API-Knowledge Aware Search-Based Software Testing: Where, What, and How

Xiaoxue Ren
Zhejiang University
China
xxren@zju.edu.cn

Xinyuan Ye
Australian National University
Australia
xinyuan.ye@anu.edu.au

Yun Lin
Shanghai Jiao Tong University
China
lin_yun@sjtu.edu.cn

Zhenchang Xing
CSIRO's Data61 & Australian National University
Australia
Zhenchang.Xing@anu.edu.au

Shuqing Li
Chinese University of Hong Kong
China
sqli21@cse.cuhk.edu.hk

Michael R. Lyu
Chinese University of Hong Kong
China
lyu@cse.cuhk.edu.hk

ABSTRACT

Search-based software testing (SBST) has proved its effectiveness in generating test cases to achieve its defined test goals, such as branch and data-dependency coverage. However, to detect more program faults in an effective way, pre-defined goals can hardly be adaptive in diversified projects. In this work, we propose Kat, a novel knowledge-aware SBST approach to generate on-demand assertions in the program under test (PUT) based on its used APIs. Kat constructs an API knowledge graph from the API documentation to derive the constraints that the client codes need to satisfy. Each constraint is instrumented into the PUT as a program branch, serving as a test goal to guide SBST to detect faults. We evaluate Kat with two baselines (i.e., EvoSuite and Catcher) with a close-world and an open-world experiment to detect API bugs. The close-world experiment shows that Kat outperforms the baselines in the F1-score (0.55 vs. 0.24 and 0.30) to detect API-related bugs. The open-world experiment shows that Kat can detect 59.64% and 9.05% more bugs than the baselines in practice.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

Software Testing, Test Case Generation, Knowledge Graph

ACM Reference Format:
Xiaoxue Ren, Xinyuan Ye, Yun Lin, Zhenchang Xing, Shuqing Li, and Michael R. Lyu. 2023. API-Knowledge Aware Search-Based Software Testing: Where, What, and How. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616269

1 INTRODUCTION

Search-based software testing (SBST), as the most practical software testing solution, has proved its success in effectively covering pre-defined test goals (e.g., branches, data dependencies, lines, etc.) [20, 22, 52]. By measuring the distance on how far a test case is away from achieving a specific goal, SBST can evolve the test case to minimize such a distance. State-of-the-art SBST tools such as EvoSuite [19] and Randoop [44] are highly effective in generating diversified test cases regarding branch coverage, line coverage, and data-dependency coverage in practice [22, 28, 34, 52, 62]. They are widely used in detecting software bugs in the industry [3, 5, 11, 21, 46], and serve as popular baselines to evaluate many fault detection techniques in academia [33, 47, 53].

However, to detect real-world software faults in practice, a test generator needs to be enhanced with the specified oracle to be effective. To this end, many researchers propose approaches to enhance test generation with oracles for fault detection [2, 15, 22, 28]. Generally, the oracles are derived to validate either the final test results [8, 66] or the internal execution states [23, 37]. The approaches targeting the final test results usually derive *project-specific* oracle from the comments of the target method [9, 26] and the machine learning (or deep learning) based models [1, 57]. In contrast, the approaches targeting the internal execution states focus on *general* oracles such as the manifestation of runtime exceptions [37] and the violation of pre-defined heuristics (e.g., memory access and null value access) [23].

Generally, existing solutions that infer the internal execution states often rely on heuristics to derive a general oracle. While these solutions are practical for locating pre-defined types of faults (e.g., null value access and invalid array access), their rules are not easily expandable. Additionally, defining the granularity of these rules involves a tradeoff. The more general the rules (e.g., the program terminates with runtime exception have faults), the more likely that the rule is imprecise for a specific project (e.g., some programs may expect a runtime exception on some invalid inputs). The less general the rules, the more likely we miss detecting some program faults. Inspired by previous work [51], developers can effectively resolve the Oracle problem, by adhering to API usage specifications. The API usage specifications provide a standardized framework that enables accurate and reliable inference of internal execution states. This framework presents a notable advancement, overcoming the limitations associated with heuristic-based approaches and offering a more adaptable and scalable solution.

In this work, we propose **K**nowledge-**A**ware search-based software **T**esting (Kat), a knowledge extraction-based approach to generate on-demand assertions in the PUT (program under test), validating the internal execution states when interacting with API frameworks. Our rationale is based on the understanding that well-documented descriptions, particularly the exception handling knowledge, of APIs serve as informative oracles/specifications when APIs are utilized in the PUT. Moreover, the (in)consistencies between API specification and their usage in the code can be transformed into test goals to further facilitate SBST to discover the faults. Specifically, given a target program, by parsing the description of API and library documentation, Kat aims to identify (1) ***where*** in the code to test based on the (in)consistencies between API specification in the documentation and the API usage in the code, (2) ***what*** API specification in the code to test, and (3) ***how*** to test the specification (i.e., manipulate the source code to facilitate state-of-the-art test generators).

