# PreInfer: Automatic Inference of Preconditions via Symbolic Analysis

Angello Astorga*, Siwakorn Srisakaokul*, Xusheng Xiao†, Tao Xie*
*University of Illinois at Urbana-Champaign
†Case Western Reserve University
{aastorg2, srisaka2, taoxie}@illinois.edu,{xusheng.xiao}@case.edu

***Abstract*—When tests fail (e.g., throwing uncaught exceptions), automatically inferred preconditions can bring various debugging benefits to developers. If *illegal* inputs cause tests to fail, developers can directly insert the preconditions in the method under test to improve its robustness. If *legal* inputs cause tests to fail, developers can use the preconditions to infer failure-inducing conditions. To automatically infer preconditions for better support of debugging, in this paper, we propose PREINFER, a novel approach that aims to infer *accurate* and *concise* preconditions based on symbolic analysis. Specifically, PREINFER includes two novel techniques that prune irrelevant predicates in path conditions collected from failing tests, and that generalize predicates involving collection elements (i.e., array elements) to infer desirable quantified preconditions. Our evaluation on two benchmark suites and two real-world open-source projects shows PREINFER's high effectiveness on precondition inference and its superiority over related approaches.**

***Keywords*-precondition inference; dynamic symbolic execution; symbolic analysis; path conditions;**

## I. INTRODUCTION

With recent research advances in automatic test generation based on dynamic symbolic execution [1], [2], powerful test generation tools are now at the fingertips of developers in industry. For example, Pex [2], [3] has been shipped as *IntelliTest* [4] in Visual Studio 2015 and 2017 Enterprise Edition, benefiting numerous developers. These test generation tools allow developers to automatically generate tests for the code under test, comprehensively covering various program behaviors to achieve high code coverage. However, even with these tools, developers still need to perform time-consuming debugging tasks when unexpected failing tests are generated. They must determine whether the inputs are reasonably expected to be witnessed in real scenarios; if so, the developers must isolate the causes of the failures, possibly through more tests.

To assist such debugging tasks, automatically inferred preconditions bring various benefits. When tests containing *illegal* inputs fail, developers can directly insert the preconditions in the method under test to improve its robustness. When tests containing *legal* inputs fail, developers can use the preconditions to further infer failure-inducing conditions. Generally automatic inference of preconditions aims to infer preconditions that are *sufficient*[1] and *necessary*[2]. However, to be conveniently usable by developers, preconditions should also be *succinct* (e.g., including a minimal number of predicates). More succinct preconditions typically incur less runtime overhead during runtime monitoring, and also incur less human effort during manual inspection.

To automatically infer preconditions, in this paper, we present PREINFER, a novel approach based on symbolic analysis that aims to guard against failures exposed by the generated failing tests without blocking passing tests. PREINFER leverages symbolic analysis to capture an execution path with path condition $\rho$, which is a sequence of conjuncted predicates $(\phi_1, \phi_2, \dots, \phi_{|\rho|})$ collected from the executed branch conditions in the path. Note that the variables appearing in each predicate in $\rho$ are program inputs, i.e., symbolic inputs, instead of local variables inside the program body. In particular, PREINFER takes as input a test suite containing both passing and failing tests. It uses path conditions from failing tests (in short as failing path conditions) to infer a succinct condition $\alpha$ that is a generalization (i.e., summary) of the witnessed unsafe states. To avoid overgeneralizing, PREINFER uses path conditions from passing tests to identify states that should not satisfy $\alpha$. Naturally, PREINFER outputs $\neg\alpha$ as the inferred precondition.

To infer the condition $\alpha$, we choose the divide-and-conquer strategy by disjuncting these failing path conditions to form $\alpha$. Unfortunately, such disjunction introduces an $\alpha$ with a large number of predicates, compromising the goal of obtaining a *succinct* precondition. Alternatively, to aim for a sufficient and necessary precondition, we can always keep only the *last-branch predicate*, i.e., $\phi_{|\rho|}$, for each failing path condition and disjunct these reduced path conditions to produce $\alpha$. $\phi_{|\rho|}$ in a failing path condition $\rho$ for path $p$ corresponds to the $p$-assertion-violating condition (e.g., the assertion-violating condition in path $p$) at the assertion-

[1]Preconditions are *sufficient* if their satisfaction ensures that the program's execution does not produce runtime failures: sufficient preconditions block all illegal inputs, but can possibly block legal inputs.

[2]Preconditions are *necessary* if their violation ensures that the program's execution produces runtime failures: necessary preconditions block only illegal inputs but may not block all of them.

containing location[3]. In particular, $\phi_{|\rho|}$ expresses the $p$-assertion-violating condition at the assertion-containing location in terms of the program inputs, i.e., symbolic inputs, to form a symbolic expression.

However, keeping only the $p$-assertion-violating condition ($\phi_{|\rho|}$) causes two main issues. (1) *Location reachability*. Without keeping some other predicates in $\rho$ to constrain the test generation, some further generated tests do not even reach the assertion-containing location, and thus cannot violate the assertion. (2) *Expression preservation*. Without keeping some other predicates in $\rho$ to constrain the test generation, even when a further generated test can reach the assertion-containing location, the last-branch predicate $\phi_{|\rho'|}$ (corresponding to the $p$-assertion-violating condition) of its path condition $\rho' : (\phi_1, \phi_2, \ldots, \phi_{|\rho'|})$ is *different* from $\phi_{|\rho|}$ in $\rho$ (in terms of their symbolic expressions). Thus, a generated test satisfying $(\phi_1 \wedge \phi_2 \wedge \cdots \wedge \neg\phi_{|\rho'|} \wedge \phi_{|\rho|})$ reaches the assertion-containing location but cannot violate the assertion.

To address these two issues, PREINFER prunes irrelevant predicates from each failing path condition (one failing path condition at a time) to produce a reduced failing path condition. Then, PREINFER computes the disjunctions of these reduced failing path conditions to infer $\alpha$. PREINFER includes a technique of dynamic predicate pruning, which keeps only the predicates (in a failing path condition $\rho$ for path $p$) that are necessary for helping achieve location reachability and expression preservation with respect to the $p$-assertion-violating condition. Specifically, we define two relations to describe predicates in a failing path condition $\rho$: predicates ensuring location reachability are in a ***c-depend*** relation and predicates ensuring expression preservation are in a ***d-impact*** relation. Naturally, based on these relations, PREINFER prunes predicates not in either of these two relations.

Furthermore, we observe that for those programs that loop over a collection-based data structure such as an array, a failing path condition can contain a lot of predicates on multiple array elements. We name these predicates ***overly specific predicates***. They usually cannot be pruned by the technique of dynamic predicate pruning since most of them are in c-depend or d-impact relations, posing challenges for inferring sufficient, necessary, and succinct preconditions. To address such issue, PREINFER includes the technique of collection-element generalization to deal with the cases when a failure is dependent on the iterations of looping over collection elements, and thus a quantified precondition is needed.

In summary, this paper makes the following main contributions:

[3]Such assertion can be of an implicit-check type such as NullReference and DivideByZero (which can be automatically inserted by tools such as Pex before test generation) or an explicit-check type, which is for an explicitly written assertion statement.

```
1  public int example(string[] s, int a, int b,
       int c, int d) {
2    /*ground-truth precondition for exception
         at Lines 14-15*/
3  ¬(((c > 0 ∧ d + 1 > 0) ∨ (c ≤ 0 ∧ d > 0)) ∧ s == null)
4    /*ground-truth precondition for exception
         at Lines 16-17*/
5  ¬((((c > 0 ∧ d + 1 > 0) ∨ (c ≤ 0 ∧ d > 0)) ∧ s! = null)∧
       ∃i, (i < s.length ∧ s[i] == null))
6    int sum = 0;
7    if (a > 0)
8      b++;
9    if (c > 0)
10     d++;
11   if (b > 0)
12     sum++;
13   if (d > 0) {
14     assert(s != null);/*implicit assert:
           NullReference exception*/
15     for (var i = 0; i < s.Length; i++) {
16       assert(s[i] != null);/*implicit
             assert: NullReference exception*/
17       sum += s[i].Length;
18   }
19   return sum;
20 }
```

Figure 1. Example method under test

- A technique of dynamic predicate pruning to effectively prune path-condition predicates to infer preconditions.
- A technique of collection-element generalization to effectively generalize path-condition predicates to infer quantified preconditions.
- A tool implementation of PREINFER as an extension of Pex and an evaluation of PREINFER on two sets of benchmarks and two real-world open-source projects; the evaluation results show PREINFER's high effectiveness on precondition inference and its superiority over related approaches.

