UnitCon: Synthesizing Targeted Unit Tests for Java Runtime Exceptions

SUJIN JANG, KAIST, Korea
YEONHEE RYOU, KAIST, Korea
HEEWON LEE, KAIST, Korea
KIHONG HEO, KAIST, Korea

We present UnitCon, a system for synthesizing *targeted unit tests* for runtime exceptions in Java programs. Targeted unit tests aim to reveal a bug at a specific location in the program under test. This capability benefits various tasks in software development, such as patch testing, crash reproduction, or static analysis alarm inspection. However, conventional unit test generation tools are mainly designed for regression tests by maximizing code coverage; hence they are not effective at such target-specific tasks. In this paper, we propose a novel synthesis technique that effectively guides the search for targeted unit tests. The key idea is to use static analysis to prune and prioritize the search space by estimating the semantics of candidate test cases. This allows us to efficiently focus on promising unit tests that are likely to trigger runtime exceptions at the target location. According to our experiments on a suite of Java programs, our approach outperforms the state-of-the-art unit test generation tools. We also applied UnitCon for inspecting static analysis alarms for null pointer exceptions (NPEs) in 51 open-source projects and discovered 21 previously unknown NPE bugs.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Program analysis, Program synthesis, Software testing

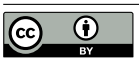
ACM Reference Format:
Sujin Jang, Yeonhee Ryou, Heewon Lee, and Kihong Heo. 2025. UnitCon: Synthesizing Targeted Unit Tests for Java Runtime Exceptions. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE092 (July 2025), 24 pages. https://doi.org/10.1145/3729362

1 Introduction

Automatic generation of unit tests is an important task in software engineering. Unit tests are used to validate the correctness of individual units of code, such as methods or classes. They are essential for ensuring the quality of software systems and are widely used in practice. Since writing unit tests is labor-intensive, researchers have developed various techniques for automatic unit test generation, such as Randoop [Pacheco and Ernst 2007] and EvoSuite [Fraser and Arcuri 2011].

However, existing unit test generation is not well suited for generating *targeted* tests. Conventional *untargeted* unit test generation mainly aims to generate regression tests that exercise various paths of the program to check the overall correctness. On the other hand, the goal of targeted test generation is to expose bugs at a specific program location (e.g., line). Such targeted tests are useful for various tasks in practice, such as continuous integration, patch testing [Böhme et al. 2013; Marinescu and Cadar 2013], crash reproduction [Jin and Orso 2012; Pham et al. 2015], or static analysis alarm inspection [Christakis et al. 2016].

Authors' Contact Information: Sujin Jang, KAIST, Daejeon, Korea, sujin.jang@kaist.ac.kr; Yeonhee Ryou, KAIST, Daejeon, Korea, yeonhee.ryou@kaist.ac.kr; Heewon Lee, KAIST, Daejeon, Korea, heewon.lee@kaist.ac.kr; Kihong Heo, KAIST, Daejeon, Korea, kihong.heo@kaist.ac.kr.



ACM 2994-970X/2025/7-ARTFSE092
https://doi.org/10.1145/3729362

In real-world, the central challenge of targeted unit test generation is to handle the huge search space of candidate unit tests. Even though a target location is specified, the search space of candidate unit tests is still enormous. Such unit tests require a sequence of method calls that set up the program state to reach the target location and trigger the bug at the point. This makes the search space of candidate unit tests exponentially large. Thus it is infeasible to enumerate all possible unit tests with a naive approach.

In this paper, we present UnitCon, the first system for fine-grained targeted unit test generation. UnitCon is designed to trigger *runtime exceptions* at a specific program statement. Given a program location where a potential exception may occur, UnitCon synthesizes a unit test that triggers the exception at the location. The synthesis algorithm searches for the unit test by enumerating candidates written in our domain-specific language which is a subset of Java.

Our key idea is to leverage static analysis to guide the test generation process. First, we use static analysis to infer the preconditions to reach the target exception location. This precondition is used to prioritize candidate unit tests that are likely to trigger the exception. Second, we use static analysis to estimate the semantic equivalence of candidate unit tests. The estimated equivalence is used to prune redundant candidates from the search space.

We implemented UnitCon using Facebook's Infer analyzer [Facebook 2024] and demonstrated its effectiveness on a suite of Java programs comprising 198 bugs with various exception types such as null pointer exceptions, index out of bounds exceptions, and class cast exceptions [Just et al. 2014; Lee et al. 2022]. The results show that UnitCon effectively synthesizes unit tests that trigger exceptions at the target locations. We compare UnitCon's performance to five state-of-the-art unit test generation tools: Randoop, EvoSuite, EvoFuzz, NPETest and UTBot. According to the experiments, UnitCon consistently outperforms the baselines in terms of the reproduction rate of the target exceptions. UnitCon discovers 104 (52.5%) of 198 bugs, which is 1.2–3.6 times more than the baselines within 10 minutes, on average. Furthermore, we demonstrate the effectiveness of UnitCon as an alarm inspection tool for static analysis. We applied UnitCon to 51 real-world open-source projects for inspecting the null pointer exception alarms reported by Infer and found 21 new bugs.

We summarize our contributions below:

- We present UnitCon, a new system for targeted unit test generation. UnitCon aims to synthesize Java unit tests to trigger runtime exceptions given a specific program location.
- We present an efficient algorithm to synthesize targeted unit tests using static analysis. Our algorithm uses static analysis to prioritize candidate unit tests and prune the search space of candidate unit tests.
- We demonstrate the effectiveness of UnitCon in comparison with state-of-the-art tools. UnitCon outperforms the baselines in terms of the number of bugs discovered.
- We applied UnitCon to real-world open-source projects and found new runtime exception bugs.

2 Overview

2.1 Motivating Example

We illustrate our approach with a buggy Java program in Figure 1(a). The code snippet is excerpted from JSqlParser, a widely used parser library for SQL queries in Java[1]. The library provides a set of traversal APIs to visit the hierarchical structure of SQL queries, such as the `visit` method at line 7. The `visit` method raises an null pointer exception (NPE) at line 10 when the value of `withItemsList` is `null`. Note that the value of the `withItemsList` field of the `subSelect` parameter at line 9 is returned by `subSelect.getWithItemsList()` at line 17. Moreover, the NPE

[1] https://github.com/JSQLParser/JSqlParser

```
1  public class ExpressionVisitorAdapter {
2    private SelectVisitor selectVisitor;
3
4    public void setSelectVisitor (SelectVisitor selectVisitor) { this.selectVisitor = selectVisitor; }
5
6    // Error condition: selectVisitor != null /\ subSelect != null /\ withItemsList == null
7    public void visit(SubSelect subSelect) {
8      if (selectVisitor != null) {
9        List<WithItem> withItemsList = subSelect.getWithItemsList();
10       for (WithItem item: withItemsList) { // Target NPE location
11         ...
12 }
13
14 public class SubSelect {
15   private List<WithItem> withItemsList;
16
17   public List<WithItem> getWithItemsList() { return withItemsList; }
18 }
19
20 public class Merge {
21   private SubSelect usingSelect;
22
23   public SubSelect getUsingSelect() { return usingSelect; }
24 }
```

(a) Simplified code from JSqlParser.

```
1 public void test() {
2   ExpressionVisitorAdapter recv = new ExpressionVisitorAdapter();
3   SelectVisitor selectVisitor = new TablesNamesFinder();
4   recv.setSelectVisitor(selectVisitor);
5   SubSelect subSelect = new SubSelect();
6   recv.visit(subSelect);
7 }
```

(b) Targeted unit test case generated by UnitCon.

Fig. 1. Motivating example.

is triggered only when the condition at line 8 is satisfied, i.e., both of the `selectVisitor` field of the receiver object and the `subSelect` argument are not `null`.

Our goal is to generate a unit test that triggers the NPE in the example. Such a targeted unit test should consist of a sequence of API calls that eventually call the `visit` method. The API call sequence should establish proper arguments and receiver objects to satisfy the error condition as in line 6. In this case, we should set the field `selectVisitor` to a non-`null` value, and allocate an object for the argument `subSelect` whose field `withItemsList` has `null`.

However, it is challenging to generate such a unit test because of the huge search space. Note that there are 70, 14, and 16 methods in class `ExpressionVisitorAdapter`, `SubSelect`, and `Merge`, respectively. Yet, only a few of them are relevant to the target NPE. Therefore, even if a path condition to reach the NPE point is given by a third-party tool such as a static analyzer, it is still difficult to synthesize a crashing test case that satisfies the condition. Existing automated unit test case generation tools such as EvoSuite [Fraser and Arcuri 2011] and Randoop [Pacheco and Ernst 2007] could rarely generate a crashing test case for the NPE within 10 minutes. Since their main goal is not to generate test cases for a specific program location, they consider all the methods equally important and do not prioritize the methods relevant to the NPE. For example, EvoSuite

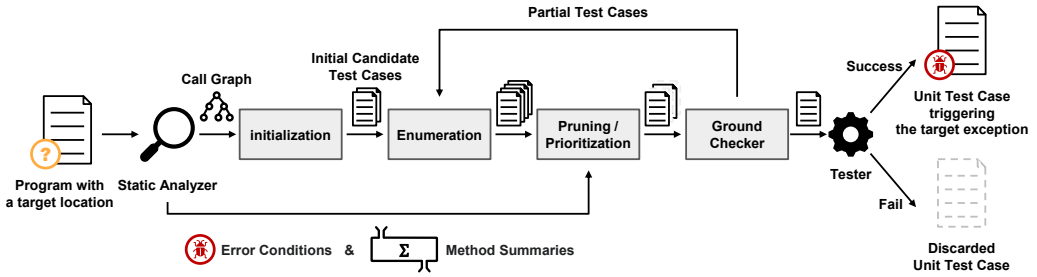


Fig. 2. Overview of UnitCon.

and Randoop generated 95 and 79,807 test cases on average, respectively. However, EvoSuite triggered the NPE only once out of 10 trials and Randoop never triggered the NPE[2].