Technically, Kat is designed to (1) construct an API exception handling knowledge graph, which is done by extracting knowledge triples from API exception triggers in the documentation (e.g., [`String.charAt(index)`, throw, `IndexOutOfBoundsException`], [`index`, <, 0] and [`index`, >, `String.length()`] will be extracted from "*String.charAt(index) throws IndexOutOfBoundsException - if the index argument is negative or not less than the length of this string*", see Part I in Fig. 1), (2) transform the knowledge graph into program constraints instrumented as additional program branches (e.g., `if(startOffset < 0){throw new IndexOutOfBoundsException();}`, and `if(startOffset >= signature.length()){throw new IndexOutOfBoundsException();}` see Part III in Fig. 1 for more details), and (3) convert those instrumented branches as test goals for SBST tools (e.g., EvoSuite) to generate test cases to cover. The instrumented program constraints and their covering test cases serve as both the fault detection results and explanations. To further improve the performance, Kat also schedules the testing budget for different test goals regarding our measured ROI (return on investment) metrics for each goal.

We conduct extensive experiments to evaluate both the quality of API exception handling knowledge graph construction and the performance of Kat to detect software faults in experimental settings and in practice. To evaluate the constructed knowledge graph, we manually check the *MIN* (computed by a statistical sampling method [58]) samples from the total 39,132 API usage constraints, which are extracted from the exceptions (also called exception triggers) of 1,821 Java SE&JDK API classes. The results demonstrate that Kat achieves an impressive average accuracy of 96.46% in extracting exception triggers and successfully transforming them into knowledge triples. To evaluate the performance of Kat to detect software faults, we compare Kat with two baselines (i.e., **Catcher** [33] and **EvoSuite** [19]) in a close-world experiment and an open-world experiment. In the close-world experiment, we would like to assess the effectiveness of Kat in detecting API-related bugs, as it is designed to be API-knowledge-aware. We gathered a close-world dataset comprising 12 real-world bugs that violated API specifications. These bugs were reported, fixed, and accompanied by corresponding patches in the GitHub repository. This dataset allows us to compare Kat with other baselines in detecting API-related bugs using evaluation metrics (i.e., precision, recall and F1-score). By executing these baselines on the close-world dataset, we can assess their performance. The results indicate that Kat achieves an impressive overall F1-score of 0.55, surpassing the state-of-the-art baseline (Catcher) by an outstanding margin of 83.33% in terms of F1-score. This comparison demonstrates the superior performance of Kat in accurately detecting API-related bugs when compared to existing approaches. In the open-world experiment, we run Kat and baselines on 21 Apache projects, which have been used in other related work [33]. As the open-world dataset does not have the ground truth reported by developers, we compare Kat to baselines by counting the number of bugs found. The results show Kat discovers 265 real-world bugs in total within two days, outperforming the baselines by finding 9.05% and 59.64% more bugs, respectively.

We summarize our contributions as follows:

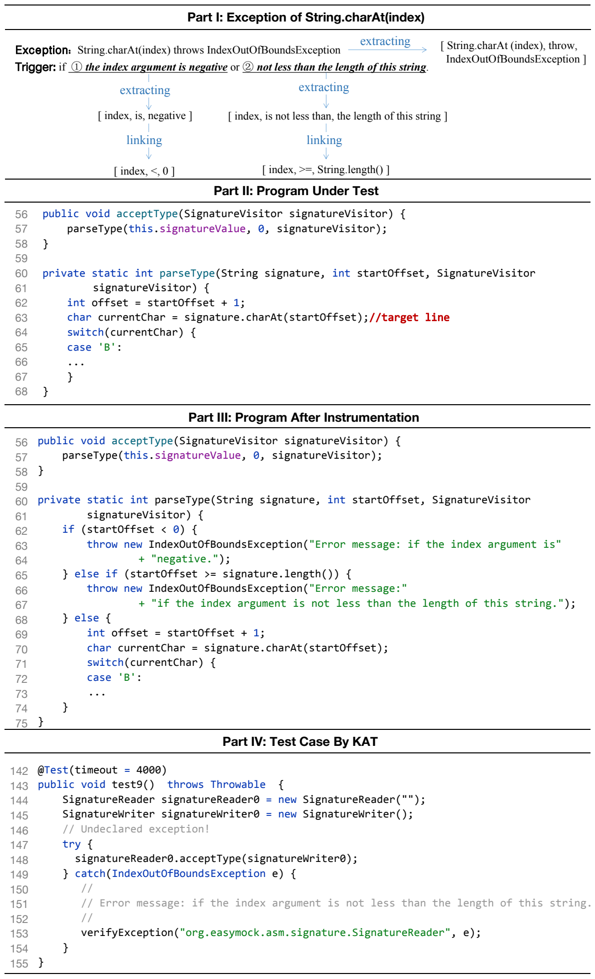
- We propose Kat, an SBST approach to detect bugs by translating the API specifications into on-demand assertions in PUT. These generated assertions serve as oracles for evaluating the internal execution states.
- We conduct comprehensive experiments to evaluate the effectiveness of Kat. The results demonstrate that Kat outperforms the state-of-the-art approaches in detecting bugs. Specifically, it achieves an impressive 83.33% improvement in F1-score and identifies 9.05% more bugs compared to Catcher.
- We report 22 new bugs in the Apache open-source projects. In addition, the dataset and tutorial to run Kat are available at https://github.com/XiaoxueRenS/KAT.

