Property Differencing for Incremental Checking

Guowei Yang1, Sarfraz Khurshid2, Suzette Person3, Neha Rungta4
1Department of Computer Science, Texas State University, San Marcos, TX 78666, USA
2Department of Electrical and Computer Engineering, University of Texas, Austin, TX 78712, USA
3NASA Langley Research Center, Hampton, VA 23681, USA
4NASA Ames Research Center, Moffett Field, CA 94035, USA
1gyang@txstate.edu, 2khurshid@utexas.edu, 3suzette.person@nasa.gov, 4neha.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

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—towards 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 $\Phi$ about program $p$ holds at a control point $l$ in $p$ and $\Phi$ implies another property $\Phi'$, $\Phi'$ 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].

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 ‘\(<\sim\)’, 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).

```java
1 assert x == y || x == z ? m = x : true; //#1
2 assert y == z ? m = y : true; //#2
3 assert x <= z && z <= y ? m = z : true; //#3
4 assert y <= z && z <= x ? m = z : true; //#4
```

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

```java
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. 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 ∈ A is an Java assertion “assert Φ” where Φ specifies one or more properties as a conjunction of clauses, Φ = φ₀ ∧ ... ∧ φₘ (m ≥ 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 differing 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 Outₚ : A → {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 result of checking a are not known.

**Definition 3.3.** Counterexample is a map Cexₚ : A → 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 Outₚ(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 → 2^{clauses} that returns the set of clauses contained in the conjunction, Φ₀ := φ₀ ∧ ... ∧ φₘ., for the assertion a ∈ A.
procedure PropertyDifferencing(A’, Cex_p, Out_p)  
1: ∀a’ ∈ A’, Cex_p(a’) := φ
2: for each a’ ∈ A’ do  
3: if matchedAssertions(a’) ̸= ⊥ then  
4: ⟨outcome, a, a’⟩ := AssertDiff(a, a’)  
5: else ⟨outcome, a, a’⟩ := ⟨Unknown, transform(a)⟩  
6: Out_p(a’):= outcome  
7: Δ_p(A’) := a, a’  
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 ⟨Unknown, transform(a’)⟩  
13: if LogicalDifference(a’, a) == φ then  
14: return ⟨Out_p(a), 0⟩  
15: if Out_p(a) == Not Fail then  
16: return ⟨Unknown, ⟨LogicalDifference(a’, a)⟩⟩  
17: else if Out_p(a) == Fail then  
18: return ⟨Unknown, ⟨LogicalDifference(a’, a)⟩⟩  

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 \( \{ \delta’ \in transform(a) \mid \Phi_a \not\Rightarrow \delta’ \} \).

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 \( \bigwedge_{i=0}^{|δ|} \delta_i \) 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 \( \bigwedge_{i=0}^{|δ|} \delta_i \) 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’, ii) the counter-example map, Cex_p’, and iii) the map of assertions differences, \( \Delta_{A’_p,A} \), 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 matchedAssertions at line 3 in Figure 2. This function takes as input an assertion statement in p’ and returns the corresponding assertion statement in p; when no corresponding assertion statement is found, it returns \( \perp \). 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 \( \Delta_{A’_p,A} \) in order to be checked against p’ (line 7). When the matchedAssertions 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 \( \Delta_{A’_p,A} \) 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 \( \Delta_{A’_p,A} \).

We also use previously generated counterexamples to potentially reduce the number of properties that need to be checked using 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’_r, such that Out_p’(a’_r) := 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., \( \Delta_{A’_p,A}(a’_r) := \emptyset \), and (c) counterexample map are updated with the error trace.

The algorithm in Figure 2 computes the map of assertion differences, \( \Delta_{A’_p,A} \). 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 \( \Delta_{A’_p,A} \) is mapped into assert statement assert \( \Phi \) 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) \[5, 50\] 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 \(C\).