2.2 Our Approach

We propose UnitCon, which synthesizes targeted unit tests to trigger runtime exceptions at a given program location. The overall architecture of UnitCon is shown in Figure 2. Given a program and a target location, UnitCon basically enumerates all possible test cases aiming for the target location and checks whether each of the synthesized tests triggers the target exception. To tackle the huge search space, we designed efficient search strategies using static analysis. As a result, UnitCon can synthesize a crashing unit test shown in Figure 1(b) within only 20 seconds. In the rest of the section, we describe how UnitCon synthesizes the targeted unit test.

2.2.1 ***Initialization.*** For the given target program location, UnitCon first collects the *error entry methods* using the call graph derived by the static analyzer. Error entry methods are public methods in the program under test that can potentially call the method enclosing the target location. Since unit tests are typically defined outside the class containing the target location, UnitCon synthesizes sequences of statements that call the public error entry methods. If the target location is in a private method, UnitCon collects the public methods that can call the private method using the call graph. If the target location is in a public method, the method itself is considered an error entry method. In the example, we collect only `visit` at line 7 as the error entry method since it is public.

Given a set of error entry methods, UnitCon generates an initial set of partial test cases each of which calls an error entry method. Such partial test cases are written in a domain-specific language (DSL) that we designed for the synthesis. The DSL is a subset of Java with placeholders for identifiers, expressions, and statements. For the example, UnitCon starts with the following partial test case T_0 for the example: ID.`visit`(ID), where each ID is a placeholder for an identifier, which can be a receiver or an argument.

2.2.2 ***Enumerative Synthesis.*** Given a set of partial test cases, UnitCon enumerates new partial unit tests by expanding the placeholders. We designed a set of rules each of which transforms a chosen partial test case into a new one. A rule $\langle \Gamma, S \rangle \rightarrow \langle \Gamma', S' \rangle$ represents a transformation of statement S in the partial test case under a type environment Γ into a new statement S' with an updated type environment Γ'. A type environment Γ is a mapping from an identifier (variable or method) to its type. The initial type environment is derived from the type information of the program under test and updated by the rules.

[2]We ran the tools 10 times due to their randomness.

$$\textsc{Recv}\ \frac{\text{fresh variable } x, \quad \Gamma[f] = \langle \tau_0, \tau_1 \rangle \to \text{void}, \quad \Gamma' = \Gamma \cdot \{x \mapsto \tau_0\}}{\langle \Gamma, ID.f(ID) \rangle \to \langle \Gamma', x := ID.M(ID); Stmt_x; x.f(ID) \rangle}$$

$$\textsc{Arg1}\ \frac{\text{fresh variable } x_1, \quad \Gamma[f] = \langle \tau_0, \tau_1 \rangle \to \text{void}, \quad \Gamma' = \Gamma \cdot \{x_1 \mapsto \tau_1\}}{\langle \Gamma, x_0.f(ID) \rangle \to \langle \Gamma', \ x_1 := Exp; x_0.f(x_1) \rangle}$$

$$\textsc{Arg2}\ \frac{\text{fresh variable } x_1, \quad \Gamma[f] = \langle \tau_0, \tau_1 \rangle \to \text{void}, \quad \Gamma' = \Gamma \cdot \{x_1 \mapsto \tau_1\},}{\langle \Gamma, x_0.f(ID) \rangle \to \langle \Gamma' \ x_1 := ID.M(ID); Stmt_{x_1}; x_0.f(x_1) \rangle}$$

Fig. 3. Selected production rules for expanding *ID*.

Figure 3 shows the selected rules that expand the placeholder *ID* for receiver and argument. For example, the rule Recv is fired when the partial test case is of the form $ID.f(ID)$ and the type of method f is $\langle \tau_0, \tau_1 \rangle \to$ void which means a void function type where τ_0 and τ_1 are the types of the receiver and the argument, respectively. If multiple rules are applicable to a given partial test case, UnitCon prioritizes the least recently generated placeholder to expand. For example, UnitCon triggers rule Recv for the initial partial test case T_0 as the return type of `visit` is void. The rule expands the placeholder and derives the following partial test case T_1:

T_1: Derived from T_0 using rule Recv

`ExpressionVisitorAdapter recv =` $ID.M(ID)$`;`
$Stmt_{\texttt{recv}}$`;`
`recv.visit(`*ID*`);`

where M and $Stmt_{\texttt{recv}}$ are placeholders for a method name and a statement, respectively. The type of `recv` is set to the type of the receiver of `visit` and the type environment is updated with the newly generated local variable `recv`. Note that the placeholder $Stmt_{\texttt{recv}}$ is added for the future expansion of the statement that uses the object `recv` as a receiver.

Since T_1 is not a grounded test case, UnitCon applies more rules to expand the placeholders in T_1. Among the multiple placeholders, UnitCon selects the least recently generated one, which is the highlighted *ID*. UnitCon applies the argument expansion rules and derives new partial test cases. Below, we present two representative cases:

T_2: Derived from T_1 using rule Arg1

`ExpressionVisitorAdapter recv =` $ID.M(ID)$`;`
$Stmt_{\texttt{recv}}$`;`
`SubSelect subSelect =` *Exp*`;`
 	
`recv.visit(subSelect);`

T_3: Derived from T_1 using rule Arg2

`ExpressionVisitorAdapter recv =` $ID.M(ID)$`;`
$Stmt_{\texttt{recv}}$`;`
`SubSelect subSelect =` $ID.M(ID)$`;`
$Stmt_{\texttt{subSelect}}$`;`
`recv.visit(subSelect);`

where *Exp* is a placeholder for an expression. The two partial test cases are the same except for the definition of `subSelect`. The former is initialized with an expression (constant or global variable), and the latter is initialized with a method call.

UnitCon iteratively performs the same process to expand the partial test cases. Once a grounded test case is derived, UnitCon checks if the test case triggers the target error. However, the search space grows exponentially for each expansion of a nonterminal. Thus, the naive enumerative search does not scale to real-world problems.

2.2.3 ***Search Space Pruning and Prioritization using Static Analysis.*** To improve the efficiency, we guide the search using static analysis results. UnitCon uses Infer [Facebook 2024] to estimate the semantics of each method as well as the conditions to trigger the target error. This in turn enables UnitCon to guide the search for a crashing test case.

First, UnitCon effectively prunes the search space by comparing the semantics between partial test cases. If two partial test cases are deemed to be semantically equivalent by the static analyzer, UnitCon discards the larger one.

Suppose T_2 and T_3 are further expanded using the corresponding rules as follows, respectively:

T_4: Derived from T_2	T_5: Derived from T_3
`ExpressionVisitorAdapter recv = ID.M(ID);`	`ExpressionVisitorAdapter recv = ID.M(ID);`
$Stmt_{\texttt{recv}}$;	$Stmt_{\texttt{recv}}$;
	`Merge merge = new Merge();`
	$Stmt_{\texttt{merge}}$;
`SubSelect subSelect = null;`	`SubSelect subSelect = merge.getUsingSelect();`
	$Stmt_{\texttt{subSelect}}$;
`recv.visit(subSelect);`	`recv.visit(subSelect);`

In T_5, the method `getUsingSelect` of the class `Merge` is used, which returns the value of the field `usingSelect`. Note that `usingSelect` is not initialized by the default constructor `Merge`. Thus, `m.getUsingSelect()` in T_5 returns `null` value according to the Java language semantics.

This behavior is captured by the static analysis. Infer estimates the semantics of each method and concludes that `Merge` and `getUsingSelect` do not initialize the field `usingSelect`. Furthermore, Infer captures that no other method creates an object for `usingSelect`. This means that `usingSelect` can return either (1) `null` if it is not initialized by any method call in $Stmt_{\texttt{merge}}$ or (2) a new object if it is initialized through the explicit call to a setter method in $Stmt_{\texttt{merge}}$. The first case is simply subsumed by T_4. For the second case, since no other method initializes `usingSelect`, the only way to initialize it is to create a new object in the unit test and call a setter method as follows:

```
...
Merge merge = new Merge();
SubSelect subSelect2 = ID.M(ID);
merge.setUsingSelect(subSelect2);
SubSelect subSelect = merge.getUsingSelect();
...
```

Note that this partial test case is subsumed by T_3 since the value of `subSelect` again depends on the choice of $ID.M(ID)$. Therefore, UnitCon can safely discard T_5.

Second, UnitCon effectively prioritizes the partial test cases that are more likely to trigger the target error. For a given target program location, the static analyzer estimates sufficient conditions for the target error. The conditions basically consist of the path conditions to reach the target location, but if possible, we also include the exception-triggering conditions for the target exception.[3] We call the conjunction of the path conditions and the exception-triggering conditions the *error condition*. The static analyzer may derive multiple error conditions for a target location. During the synthesis, UnitCon checks if the partial test case can potentially satisfy the error conditions. If so, UnitCon prioritizes the partial test case. In case of multiple error conditions, UnitCon uses all the error conditions equally for prioritization.

[3]In our implementation, we include nullness as the exception-triggering conditions if the target location contains a pointer.

Stmt	→	*ID* := *Exp*	assignment	*M*	→	*f*	methods
	|	*ID* := *ID*.*M*(*ID*)	non-void method call	*Exp*	→	*n* | null	primitive values
	|	*ID*.*M*(*ID*)	void method call		|	*g*	global constants
	|	*Stmt*; *Stmt*	sequence	τ	→	int | void | ⋯	primitive types
	|	*Skip*	no-op		|	*C*	class types
ID	→	*x*	variables		|	$\tau \times \tau \to \tau$	method types
	|	*C*	class names				

Fig. 4. Syntax and type of our domain-specific language.