2 MOTIVATION

Fig. 1 shows a buggy code example from the EasyMock project where line 63 (Part II) can throw an `IndexOutOfBoundException` when the string length of `signature` is smaller than the value of `startOffset`. To detect such a bug, existing SBST approaches (e.g., EvoSuite [19] and Catcher [33]) define test goals, guiding the search-based approach to synthesize a test case to trigger such an exception. However, we run each approach on SignatureReader class of the EasyMock project for 30 minutes, failing to report the bug. Generally, both approaches suffer from the limits of brute-force search or lack of knowledge.

Challenge of EvoSuite. It is a big challenge for EvoSuite to discover the fault because it has abundant test goals (e.g., line coverage, branch coverage, etc.) to diverge its computation resource from the goals with actual bugs. In this example, SignatureReader class has 102 branches, 164 lines, and four methods. Treating all test goals with equality is computationally exhaustive. More specifically, **EvoSuite lacks the knowledge of *where* to test**.

Challenge of Catcher. Catcher is an SBST approach, built upon EvoSuite, to detect exception-triggered faults by localizing the call site with the potential to trigger unexpected exceptions. By investigating how the approach can work on the example, we find Catcher can effectively identify that line 63 is a potential call site with an unexpected exception. However, Catcher is challenging to synthesize a test case to confirm the exception. Since there is no guidance (e.g., branch distance) to synthesize the two specific Java objects into `SignatureReader` and `SignatureWriter` with fields


Part I: Exception of String.charAt(index)
Exception: String.charAt(index) throws IndexOutOfBoundsException
extracting
[String.charAt (index), throw, IndexOutOfBoundsException]
Trigger: if ① the index argument is negative or ② not less than the length of this string.
extracting
[index, is, negative]
linking
[index, <, 0]
extracting
[index, is not less than, the length of this string]
linking
[index, >=, String.length()]
Part II: Program Under Test
Part III: Program After Instrumentation
Part IV: Test Case By KAT

Part II: Program Under Test

```
56  public void acceptType(SignatureVisitor signatureVisitor) {
57      parseType(this.signatureValue, 0, signatureVisitor);
58  }
59
60  private static int parseType(String signature, int startOffset, SignatureVisitor
61          signatureVisitor) {
62      int offset = startOffset + 1;
63      char currentChar = signature.charAt(startOffset);//target line
64      switch(currentChar) {
65      case 'B':
66      ...
67      }
68  }
```

Part III: Program After Instrumentation

```
56  public void acceptType(SignatureVisitor signatureVisitor) {
57      parseType(this.signatureValue, 0, signatureVisitor);
58  }
59
60  private static int parseType(String signature, int startOffset, SignatureVisitor
61          signatureVisitor) {
62      if (startOffset < 0) {
63          throw new IndexOutOfBoundsException("Error message: if the index argument is"
64                  + "negative.");
65      } else if (startOffset >= signature.length()) {
66          throw new IndexOutOfBoundsException("Error message:"
67                  + "if the index argument is not less than the length of this string.");
68      } else {
69          int offset = startOffset + 1;
70          char currentChar = signature.charAt(startOffset);
71          switch(currentChar) {
72          case 'B':
73          ...
74      }
75  }
```

Part IV: Test Case By KAT

```
142 @Test(timeout = 4000)
143 public void test9()  throws Throwable  {
144     SignatureReader signatureReader0 = new SignatureReader("");
145     SignatureWriter signatureWriter0 = new SignatureWriter();
146     // Undeclared exception!
147     try {
148       signatureReader0.acceptType(signatureWriter0);
149     } catch(IndexOutOfBoundsException e) {
150        //
151        // Error message: if the index argument is not less than the length of this string.
152        //
153        verifyException("org.easymock.asm.signature.SignatureReader", e);
154     }
155 }
```