## II. MOTIVATING EXAMPLE

This section illustrates how PREINFER infers a *precondition candidate* for the example method under analysis (shown in Figure 1). For illustration purposes, in Figure 1, we show two implicit assertions (Lines 14 and 16) that are dynamically inserted by the underlying test generation tool. When such tool is applied on the example method, some failing tests can be generated to cause runtime exceptions (violating the implicit assertions). For example, running the method on a generated test $t_{f_1}$ : `(s: {null}, a: 1, b: 0, c: 1, d: 0)` causes the method to throw the `NullReference` exception (Lines 16-17). For this execution, the path condition is listed in Column 1 of Table I. Note that all the predicates in Table I are collected from the executed program branches (explicit branch conditions) or the implicit branch conditions (resulted from implicit assertions) by performing symbolic analysis on the test. All the predicates consist of constraints involving symbolic

Table I
PATH CONDITION FOR THE FAILING TEST $t_{f_1}$: `(S: {NULL}, A: 1, B: 0, C: 1, D: 0)` WHERE D: D-IMPACT; C: C-DEPEND

| PC predicates | Line # Branch | Kept? | Justification |
|---|---|---|---|
| $a > 0$ | Line 7 | ✗ | ✗d; ✗c |
| $c > 0$ | Line 9 | ✓ | ✓d; ✗c |
| $b + 1 > 0$ | Line 11 | ✗ | ✗d; ✗c |
| $d + 1 > 0$ | Line 13 | ✓ | ✗d; ✓c |
| $s! = null$ | Line 14 (Implicit Branch) | ✓ | ✗d; ✓c |
| $0 < s.length$ | Line 15 | ✓ | ✗d; ✓c |
| $s[0] == null$ | Line 16 (Implicit Last Branch) | ✓ | assertion-v cond |

Table II
PATH CONDITION FOR THE FAILING TEST $t_{f_3}$: `(S: {``A'', ``A'', NULL}, A: 1, B: 0, C: 1, D: 0))` WHERE D: D-IMPACT; C: C-DEPEND

| PC predicates | Line # Branch | Kept? | Justification |
|---|---|---|---|
| $a > 0$ | Line 7 | ✗ | ✗d; ✗c |
| $c > 0$ | Line 9 | ✓ | ✓d; ✗c |
| $b + 1 > 0$ | Line 11 | ✗ | ✗d; ✗c |
| $d + 1 > 0$ | Line 13 | ✓ | ✗d; ✓c |
| $s! = null$ | Line 14 (Implicit Branch) | ✓ | ✗d; ✓c |
| $0 < s.length$ | Line 15 | ✓ | ✗d; ✓c |
| $s[0]! = null$ | Line 16 (Implicit Branch) | ✓ | ✗d; ✓c |
| $1 < s.length$ | Line 15 | ✓ | ✗d; ✓c |
| $s[1]! = null$ | Line 16 (Implicit Branch) | ✓ | ✗d; ✓c |
| $2 < s.length$ | Line 15 | ✓ | ✗d; ✓c |
| $s[2] == null$ | Line 16 (Implicit Last Branch) | ✓ | assertion-v cond |

variables resulted from the method inputs, e.g., `(s, a, b, c, d)`; we name these constraints as symbolic expressions. The predicate $s[0] == null$ in the last row comes from the implicit check on the array access at Line 16, referred to as the *last-branch predicate* ($\phi_{|\rho|}$). The predicate $d + 1 > 0$ corresponds to the branch `(d > 0)` at Line 13 because $d$ gets increased at Line 10. To block this failing test `(s: {null}, a: 1, b: 0, c: 1, d: 0)` (along with all other possible failing tests reaching that same program location such as $t_{f_3}$ : `(s: {``a'', ``a'', null}, a: 1, b: 0, c: 1, d: 0)` ), we can use the ground-truth precondition (Line 5) to filter out the tests.

To infer the precondition candidate, PREINFER first applies our technique of dynamic predicate pruning to remove predicates (from the path condition) that are *irrelevant* for helping achieve *location reachability* and *expression preservation*. Informally, the predicates that preserve location reachability are in a *c-depend* relation w.r.t. the last-branch predicate $\phi_{|\rho|}$, and the predicates that preserve expression preservation are in a *d-impact* relation w.r.t. $\phi_{|\rho|}$(formally defined in Section III-A). For each failing path condition $\rho_{f_1}$, our technique employs a backward analysis starting from $\phi_{|\rho|}$, to detect the predicates that belong in the reduced path condition, $\rho_{f_1}\prime$, which ultimately composes $\alpha$. For $\rho_{f_1}$ shown in Table I, our analysis determines that the last-branch predicate $s[0] == null$ should not be pruned since it expresses the $p$-assertion-violating condition. Our analysis proceeds by analyzing constraints $0 < s.length$, $s! = null$ up to $a > 0$ in that order. The results of the analysis can be seen in the last two columns of Table I.

For brevity, we illustrate only how our technique of dynamic predicate pruning prunes $a > 0$, derived from a conditional $c$. Note that $b + 1 > 0$ is pruned similarly. Our technique prunes $a > 0$ if the analysis can establish $a > 0$ as irrelevant. A predicate is irrelevant if it is "✗c-depend" (i.e., **not** in a c-depend relation) and also is "✗d-impact" (i.e., **not** in a d-impact relation). To check whether $a > 0$ is ✗c-depend, our analysis considers another prefix-sharing path condition from an available passing test $t_{p_1}$ that also reaches the same last-branch predicate (i.e., a passing path that shares the same prefix as $t_{f_1}$ before $c$ but takes $c$'s the other branch not taken by $t_{f_1}$). In this case, the path condition for $t_{p_1}$ is $a < 1 \wedge c > 0 \wedge b > 0 \wedge d + 1 > 0 \wedge s! = null \wedge 0 < s.length \wedge s[0]! = null \wedge 2 > s.length$. Our analysis establishes that $a > 0$ is ✗c-depend since a path from either $a > 0$ or $a < 1$ can still reach the last-branch predicate $s[0] == null$ (e.g., $s[0]! = null$ in $t_{p_1}$). To check whether $a > 0$ is ✗d-impact, our analysis considers another prefix-sharing path condition from an available failing test $t_{f_2}$ that also reaches the last-branch predicate. In this case, the path condition for $t_{f_2}$ is $a > 1 \wedge c > 0 \wedge b > 0 \wedge d + 1 > 0 \wedge s! = null \wedge 0 < s.length \wedge s[0] == null$. Our analysis establishes that $a > 0$ is ✗d-impact since a path from either $a > 0$ or $a > 1$ does not change the symbolic expression at the last-branch predicate. After our technique analyzes every predicate in $\rho_{f_1}$, the newly reduced path condition is $\rho_{f_1}\prime := c > 0 \wedge d+1 > 0 \wedge s! = null \wedge 0 < s.length \wedge s[0] == null$.

However, after we apply the technique of dynamic predicate pruning, the reduced path conditions include overly specific predicates that are in either c-depend or d-impact relations across all failing tests. To illustrate these predicates, consider the reduced path conditions for test $t_{f_1}$, $\rho_{f_1\prime} = c > 0 \wedge d+1 > 0 \wedge s! = null \wedge 0 < s.length \wedge s[0] == null$ and for test $t_{f_3}$, $\rho_{f_3\prime} = c > 0 \wedge d + 1 > 0 \wedge s! = null \wedge 0 < s.length \wedge s[0]! = null \wedge 1 < s.length \wedge s[1]! = null \wedge 2 < s.length \wedge s[2] == null$. The predicates involving a relation over a constant and length of the array (e.g., $1 < s.length$) and a relation over an element of the array and some value (e.g., $s[1]! = null$) are in c-depend relations and should not be pruned.