Suppose UnitCon derives the following partial test cases after several more iterations from T_3:

T_6: Derived from T_3

```
...
SelectVisitor selectVisitor = null;

recv.setSelectVisitor(selectVisitor);
SubSelect subSelect = new SubSelect();
Stmt_subSelect;
recv.visit(subSelect);
```

T_7: Derived from T_3

```
...
SelectVisitor selectVisitor = new TablesNamesFinder();
Stmt_selectVisitor;
recv.setSelectVisitor(selectVisitor);
SubSelect subSelect = new SubSelect();
Stmt_subSelect;
recv.visit(subSelect);
```

According to the sufficient error condition derived by the static analyzer, the `selectVisitor` field of `recv` object should not be `null`. However, T_6 initializes `selectVisitor` with `null` through `setSelectVisitor()` function. On the other hand, T_7 initializes the field with a new object, `new TablesNamesFinder()`, which is a subtype instance of `SelectVisitor`. Thus, UnitCon prioritizes T_7 over T_6. However, we do not discard T_6 from the candidate set. Even though T_6 does not satisfy the condition, it still has a chance to change its semantics to meet the condition by further expansion. Moreover, static analysis results may not be accurate due to the unsoundness or incompleteness in practice. Therefore, UnitCon does not discard the partial tests but conservatively prioritizes more promising partial test cases.

These pruning and prioritization strategies enable UnitCon to efficiently search for a crashing test case. As a result, UnitCon can synthesize 3.6× and 1.3× more crashing test cases on average than Randoop and EvoSuite within 10 minutes, respectively, in our experiments.

3 The Overall Synthesis Algorithm

In this section, we formalize UnitCon and explain the detailed process. Given a program P and a target error E, UnitCon synthesizes a unit test case that triggers the target error. Each unit test for the target error is a method designed to invoke an *error entry method*, which is a public method that can potentially call the method enclosing the target error location. Thus UnitCon synthesizes a series of statements to construct the receiver and arguments of the error entry method.

In the rest of this section, we first define our DSL for targeted unit test synthesis. Then, we present our production rules for expanding the placeholders in the DSL. Finally, we describe our enumerative synthesis algorithm.

3.1 Domain-Specific Language

UnitCon synthesizes a unit test case that consists of a sequence of statements written in our DSL shown in Figure 4. The language is designed to be simple yet expressive enough to describe

common unit tests in Java; it includes assignment, method call[4], sequence, and no-op. We excluded conditional statements and loops because they are rarely used in unit tests in practice. We further assume that all fields are accessed via getters and setters following a common practice in Java.

A unit test written in the DSL consists of non-terminal and terminal symbols. Non-terminal symbols (e.g., *Stmt*) are placeholders for statements, methods, expressions, identifiers, and no-op, respectively. Terminal symbols (e.g., f) denote actual fragments of the program. A candidate unit test is *grounded* if it does not contain any non-terminal symbols. The grounded candidate can be executed within the original program and checked whether it triggers the target error.

UnitCon uses terminal symbols collected from the target program, error conditions, and pre-defined sets. We collect the methods, global constants, and classes defined in the program. For primitive values, we assume a predefined set of primitive values is given. In the implementation, we used 35 typical primitive values such as 0 or `'x'`. We also collect constant values used in the target program and their simple combinations such as arithmetic additions or string concatenations.

Our language has the same type system as Java. Figure 4 shows the types considered in our DSL. We will use class names and class types interchangeably. A method type consists of a receiver type, an argument type, and a return type. For two types τ_1 and τ_2, we define the subtype relation $\tau_1 \preccurlyeq \tau_2$ if and only if τ_1 is equal to or a subtype of τ_2.

3.2 Type-guided Production Rule

UnitCon iteratively replaces non-terminal symbols in the candidate test cases using a set of production rules. A rule $\langle \Gamma, S \rangle \rightarrow \langle \Gamma', S' \rangle$ denotes that a statement S in a given partial program with a type environment Γ can be expanded to a new statement S' with an updated type environment Γ'. We use the notation $\langle \Gamma, A \rangle \xrightarrow{r} \langle \Gamma', B \rangle$ to denote a specific production instantiated from rule r, which transforms code fragment A to B. We omit r when it is clear from the context. In total, UnitCon has 19 rules which are provided in the supplementary materials.

For each partial program, UnitCon applies possible production rules if there is a corresponding pattern in the program. If there exist multiple possible patterns, we expand the least recently generated placeholder according to the pre-defined production rules. When a code fragment A exists in a partial program p and a production $\langle \Gamma, A \rangle \xrightarrow{r} \langle \Gamma', B \rangle$ is applicable, we use the notation $p[B/A]$ to denote the new partial program where the fragment A in p is replaced with B.

Our rules are also based on type constraints managed by a type environment associated with each partial program. A type environment Γ stores the types of variables and methods in the target program P and updated during the synthesis. For example, $\Gamma[x]$ denotes the type of the variable x. Once a rule is applied, UnitCon derives the resulting (partial) test cases and the updated type environments, which include the types of newly introduced variables and methods by the rule.

3.3 Enumerative Synthesis

We describe our algorithm based on enumerative program synthesis in Algorithm 1. The algorithm starts with a static analysis that computes the initial type environment Γ_0, call graph CG, method summaries Σ, and error condition φ of the program and target error (line 2). Γ_0 contains the type information of all public methods and global constants in P. The call graph CG describes the call relations between methods in P. The method summaries Σ and the error condition φ are used for the search space pruning (line 14) and prioritization strategy (line 17), which will be described in the next section.

UnitCon first collects the error entry methods that can potentially reach the target error E by using the call graph CG (line 3). If a method enclosing the target error location is a public method,

[4]For brevity, we only consider one argument in this example.

Algorithm 1: UnitCon($\mathcal{A}$, P, E) where $\mathcal{A}$ is a static analyzer, P and E are the target program and the error.

```
1  Procedure UnitCon(A, P, E)
2      ⟨Γ0, CG, Σ, φ⟩ ← A(P, E)                                  ▷ Static analysis
3      F_e ← CollectErrorEntry(E, CG)
4      Q ← {⟨Γ0, ID.f_e(ID), 0⟩ | f_e ∈ F_e}                     ▷ Initialization
5      while not timeout and Q ≠ ∅ do
6          ⟨Γ, p, c⟩ ← PriorityDequeue(Q)
7          if Ground(p) then
8              if Reproduce(p, E) then return p                  ▷ Success
9          else
10             Q ← Q ∪ Unroll(Γ, Σ, φ, p, c)                     ▷ Enumeration
11 Procedure Unroll(Γ, Σ, φ, p, c)
12     Q ← ∅
13     R ← GetProductions(Γ, p)
14     R' ← Prune(Γ, Σ, R)                                       ▷ Pruning
15     forall ⟨Γ, A⟩ -r-> ⟨Γ', B⟩ ∈ R' do
16         p' ← p[B/A]
17         c' ← c + SynCost(A, B) + SemCost(B, p', φ)            ▷ Prioritization
18         Q ← Q ∪ {⟨Γ', p', c'⟩}
19     return Q
```

we collect only the method as an error entry. Otherwise, we collect all the public methods that can transitively call the target method through only private methods.

For each collected error entry method f_e, we initialize a set of tuples Q (line 4). Each tuple consists of the initial type environment, a partial test case, and a cost. Each initial test case contains only a method call to the error entry method where the receiver and arguments of the call are placeholders *ID*. The rest of the algorithm iteratively replaces such placeholders with actual statements, methods, expressions, and identifiers. The cost of each candidate test case is used to prioritize the candidates in Q and is initialized to 0.

The synthesis algorithm searches for a grounded program that triggers the target error. If found, the unit test is returned (line 8). Otherwise, the algorithm instantiates a rule that can be applied to the placeholder and derives all possible productions (line 13). Among the instantiated productions, UnitCon prunes the redundant ones using function Prune (line 14). Then UnitCon derives new unit tests by applying the remaining production rules to the current partial test and adds them to Q with the cost (line 15). UnitCon repeats this process until there are no more candidates or the time limit is reached. The details of the pruning and prioritization will be explained in the next section.

4 Guided Search Strategies via Static Analysis

In this section, we describe our search strategies to guide the enumerative synthesis algorithm using static analysis. We first formalize the static analysis used in UnitCon. Then we explain how UnitCon prunes and prioritizes the candidate test cases using Prune (line 14), SynCost and SemCost (line 17) based on the static analysis results.

$$\textsc{NonVoidMethod}\ \frac{f \in \mathcal{F},\ \ \Gamma[f] = \langle \tau_0, \tau_1 \rangle \rightarrow \tau_2,\ \ \tau_2 \preccurlyeq \Gamma[x]}{\langle \Gamma, x := ID.M(ID) \rangle \rightarrow \langle \Gamma, x := ID.f(ID) \rangle}$$

$$\textsc{VoidMethod}\ \frac{f \in \mathcal{F},\ \ \Gamma[f] = \langle \tau_0, \tau_1 \rangle \rightarrow \mathsf{void},\ \ \Gamma[x] \preccurlyeq \tau_0}{\langle \Gamma, x.M(ID) \rangle \rightarrow \langle \Gamma, x.f(ID) \rangle}$$

Fig. 5. Production rules for expanding the placeholder M to the grounded method name f where $\mathcal{F}$ is the set of pre-collected methods from the target program.