Figure 1: Example of a Bug in SignatureReader class of Easy-Mock Project

of specific value for test inputs (see Part IV). As a result, Catcher has to randomly generate test cases regarding the method calls of the `SignatureReader` class, with a slim chance of satisfying the specific constraints when synthesizing the Java objects. More specifically, **Catcher lacks the knowledge of *how* to test.**

To address the aforementioned challenges, we propose KAT, which is a knowledge-aware approach to (1) extract constraints from the API documentation and construct an exception handling knowledge graph to validate call sites of APIs in the PUT, (2) translate the constraints as program conditions instrumented in the PUT, and (3) guide SBST to synthesize the test cases covering test goals of the instrumented program conditions.

Specifically, KAT statically detects the (in)consistencies between API exception handling knowledge and the PUT. In Fig. 1, the API in the target line (i.e., line 63 of Part II) is `String.charAt(index)`, which has two constraints that may trigger `IndexOutOfBoundsException`, i.e., [`index`, <, `0`] and [`index`, >=, `String.length()`]. With static analysis, KAT transforms the above two constraints into triples [`startOffset`, <, 0] and [`startOffset`, >=, `signature.length()`] (see Part I in Fig. 1). Considering what and how to test, KAT transforms the inconsistencies into programs, and instruments additional branches into the PUT (i.e., lines 62-67 in Part III). With the instrumented branches, KAT searches with branch coverage to find bugs. Also, KAT computes the confidence score of the code instrumented by considering both knowledge confidence and instrumentation confidence (see section 3.2.2 and 3.3.2). With the confidence score, KAT can tell SBST how to allocate time budget to detect bugs. In this case, KAT is with a pretty high score and is confident in finding bugs. Part IV is the test cases generated by KAT, which can catch `IndexOutOfBoundsException` in the PUT by setting `signatureReader0` as a null object, which existing SBST methods cannot simulate.

3 APPROACH

Fig. 2 presents the overview of KAT, which consists of three main phases, i.e., *construction of API exception handling knowledge graph* (Phase-I), *static detection of potential bugs* (Phase-II), and *specification violation as test goals* (Phase-III). In Phase-I, we first extract API exception handling knowledge from Java SE&JDK API documentation [30], and then construct them into a knowledge graph. In Phase-II, KAT performs static analysis to detect potential bugs by checking whether the PUT violates the constraints in API handling knowledge (for *where*). In Phase-III, KAT transforms the violations into test goals to generate and validate test cases via code instrumentation (for *what*). Besides, KAT also schedules the testing budget for different test goals (for *how*). More details are described as follows.

3.1 Knowledge Graph Schema

KAT aims to generate on-demand assertions in the PUT according to the knowledge defined in the official API documentation. Thus, we construct an API exception handling knowledge graph based on the official Java SE&JDK API documentation [30], which with relatively complete contents and high quality [29, 50, 68].

The entities of the API exception handling knowledge graph include ***API components*** (*module*, *package*, *class*, *method*, *parameter*, *return-value*, and *exception*) declared in Java official documentation and ***value literals*** (*-1*, *null*, *true* and range like *negative* or *[0-9]*) extracted from textual usage directives in documentation. The relations between entities include ***declaration relation*** and ***constraint relation***. The declaration relations are between API components, including (*hasClass*, *hasMethod*, *hasParameter*, *return* and *throw*). The constraint relations include *constraint-enriched return* and *throw*, which are attached with conditions extracted from return contents and exception triggers in the documentation. For instance, Part I in Fig. 1 shows an exception handling sentences of `String.indexOf(index)`, which has the following two exception triggers (i.e., "*the index argument is negative*", and "*the index argument not less than the length of this string*").

3.2 Construction of API Exception Handling Knowledge Graph

The official Java SE&JDK API documentation contains semi structured API specifications, including API hierarchies (i.e., API declaration graph in this paper) and textual usage directives. To detect bugs violating API specifications, we focus on exception triggers