To address overly specific predicates introduced by collection-based data structures, PREINFER further includes our technique of collection-element generalization on all the failing path conditions to generalize these predicates with a quantified condition. Our technique matches each path condition against pre-defined generalization templates such as $\exists x, (A(x) \wedge B(x))$ where $A$ and $B$ are a set of predicates. Predicates in $A$ constrain the domain of the formula (typically the integer domain used to iterate over collection structures) and predicates in $B$ are those expressing the violated property that causes the assertion violation. For this example, the quantified constraint resulted from the generalization is $\exists i, (i < s.length \wedge s[i] == null)$, where $i < s.length$ instantiates a predicate in $A(i)$, and $s[i] ==$

$null$ instantiates a predicate in $B(i)$, since every failing path condition contains predicate $i < s.length \wedge s[i] == null$ for some value of $i$.

Overall, PREINFER first applies our technique of dynamic predicate pruning to remove some predicates and then applies our technique of collection-element generalization for deriving $\alpha := ((c \leq 0 \wedge d > 0 \wedge s! = null) \wedge \exists i, (i < s.length \wedge s[i] == null)) \vee ((c > 0 \wedge d + 1 > 0 \wedge s! = null) \wedge \exists i, (i < s.length \wedge s[i] == null))$. In the end, PREINFER infers the ground-truth precondition $(\neg \alpha)$ at Line 5 of Figure 1.

## III. Preliminaries and Problem Definition

In this section, we formally define the terms used in the paper and the problem that PREINFER intends to address.

**Definition 1.** A *method-entry state* s for the method under test m is a concrete-value assignment over the method input — the variables being used in m (e.g., the parameters of m, the receiver object's fields) before invocation.

A *method execution* of method m with its method-entry state s is denoted as m[[s]].

**Definition 2.** An *assertion-containing location* e denotes a program location containing an assertion check[4]. When a method execution reaches an assertion-containing location and violates the assertion check, the execution aborts with an exception.

The exception can be (1) implicit such as *DivideByZeroException* or (2) explicit, from explicitly written assertion checks, such as `Assert.IsTrue()`.

We assume that the method under test m is deterministic and sequential. A method execution m[[s]] is *failing* if it reaches an assertion-containing location and aborts due to an assertion violation; otherwise, m[[s]] is *passing*. Therefore, the set of all possible method executions $I_{all}$ of m can be partitioned into two disjoint subsets $I_{all} = I_{fail} \cup I_{pass}$ where $I_{fail}$ is the set of all the failing method executions and $I_{pass}$ is the set of all the passing method executions. Next we define a precondition candidate and its relationship with method executions.

**Definition 3.** A *precondition candidate* $\psi$ of method m is a predicate over the parameters of method m. The complexity of $\psi$, denoted as $|\psi|$, is the number of logical connectives and quantifiers in $\psi$. The evaluation of $\psi$ under the assignment from method-entry state s is denoted as s($\psi$).

**Definition 4.** A precondition candidate $\psi$ *validates* method execution m[[s]] if s($\psi$) is true. Let $I^{\psi}$ be the set of all possible method executions validated by precondition candidate $\psi$, i.e., $I^{\psi} = \{\mathtt{m}[[\mathtt{s}]] \mid \mathtt{s}(\psi)\}$.

[4]In our work, we consider both explicit assertion checks written by developers in production code or test code and implicit assertion checks automatically inserted by the language runtime or the underlying Pex test generation tool [5].

A precondition $\psi$ prevents a failing test (method inputs) if its method execution is not in $I^{\psi}$. We denote ideal precondition candidates as *sufficient*, *necessary*, and *succinct*. A *sufficient* precondition candidate is a precondition that invalidates all method executions in $I_{fail}$, but may also invalidate some method executions in $I_{pass}$. That is, a sufficient precondition prevents all failing tests and possibly some passing tests. Dually, a *necessary* precondition candidate validates all method executions in $I_{pass}$, but may also validate some method executions in $I_{fail}$. That is, a necessary precondition prevents only failing tests but possibly not all of them. Formally, $\psi$ is sufficient if and only if $I^{\psi} \cap I_{fail} = \emptyset$; $\psi$ is necessary if and only if $I_{pass} \subset I^{\psi} \equiv \overline{I^{\psi}} \cap I_{pass} = \emptyset$ ($\overline{I^{\psi}}$ denotes the complement of $I^{\psi}$). It is also desirable for $\psi$ to have the relatively low complexity $|\psi|$ with respect to the ideal complexity (the complexity of the ground-truth precondition $\psi^*$). The reason is that more succinct precondition candidates generally incur lower runtime-checking cost and lower human efforts for inspection and understanding.

We next define path conditions used to infer precondition candidates. Given method execution m[[s]], the execution path $p$ can be captured by path condition, $\rho$. The path condition is a conjunction of predicates $\rho = \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_{|\rho|}$ collected from the executed branch conditions in m and its (direct and indirect) callee methods along the executed path $p$. Note that $\phi_1$ is from the predicate in the first branch appearing in the executed path and $\phi_{|\rho|}$ is the *last-branch predicate*. This predicate is derived from the last branch in the executed path and is also an assertion-check predicate when $p$ is a failing. A path condition $\rho$ can also be viewed as a list of predicates $[\phi_1, \phi_2, \ldots, \phi_{|\rho|}]$. More precisely, $\rho$ is a logic formula that characterizes the inputs for which method m executes along execution path $p$. Each variable appearing in $\rho$ is from the method input (defined in its method-entry state s), while each predicate appearing in $\rho$ is over some first-order theory. A path condition $\rho$ for execution path $p$ is *sound* if every variable assignment satisfying $\rho$ defines an execution of m that follows $p$ [6]. In this work, we assume that a path condition is sound.

Given path $p$ that reaches assertion-containing location e (without considering the path executed after e) in a method m, a *p-assertion-violating condition* is a predicate over the method input defined in method-entry state s such that if any $p$-following inputs (i.e., those inputs whose execution follows $p$) satisfy the predicate, then the assertion check in e fails and the corresponding method execution m[[s]] is failing; otherwise, the assertion in $e$ is not violated and thus m[[s]] is a passing one.

### A. Problem Outline and Overview

In this section, we first elaborate on our target problem for clarity purposes. Then, we formally define two key components of our dynamic predicate pruning technique as relations namely *c-depend* and *d-impact*. We conclude by illustrating the abstraction of our collection-element generalization technique to summarize *overly specific* predicates. Overly specific predicates are those derived from conditions in branches located in loops including the loop header. In this work, we focus on those overly specific predicates that contain collection elements and integer values that change during each iteration of loops [7]. Given a failing path condition, $\rho_{f_k}$, all inputs satisfying $\rho_{f_k}$ will induce executions that reach an assertion-containing location, $e$, and satisfy its $p$-assertion-violating condition. Given a passing path condition, $\rho_{p_i}$, all inputs satisfying $\rho_{p_i}$ will induce executions that do not reach $e$ or reach $e$ but do not satisfy the $p$-assertion-violating condition. Thus, $\rho_{f_k} \wedge \rho_{p_i}$ is unsatisfiable. Based on such observation, the goal of PREINFER is to infer a condition that is a generalization from the initial failing runs (by pruning irrelevant predicates) but is precise enough to avoid capturing behavior from passing runs. Formally, PREINFER aims to infer a condition $\alpha$ with the lowest complexity (i.e., with the minimum $|\alpha|$) that satisfies the following criteria:

- $\forall \rho_{f_k} \in P_{fail}$, $\rho_{f_k} \implies \alpha$
- $\forall \rho_{p_i} \in P_{pass}$, $\rho_n \implies \neg\alpha$

Note that the given set $P_{fail}$ typically includes only a subset of all possible failing path conditions due to the limited resources allocated to the used test generation tool. A program with loops can also contain infinitely many paths. Intuitively, in the best case of being able to generalize perfectly, $\alpha$ captures all failing path conditions in the given $P_{fail}$ and even all other failing path conditions not in $P_{fail}$. In the worse case of not being able to generalize, $\alpha$ captures only failing path conditions in the given set $P_{fail}$. To produce $\alpha$, given an input set of failing path conditions $\{\rho_{f_1}, ..., \rho_{f_k}\}$, our algorithm removes the predicates from each $\rho_{f_i}$ to produce a reduced set of path conditions $P_{fail}' = \{\rho_{f_1}', ..., \rho_{f_k}'\}$ such that

- $|\rho_{f_i}'| \leq |\rho_{f_i}|$
- $\forall \rho_{p_i} \in P_{pass} \forall \rho_{f_k}' \in P_{fail}', \neg(\rho_{p_i} \wedge \rho_{f_k}')$, i.e., $\rho_{p_i} \wedge \rho_{f_k}'$ is unsatisfiable.