4.1 Static Analysis

For a given target program, the static analyzer computes the method summaries Σ and the error condition φ. A summary table Σ is a mapping from methods to summaries where a method summary σ is a set of states:

$$\begin{array}{rcrcl} \Sigma & \in & \mathit{Method} & \rightarrow & \mathit{Summary} \\ \sigma & \in & \mathit{Summary} & = & \wp(\mathit{State}) \\ s & \in & \mathit{State} & = & \mathit{PC} \times \mathit{Memory} \\ \pi & \in & \mathit{PC} & = & \wp(\mathit{SymVal} \otimes \mathit{SymVal}) \\ m & \in & \mathit{Memory} & = & \mathit{Var} \rightarrow \mathit{SymVal} \\ v & \in & \mathit{SymVal} & = & \mathbb{Z} \cup \mathit{Obj} \cup \{\mathsf{null}\} \cup \mathit{Symbol} \end{array}$$

A state s is a pair of a path condition and a memory. A path condition π is a set of constraints over symbolic values to reach the exit of a method or the target error location. A memory m is a mapping from variables to symbolic values. A symbolic value *SymVal* is a set of integers, objects, null, and symbols. A symbol represents a parameter or a field. An error condition φ is a formula representing the condition for the target error E, which is obtained by the summary of the error entry method call, Each error condition is a conjunction of the path condition to reach the error location and the nullness of the target object. We abuse the notation φ to denote a set of formulas if there exist multiple error conditions.

We used an analyzer based on symbolic execution. The analyzer symbolically enumerates a fixed number of paths in the method body and computes the states at the end of each path. We unroll loops a finite number of times. For example, the following table shows the method summaries of `getUsingSelect` and `selectVisitor` in Figure 1(a):

Method	Path Condition	Memory
`getUsingSelect`	True	{`ret` ↦ **usingSelect**}
`visit`	selectVisitor ≠ null ∧ subSelect ≠ null	{`subSelect` ↦ **subSelect** `selectVisitor` ↦ **selectVisitor** `withItemsList` ↦ **withItemsList**}

where the names in typewriter font are program variables (e.g., `ret`) and those in bold font are symbolic values (e.g., **usingSelect**). The variable `ret` denotes the return value. Note that the summary of error entry method `visit` has the path condition and the memory at the error location.

4.2 Search Space Pruning

We first use the static analysis results to prune out the partial unit tests that are semantically equivalent to others. UnitCon's pruning strategy is applied to the production rules shown in Figure 5 which expand the placeholder M to a concrete method name f. Note that these are the most crucial rules that can generate a huge number of new partial test cases because of the large number of methods and class inheritances in real-world Java programs.

When pruning productions instantiated from the rules, UnitCon considers two criteria: Prune_{rel} and Prune_{self}. The Prune_{rel} criterion checks whether two instantiations derive semantically equivalent test cases. Suppose the NonVoidMethod rule is instantiated to the following two productions:

$$p_1 : \langle \Gamma, x := ID.M(ID) \rangle \rightarrow \langle \Gamma, x := ID.f(ID) \rangle$$
$$p_2 : \langle \Gamma, x := ID.M(ID) \rangle \rightarrow \langle \Gamma, x := ID.f'(ID) \rangle$$

In this case, one of the productions is pruned out if the summaries of the two methods are equivalent:

$$\mathsf{Prune}_{rel}(p_1, p_2) \iff \Sigma(f) = \Sigma(f')$$

where the equivalence between two summaries σ and σ' holds if $\sigma \sqsubseteq \sigma' \wedge \sigma' \sqsubseteq \sigma$ where relation $\sigma \sqsubseteq \sigma'$ is defined as follows:

$$\forall \langle \pi, m \rangle \in \sigma. \exists \langle \pi', m' \rangle \in \sigma'.\ (\pi \iff \pi')\ \wedge\ \forall y. m(y) = m'(y)$$

For simplicity, we assume that the parameters are consistently named in the method summaries. [5] Intuitively, we consider two method summaries as equivalent if they have equivalent pairs of path conditions and memories.

The Prune_{self} criterion checks whether an instantiation derives a partial test case that is semantically equivalent to itself. Suppose the NonVoidMethod rule is instantiated to the following production: $\langle \Gamma, x := ID.M(ID) \rangle \rightarrow \langle \Gamma, x := ID.f(ID) \rangle$. In this case, $\mathsf{Prune}_{self}(p)$ holds if the following conditions hold:

(1) f returns a field of its receiver,
(2) f does not allocate a new object to the field,
(3) no other method that can generate the receiver of f allocates a new object to the field, and
(4) no other member method of the receiver object allocates a new object to the field.

The intuition behind Prune_{self} is that we discard candidate test cases that just propagate objects through method calls and do not allocate new objects to the fields.

Suppose the following production is applicable to a partial test case and the conditions for Prune_{self} hold:

`SubSelect subSelect =` $ID.M(ID)$ $\rightarrow$ `SubSelect subSelect =` ID`.getUsingSelect()`

The conditions indicate that `getUsingSelect` returns the value of the field `usingSelect`, but does not allocate a new object to the field. Also, no other method that can return an object whose type is a subtype of the receiver type of `getUsingSelect` (e.g., the constructor `Merge`) allocates a new object to the field. Finally, no other member method of the receiver object (e.g., `Merge.setUsingSelect`) allocates a new object to the field. In this case, the production is pruned.

One example that can be derived from such a redundant production is as follows:

```
Merge merge = new Merge();
Stmt_merge;
SubSelect subSelect = merge.getUsingSelect();
...
```

Because of the above conditions, `Merge`, `getUsingSelect` and all the other member methods of `merge` do not allocate an object to the field `usingSelect`. Then, the only way to allocate an object is to call a method in the unit test and then assign it to the field using a setter method. Note that such test cases are semantically equivalent to the initial one in the pruned production, in that the value of `subSelect` is determined by the placeholder $ID.M(ID)$. Thus, we prune out this case.

[5] This assumption is only for the sake of brevity. Our implementation handles arbitrary Java programs.

4.3 Search Space Prioritization

The second strategy is to prioritize the partial test cases. The key idea is to use the method summaries provided by the static analyzer to estimate the chance of triggering the target error. UnitCon prioritizes the smaller candidate test cases that are more likely to satisfy the error condition φ. The likelihood is estimated by the accumulated cost for productions applied so far (line 17). The cost for each production is computed as the sum of the syntactic and semantic costs.

The syntactic cost represents the size of the partial test case, following the conventional enumerative synthesis algorithms. When a part A of the partial test case is replaced by another part B by a rule, the cost of this production $\mathsf{SynCost}(A, B)$ is defined as $|B| - |A|$ where $|A|$ and $|B|$ denote the number of nodes in the abstract syntax tree of A and B, respectively.

To define the semantic cost, we first define the set of error conditions φ_i at the i-th statement of partial test case p, named p_i. The set φ_i is a set of error conditions that are backpropagated from the error entry method call to the i-th statement. Suppose p_n is the last statement of p which is the error entry method call and f is the name of the method called at p_n. Then, the set of propagated error condition φ_i at p_i represents the preconditions that are required to satisfy one of the error conditions at the error entry method call p_n. The propagated error condition φ_i at p_i is inductively defined from φ_{i+1} as follows:

$$\varphi_i = \begin{cases} \varphi & \text{if } i = n \\ \{\pi \Longrightarrow \pi' \mid \langle \pi, m \rangle \in \Sigma(f) \ \wedge \exists \pi' \in \varphi_{i+1}.m \Longrightarrow \pi'\} & \text{if } p_i \text{ is a call to a grounded method } f \\ \{\pi[e/x] \mid \pi \in \varphi_{i+1}\} & \text{if } p_i \text{ is an assignment of the form } x := e \\ \varphi_{i+1} & \text{otherwise } (i.e., p_i \text{ is not grounded}) \end{cases}$$

For simplicity, we assume that the symbolic values of parameters and return values are consistently named. [6] If p_i is a method call, φ_i is a set of the weakest preconditions of f each of which satisfies an error condition in φ_{i+1}. The precondition of f is computed using the method summary $\Sigma(f)$ provided by the static analyzer. If p_i is a grounded assignment as $x := e$, we also compute the weakest precondition that satisfies the error conditions by simply replacing the name x in each of the condition π in φ_{i+1} with e. If p_i is not grounded yet, the error condition φ_{i+1} is propagated to p_i.

Similarly, we define the set of post states $\mathsf{Post}(p_i)$ at the p_i. The set $\mathsf{Post}(p_i)$ is a set of memories which are propagated from the first statement of the partial test case to the p_i, and it is inductively defined from $\mathsf{Post}(p_{i-1})$ as follows:

$$\mathsf{Post}(p_i) = \begin{cases} \{m_\iota\} & \text{if } i = 0 \\ \{m \mid \langle \pi, m \rangle \in \Sigma(f) \wedge \exists m' \in \mathsf{Post}(p_{i-1}).m' \Longrightarrow \pi\} & \text{if } p_i \text{ is a call to a grounded method } f \\ \{m\{x \mapsto e\} \mid m \in \mathsf{Post}(p_{i-1})\} & \text{if } p_i \text{ is an assignment of the form } x := e \\ \mathsf{Post}(p_{i-1}) & \text{otherwise } (i.e., p_i \text{ is not grounded}) \end{cases}$$

For a base case, $\mathsf{Post}(p_0)$ is a singleton set of an empty memory m_ι. If p_i is a method call, the post state can be obtained from the method summary $\Sigma(f)$. If p_i is a grounded assignment, the post-state is the state after the assignment. If p_i is not grounded, the post state $\mathsf{Post}(p_{i-1})$ is propagated to p_i.

Finally, we define the semantic cost which measures the likelihood of triggering the target error. Suppose B is the newly introduced part at the b-th statement of p. Then the semantic cost SemCost is defined as follows:

$$\mathsf{SemCost}(B, p, \varphi) = \begin{cases} 0 & \text{if } \forall m \in \mathsf{Post}(p_b), \forall \{x \mapsto v\} \in m, \forall \pi \in \varphi_{b+1}.\ v \notin \pi \\ -|p_b| & \text{if } \exists m \in \mathsf{Post}(p_b).\ \exists \pi \in \varphi_{b+1}.\ m \Longrightarrow \pi \\ |p_b| & \text{otherwise} \end{cases}$$

[6] This assumption is only for the sake of brevity. Our implementation handles arbitrary Java programs.