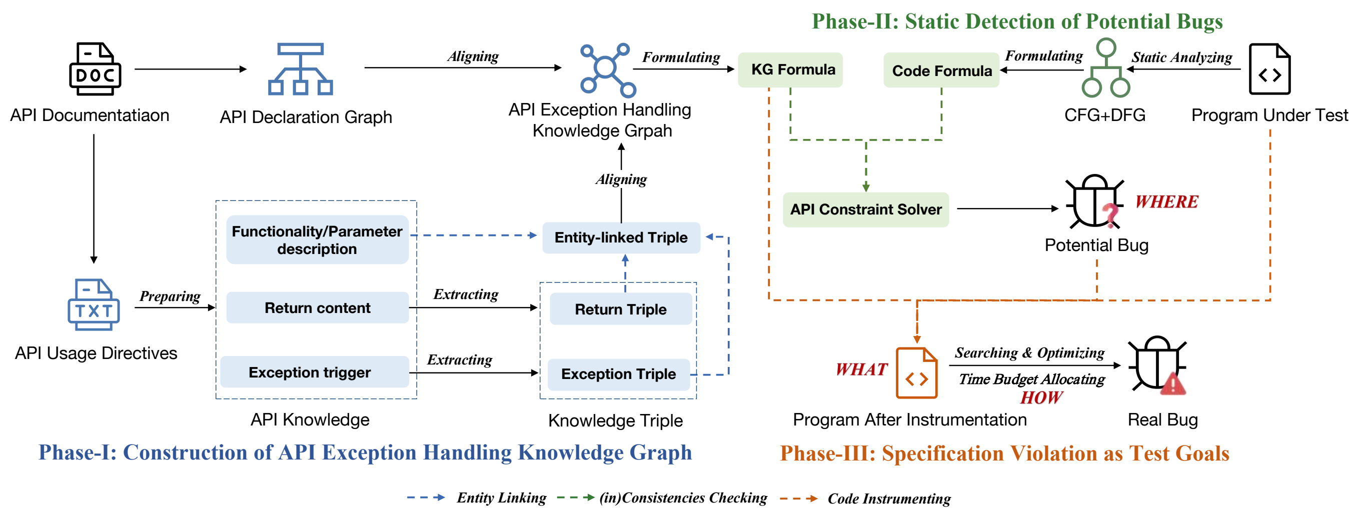


Figure 2: The Overall Framework of KAT

and build an API exception handling knowledge graph to represent the semantics of API usage in the PUT. The API exception handling knowledge graph consists of two parts: *the API declaration graph*, which can be accessed directly from the documentation, and *knowledge triples* extracted from textual usage directives. By aligning triples into the API declaration graph, we can finally obtain the API exception handling knowledge graph. The API declaration graph needed by us is the same as the generic API knowledge graph in [36, 39], and we follow their treatments to obtain the API declaration graph. Therefore, the central part of knowledge graph construction lies in knowledge triple extraction and alignment. Details are as follows:

3.2.1 *API Knowledge Preparation.*

We first do some preliminary work for constructing the knowledge graph, including knowledge collection and pre-processing.

Knowledge collection. We collect three types of textual API usage directives: *descriptions* (both functionality and parameter description), *return contents* and *exceptions*. Descriptions are the basic knowledge of APIs, which are usually composed of several descriptive sentences. We focus on the first sentence, commonly recognized as the most helpful knowledge [36]. It is usually a functionality summary or overall description. Return contents and exceptions are critical knowledge to triple extraction, which may contain essential conditions/triggers causing potential bugs.

Knowledge pre-processing. We pre-process the collected knowledge as follows. (1) First, we resolve coreferences, including both general coreferences (e.g., *it* and *them*) and domain-specific coreferences (e.g., *the arguments*). We use neuralcoref [43] for general coreferences, which is a state-of-the-art coreference resolution method. We use self-defined patterns to resolve domain-specific coreferences. Our observation is that conditions/triggers in return contents/exceptions often use "*any/either/all (of the) parameter(s)/argument(s)*" to represent their parameters. For example, the trigger of `NullPointerException` in `EnumSet.of(e1, e2)`, "*if any parameters are null*", means either `e1` or `e2` are null. Thus, this trigger should be resolved into "*if* `e1` *or* `e2` *are null*". (2) Then, we split clauses by parsing the syntax dependency tree of trigger sentences with Spacy [61]. We detect conjunctions of the sentences and complete the omitted parts in the subordinate clause based on the elements in the main clause. In the example above, the trigger after resolution can be further split into two triggers (i.e., "*if* `e1` *is null*" and "*if* `e2` *is null*"), and the relation between triggers is *OR*.

3.2.2 *Knowledge triple Extraction & Alignment.*

To construct the API exception handling knowledge graph, we first represent the knowledge in the format of triples via entity-relation extraction approaches, and then align the triples into the declaration graph by linking entities.

Triple Extraction. Return contents and exceptions we collected from API documentation often come with constraints, which are called return conditions and exception triggers, respectively. Return contents are helpful when analyzing the data flow, and exceptions are closely related to bugs violating API specifications. Hence, we need to extract knowledge triples from return contents and exceptions, as well as their constraints.