The algorithm then computes a condition $\alpha = \rho_{f_1}' \vee \cdots \vee \rho_{f_k}'$. By removing predicates from each $\rho_{f_i}$ to form $\rho_{f_i}'$, the resulting set of $\rho_{f_i}'$ can include duplicate predicates, and these duplicates are removed, further simplifying $\alpha$. It is obvious to see that $\forall \rho_{p_i} \in P_{pass}, \neg(\rho_{p_i} \wedge \alpha)$, so the result condition $\alpha$ is a generalization from all the failing runs and does not capture behavior from all the passing runs. Next, we define two binary relations to determine the predicates that can be removed. Predicates in path conditions can be in a *c-depend* or a *d-impact* relation w.r.t. the last-branch predicate.

**Definition 5.** A predicate $\phi_i$ in a failing path $\rho_{f_i} = [\phi_1, \phi_2, \ldots, \wedge\phi_{i-1}, \wedge\phi_i, \ldots, \phi_{|\rho|}]$ is in a ***c-depend*** relation w.r.t. the last-branch predicate ($\phi_{|\rho|}$) if the concrete executions of all inputs that satisfy $\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_{i-1} \wedge \neg\phi_i$ do not reach the assertion-containing location.

To decide whether predicate $\phi_i$ is in c-depend, we consider only the executions that reach the branch condition of $\phi_i$. Those executions can be partitioned into two sets: the executions whose evaluation of $\phi_i$ is true and the executions whose evaluation of $\phi_i$ is false. Now we can conclude that predicate $\phi_i$ is in c-depend if and only if only one set contains a path reaching the assertion-containing location.

**Definition 6.** A predicate $\phi_i$ in a failing path $\rho_{f_i} = [\phi_1, \phi_2, \ldots, \wedge\phi_{i-1}, \wedge\phi_i, \ldots, \phi_{|\rho|}]$ is in a ***d-impact*** relation w.r.t. the last-branch predicate ($\phi_{|\rho|}$) if there exists an input that satisfies $\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_{i-1} \wedge \neg\phi_i$, and whose concrete execution reaches the assertion-containing location but whose symbolic expression of the $p$-assertion-violating condition (from the assertion-containing location) is different than the symbolic expression of $\phi_{|\rho|}$.

These two relations define membership of predicates in the reduced path conditions. The reduced path conditions, composed of only predicates in c-depend or d-impact relations, ensure *location reachability* and *expression preservation*. In other words, predicate $\phi_i \in \rho_{f_i}'$ if and only if $\phi_i$ is in a c-depend or d-impact relation. Intuitively, the aim is to identify and remove those predicates whose valuation (true, false) is irrelevant to whether or not the paths containing those predicates reach the assertion-containing location (location reachability) and the symbolic expressions of the predicates from the assertion-containing location of those paths are the same (expression preservation).

After pruning, each predicate $\phi$ in each reduced path condition $\rho_{f_1}'$ from the set $\{\rho_{f_1}', ..., \rho_{f_k}'\}$ is either in a c-depend or d-impact relation. However, a subset of these predicates may be repetitive only differing by a factor of the iteration count of some loop. We denote these predicates as *overly specific*. These *overly specific* predicates provide an opportunity to infer more succinct preconditions while retaining precision. Our approach includes one additional generalization step to summarize these predicates into more concise representations. To conduct generalization, we want to construct formulas of the form (1) $\exists x_1, \ldots, x_n(A(x_1, \ldots, x_n) \wedge B(x_1, \ldots, x_n))$ or (2) $\forall x_1, \ldots, x_n(A(x_1, \ldots, x_n) \rightarrow B(x_1, \ldots, x_n))$ where A and B are sets of predicates in terms of the bounded variables. In particular, predicates in A restrict the domain of the bounded variables, $x_1, \ldots, x_n$, while predicates in B express the violated property(ies) in terms of the bounded variables. Then, for each $\rho_{f_i}'$, we instantiate formulas of the preceding form 1 or 2, based on the predicates in $\rho_{f_i}'$, to create candidate formulas. Then, we choose a candidate $C$ based on the number of subsumed *overly specific* predicates in $\rho_{f_i}'$. Finally, our generalization step uses $C$ to construct $\rho_{f_i}'' = [\phi_1, \exists i.\phi_A(i) \wedge \phi_B(i), \ldots, \phi_{|\rho|}]$ or

**Algorithm 1** Dynamic Predicate Pruning

```
1:  function PREDICATEPRUNING($p_f, P_{pass}, P_{fail}$)
2:      $SP$ ←Map from a path to its slice
3:      for each path $p \in P_{pass} \cup P_{fail} \cup \{p_f\}$ do
4:          $SP[p] \leftarrow Last(p)$
5:          $p \leftarrow p \setminus Last(p)$
6:      end for
7:      while $p_f \neq \varnothing$ do
8:          $b \leftarrow Last(p_f)$
9:          if $IsC\text{-}Depend(SP, b, p_f, P_{pass}) \vee$
10:             $IsD\text{-}Impact(SP, b, p_f, P_{fail})$ then
11:             Add $b$ to $SP[p_f]$
12:             for each path $p \in P_{pass} \cup P_{fail} \wedge b \in p$ do
13:                 Add $b$ to $SP[p]$
14:             end for
15:         else
16:             for each path $p \in P_{pass} \cup P_{fail} \cup \{p_f\} \wedge b \in p$ do
17:                 $p \leftarrow p \setminus b$
18:             end for
19:         end if
20:     end while
21:     return $SP[p_f]$
22: end function
```

${\rho_f}_i'' = [\phi_1, \forall i.\phi_A(i) \rightarrow \phi_B(i), \ldots, \phi_{|\rho|}]$ such that:

- $|{\rho_f}_i''| \leq |{\rho_f}_i'| \leq |\rho_{f_i}|$
- $\alpha_{gen} = {\rho_f}_1'' \vee \cdots \vee {\rho_f}_k''$
- $\forall \rho_{p_i} \in P_{pass}, \neg(\rho_{p_i} \wedge \alpha_{gen})$, i.e., each $(\rho_{p_i} \wedge \alpha_{gen})$ is unsatisfiable.

Finally, our approach produces $\neg\alpha_{gen}$ as a precondition candidate.

## IV. Approach

PREINFER takes as input a method `m`, an assertion-containing location `e`, and a test suite containing passing tests and failing tests (e.g., satisfying the $p$-assertion-violating condition at $e$). PREINFER executes the tests and collects the path conditions of the tests. Then, PREINFER applies the technique of dynamic predicate pruning to remove *irrelevant* predicates and the technique of collection-element generalization to generalize overly specific predicates to infer a precondition candidate for each assertion-containing location `e` in `m`.