```plaintext
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:   if (y == 0) b = y+1;
6:   else b = y+2; // change
7: }
```

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) \(x > 0 \land y = 0\), (b) \(x > 0 \land y \neq 0\), (c) \(x \leq 0 \land y = 0\), and (d) \(x \leq 0 \land y \neq 0\). 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. \[5, 45, 50\].

#### 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 \((C_{p_f} \text{ and } I_{p_f})\) 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 \(\text{ImpactAnalysis}\) 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 \(\text{DirectedSymbolicExecution}\) 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 violations of 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 \(\Phi_a\) in assertion \(a\) implies a clause, \(\phi'\), (RQ2) we (1) create a symbolic driver method with body “if (\(\Phi_a\)) assert \(\phi'\);” where \(\Phi_a\) is the conjunction of clauses in \(a\), and the formal parameters are based on the free variables of \(\Phi_a\) and \(\phi'\); 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 \(\Phi_a\) and \(\phi'\). If this formula holds, then the clauses in \(a\) imply \(\phi'\) 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 \(\Phi_a\) or \(\phi'\) 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 \(m(\ldots)\) (based on method name and arguments) in “if (\(\Phi_a\)) assert \(\phi'\);”, declare a fresh boolean variable and initialize it using a non-deterministic choice (i.e., \(\text{Verify.getBoolean()}\) 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 \(\text{trityp}\) 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 \(\text{trityp}\) program is implemented by the \(\text{Triang}\) method using 53 lines of Java code which seem deceptively simple, but are non-trivial to reason about. The assertions for this artifact were developed following an evolutionary process.

\footnote{In the textbook by Ammann and Offutt (Figures 3.2–3.3, Lines 29–81, [2])}
we create a symbolic execution
them in this work to create multiple assertion versions.

Some of the data structure invariants
in order:

- Root is largest
- Max-heap property
- Reverse-sorted array is max-heap

**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:

```plaintext
/*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 <= 0 || B <= 0 || C <= 0 ? T==4 : true;
/*A6*/ assert A==A && B < A+B ? T==2 : true;
/*A7*/ assert A==B && C < A+B ? T==2 : true;
/*A8*/ assert B+C >= A ? T==4 : true;
```

**Version V2**
The analysis of the trityp example revealed that assertions A2–A8 are invalid in the initial version and are therefore edited to:

```plaintext
```

**Version V3**
After checking version V2, assertions A6′–A8′ in V2 are still invalid and edited to:

```plaintext
/*A0*/ assert A==A && B < C ? T==4 : true;
/*A1*/ assert A==A && B < C ? T==4 : true;
/*A2*/ assert A==A && B < C ? T==4 : true;
/*A3*/ assert A==A && B < C ? T==4 : true;
/*A4*/ assert A==A && B < C ? T==4 : true;
/*A5*/ assert A==A && B < C ? T==4 : true;
/*A6*/ assert A==A && B < C ? T==4 : true;
/*A7*/ assert A==A && B < C ? T==4 : true;
/*A8*/ assert A==A && B < C ? T==4 : true;
```

V3 of the assertions thus includes the following 10 assertions in order: A0, A1, A2–A8, A9.

**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 [10]. 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 1068 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 execution driver followed by a sequence of `assert` statements, where V_i contains “assert I_i;”, and version V_i (1 ≤ i ≤ k) contains “assert I_i;...;assert I_k”. Thus, each new version adds the corresponding new assertion.

Table 2: Results for checking the trityp assertions.

<table>
<thead>
<tr>
<th>Version</th>
<th>CheckFull</th>
<th>CheckDiff</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#SMT</td>
<td>#States</td>
</tr>
<tr>
<td>V1</td>
<td>812</td>
<td>821</td>
</tr>
<tr>
<td>V2</td>
<td>763</td>
<td>767</td>
</tr>
<tr>
<td>V3</td>
<td>796</td>
<td>787</td>
</tr>
</tbody>
</table>

Across all versions: “T o = new T(); o.add(x1); ...; o.add(x5);” for symbolic variables x1,...,x5 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,...,I_k and versions be V_1,...,V_k. Each version consists of the symbolic execution driver followed by a sequence of `assert` statements, where V_i contains “assert I_i;”, and version V_i (1 ≤ i ≤ k) contains “assert I_i;...;assert I_k”. Thus, each new version adds the corresponding new assertion.
evant and valid invariants generated by Daikon. The first
variants reported by Daikon, the number of assertion ver-
ified assertions and lists the methods we consider, the
number of SMT solver calls, number of states explored, and the time
outperforms checking full assertions in terms of the number
of assertions that correspond to one relevant valid invariant. We use in-
variants also selected at random. Daikon’s invariant generation is particularly e

Table 3: Subjects with mechanically synthesized assertions using Daikon.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Method</th>
<th># Tests</th>
<th># Invariants</th>
<th>Version generation</th>
<th># Invariants in each version</th>
</tr>
</thead>
<tbody>
<tr>
<td>WBS</td>
<td>update</td>
<td>1000</td>
<td>4</td>
<td>V1: 1, V2: 2, V3: 3, V4: 4</td>
<td></td>
</tr>
<tr>
<td>TCAS</td>
<td>alt_sep_test</td>
<td>1008</td>
<td>30</td>
<td>V1: 16, V2: 20, V3: 24, V4: 30, V5: 35</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Non_Crossing_Biased_Cimb</td>
<td>1008</td>
<td>5</td>
<td>V1: 10, V2: 20, V4: 40, V5: 63</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Non_Crossing_Biased_Descend</td>
<td>1008</td>
<td>6</td>
<td>V1: 16, V2: 25, V3: 36, V4: 45, V5: 66</td>
<td></td>
</tr>
<tr>
<td>Apollo</td>
<td>Reaction_set_Control0.Main1</td>
<td>1000</td>
<td>5</td>
<td>V1: 1, V2: 2, V3: 3, V4: 4, V5: 5</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Results for checking data structure assertions.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Version</th>
<th>CheckFull</th>
<th>CheckDiff</th>
</tr>
</thead>
<tbody>
<tr>
<td>SortedList</td>
<td>V1</td>
<td>208</td>
<td>118.2%</td>
</tr>
<tr>
<td></td>
<td>V2</td>
<td>213</td>
<td>114.9%</td>
</tr>
<tr>
<td></td>
<td>V3</td>
<td>1438</td>
<td>113%</td>
</tr>
<tr>
<td></td>
<td>V4</td>
<td>8958</td>
<td>112%</td>
</tr>
<tr>
<td></td>
<td>V5</td>
<td>10318</td>
<td>111%</td>
</tr>
<tr>
<td>HeapArray</td>
<td>V1</td>
<td>1102</td>
<td>59.7%</td>
</tr>
<tr>
<td></td>
<td>V2</td>
<td>3510</td>
<td>33.3%</td>
</tr>
<tr>
<td></td>
<td>V3</td>
<td>4292</td>
<td>25.7%</td>
</tr>
<tr>
<td>BST</td>
<td>V1</td>
<td>2162</td>
<td>49.2%</td>
</tr>
<tr>
<td></td>
<td>V2</td>
<td>9364</td>
<td>44.3%</td>
</tr>
<tr>
<td></td>
<td>V3</td>
<td>16566</td>
<td>41.4%</td>
</tr>
<tr>
<td></td>
<td>V4</td>
<td>36256</td>
<td>40.7%</td>
</tr>
</tbody>
</table>

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 not 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 versions in each method. 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.

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: Results for checking WBS, TCAS, and Apollo.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Version</th>
<th>CheckFull</th>
<th>CheckDiff</th>
</tr>
</thead>
<tbody>
<tr>
<td>WBS</td>
<td>V1</td>
<td>2126</td>
<td>118.2%</td>
</tr>
<tr>
<td></td>
<td>V2</td>
<td>9364</td>
<td>114.9%</td>
</tr>
<tr>
<td></td>
<td>V3</td>
<td>16566</td>
<td>113%</td>
</tr>
<tr>
<td></td>
<td>V4</td>
<td>36256</td>
<td>112%</td>
</tr>
<tr>