We abuse the notation $v \in \pi$ to denote the symbolic value v is used in the condition π. Each of the three cases in the definition is explained as follows:

(1) UnitCon assigns the zero cost to p_b if the propagated φ_{b+1} is irrelevant to the post-condition of p_b. This happens due to one of the two reasons: (i) they are actually irrelevant, or (ii) the relevance has not been revealed yet since some of the subsequent statements are not grounded. Thus, we do not assign any cost in this case.
(2) If the current statement p_b has a chance to satisfy the error condition in φ_{b+1}, UnitCon discounts the cost by the size of p_b.
(3) Otherwise, it assigns the cost by the size of p_b.

Note that we do not prune the partial test case even if its estimated semantics contradict the error condition φ, as the partial test case could be extended further to meet the error condition later.

Suppose the following production r is applied at the b-th statement of a partial test case p and the error condition φ is `recv.selectVisitor` $\neq$ `null` at the error entry method call:

r: `SelectVisitor selectVisitor =` $Exp \longrightarrow$ `SelectVisitor selectVisitor = null`

i	p_i	φ_i
b :	`SelectVisitor selectVisitor = null;`	
$b+1$:	`recv.setSelectVisitor(selectVisitor)`	`selectVisitor` $\neq$ `null` $\Longrightarrow \varphi_n$
$b+2$:	$Stmt_{\texttt{recv}}$	φ_n
n :	`recv.visit(subSelect);`	`recv.selectVisitor` $\neq$ `null`

UnitCon propagates the error condition to the previous statements. Since the statement p_b is a grounded assignment, we can simply compute the post state of p_b, i.e., {`selectVisitor` = `null`}. Since this state contradicts the propagated error condition φ_{b+1}, UnitCon assigns the positive semantic cost to p by the size of p_b. However, if p_b is `SelectVisitor selectVisitor = new SelectVisitor();`, the post state of p_b satisfies the propagated error condition φ_{b+1}. In this case, UnitCon discounts semantic cost to p' by the size of p'_b.

Note that static analyzers may not accurately estimate method summaries or error conditions in practice when the target program uses complex language features such as recursion or external libraries without the source code. In such cases, we heuristically assign 0 to the cost of production to prevent such inaccurate analysis results from misguiding the search.

5 Evaluation

We aim to answer the following research questions:

RQ1 How effective is UnitCon in reproducing the target runtime exceptions?
RQ2 How effective is UnitCon in revealing unknown bugs?
RQ3 How do the pruning and prioritization strategies impact the performance of UnitCon?

5.1 Experimental Setup

We implemented UnitCon in 8K lines of OCaml and Python code. UnitCon uses the Infer framework for the static analysis. All the experiments were conducted on Linux machines with 512GB RAM and Intel Xeon 2.90GHz.

We compare UnitCon with five state-of-the-art unit test generation tools: EvoSuite [Fraser and Arcuri 2011], EvoFuzz [Moon and Jhi 2024], NPETest [Lee et al. 2024], UTBot [Ivanov et al. 2023] and Randoop [Pacheco and Ernst 2007]. EvoSuite and Randoop are the representative tools of search-based software testing and random testing for Java programs, and have been continuously evolving until now. EvoFuzz and NPETest are tools built on top of the latest snapshot of EvoSuite. We used the latest versions of EvoSuite (1.2), UTBot (2024.6.3-228072f), and Randoop (4.3.2).

Table 1. Benchmark characteristics. Column **Projs** denotes the number of projects in benchmarks. Columns **Files** and **LOC** denote the average number of Java files and their lines of code. Columns **Classes**, **Methods**, and **Global** indicate the average number of classes, methods, and global constants, respectively.

Benchmarks	Projs	Files	LOC	Classes	Methods	Global
Bears [Madeiral et al. 2019]	14	577	43,197	808	3,917	102
Genesis [Long et al. 2017]	16	566	43,850	963	4,563	170
NPEX [Lee et al. 2022]	59	1,170	97,786	1,291	8,263	270
VFix [Xu et al. 2019]	29	388	50,770	385	4,133	241
Defects4J [Just et al. 2014]	80	686	107,268	867	9,123	216
Total	198	769	86,513	926	7,399	224

We configured EvoSuite, its variants, and UTBot not to use mock objects for a fair comparison with UnitCon. To focus on the error detection capability, we configured Randoop to use the error-revealing mode rather than the regression mode and turned on the npe-on-null-input option that does not ignore NPEs when the method argument is `null`.

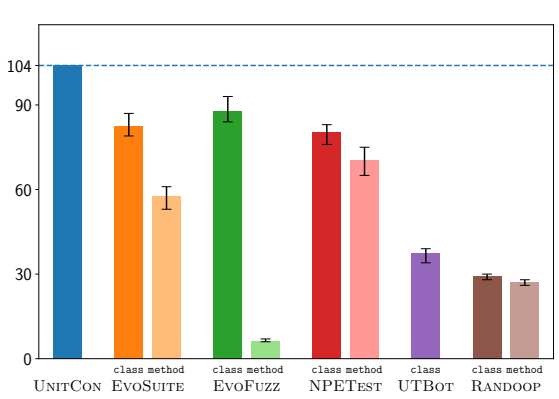
Since our goal is to trigger errors at a specific location, we used the configurations for targeted unit test generation provided by the tools when available. EvoSuite and Randoop provide the options to specify the target method or class. EvoFuzz and NPETest inherit the same options from EvoSuite. UTBot only provides the option to specify the target class.

We evaluated the tools on a collection of runtime exceptions from open-source Java projects. The benchmark projects are collected from the previous studies on program repair [Lee et al. 2022; Long et al. 2017; Madeiral et al. 2019; Xu et al. 2019] and Defects4J [Just et al. 2014]. Each project in the benchmarks has one target runtime exception bug. Among the 227 bugs in the benchmarks, we excluded one case that is not reproducible on recent versions of Java supported by EvoSuite. Additionally, we excluded 12 cases that are duplicated across two benchmark sets. We also excluded 12 cases where the target errors are triggered in the test cases rather than in the main program. Finally, we excluded 4 cases that are considered duplicates based on the target error location. Table 1 summarizes the statistics of the benchmarks which indicates the search space of the test case generation.

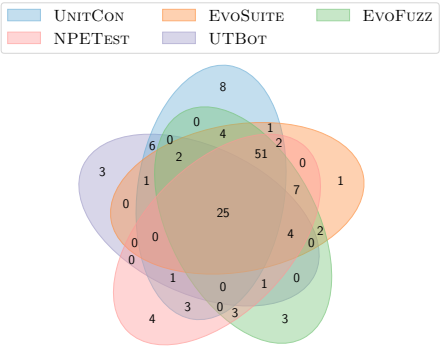
5.2 RQ1. Effectiveness of Target Error Reproduction

In this section, we evaluate the performance of UnitCon compared to the baselines in reproducing the target errors. We measure the number of successful test case generations within a given time limit. We set the time limit to 10 minutes for each benchmark following the experimental setting of previous studies [Galeotti et al. 2013; Lemieux et al. 2023; Ma et al. 2015]. Since the baselines are based on randomness, we ran each tool 10 times with different random seeds. For fair comparisons, we include the static analysis time in the running time; i.e., UnitCon runs both the static analyzer and the synthesizer within the time limit.

Figure 6(a) shows the effectiveness of UnitCon in reproducing the target exceptions compared to the baselines. Since UnitCon is based on a deterministic search algorithm, the results are consistent across the 10 runs. Conversely, the other tools are based on randomized algorithms. Thus, we represent both the maximum and minimum number of successfully reproduced target errors observed in a single run. UnitCon generated error-triggering test cases for 104 out of 198 (52.5%) projects within 10 minutes. On the other hand, the other tools succeeded in at least 29.2 (14.7%) and at most 87.7 (44.3%) cases, on average. Note that the best results in a single run of each tool range from 30 to 93, which is much lower than the results of UnitCon. Additionally, all



(a) The number of reproduced errors by each tool. The x-axis represents the name of each tool, and the y-axis represents the average number of successfully reproduced target errors. The whiskers of each bar denote the maximum and minimum number of reproduced errors out of 10 experiments. Bars with class and method indicate the target class and method option for supporting tools, respectively.



(b) The relationships among the reproduced errors by each tool. For the baselines, we report the number of errors reproduced at least once in 10 experiments.

Fig. 6. Performance comparison of UnitCon and the baselines in reproducing the target errors.

baselines, except for UTBot, reproduced fewer errors when they were executed with the target method option instead of the target class option. In total, UnitCon reproduces 1.2–3.6× more target errors than baselines.

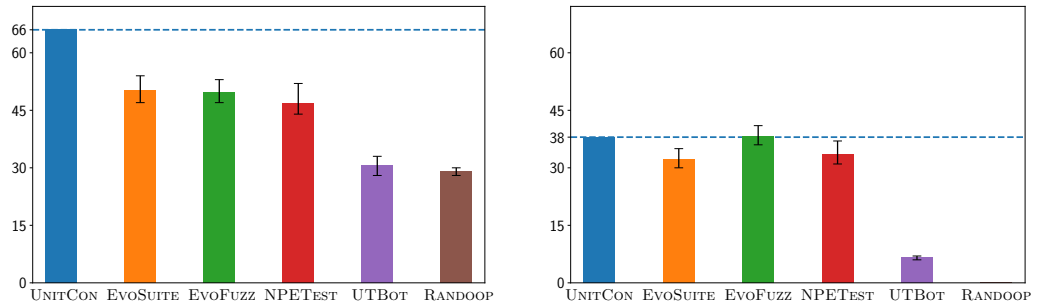
Figure 6(b) illustrates the relationships among the reproduced errors by each tool. We excluded the target method option results from Figure 6(b) for simplicity. The baselines reproduced different errors for the 10 experiments because of their randomness. Therefore, we present the errors that were reproduced at least once. In all cases where Randoop succeeded, either UnitCon or the other tools succeeded at least once. We excluded Randoop from the Venn diagram because it did not reproduce any errors that were not reproduced by the other tools.