### A. Dynamic Predicate Pruning

Algorithm 1 shows the technique of dynamic predicate pruning, which detects irrelevant predicates in all the failing path conditions. For each failing path condition $\rho_f = [\phi_1, \phi_2, \ldots, \phi_j, \ldots, \phi_{|\rho_f|}]$, the algorithm checks the predicates one by one in a backward manner, starting from the last predicate $\phi_{|\rho_f|}$ (Lines 3-6). To determine whether a predicate $\phi_j$ is not in a c-depend relation, we first assume that $\phi_j$ is in a c-depend relation, and then try to find a contradiction. Our technique searches for a passing path condition $\rho_p$ such that $\rho_p$ shares the same prefix as $\rho_f$ up to $\phi_j$, but $\rho_p$ deviates at $\phi_j$ and still reaches the assertion-containing location `e`. That is, $\rho_p := [\phi_1, \phi_2, \ldots, \neg\phi_j, \ldots, \phi_e, \ldots, \phi_{|\rho_p|}]$ where $\phi_e$ is the predicate derived from `e`. If there exists such a path, $\phi_j$ is not in a c-depend relation. Next our technique determines whether the predicate $\phi_j$ is in a d-impact relation. First, we assume that the predicate $\phi_j$ is not in a d-impact relation. Our technique searches for a failing path condition ${\rho_f}'$ such that ${\rho_f}'$ shares the same prefix as $\rho_f$ up to $\phi_j$, but ${\rho_f}'$ deviates at $\phi_j$ and eventually reaches the assertion-containing location `e` to cause assertion violation, and the symbolic expression at the last-branch predicate in ${\rho_f}'$ is not the same as in $\rho_f$. That is, ${\rho_f}' = [\phi_1, \phi_2, \ldots, \neg\phi_j, \ldots, \phi_{|{\rho_f}'|}]$ and $\phi_{|{\rho_f}'|} \not\equiv \phi_{|\rho_f|}$. If there exists such a path, $\phi_j$ is in a d-impact relation. If a predicate is neither in a c-depend relation nor a d-impact relation, our technique removes the predicate from its path condition.

### B. Collection-Element Generalization

The presence of input-dependent loops and collection structures in the execution causes path conditions to contain many overly specific predicates over the indices of collection structures (e.g., loop variables). Since these predicates are in either c-depend or d-impact relations, our technique of dynamic predicate pruning cannot prune these predicates, and thus the generated precondition can contain a lot of these overly specific predicates. Based on our empirical observations, the overly specific predicates over the indices of collection structures often follow certain patterns, presenting opportunities to summarize these predicates using a quantified constraint over the indices of collection structures. Based on this insight, we describe how to produce preconditions with a quantified condition for the failing path conditions.

For each individual predicate $pred$ from the precondition candidate generated by the technique of dynamic predicate pruning, the technique of collection-element generalization chooses which predicates can be generalized with a quantified constraint over array index. The quantified constraint is of the form $\forall x.(A(x) \rightarrow B(x))$ or $\exists x.(A(x) \wedge B(x))$ where $A$ and $B$ are sets of predicates. Predicates in $A$ constrain the domain of the constraint (typically the integer domain used to iterate over collection structures), and predicates in $B$ are those expressing the violation of a property for causing failures.

Our technique breaks down the problem of inferring a quantified constraint into two general steps. First, our technique selects predicates that express the violation of a property (belonging to set $B$). Then, our technique selects a quantifier based on whether all eligible collection elements witness the violation of the property expressed by the predicates in set $B$. Finally, our technique selects predicates that restrict the domain of the constraint denoting the eligible collection elements. Next we describe in detail the technique of collection-element generalization.

**Identifying the violation of a property.** Similar to the technique of dynamic predicate pruning, the last-branch predicate is the pivot point for our generalization. As mentioned earlier, typically an assertion-violation failure is control-dependent on the branch condition represented as the last-branch predicate. Thus, in the cases where an assertion-violation failure is dependent on collection structures, the last-branch predicate likely expresses the violation of the property.

For example, the path condition of the failing test $t_{f_3}$ (Table II) includes $s[2] == null$ as the last-branch predicate. This predicate indicates that the program execution accesses $s[2]$ and results in an exception. Thus, $s[2] == null$ expresses the violation of the property: *collection element should not be null*. Consider another example program where each element of a collection $arr$ (starting from the first element) is used as a denominator in division. When the first three elements are not 0 and the fourth element is 0, the last-branch predicate would be $arr[3] == 0$, expressing the violation of the property: *no collection element should be zero*.

**Inferring the range of collection index.** Based on our empirical observations, our technique currently focuses on two common types of generalization templates, *Existential Template* and *Universal Template*, to infer the range of collection index. But new types of templates can be easily added as long as they operate over the predicates from failing path conditions. For a path condition $\rho$, our technique first generalizes the identified violation of the property as a predicate $\phi$, and then try to instantiate the generalization templates.

- **Existential Template.** For a path condition $\rho$ over a visited collection $a$ in the program under test, if only the last visited collection element $a[i]$ satisfies a predicate $\phi$, represented as $\phi(a[i])$, while all the previously visited collection elements $a[j]$ do not satisfy $\phi$, represented as $\neg\phi(a[j])$, then the following generalized property-violation condition can be inferred:
$$\varphi_\rho = \exists i, (0 \leq i < a.length) \wedge \phi(a[i])$$
- **Universal Template.** For a visited collection $a$ in the program under test, if all visited elements of $a$ satisfy a predicate $\phi$, represented as $\phi(a[i])$, the following generalized property-violation condition can be inferred:
$$\varphi_\rho = \forall i, (0 \leq i < a.length) \rightarrow \phi(a[i])$$

In the Existential Template, an exception (i.e., an assertion violation) from an assertion-containing location within a loop is triggered by the value of a collection element. The collection element is always the last-visited one since the program aborts with an exception. Thus, the inferred property-violation condition is that *there exists an collection element that satisfies the predicate*. Dually, in the Universal Template, an exception from an assertion-containing location within a loop is triggered because all elements satisfy the predicate, although the given path may not visit all elements in the collection.

**Example**. Consider the method under test `example` in Figure 1. In Lines 15-17, a loop iterates over the array elements to compute the sum of the elements' lengths. One failing path condition caused by the implicit assertion violation on Line 16 can contain predicates $s \neq null \wedge 0 < s.length \wedge s[0] \neq null \wedge 1 < s.length \wedge s[1] \neq null \wedge 2 < s.length \wedge s[2] == null$. Note that $s[2] == null$ causes the NullReference exception. These predicates can be summarized using the Existential Template with predicate $s[i] == null$ (i.e., the violation of the property) as $\exists i, (i < s.length \wedge s[i] == null)$.

Our technique can be easily extended with more templates on the predicates over a collection index. Consider a program that iterates over the even-numbered elements of an array, $a$, to check that $\phi(a[i])$ holds for every even $i$. We can summarize these predicates by adding the following template:

$$\varphi_\rho = \forall i, (0 \leq i < a.length \wedge i\%2 == 0) \rightarrow \phi(a[i])$$

To instantiate this template for a path condition $\rho$, our technique can check that for every even index $i$, $\phi(a[i])$ must hold.

## V. Evaluation

We implement our PreInfer approach as a prototype on top of Pex [2], [3], an industrial test generation tool. To assess PreInfer's effectiveness, we compare PreInfer with two related state-of-the-art approaches for precondition inference (DySy [8] and FixIt [9]). In particular, we conduct an evaluation of PreInfer and the related approaches on two benchmark suites and two real-world open-source projects. This comparison helps characterize the strengths of PreInfer compared with the previous related work. We intend to answer the following two research questions:

- **RQ1:** How effective is PreInfer in inferring preconditions compared to the related state-of-the-art approaches (DySy [8] and FixIt [9])?
- **RQ2:** How complex are the preconditions inferred by PreInfer compared to those inferred by DySy and FixIt?

### A. Evaluation Subjects

We select four evaluation subjects (written in C#) from GitHub [10]. Table III shows the characteristics of each evaluation subject. Additionally, these four subjects are classified into three categories: open-source projects, well-studied benchmarks, and manually-constructed benchmarks.