To extract knowledge triples from textual sentences, we employ Spacy [61] to parse sentences into semantic dependency trees, from which we identify entities and relations. Since official documentations usually have a relatively fixed expression, we summarize three common sentence structures of exception triggers and return conditions, including subject-verb (S-V) (e.g.,"*if the character does not occur*"), subject-verb-predicative (S-V-P) (e.g., "*the beginIndex is negative*"), and subject–verb–object (S-V-O) (e.g., "*beginIndex is larger than endIndex*"). When extracting knowledge triples, we first detect the main verbs and categorize the sentences into the corresponding structures. Then, by detecting nouns linked by the verbs, which are the entities, we can obtain the entity-relation triples from sentences. Note that compound nouns and noun phrases often appear in the documentation. We need to merge them into one noun with Spacy [61] before extracting triples. For example, "*the length of the string*" should be merged into one entity.

Inspired by [51, 70], to make the knowledge graph better serve bug detection with specific bug types, we further classify exception triggers into four types, i.e., *T1: nullness prohibition*, *T2: range limitation*, *T3: call dependency*, and *T4: other restriction*. They correspond to four types of bugs violating API specifications, i.e., *B1: missing null check*, *B2: missing range check*, *B3: missing call*, and *B4: missing* *try-catch*. After observation, *T2* and *T3* triggers are consistent in expression, which can be identified by detecting key patterns. We compare the two pattern sets defined in [42, 69] and obtain 15 key patterns (available in https://kat4cs.com/) related to our categories for trigger classification (e.g., "*null*", "*negative*", ">" and "*less than*"). *T3* and *T4* cannot be identified through pattern matching due to the absence of templated expressions. However, they can be resolved by linking entities with semantic similarities, referring to the triple alignment part for details. More specifically, the knowledge triples of triggers in Fig. 1 both belong to $T2$ (i.e., [`index, is, negative`] and [`index, is not less than, the length of this string`]), which may cause missing range check bugs.

Triple Alignment. As API usage directives are organized in unstructured natural language, which has different formats from structured knowledge in the API declaration graph, we perform entity linking over API usage directives to align the extracted triples to the entities in the API declaration graph. For preparation, we train a domain-specific word embedding model to evaluate the similarity between entities/sentences. Specifically, we use gensim [25] to fine-tune Google's pre-trained Word2vec model [65] with all texts from official Java documentation.

For alignment, we compute a confidence score for a triple with respect to the entity to judge whether to link. The confidence score is calculated with the following steps: (1) Direct URL linking: check whether there is a direct URL link with the entity. If any, match the entity to the object pointed by the link and denote the knowledge confidence as 1.0; (2) Intra-method linking: compute the similarity between the entity in the triple and the parameters in the same method, considering parameter names first and then descriptions. Once the similarity score is larger than 0.8, stop matching, and the similarity score is denoted as the knowledge confidence. (3) Intra-class linking: compute the similarity between the entity in the triple and methods in the same class similar to strategy (2). Stop matching when similarity is more than 0.8 and denote its knowledge confidence score. (4) Inter-class linking: compute the similarity between the entity in triple with related methods in other classes. The related methods include inherited methods declared in other classes or the parameter's functionality methods. Selecting the matched entity with the highest score (> 0.5), which is also denoted as its knowledge confidence score. The threshold here is different from 0.8, because the expressions in the same class are more similar, although they may not be related, while the expressions in different classes are more different, although they may be related. To avoid mismatching, we choose different thresholds. We also identify *T3* and *T4* triggers in this step. After detecting *T1* and *T2* triggers by matching key patterns, we continue to identify types of the rest triggers by linking entities after extracting triples. If the entity in a triple can be linked with another method, and the trigger is to check the state of the linked method, this triple can be identified as *T3*, otherwise *T4*. For example the triple [`Iterator.hasnext(), ==, true`] extracted from "*Iterator.next() throws NoSuchElementException if the iteration has no more elements*" is identified as *T3*, because "*has no more elements*" is linked with the `false` state of `Iterator.hasnext()`.

we finally align all knowledge triples into the API declaration graph by the linked entities, which forms our API exception handling knowledge graph. To better display the knowledge graph, we use heuristic rules to symbolize relations. Take the trigger "*the index argument is negative*" of `String.charAt(index)` as an example, the entities "*the index argument*" are ideally linked with the parameter of a 1.0 confidence score. Thus, the knowledge triple is [`index`, is, `negative`], which can be further represented as [`index`, <, 0].

3.3 Static Detection of Potential Bugs

After constructing the API exception handling knowledge graph, we use static analysis to detect the (in)consistencies between the knowledge graph and PUT to find potential bugs (referring to *where to test*). Specifically, it includes the following two steps: programs parsing the PUT to obtain flow information, and (in)consistencies checking to detect potential bugs.