UnitCon consistently reproduced 104 errors, while the baselines reproduced 30–102 errors at least once. The number of errors successfully reproduced by UnitCon is similar to that of EvoSuite and its variants. However, unlike these tools, UnitCon's results are based on a single run, whereas EvoSuite and its variants accumulated the results of 10 executions. Additionally, UnitCon exclusively reproduced 8 errors that none of the other tools found, which is the highest number of uniquely reproduced errors among the six tools.

There are 28 cases where UnitCon failed to reproduce the target exception while the other tools succeeded at least once across the 10 experiments. One of the main reasons is the limitation of the underlying static analyzer. The analyzer failed to capture the semantics of the target program when a complex feature (e.g., lambda function) is used. We conjecture this limitation can be resolved by using a more sophisticated analyzer. Another reason is the limitation of the search space. There are cases where the target errors are triggered by only using methods with the default modifier. Since our goal is to generate error-triggering unit tests using public methods, UnitCon fails to find such cases. On the other hand, the other tools are designed to generate a regression test suite for a given

Table 2. Effectiveness of UnitCon in generating test cases for different types of runtime exceptions. Column **Type** indicates the name of exception. Column **Succ.** and **Total** indicate the number of successful cases and the total number of cases, respectively.

Type	Succ.	Total	Type	Succ.	Total
ArrayIndexOutOfBoundsException	3	7	NumberFormatException	2	3
ArrayStoreException	0	2	OutOfMemoryError	0	3
ClassCastException	2	3	RuntimeException	0	4
IllegalArgumentException	11	14	StackOverflowError	0	2
IllegalStateException	3	6	StringIndexOutOfBoundsException	3	7
IndexOutOfBoundsException	2	2	UnsupportedOperationException	0	2
NullPointerException	66	122	Others (user-defined exceptions)	12	21



(a) The number of reproduced NPEs (b) The number of reproduced non-NPE exceptions

Fig. 7. The performance of each tool in reproducing different types of runtime exceptions. The notations are the same as Figure 6(a)

project. This in turn enables the other tools to trigger the target error by using methods with the default modifier.

UnitCon is effective in reproducing various types of runtime exceptions. Table 2 shows the number of successful test case generations for each exception type. Among the 14 exception types, UnitCon reproduced at least one target error for 9 exception types, including user-defined exceptions.

UnitCon generally outperforms baselines regardless of the exception types. Figure 7 illustrates the number of successful test case generations by each tool for exceptions categorized into NPEs and non-NPE exceptions. On average, UnitCon reproduced 1.3–2.3× more NPEs than the baselines. For non-NPE exceptions, UnitCon reproduced similarly to other tools except UTBot and Randoop. Notably, Randoop failed to discover any non-NPE exceptions.

In summary, UnitCon outperforms the baselines in reproducing the target errors. This result demonstrates that UnitCon can be used as a complementary tool to existing ones for various tasks such as validating the results of static analysis tools and reproducing the bugs reported by users.

5.3 RQ2. Effectiveness of Revealing Unknown Bugs

We evaluate the effectiveness of UnitCon in revealing unknown bugs in real-world projects. We used UnitCon to inspect the alarm reports of Infer's Pulse engine on a suite of the latest versions of our benchmarks. In total, we collected 51 projects which include popular libraries such as Apache Commons IO. Infer, by default, only reports *manifest* alarms which have strong evidence of being

Table 3. Reports for bugs discovered by UnitCon. Column **Project** and **Report** indicate the project in which the bug was discovered and its corresponding report, respectively. Column **Status** indicates the current status of the report as handled by the project's maintainers.

Project	Report	Status	Project	Report	Status
Activiti	Issue 4553	Open	Apache Johnzon	Issue 123	Patched
Apache Commons BCEL	Issue 289	Patched	Apache Karaf	Issue 1825	Patched
Apache Commons Configuration	Issue 355	Patched	Apache Karaf	Issue 1826	Patched
Apache Commons Configuration	Issue 365	Patched	Apache PDFBox	Issue 178	Patched
Apache Commons Configuration	Issue 368	Patched	Apache TsFile	Issue 50	Open
Apache Commons Configuration	Issue 381	Patched	Feign	Issue 2304	Patched
Apache Commons Configuration	Issue 382	Rejected	JSqlParser	Issue 1965	Patched
Apache Commons DBCP	Issue 352	Patched	Kubernetes Java Client	Issue 3081	Patched
Apache Commons IO	Issue 569	Rejected	Nutz	Issue 1613	Open
Apache Commons Math	Issue 236	Open	OpenGrok	Issue 4542	Patched
Apache Johnzon	Issue 117	Patched			

```
1  public class InstrFactory {
2    public InvokeInstr createInvoke(String className, String name, Type retType,
3                                    Type[] argTypes, short kind, boolean useInterface) {
4      if (kind != Const.INVOKESPECIAL || ...) {
5        throw new IllegalArgumentException(...);
6      }
7      ...
8      String sign = Type.getSignature(retType, ...);
9      // NPE if argTypes is null
10     for (Type argType : argTypes)
11     ...
12   }
13 }
```

(a) Simplified code from Apache Commons BCEL.

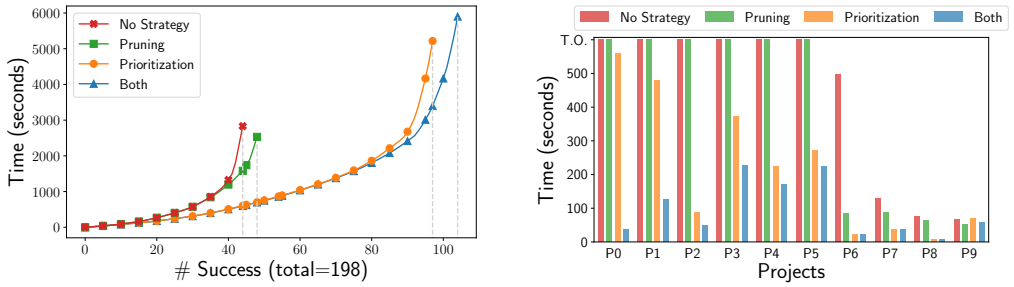
```
1 public static void main() {
2   InstrFactory recv = new InstrFactory(...);
3   boolean useInterface = true;
4   short kind = Const.INVOKESPECIAL;
5   Type[] argTypes = null;
6   Type retType = LONG_Upper.theInstance();
7   String name = "";
8   recv.createInvoke(name, name, retType, argTypes, kind, useInterface);
9 }
```

(b) Generated test triggering the NPE in Figure 8(a).

Fig. 8. A new NPE discovered by UnitCon.

true positives observed by the analysis [Le et al. 2022]. While this strategy is reasonable to avoid false positives, it may miss many actual errors in practice. Thus, for our bug-revealing experiment, we configured Infer to report all potential NPE alarms and run UnitCon to generate test cases targeting each alarm for 5 minutes. Note that UnitCon supports other types of runtime exceptions, but we focused on NPEs for this experiment because they are the main target of Pulse.

Among the 4,290 NPE alarms, UnitCon generated crashing test cases for 256 alarms. We randomly selected 10% (26) of the alarms and manually inspected the generated test cases. According to the inspection, those alarms are caused by 21 unique critical bugs. We reported the bugs and provided the test cases generated by UnitCon. Table 3 shows the list of reported unknown bugs. Among



(a) The x-axis represents the number of crashing test cases generated by UnitCon within the time limit, and the y-axis represents the accumulated time to synthesize the test cases.

(b) The performance of different strategies on the 10 target errors.

Fig. 9. The impact of search space pruning and prioritization.

them, 12 bugs have been patched within 2 weeks, with a total of 15 fixed. There are 4 reports still open. Only 2 reports were rejected. One case is because the maintainer decided to deprecate the feature that caused the NPE. In the other case, the maintainer determined that the NPE was the intended behavior. Notice that Infer with the default configuration reported only 9 alarms, including 6 false positives. In contrast, UnitCon discovered 256 crashes combined with Infer.

Figure 8 shows a reported bug that demonstrates the effectiveness of UnitCon. Observe that an NPE is triggered at line 10 if the parameter `argTypes` is `null`. To reach this line, the test case must not execute the true branch of the conditional statement at line 4 and successfully pass the method call to `Type.getSignature` at line 8. Infer correctly reported this potential bug along with the error condition $\{\texttt{argTypes} = \texttt{null} \wedge \texttt{kind} = 183\}$, where 183 is the constant used in line 4 and is the same value as global constant `Const.INVOKESPECIAL`.

UnitCon successfully generated the test case in Figure 8(b) within 40 seconds, while EvoSuite and Randoop did not find any error-triggering test cases within 5 minutes. Note that the search space is huge: the error entry method has 6 parameters having 5 different types, among which `kind` could take any integer value, and 73 different methods return `Type` objects. However, UnitCon effectively prioritized the partial test cases assigning `Const.INVOKESPECIAL` to `kind` and `null` to `argTypes`. Additionally, UnitCon pruned 23 out of 73 productions calling the methods to generate `Type` objects, effectively reducing the search space.

The overall results demonstrate that UnitCon can be used to reveal unknown bugs in real-world programs. Especially, combined with static analysis tools, UnitCon can be used to validate the results of the static analysis tools.

5.4 RQ3. Impact of Guided Search Strategies

We now evaluate the impact of the pruning and prioritization strategies of UnitCon. To measure the impact of the strategies, we experimented with four variants of UnitCon: (1) basic enumeration with no strategy, (2) pruning only, (3) prioritization only, and (4) both strategies. Each version of UnitCon experimented with a time budget of 10 minutes per benchmark for the same 198 benchmarks in RQ1.