Table III
CHARACTERISTICS OF EVALUATION SUBJECTS

| Subject | #Classes | #Methods | #Lines | #Files |
|---|---|---|---|---|
| Algorithmia | 91 | 525 | 18249 | 95 |
| CodeContracts | 4 | 150 | 1965 | 7 |
| DSA | 48 | 457 | 9468 | 61 |
| SVComp | 4 | 11 | 1219 | 14 |

Table IV
AVERAGE BLOCK COVERAGE ACHIEVED BY PEX FOR ALL THE METHODS IN EACH EVALUATION SUBJECT

| Subject | Average Block Coverage |
|---|---|
| Algorithmia | 65.41% |
| CodeContracts | 99.20% |
| DSA | 100.00% |
| SVComp | 95.61% |

*1) Open-Source Projects:* We select two open-source projects, Data Structures and Algorithms (DSA) [11] and Algorithmia [12], used in previous studies. DSA is used in previous empirical studies of specifications [13], while Algorithmia is used in previous empirical studies of structural test generation [14].

*2) Well-Studied Benchmarks:* We select array-examples, loop-acceleration, and array-industry-pattern benchmark suites from SV-COMP [15]. We choose SV-COMP because its benchmarks are well studied, often used in evaluating software verification tools [16], [17], and contain non-trivial quantified preconditions. From the selected benchmark suites, we further exclude programs that do not have preconditions or that we are not able to translate to C#.

*3) Manually-Constructed Benchmarks:* We extract regression tests for the static analyzer cccheck [18] to construct a benchmark suite. The static analyzer infers preconditions for .NET programs. For the suite, we include only tests that stress-test the precondition inference algorithms in cccheck, without including other tests that stress-test other components (e.g., abstract domains).

## *B. Evaluation Setup*

For each evaluation subject, we use Pex to generate tests for each public method in the subject. If there exists a generated test that triggers an uncaught runtime exception at an assertion-containing location, we denote the test as failing and the method as an exception-throwing method. Note that multiple assertion-containing locations can be triggered in a method. The final set of methods used in our evaluation contains all the exception-throwing methods in each evaluation subject. Furthermore, the total number of exception-throwing locations used in our evaluation is the number of triggered assertion-containing locations across all methods in the final set. Column *#ACL* in Table V shows the total number of exception-throwing locations evaluated per subject. In total, our evaluation subjects include 188 exception-throwing locations, among which 33 are from collection-element cases, as shown in Column *#ACL* in Table VI.

To ensure fair comparison between different inferred preconditions for an assertion-containing location, we use the same set of Pex-generated tests for different approaches under comparison to check whether an inferred precondition blocks all the failing tests in the set and allows all the passing tests in the set. In particular, we use Pex to produce a set of tests $T$ for the method under test $m$. Given an assertion-containing location $e$ in $m$, we partition $T$ into $T_{pass}$ and $T_{fail}$: (1) we assign a test $t$ to $T_{pass}$ if $t$'s execution does not reach the assertion-containing location $e$, or reaches $e$ but does not violate the assertion in $e$, and (2) we assign $t$ to $T_{fail}$ if $t$'s execution does reach the assertion-containing location $e$ and violate the assertion in $e$. Table IV shows the average block coverage achieved by the tests generated by Pex for all the methods in each evaluation subject.

To assess the quality of an inferred precondition for each assertion-containing location, we use two metrics.

**Correctness.** We manually inspect an inferred precondition against a ground-truth precondition. In other words, we check whether the inferred precondition is semantically equivalent to the ground-truth precondition. To obtain a ground-truth precondition for each assertion-containing location, we employ the following steps. Initially, an author of this paper inspects the source code of the method containing the assertion-containing location and derives a precondition, $pred$. For difficult cases, where the author is unsure of the correctness of $pred$, another author is engaged. If both authors cannot reach a consensus, then we test the strength of $pred$ and $\neg pred$ using Pex. If $pred$ is 'likely' perfect, then inserting $pred$ at the entry point of the method should invalidate all failing runs, while inserting $\neg pred$ should validate only failing runs. We can only be certain $pred$ is 'likely' perfect because Pex may not explore all execution paths.

**Complexity.** Unlike our methodology for manually judging correctness of an inferred precondition, we rely on tool automation that parses a string representation of the inferred precondition to compute complexity. In particular, we measure the complexity of an inferred precondition $\psi$ of a method by calculating a *relative complexity*, which represents the percentage difference between its *complexity* ($|\psi|$) and the *complexity* of the method's ground-truth precondition ($|\psi^*|$) generated manually. In other words, the relative complexity of $\psi$ is

$$|\frac{|\psi| - |\psi^*|}{|\psi^*|}|$$

The lower the relative complexity is, the more succinct the precondition is, i.e., a perfect inferred precondition has the relative complexity equal to zero.

Table V
COMPARISON OF PRECONDITIONS GENERATED BY THE THREE APPROACHES ON ALL THE SUBJECTS

| **Namespace** | **Exception Location** | ***#ACL*** | **PREINFER** | | | **FixIt** | | | **DySy** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | # Suff | # Nece | # Both | # Suff | # Nece | # Both | # Suff | # Nece | # Both |
| **Algorithmia.Sorting** | Before loop | 3 | 0 | 0 | 3 | 0 | 0 | 1 | 2 | 0 | 0 |
| | Inside loop | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | After loop | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | **Total** | **3** | 0 | 0 | **3** | 0 | 0 | **1** | 2 | 0 | **0** |
| **Algorithmia.GeneralDataStr** | Before loop | 13 | 0 | 0 | 12 | 5 | 0 | 7 | 4 | 0 | 8 |
| | Inside loop | 5 | 0 | 1 | 4 | 0 | 0 | 4 | 1 | 1 | 3 |
| | After loop | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | **Total** | **18** | 0 | 1 | **16** | 5 | 0 | **11** | 5 | 1 | **11** |
| **DSA.Algorithm** | Before loop | 11 | 0 | 0 | 11 | 0 | 2 | 7 | 4 | 0 | 5 |
| | Inside loop | 17 | 0 | 7 | 9 | 3 | 1 | 8 | 11 | 1 | 5 |
| | After loop | 5 | 0 | 0 | 3 | 0 | 2 | 0 | 4 | 1 | 0 |
| | **Total** | **33** | 0 | 7 | **23** | 3 | 5 | **15** | 19 | 2 | **10** |
| **CodeContracts.ExamplesPuri** | Before loop | 14 | 0 | 1 | 13 | 6 | 0 | 5 | 2 | 4 | 8 |
| | Inside loop | 18 | 0 | 1 | 17 | 5 | 0 | 6 | 5 | 1 | 9 |
| | After loop | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | **Total** | **32** | 0 | 2 | **30** | 11 | 0 | **11** | 7 | 5 | **17** |
| **CodeContracts.PreInference** | Before loop | 58 | 0 | 0 | 58 | 27 | 0 | 18 | 3 | 4 | 33 |
| | Inside loop | 21 | 0 | 1 | 18 | 8 | 0 | 7 | 12 | 0 | 6 |
| | After loop | 7 | 0 | 1 | 6 | 5 | 0 | 1 | 4 | 1 | 0 |
| | **Total** | **86** | 0 | 2 | **82** | 40 | 0 | **26** | 19 | 5 | **39** |
| **CodeContracts.ArrayPurityI** | Before loop | 2 | 0 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 2 |
| | Inside loop | 3 | 0 | 2 | 1 | 2 | 0 | 0 | 1 | 0 | 1 |
| | After loop | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | **Total** | **5** | 0 | 2 | **3** | 2 | 0 | **2** | 1 | 0 | **3** |
| **SVComp.SVCompCSharp** | Before loop | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Inside loop | 9 | 0 | 3 | 2 | 0 | 0 | 0 | 9 | 0 | 0 |
| | After loop | 2 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |
| | **Total** | **11** | 0 | 4 | **3** | 1 | 0 | **1** | 10 | 0 | **0** |
| **Total** | | **188** | 0 | 18 | **160** | 62 | 5 | **67** | 63 | 13 | **80** |