3.3.1 *Programs parsing.*

To obtain flow information, we adopt Soot [60], a popular framework for analyzing and transforming Java programs, to parse the PUT. First, we use soot to transform the PUT into an intermediate representation as Jimple, which can essentially simplify flow analysis [63]. Then, we construct control flow and data flow graphs from Jimple files. With the flow graphs, we can traverse and record various flow information about the PUT, e.g., value assignment, variable type, handled exceptions, method invocations, and conditional branch, which can help transform the knowledge graph into program constraints.

3.3.2 *(In)Consistencies Checking.*

With the flow graphs, we first use the knowledge graph to detect target APIs in the PUT. The target APIs refer to the APIs with exception triples in the knowledge graph. Then, we collect corresponding arguments and constraints of target APIs in the PUT according to entities and relations in exception triples, which form the code formula (i.e., f_{Code}). After that, we also transform constraints of target APIs in the knowledge graph into program constraints by replacing entities with arguments in the PUT, which forms the KG formula (i.e., f_{KG}). Take the PUT in Fig. 1 as an example, it shows a process of `acceptType` calls `parseType`. The target API is `String.charAt(index)` in line 63, and we can traverse the flow graphs and obtain its code formula and KG formula (i.e., f_{KG}) as follows:

$$f_{Code} = \begin{cases} index = setOffset = 0 \\ String.length() = signature.length() \\ \qquad\qquad\qquad\quad = this.signature.length() \end{cases}$$

$$f_{KG} = OR\ (index < 0, index >= String.length())$$

Then, to check the (in)consistencies of the two formulas, we adopt an SMT solver (i.e., Z3 [49]) to check whether there is a solver that can trigger the exception, namely, whether there is a solver in f_{Code} can meet one of the constraints in f_{KG}. Therefore, we feed the arguments in f_{Code} into f_{KG}, obtaining a series of formulas under (in)consistencies checking. For example, when feeding f_{Code} of the PUT in Fig. 1 into its corresponding f_{KG}, we can get the (in)consistencies checking formula: $OR\ (0 < 0, 0 >=$ `this.signature.length()`$)$. By evaluating the (in)consistencies checking formula with Z3, we can find whether there is a potential bug or not. The output of Z3 should be *sat* or *unsat*. *unsat* means the API usage in the PUT meets the corresponding constraints in API specifications and does not trigger any exceptions, while *sat* means potential bugs may happen and will be marked as ***where*** to test. Specifically, in the PUT of Fig. 1, if `this.signature.length()<= 0`, `IndexOutOfBoundsException` might be thrown in line 63, see the test case by KAT (Part IV). Hence, the solver of the (in)consistencies checking formula should be *sat*, marking a potential bug in line 63.

Note that, we also compute a confidence score for detecting potential bugs (called bug confidence), which has three levels: 0.0 means no potential bugs because no target APIs are found; 1.0 means perfect confidence, it is when the determined/inferred argument value can meet the constraints; 0.5 is borderline confidence, it is when the argument is an uncertain input, and its value cannot be inferred from the knowledge graph. Specifically, the (in)consistencies checking formula of PUT in Fig. 1 is with borderline confidence (0.5) because the occurrence of the exception largely depends on the input of `signature`, which may be possible to trigger an exception.

3.4 Specification Violation as Test Goals

After finding potential violations by checking (in)consistencies between the PUT and knowledge triples, we transform the violations into code snippets for instrumentation, which serve as additional branches to cover.

3.4.1 Code Instrumentation. Considering the different types of triggers we extracted from the documentation, we first summarize the general expression of human patches and design different instrumentation patterns for each, which reflect ***what*** to test. Then, we use Javassist [10], a Java bytecode engineering toolkit, to instrument code snippets into the class files of projects. The instrumented code snippets serve as additional branches for SBST to generate test cases covering test goals.

Specifically, the instrumentation patterns are as follows: (1) For nullness prohibition and range limitation, we translate the violation into `if ([condition]) {throw new [Exception];}`, e.g., the violation of triple [`index`, <, 0] in Fig. 1 is converted into `if(index < 0){throw new IndexOutOfBoundsException;}`. (2) For call dependency, different from specific range and null, we add a state checking to check the state of the dependent API call, i.e., `if([API] == [State]){throw new [Exception];}`. For example, if the violate knowledge triple `[Iterator.hasNext(), false, Iterator.next()]` can be converted into `if(hasNext() == false){throw new NoSuchElementException;}`. (3) For other restrictions, we wrap the target line with `try{[Target]} catch([Exception] e)`. Note that the *OR/AND* relationship exists among different violations, so the positional relationship between instrumented code snippets should be considered. For example, the instrumented code snippets in Fig. 1 (see lines 62-68 in Part III) check two potential violations with the *OR* relationship.