Figure 9(a) summarizes the results in a cactus plot. Each line is labeled with the strategy combination, No strategy, Pruning, Prioritization, and Both, respectively. The results show that both the pruning and prioritization strategies are crucial for efficiently synthesizing targeted test cases. When no strategy is applied, UnitCon only succeeded in 44 cases. However, UnitCon with the

pruning and prioritization strategies succeeded in 48 and 97 cases, respectively. When both strategies are applied, UnitCon succeeded in 104 cases, which is 2.4 times more than the case with no strategy.

While the prioritization improves the number of successful test case generations, the pruning strategy also reduces the time required to synthesize the test cases. Figure 9(b) shows the detailed results for 10 cases that have the biggest performance gap depending on the Pruning strategy. The x-axis represents 10 example projects, and the y-axis represents the time required to reproduce the target error. For the example project P0 in Figure 9(b), UnitCon reproduces the target error in 566.0 seconds only with the prioritization strategy, while it takes 43.8 seconds with both strategies.

In summary, the results show that both of the strategies are essential for the effectiveness of UnitCon. The prioritization strategy is more effective in terms of the number of successful test case generations, but the performance can be further improved by the pruning strategy.

6 Related Work

Automated Unit Test Generation. There is a large body of research on automated unit test generation, but their main goal is different from UnitCon. EvoSuite [Fraser and Arcuri 2011] and Randoop [Pacheco and Ernst 2007] are the state-of-the-art tools in this field. Randoop randomly generates test cases using execution feedback from the program under test. EvoSuite uses a search-based approach to generate test cases using a genetic algorithm. Symstra [Xie et al. 2005], CUTE [Sen et al. 2005] and UTBot [Ivanov et al. 2023] employ symbolic and concolic execution to generate unit tests. These tools mainly aim to generate tests to achieve high coverage of the entire program. Instead, the goal of UnitCon is to generate a test case that triggers an error at a specific location.

Researchers have proposed various search strategies to enhance the performance of Randoop and EvoSuite. There have been several approaches to improve search algorithms of EvoSuite using large language model [Lemieux et al. 2023], fuzzing [Moon and Jhi 2024], many-objective genetic algorithm [Panichella et al. 2018], and control/data dependency analysis [Lin et al. 2021]. There also exist other approaches to effectively generate test cases that trigger errors while increasing code coverage of the entire program. SBST_{DPG} [Perera et al. 2020] guides the search of EvoSuite using defect prediction models based on code change history. NPETest [Lee et al. 2024] guides EvoSuite to generate test cases that trigger NPEs by utilizing static and dynamic analysis. GRT [Ma et al. 2015] integrated static and dynamic analysis results to Randoop. While these approaches are effective in increasing code coverage and finding errors in the entire program, UnitCon employs a detailed static analysis to generate test cases that trigger errors at specific locations. We demonstrated UnitCon outperforms the latest version of EvoSuite, Randoop and their variants (NPETest and EvoFuzz) with recently proposed search strategies and the error-revealing mode.

Program Synthesis. Our synthesis algorithm is inspired by syntax-guided program synthesis [Alur et al. 2013]. Recently, there have been approaches to improving the performance of program synthesis using static analysis [Guria et al. 2023; So and Oh 2017; Yoon et al. 2023]. Similar to UnitCon, they use static analysis to guide the search for solutions by reducing the search space. However, they aim to synthesize programs that fit given input-output examples. Instead, UnitCon synthesizes unit tests that trigger target runtime exceptions. Furthermore, the existing work uses simple static analyses to estimate the semantics of candidate programs. However, UnitCon analyzes the target program under test by using a detailed analysis.

Directed Test Generation. Directed test generation has been mainly explored in the context of symbolic execution and fuzzing. Researchers have proposed various techniques for directed symbolic execution for bug finding [Chen et al. 2020; Godefroid et al. 2005; Kim et al. 2019], patch

testing [Marinescu and Cadar 2013], incremental analysis [Person et al. 2011], and static analysis alarm inspection [Christakis et al. 2016; Ma et al. 2011]. Recently, researchers have also applied the concept of directed testing to fuzzing [Böhme et al. 2017; Chen et al. 2018; Du et al. 2022; Huang et al. 2022; Kim et al. 2023]. Similar to UnitCon, these techniques aim to produce test inputs that trigger errors at specific points. However, their main focus is to generate primitive types of inputs (e.g., integers) or simple data structures (e.g., list). In contrast, UnitCon aims to generate executable unit tests including a series of method calls. Because of this fundamental difference, tool competitions (e.g., SBFT2023 [SBFT 2023]) have separated tracks for unit test generation and fuzzing.

7 Limitations and Threats to Validity

Although we carefully designed our experiment, there are still limitations that we need to consider. First, our implementation entails the limitations of the underlying static analyzer such as the lack of support for the full Java standard libraries and recursion. Such limitations may affect the accuracy of the analysis results, which in turn degrade the effectiveness of UnitCon. However, we handle this issue by heuristically adjusting the priority of candidates as described in Section 4.

The randomness of the baselines is another threat to the validity of our experiment. We fixed the random seed to ensure the reproducibility of the experiment, but it may still affect the results.

Another threat to validity is the generalizability of our experimental results. We used Infer for the static analysis and therefore the results may not apply to other static analyzers. However, UnitCon relies not on the internal execution structure of the analyzer, but on the analysis results that provide the error trace that most of the bug-finding tools provide. Thus we believe that UnitCon can be easily extended to work with other static analyzers. Also, our experimental benchmarks may not be generalizable to other projects. To mitigate this threat, we used four different benchmarks from the previous studies. Moreover, we experimented UnitCon with the latest versions of the benchmark programs and received substantial feedback from their maintainers during the experiment on RQ2. The results still show the effectiveness of UnitCon.

8 Conclusion

We presented UnitCon, a system that effectively synthesizes targeted unit test cases for runtime exceptions in Java programs using static analysis. The key idea of this system is to prune and prioritize the search space by estimating the semantics of candidate test cases through static analysis. We evaluated this system on a suite of large Java programs and successfully reproduced runtime exceptions in 104 out of 198 projects. This represents 1.2–3.6 times more successful reproductions than the baselines within the same time limit. Additionally, we demonstrated that UnitCon can effectively discover real-world runtime exception errors combined with static analysis and discovered 21 previously unknown runtime exception errors in 51 open-source projects.

9 Data Availability

UnitCon is available as an archived version on Zenodo [Jang et al. 2025]. This provides the data and the scripts for reproducibility of experiments reported in the paper. Furthermore, UnitCon and the definition of full synthesis rules are open-sourced on our website: https://prosys.kaist.ac.kr/unitcon.

Acknowledgements

This work was partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2021-NR060080, 50%) and the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.RS-2025-02263143, Development of Ground Station Security Threat Response Technology for Space Cybersecurity, 50%).

A Synthesis Rules

We defined 19 production rules to expand the partial test cases written in our DSL using the following pre-defined sets in the target program:

$f \in$	$\mathcal{F}$	a pre-defined set of methods
$n \in$	Primitive	a pre-defined set of primitive values
$g \in$	Globals	a pre-defined set of global constants
$C \in$	ClassName	a pre-defined set of class names

A.1 Skip rule

For a partial program *Skip*, we can replace *Skip* with no-op. ϵ indicates no-op.

$$\frac{}{\langle \Gamma, \mathit{Skip} \rangle \rightarrow \langle \Gamma, \epsilon \rangle} \tag{1}$$

A.2 Constant rules

For a partial program $x := \mathit{Exp}$ with a concrete variable x, we can synthesize Exp under three constant rules.

$$\frac{n \in \text{Primitive}, \Gamma[x] = \text{Primitive}}{\langle \Gamma, x := \mathit{Exp} \rangle \rightarrow \langle \Gamma, x := n \rangle} \tag{2}$$

$$\frac{g \in \text{Globals}, \ \Gamma[g] \preccurlyeq \Gamma[x]}{\langle \Gamma, x := \mathit{Exp} \rangle \rightarrow \langle \Gamma, x := g \rangle} \tag{3}$$

$$\frac{\Gamma[x] \neq \text{Primitive}}{\langle \Gamma, x := \mathit{Exp} \rangle \rightarrow \langle \Gamma, x := \text{null} \rangle} \tag{4}$$

A.3 Function call in assignment rules

For a partial program $x := \mathit{ID}.M(\mathit{ID})$ with a concrete variable x, we can synthesize the left M.

$$\frac{f \in \mathcal{F}, \ \Gamma[f] = (\tau_0, \tau_1) \rightarrow \tau_2, \ \tau_2 \preccurlyeq \Gamma[x]}{\langle \Gamma, x := \mathit{ID}.M(\mathit{ID}) \rangle \rightarrow \langle \Gamma, x := \mathit{ID}.f(\mathit{ID}) \rangle} \tag{5}$$

A.4 Receiver in assignment rules

For a partial program $x := \mathit{ID}.f(\mathit{ID})$ with concrete variables x, f, we can synthesize the left ID under three rules.

$$\frac{C \in \text{ClassName}, \ \Gamma[f] = (\tau_0, \tau_1) \rightarrow \tau_2, \ \Gamma[C] \preccurlyeq \tau_0}{\langle \Gamma, x := \mathit{ID}.f(\mathit{ID}) \rangle \rightarrow \langle \Gamma, x := C.f(\mathit{ID}) \rangle} \tag{6}$$

$$\frac{\text{fresh variable } x_1, \ \Gamma[f] = (\tau_0, \tau_1) \rightarrow \tau_2, \ \tau_2 \preccurlyeq \Gamma[x_0]}{\langle \Gamma, x_0 := \mathit{ID}.f(\mathit{ID}) \rangle \rightarrow \langle \Gamma \cdot \{x_1 \mapsto \tau_0\}, x_1 := \mathit{Exp}; x_0 := x_1.f(\mathit{ID}) \rangle} \tag{7}$$

$$\frac{\text{fresh variable } x_1, \ \Gamma[f] = (\tau_0, \tau_1) \rightarrow \tau_2, \ \tau_2 \preccurlyeq \Gamma[x_0]}{\langle \Gamma, x_0 := \mathit{ID}.f(\mathit{ID}) \rangle \rightarrow \langle \Gamma \cdot \{x_1 \mapsto \tau_0\}, \ x_1 := \mathit{ID}.M(\mathit{ID}); \mathit{Stmt}_{x_1}; x_0 := x_1.f(\mathit{ID}) \rangle} \tag{8}$$