Table VI
COMPARISON OF PRECONDITIONS GENERATED BY THE THREE APPROACHES FOR THE COLLECTION-ELEMENT CASES ON ALL THE SUBJECTS

| **Subject** | ***#ACL*** | **PREINFER** | | | **FixIt** | | | **DySy** | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | # Suff | # Nece | # Both | # Suff | # Nece | # Both | # Suff | # Nece | # Both |
| **Algorithmia** | 2 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| **CodeContracts** | 19 | 0 | 3 | 14 | 7 | 0 | 0 | 12 | 1 | 0 |
| **DSA** | 4 | 0 | 3 | 1 | 0 | 0 | 0 | 2 | 2 | 0 |
| **SVComp** | 8 | 0 | 3 | 2 | 0 | 0 | 0 | 7 | 0 | 0 |
| **Total** | **33** | 0 | 10 | **17** | 7 | 0 | **0** | 21 | 4 | **0** |

### *C. RQ1: How effective is* PREINFER *in inferring preconditions compared to the related state-of-the-art approaches (DySy and FixIt)?*

The motivation behind RQ1 is to investigate the correctness of the preconditions generated by PREINFER compared to DySy and FixIt in the context of open-source projects and benchmark suites. Furthermore, we break down each case by the location of the assertion-containing location relative to a loop. Table V contains the results of our evaluation. Column *#ACL* shows the number of exception-throwing locations being evaluated. For each approach, Columns *#Suff*, *#Nece*, and *#Both* indicate the number of the preconditions that are only sufficient, only necessary, and both sufficient and necessary, respectively.

Across both open-source projects, PREINFER is more effective than DySy and FixIt. For Algorithmia, PREINFER can infer preconditions that are both sufficient and necessary in over 90% of the cases compared to 57% of the cases for Fix-It and 56% of the cases for DySy. Table VI shows the comparison of preconditions generated by the three approaches for cases where target preconditions must contain existential or universal quantifiers (i.e., the collection-element cases). As shown in Table VI, FixIt cannot handle any single case, since FixIt uses only the last-branch predicate to form a precondition. FixIt does not infer a precondition from multiple branch conditions and has no notion of a quantifier. Our PREINFER approach can handle 17 out of the 33 cases. Figure 2 shows a code example (from DSA), which

```
1  public static string ReverseWords(this
       string value)
2  {
3    int last = value.Length - 1;
4    int start = last;
5    StringBuilder sb = new StringBuilder();
6    while (last >= 0) {
7       while (start >= 0 &&
            char.IsWhiteSpace(value[start])) {
8          start--;
9       }
10      last = start;
11      while (start >= 0 &&
            !char.IsWhiteSpace(value[start])) {
12         start--;
13      }
14      for (int i = start + 1; i < last + 1;
            i++) {
15         sb.Append(value[i]);
16      }
17      if (start > 0) {
18         sb.Append(' ');
19      }
20      last = start - 1;
21      start = last;
22   }
23   if (char.IsWhiteSpace(sb[sb.Length - 1]))
        {
24      sb.Length = sb.Length - 1;
25   }
26   return sb.ToString();
27 }
```

Figure 2. An example of a method under study from DSA

PREINFER can handle. The method `ReverseWords` takes an input string representing a sequence of words separated by some whitespaces. The method returns a new string representing the reverse of the sequence and removes some whitespaces. If we pass an empty string as part of the input to the method, the method throws an exception *IndexOutOfRangeException* on Line 23 as `sb.Length - 1` being negative. A ground-truth precondition to prevent this exception is `value == null || ∃i,(i < value.Length ∧ char.IsWhiteSpace(value[i]) == false)`. Each failing path condition for this exception contains some predicates indicating that all the characters in `value` must be whitespaces. Our technique of collection-element generalization is able to generalize all the failing paths by using the quantified predicate, `∀i,(i < value.Length ⟹ char.IsWhiteSpace(value[i]) == true)`. Thus, the resulting precondition matches the ground-truth precondition.

One limitation of PREINFER is that if a path condition does not contain all the predicates that are needed to infer an existential or universal template, PREINFER cannot generalize anything, so the collection-element generalization technique cannot work for such cases. For example, to infer a template, `∀i,(i + 1 < 3) ⟹ s[i + 1] == 'a'`, each failing path condition must contain predicate `s[j + 1] == 'a'` if it contains predicate `j + 1 < 3`. However, the failing path condition may contain predicate `s[1 + j] == 'a'` instead, which is semantically the same, but syntactically different. One way to improve our technique is to use a constraint solver to help determine predicate equivalence instead of using the raw string representations of the predicates appearing in path conditions. In almost all cases, PREINFER can outperform both DySy and FixIt; however, for some cases that contain complex loops that PREINFER cannot handle but the correct precondition is just the negation of the last-branch predicate, PREINFER fails to resolve and simplify the long path conditions for the loops, since our technique of collection-element generalization works for only some specific kinds of loops matching the two templates. When the technique fails to generalize some failing paths related to loops, the failing paths are too specific. Thus, the precondition generated from those failing paths is not sufficient (i.e., does not block all failing paths). FixIt, which directly infers a precondition from the last-branch predicate, can address such cases. Moreover, for some cases where Pex cannot generate any passing path, PREINFER cannot infer anything, whereas DySy can infer correct preconditions for such cases. To address such cases, we can improve PREINFER by slightly modifying our technique to skip all the steps that require passing paths.

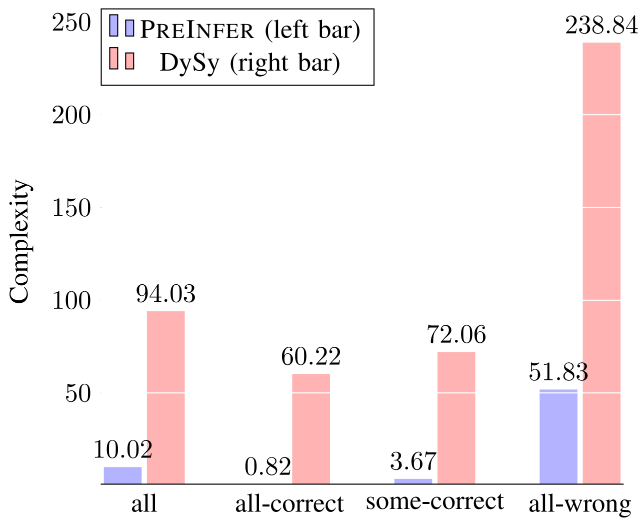

Figure 3. Average relative complexity of preconditions inferred by PREINFER and DySy in four categories across all the subjects.

### D. RQ2: How complex are the preconditions inferred by PREINFER compared to those inferred by DySy and FixIt?

One may be interested to see how complex the preconditions inferred by each approach under study are. For FixIt, the *average relative complexity* of correct preconditions inferred by FixIt is 0.19, and the average relative complexity of incorrect preconditions inferred by FixIt is 0.76. Thus, the relative complexity of preconditions inferred by FixIt is very low, regardless of their correctness. By design, preconditions inferred by FixIt are not complex because they do not capture potential control or data dependencies with the assertion-containing location. Preconditions inferred by FixIt are generated directly from last-branch predicates. Thus, the average relative complexity of preconditions inferred by FixIt is close to zero. However, the consequences of FixIt's design significantly compromise the accuracy of the preconditions as shown in Table V.

Figure 3 shows that the average relative complexity of preconditions inferred by PREINFER and DySy in four categories. The category "all" consists of all the *ACLs*. The category "all-correct" consists of all the *ACLs* that both approaches can infer correct preconditions (*sufficient* and *necessary*). The category "some-correct" consists of all the *ACLs* that at least one approach infers correct preconditions, and the category "all-wrong" consists of all the *ACLs* that none of the approaches can infer correct preconditions. In all four categories, the average relative complexity of preconditions inferred by PREINFER is a lot lower than that of preconditions inferred by DySy. On average across the four categories, the average relative complexity of preconditions inferred by PREINFER is about 9.70% of that of preconditions inferred by DySy. Thus, preconditions inferred by PREINFER are less complex than the ones inferred by DySy. One possible reason is that failing path conditions tend to have a fewer number of predicates than passing path conditions, since the execution stops early when it throws an exception. Thus, using failing path conditions instead of passing path conditions (used by DySy) to infer preconditions can reduce the complexity of the preconditions. Moreover, PREINFER also has some heuristic to prune unnecessary predicates from each failing path, helping reduce the complexity of the inferred preconditions. As shown in Figure 3, the average relative complexity of the preconditions inferred by both approaches is getting higher for a harder case. In the category "all-wrong", the average relative complexity of the preconditions inferred by both approaches is much higher than that in the category "all-correct", since the category "all-wrong" usually contains loop cases, which have long path conditions.