<td></td>
<td>V5</td>
<td>10318</td>
<td>111%</td>
</tr>
<tr>
<td>TCAS</td>
<td>V1</td>
<td>4270</td>
<td>59.7%</td>
</tr>
<tr>
<td></td>
<td>V2</td>
<td>3014</td>
<td>33.3%</td>
</tr>
<tr>
<td></td>
<td>V3</td>
<td>193</td>
<td>25.7%</td>
</tr>
<tr>
<td></td>
<td>V4</td>
<td>5494</td>
<td>49.2%</td>
</tr>
<tr>
<td></td>
<td>V5</td>
<td>6582</td>
<td>44.3%</td>
</tr>
<tr>
<td>TCAS</td>
<td>V1</td>
<td>29731</td>
<td>59.7%</td>
</tr>
<tr>
<td></td>
<td>V2</td>
<td>8934</td>
<td>33.3%</td>
</tr>
<tr>
<td></td>
<td>V3</td>
<td>10854</td>
<td>25.7%</td>
</tr>
<tr>
<td></td>
<td>V4</td>
<td>11910</td>
<td>49.2%</td>
</tr>
<tr>
<td></td>
<td>V5</td>
<td>6937</td>
<td>44.3%</td>
</tr>
<tr>
<td>Apollo</td>
<td>V1</td>
<td>1014</td>
<td>59.7%</td>
</tr>
<tr>
<td></td>
<td>V2</td>
<td>1557</td>
<td>33.3%</td>
</tr>
<tr>
<td></td>
<td>V3</td>
<td>1728</td>
<td>25.7%</td>
</tr>
<tr>
<td></td>
<td>V4</td>
<td>7330</td>
<td>49.2%</td>
</tr>
<tr>
<td></td>
<td>V5</td>
<td>29731</td>
<td>44.3%</td>
</tr>
</tbody>
</table>

Table 5 presents the results of checking full assertions and checking assertion differences on WBS, TCAS, and Apollo. TCAS refers to first TCAS method listed in Table 3, i.e., TCAS.alt_sep_test, TCAS2 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.
Thus, some cells have 0 SMT calls – SPF state exploration implication check optimization which uses non-deterministic. Therefore, the resulting implication check is handled by our Apollo invariants, invoke Daikon library methods, e.g., for implication checking. Some invariants, including all of the second experiment. All implications for the first 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.5657, arg1), and therefore, the resulting 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

Table 6: Implication checking results using symbolic execution. “Exp.” is the first experiment. “Exp.” is the second experiment.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Check implication $\phi_a \Rightarrow \phi_{a'}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># SMT</td>
</tr>
<tr>
<td>Exp: WBS</td>
<td>4</td>
</tr>
<tr>
<td>Exp: TCAS</td>
<td>2</td>
</tr>
<tr>
<td>Exp: TCAS</td>
<td>4</td>
</tr>
<tr>
<td>Exp: TCAS</td>
<td>2</td>
</tr>
<tr>
<td>Exp: TCAS</td>
<td>0</td>
</tr>
<tr>
<td>Exp: TCAS</td>
<td>2</td>
</tr>
<tr>
<td>Exp: TCAS</td>
<td>4</td>
</tr>
<tr>
<td>Exp: Apollo</td>
<td>0</td>
</tr>
</tbody>
</table>

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 $\Phi_a$ and $\Phi_{a'}$ contain a single clause $\phi_a$ and $\phi_{a'}$ respectively. Hence, for each assertion pair we symbolically execute “if $\phi_a$ 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 $\phi_a$ at random from the set of relevant valid invariants. In the second experiment, we create pairs by applying mechanical transformations [1]; for invariant $\phi_a$ selected at random: (1) if $\phi_a$ contains the $>$ comparison operator, we create $\phi_{a'}$ by mutating $>$ to $\geq$ to form assertion pair $(a, a')$; (2) if $\phi_a$ contains the $<$ comparison operator, we create $\phi_{a'}$ by mutating $<$ to $\leq$ to form assertion pair $(a, a')$; (3) if $\phi_a$ contains a numeric expression of the form $'i \text{ op } r'$ we generate $\phi_{a'}$ by adding a constant $c \in \{-100, \ldots, 100\}$, selected at random, to both sides of the expression in $\phi_a$ to form assertion pair $(a, a')$; and (4) if $\phi_a$ invokes a Daikon library method, we select an invariant $\phi_{a'}$ ( $\neq \phi_a$) at random to create assertion pair $(\langle \phi_a \land \phi_{a'} \rangle \lor \langle \phi_a \land \neg \phi_{a'} \rangle, a)$.

Table 7: Results for checking regression errors.

<table>
<thead>
<tr>
<th>Version</th>
<th>SPF</th>
<th>DiSE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># SMT</td>
<td># States</td>
</tr>
<tr>
<td>WBS</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>TCAS</td>
<td>3238</td>
<td>3239</td>
</tr>
<tr>
<td>TCAS</td>
<td>2198</td>
<td>2199</td>
</tr>
<tr>
<td>TCAS</td>
<td>7959</td>
<td>7959</td>
</tr>
<tr>
<td>Apollo</td>
<td>158</td>
<td>159</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Version</th>
<th>SPF</th>
<th>DiSE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># SMT</td>
<td># States</td>
</tr>
<tr>
<td>WBS</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>TCAS</td>
<td>1578</td>
<td>1579</td>
</tr>
<tr>
<td>TCAS</td>
<td>2198</td>
<td>2199</td>
</tr>
<tr>
<td>TCAS</td>
<td>7062</td>
<td>7063</td>
</tr>
<tr>
<td>Apollo</td>
<td>158</td>
<td>159</td>
</tr>
</tbody>
</table>

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'. For TCAS', 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', 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 co-evolve. 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