A.5 Argument in assignment rules

For a partial program $x_0 := x_1.f(ID)$ with concrete variables x_0, x_1, f, we can synthesize the ID under two rules.

$$\frac{\text{fresh variable } x_2,\ \Gamma[f] = (\tau_0, \tau_1) \to \tau_2,}{\langle \Gamma, x_0 := x_1.f(ID) \rangle \to \langle \Gamma \cdot \{x_2 \mapsto \tau_1\},\ x_2 := Exp; x_0 := x_1.f(x_2) \rangle} \tag{9}$$

$$\frac{\text{fresh variable } x_2,\ \Gamma[f] = (\tau_0, \tau_1) \to \tau_2,}{\langle \Gamma, x_0 := x_1.f(ID) \rangle \to \langle \Gamma \cdot \{x_2 \mapsto \tau_1\},\ x_2 := ID.M(ID); Stmt_{x_2}; x_0 := x_1.f(x_2) \rangle} \tag{10}$$

A.6 Void statement rules

For a partial program $x_0 := x_1.f(x_2); Stmt_{x_0}$ with concrete variables x_0, x_1, x_2, f, we can synthesize the $Stmt_{x_0}$ under two rules.

$$\frac{}{\langle \Gamma, x_0 := x_1.f(x_2); Stmt_{x_0} \rangle \to \langle \Gamma, x_0 := x_1.f(x_2); Skip \rangle} \tag{11}$$

$$\frac{}{\langle \Gamma, x_0 := x_1.f(x_2); Stmt_{x_0} \rangle \to \langle \Gamma, x_0 := x_1.f(x_2); Stmt_{x_0}; x_0.M(ID) \rangle} \tag{12}$$

A.7 Function call in void statement rules

For a partial program $ID.M(ID)$ or $x.M(ID)$ with a concrete variable x, we can synthesize the M.

$$\frac{f \in \mathcal{F},\ \Gamma[f] = (\tau_0, \tau_1) \to \text{void},}{\langle \Gamma, ID.M(ID) \rangle \to \langle \Gamma, ID.f(ID) \rangle} \tag{13}$$

$$\frac{f \in \mathcal{F},\ \Gamma[f] = (\tau_0, \tau_1) \to \text{void},\ \Gamma[x] \preccurlyeq \tau_0}{\langle \Gamma, x.M(ID) \rangle \to \langle \Gamma, x.f(ID) \rangle} \tag{14}$$

A.8 Receiver in void statement rules

For a partial program $ID.f(ID)$ with a concrete variable f, we can synthesize the left ID under three rules.

$$\frac{C \in \text{ClassName},\ \Gamma[f] = (\tau_0, \tau_1) \to \text{void},\ \Gamma[C] \preccurlyeq \tau_0}{\langle \Gamma, ID.f(ID) \rangle \to \langle \Gamma, C.f(ID) \rangle} \tag{15}$$

$$\frac{\text{fresh variable } x,\ \Gamma[f] = (\tau_0, \tau_1) \to \text{void}}{\langle \Gamma, ID.f(ID) \rangle \to \langle \Gamma \cdot \{x \mapsto \tau_0\},\ x := Exp; x.f(ID) \rangle} \tag{16}$$

$$\frac{\text{fresh variable } x,\ \Gamma[f] = (\tau_0, \tau_1) \to \text{void}}{\langle \Gamma, ID.f(ID) \rangle \to \langle \Gamma \cdot \{x \mapsto \tau_0\},\ x := ID.M(ID); Stmt_x; x.f(ID) \rangle} \tag{17}$$

A.9 Argument in void statement rules

For a partial program $x.f(ID)$ with concrete variables x, f, we can synthesize the left ID under two rules.

$$\frac{\text{fresh variable } x_1,\ \Gamma[f] = (\tau_0, \tau_1) \to \text{void}}{\langle \Gamma, x_0.f(ID) \rangle \to \langle \Gamma \cdot \{x_1 \mapsto \tau_1\},\ x_1 := Exp; x_0.f(x_1) \rangle} \tag{18}$$

$$\frac{\text{fresh variable } x_1,\ \Gamma[f] = (\tau_0, \tau_1) \to \text{void}}{\langle \Gamma, x_0.f(ID) \rangle \to \langle \Gamma \cdot \{x_1 \mapsto \tau_1\},\ x_1 := ID.M(ID); Stmt_{x_1}; x_0.f(x_1) \rangle} \tag{19}$$

References

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-Guided Synthesis. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD)*.

Marcel Böhme, Bruno C. d S. Oliveira, and Abhik Roychoudhury. 2013. Regression Tests to Expose Change Interaction Errors. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. SAVIOR: Towards Bug-Driven Hybrid Testing. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*.

Maria Christakis, Peter Müller, and Valentin Wüstholz. 2016. Guiding Dynamic Symbolic Execution toward Unverified Program Executions. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. 2022. Windranger: A Directed Greybox Fuzzer Driven by Deviation Basic Blocks. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

Facebook. 2024. Infer Static Analyzer. https://fbinfer.com/

Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. 2013. Improving Search-Based Test Suite Generation with Dynamic Symbolic Execution. In *International Symposium on Software Reliability Engineering (ISSRE)*.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*.

Sankha Narayan Guria, Jeffrey S. Foster, and David Van Horn. 2023. Absynthe: Abstract Interpretation-Guided Synthesis. *Proceedings of the ACM on Programming Languages (PACMPL)* 7, PLDI (2023).

Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. BEACON: Directed Grey-Box Fuzzing with Provable Path Pruning. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*.

Dmitry Ivanov, Alexey Menshutin, Maxim Pelevin, Daniil Stepanov, Denis Fokin, Yury Kamenev, Egor Kulikov, Artemiy Kononov, Sergey Pospelov, Ivan Volkov, Alena Lisevych, Timur Yuldashev, Nikita Stroganov, and Andrey Tarbeev. 2023. UTBot at the SBFT 2023 Java Tool Competition. In *IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*.

Sujin Jang, Yeonhee Ryou, Heewon Lee, and Kihong Heo. 2025. UnitCon: Synthesizing Targeted Unit Tests for Java Runtime Exceptions. https://doi.org/10.5281/zenodo.15205112

Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing Field Failures for in-House Debugging. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*.

Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. 2023. DAFL: Directed Grey-Box Fuzzing Guided by Data Dependency. In *Proceedings of the USENIX Security Symposium (Security)*.

Yunho Kim, Shin Hong, and Moonzoo Kim. 2019. Target-Driven Compositional Concolic Testing with Function Summary Refinement for Effective Bug Detection. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding Real Bugs in Big Programs with Incorrectness Logic. *Proceedings of the ACM on Programming Languages (PACMPL)* 6, OOPSLA1 (2022).

Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2022. NPEX: Repairing Java Null Pointer Exceptions without Tests. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

Myungho Lee, Jiseong Bak, Seokhyeon Moon, Yoonchan Jhi, and Hakjoo Oh. 2024. Effective Unit Test Generation for Java Null Pointer Exceptions. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

Yun Lin, You Sheng Ong, Jun Sun, Gordon Fraser, and Jin Song Dong. 2021. Graph-Based Seed Object Synthesis for Search-Based Unit Testing. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

Fan Long, Peter Amidon, and Martin C. Rinard. 2017. Automatic Inference of Code Transforms for Patch Generation. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. 2011. Directed Symbolic Execution. In *Proceedings of the International Conference on Static Analysis (SAS)*.

Lei Ma, Cyrille Artho, Cheng Zhang, Hiroyuki Sato, Johannes Gmeiner, and Rudolf Ramler. 2015. GRT: Program-analysis-guided Random Testing (T). In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

Fernanda Madeiral, Simon Urli, Marcelo de Almeida Maia, and Martin Monperrus. 2019. BEARS: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*.

Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: High-Coverage Testing of Software Patches. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

Seokhyeon Moon and Yoonchan Jhi. 2024. EvoFuzz at the SBFT 2024 Tool Competition. In *Proceedings of the ACM/IEEE International Workshop on Search-Based and Fuzz Testing (SBFT)*.

Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-Directed Random Testing for Java. In *Proceedings of ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Transactions on Software Engineering* 44, 2 (2018).

Anjana Perera, Aldeida Aleti, Marcel Böhme, and Burak Turhan. 2020. Defect Prediction Guided Search-Based Software Testing. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. 2011. Directed Incremental Symbolic Execution. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*.

Van-Thuan Pham, Wei Boon Ng, Konstantin Rubinov, and Abhik Roychoudhury. 2015. Hercules: Reproducing Crashes in Real-World Application Binaries. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

SBFT. 2023. Search-Based and Fuzz Testing. https://sbft23.github.io/tools/

Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

Sunbeom So and Hakjoo Oh. 2017. Synthesizing Imperative Programs from Examples Guided by Static Analysis. In *Proceedings of the International Conference on Static Analysis (SAS)*.

Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. 2005. Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

Xuezheng Xu, Yulei Sui, Hua Yan, and Jingling Xue. 2019. VFix: Value-Flow-Guided Precise Program Repair for Null Pointer Dereferences. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

Yongho Yoon, Woosuk Lee, and Kwangkeun Yi. 2023. Inductive Program Synthesis via Iterative Forward-Backward Abstract Interpretation. *Proceedings of the ACM on Programming Languages (PACMPL)* 7, PLDI (2023).

Received 2024-09-13; accepted 2025-04-01