## VI. RELATED WORK

**Dynamic Invariant Detection.** Ernst et al. [19] propose the Daikon approach for dynamically detecting likely program invariants through the execution of tests. Daikon infers invariants based on patterns of variable values matching predefined templates. DySy [8] and Vigilante [20] infer preconditions using the disjunction of the path conditions from a set of passing and failing tests, respectively. In the presence of loops, these approaches produce very long and complex preconditions. Unlike their approaches that leverage only passing or only failing tests, our approach leverages both passing and failing tests, along with white-box information (e.g., path conditions), allowing for more pruning opportunities. Additionally, our approach summarizes overly specific predicates introduced by loops.

**Static Inference of Preconditions or Input Filters.** The approach of precondition inference in cccheck [18], [21] is based on abstract interpretation. Its underlying static analysis is undesirably more conservative and less precise than dynamic analysis (being used in our approach). Bouncer [22] generates input filters to block exploit inputs. It combines both static and dynamic analyses to perform precondition slicing for pruning irrelevant predicates in path conditions. SIFT [23] is a sound approach for input-filter generation to block integer overflow vulnerabilities. SIFT uses static analysis to derive symbolic constraints on how the sizes of memory blocks are allocated to generate input filters that block integer-overflow-causing inputs. Seghir et al. [24] propose an approach to infer preconditions by iteratively refining an over-approximation of the set of safe and unsafe states until they become disjoint. The refinement process is to add predicates such that a safe state and an unsafe state cannot share their initial state. Unlike these approaches based on static analysis, our approach leverages only the dynamically collected path conditions to infer preconditions and requires no static analysis (which often faces significant challenges in analyzing real-world code bases). Moreover, our approach includes a technique to infer quantified preconditions for summarizing overly specific predicates introduced by loops, while these approaches do not handle loops specially.

**Black-box Learning of Specifications**. Among black-box learning approaches for inferring specifications, Gehr et al. [25] use random sampling to generate a fixed set of tests and use heuristics to extract diverse samples from their initial set to learn commutativity specifications. Similarly, Padhi et al. [26] learn preconditions, but also include techniques to expand the initial features needed for learning. Sankaranarayanan et al. [27] use an initial set of predicates to obtain a partial truth table and then use a decision-tree-learning algorithm to learn preconditions. Our approach differs from these black-box approaches in that our approach is a white-box one and can infer rich properties related to collection elements.

## VII. CONCLUSION

In this paper, we have presented PREINFER, a novel approach of path-condition analysis for automatic precondition inference. PREINFER infers preconditions that guard against failures exposed by the generated failing tests without blocking passing tests. In particular, PREINFER prunes predicates that are irrelevant for helping achieve location reachability and expression preservation, and generalizes overly specific predicates involving collection elements as quantified formulas over the indices of collection structures.

Our evaluation on a set of benchmarks from CodeContracts and SVComp along with two real-world open-source projects (DSA and Algorithmia) shows PREINFER's high effectiveness on precondition inference and its superiority over two related state-of-the-art approaches.

## ACKNOWLEDGMENT

This work was supported in part by National Science Foundation under grants no. CCF-1409423, CNS-1513939, CNS1564274, and the GEM fellowship.

## REFERENCES

[1] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proc. PLDI*, 2015, pp. 213–223.

[2] N. Tillmann and J. De Halleux, "Pex: White box test generation for .NET," in *Proc. TAP*, 2018, pp. 134–153.

[3] N. Tillmann, J. de Halleux, and T. Xie, "Transferring an automated test generation tool to practice: from Pex to Fakes and Code Digger," in *Proc. ASE*, 2014, pp. 385–396.

[4] Microsoft. (2015) Generate smart unit tests for your code. [Online]. Available: https://msdn.microsoft.com/library/dn823749

[5] Microsoft. (2009) Exploring implicit branches. [Online]. Available: https://social.msdn.microsoft.com/Forums/en-US/c92907eb-3b2d-4c30-abcd-93ec1c120b00/exploring-implicit-branches?forum=pex

[6] P. Godefroid, "Higher-order test generation," in *Proc. PLDI*, 2011, pp. 258–269.

[7] P. Godefroid and D. Luchaup, "Automatic partial loop summarization in dynamic test generation," in *Proc. ISSTA*, 2011, pp. 23–33.

[8] C. Csallner, N. Tillmann, and Y. Smaragdakis, "DySy: Dynamic symbolic execution for invariant inference," in *Proc. ICSE*, 2018, pp. 281–290.

[9] N. Tillmann and J. D. Halleux, "Parameterized unit testing with Microsoft Pex (Long Tutorial)," 2010. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.159.4711

[10] GitHub, "GitHub," https://github.com.

[11] lukadt, "Data Structures and Algorithms (DSA)." [Online]. Available: https://github.com/lukadt/Data-Structures-and-Algorithms-DSA

[12] SolutionsDesign, "Algorithmia." [Online]. Available: https://github.com/SolutionsDesign/Algorithmia/tree/master

[13] N. Polikarpova, C. A. Furia, Y. Pei, Y. Wei, and B. Meyer, "What good are strong specifications?" in *Proc. ICSE*, 2013, pp. 262–271.

[14] X. Xiao, S. Li, T. Xie, and N. Tillmann, "Characteristic studies of loop problems for structural test generation via symbolic execution," in *Proc. ASE*, 2013, pp. 246–256.

[15] Software and Computational Systems Lab, "Collection of verification tasks." [Online]. Available: https://github.com/sosy-lab/sv-benchmarks

[16] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa, "TRACER: A symbolic execution tool for verification," in *Proc. CAV*, 2012, pp. 758–766.

[17] X. Xie, B. Chen, Y. Liu, W. Le, and X. Li, "Proteus: Computing disjunctive loop summary via path dependency analysis," in *Proc. FSE*, 2016, pp. 61–72.

[18] P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo, "Automatic inference of necessary preconditions," in *Proc. VMCAI*, 2013, pp. 128–148.

[19] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *Proc. ICSE*, 2001, pp. 99–123.

[20] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-end containment of Internet Worms," in *Proc. SOSP*, 2015, pp. 133–147.

[21] P. Cousot, R. Cousot, and F. Logozzo, "Precondition inference from intermittent assertions and application to contracts on collections," in *Proc. VMCAI*, 2011, pp. 150–168.

[22] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado, "Bouncer: Securing software by blocking bad input," in *Proc. SOSP*, 2007, pp. 117–130.

[23] F. Long, S. Sidiroglou-Douskos, D. Kim, and M. Rinard, "Sound input filter generation for integer overflow errors," in *Proc. POPL*, 2014, pp. 439–452.

[24] M. N. Seghir and D. Kroening, "Counterexample-guided precondition inference," in *Proc. ESOP*, 2013, pp. 451–471.

[25] T. Gehr, D. Dimitrov, and M. T. Vechev, "Learning commutativity specifications," in *Proc. CAV*, 2015, pp. 307–323.

[26] S. Padhi, R. Sharma, and T. Millstein, "Data-driven precondition inference with learned features," in *Proc. PLDI*, 2016, pp. 42–56.

[27] S. Sankaranarayanan, S. Chaudhuri, F. Ivančić, and A. Gupta, "Dynamic inference of likely data preconditions over predicates by tree learning," in *Proc. ISSTA*, 2008, pp. 295–306.