicic, M. Harman, and R. M. Hierons. Backward conditioning: A new program specialisation technique and its application to program comprehension. In *IWPC*, pages 89–97, 2001.

[22] P. Godefroid. Model checking for programming languages using verisoft. In *POPL*, pages 174–186, 1997.

[23] P. Godefroid. Compositional dynamic test generation. In *POPL*, pages 47–54, 2007.

[24] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, pages 213–223, 2005.

[25] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: a new algorithm for property checking. In *FSE*, pages 117–127, 2006.

[26] M. Harman and S. Danicic. Using program slicing to simplify testing. *Software Testing, Verification and Reliability*, 5(3):143–162, 1995.

[27] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java software. In *OOPSLA*, pages 312–326, 2001.

[28] T. A. Henzinger, R. Jhala, R. Majumdar, M. A. A, S. E. Engineering, and C. Sciences. Extreme model checking. In *In International Symposium on Verification: Theory and Practice, LNCS 2772*, pages 332–358. Springer, 2003.

[29] G. J. Holzmann. The model checker spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[30] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *ISSTA*, 2000.

[31] J. A. Jones. *Semi-Automatic Fault Localization*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, 2008.

[32] S. Joshi, S. K. Lahiri, and A. Lal. Underspecified harnesses and interleaved bugs. In *POPL*, pages 19–30, 2012.

[33] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, pages 553–568, 2003.

[34] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[35] S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In *ESEC/FSE 2013*, pages 345–355. ACM, 2013.

[36] S. K. Lahiri, K. Vaswani, and T. Hoare. Differential static analysis: Opportunities, applications, and challenges. In *FoSER*, pages 201–204, 2010.

[37] 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.

[38] D. Marinov and S. Khurshid. Testera: A novel framework for automated testing of java programs. In *ASE*, pages 22–, 2001.

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

[40] D. Ohst, M. Welle, and U. Kelter. Differences between versions of uml diagrams. In *ESEC/FSE-11*, pages 227–236, New York, NY, USA, 2003. ACM.

[41] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *SIGSOFT FSE*, pages 241–251, 2004.

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

[43] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta. Symbolic Pathfinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering*, pages 1–35.

[44] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *FSE*, pages 226–237, 2008.

[45] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *PLDI*, pages 504–515, 2011.

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

[47] D. A. Ramos and D. R. Engler. Practical, low-effort equivalence verification of real code. In *SPIN*, pages 669–685, 2011.

[48] X. Ren, B. G. Ryder, M. Stoerzer, and F. Tip. Chianti: a change impact analysis tool for Java programs. In *ICSE*, pages 664–665, 2005.

[49] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, 1997.

[50] N. Rungta, S. Person, and J. Branchaud. A change impact analysis to characterize evolving program behaviors. In *ICSM*, 2012.

[51] O. Sery, G. Fedyukovich, and N. Sharygina. Incremental upgrade checking by means of interpolation-based function summaries. In *FMCAD*, pages 114–121, 2012.

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

[53] M. Souza, M. Borges, M. d'Amorim, and C. S. Păsăreanu. CORAL: solving complex constraints for Symbolic PathFinder. In *NFM*, pages 359–374, 2011.

[54] M. Staats and C. Păsăreanu. Parallel symbolic execution for structural test generation. In *ISSTA*, pages 183–194, 2010.

[55] F. Taibi, M. J. Alam, and J. Abdullah. On differencing object-oriented formal specifications. *Journal of Object Technology*, 9(1):183–198, Jan. 2010.

[56] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: Reducing, rusing and recycling constraints in program analysis. In *ESEC/FSE*, 2012.

[57] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *ASE*, Grenoble, France, 2000.

[58] Z. Xing and E. Stroulia. Differencing logical uml models. *Automated Software Eng.*, 14(2):215–259, June 2007.

[59] G. Yang, M. B. Dwyer, and G. Rothermel. Regression model checking. In *ICSM*, pages 115–124, 2009.

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

[61] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid. Invariant discovery guided by symbolic execution. In *Java PathFinder Workshop*, 2013.