3.4.2 Time Budget Allocation. According to the previous study [47], time budget is important. We design a strategy to allocate different time budgets concerning the confidence of the additional branch instrumented, which is ***how*** to test. This confidence is an average of knowledge confidence in section 3.2.2 and bug confidence in section 3.3.2. For each class under test, we allocate a total time, and each method in the class is allocated with a minimum time. Then, we continue to allocate the remaining time for methods, according to their confidence in the additional branch instrumented. As a result, the time budget of method m is $t_m = \frac{c_m}{\sum_{i=0}^{n} c_i}(T - t_{min} \times n) + t_{min}$, where T is the total time budget allocated to each class under test and the default value is 200 seconds, t_{min} is the minimum time allocated to each method in the class (2 seconds by default), n is the number of methods in a class, and c_m refers to the confidence score of method m.

3.5 Implementation

We implemented KAT on top of EvoSuite [19], a state-of-the-art Java testing framework. Many search algorithms have been proposed for EvoSuite, and we choose DynaMOSA as the search algorithm of KAT, which optimizes multiple coverage targets, simultaneously. DynaMOSA stands as the current state of the art and has been integrated into the EvoSuite tool [33, 38, 47].

Search algorithms have various parameters to set, but previous work [6] shows that parameter tuning SBST is extremely expensive and not necessary compared to default parameter values. Thus, we use the default settings of DynaMOSA. Moreover, considering KAT works by instrumenting additional branches, we choose branch coverage as the optimization objective.

4 EXPERIMENT SETUP

4.1 Baseline Methods

To investigate the overall performance of KAT, we select two state-of-the-art SBST approaches as baselines:

- Catcher [33]: Similar to KAT, Catcher also combines static exception propagation analysis with automatic search-based test case generation to detect crash-prone Java API misuses. The primary focus of Catcher is to narrow down the search space for automatic test case generation by honing in on API-call locations that are prone to triggering exceptions at runtime. To enhance its effectiveness, Catcher optimizes the search objective function by considering various coverage metrics, including branch coverage, line coverage, input coverage, and output coverage.
- EvoSuite [46]: It is a cutting-edge Search-Based Software Testing (SBST) approach designed specifically for generating test cases for Java classes. With its advanced techniques, EvoSuite has been shown to achieve remarkable code coverage and improved bug detection capability.

To illustrate the effectiveness of KAT's different design choices, we design two variants as follows:

- KAT$_{static}$: This variant of KAT keeps only the static analysis technique to detect bugs, i.e., Phase-I and Phase-II in Fig. 2. By comparing KAT to KAT$_{static}$, we evaluate the effectiveness of dynamic checking by generating a series of test cases (i.e., Phase-III in Fig. 2) in finding real bugs.
- KAT$_{dynamic}$: This variant of KAT keeps only the test case generation part of KAT (i.e., removing static detection with knowledge graph and specification violation as test goals). Thus, this variant is the same as EvoSuite. By comparing KAT to KAT$_{dynamic}$, we evaluate the effectiveness of detecting potential bugs and converting them into additional branches (i.e., Phase-I and Phase-II in Fig. 2) in finding API-related bugs.

4.2 Experimental Datasets

To evaluate the performance of KAT, Both close-world and open-world datasets are used to compare different evaluation metrics.

4.2.1 *Close-world Dataset.*
The close-world dataset refers to some API-related bugs with fixed patches, which means they have the ground truth and can be used to compute precision, recall and F1-score for evaluation. Given that Kat is designed as an API-knowledge-aware approach, we leverage the close-world dataset to conduct experiments specifically focused on evaluating the performance of API-related bug detection. However, upon manual inspection of Defects4j [32, 59], a popular benchmark for bug detection, only three bugs are related to Java SE&JDK API. Consequently, we followed the bug collection procedure outlined in Defects4j [32], searching for fixed bugs from historical commits on GitHub. This process resulted in the discovery of an additional nine fixed API-related bugs. Overall, our close-world dataset comprises a total of 12 reported bugs that have corresponding fixed patches. This dataset enables us to evaluate Kat's performance accurately and reliably in detecting API-related bugs.

4.2.2 *Open-world Dataset.*
The open-world dataset refers to open-source projects that may contain unreported bugs. We reuse the dataset released by Catcher [33], which consists of 21 large-scale projects from Apache. With this dataset, we can investigate the generalizability of Kat in the large-scale dataset by comparing the total number of detected bugs with Catcher and EvoSuite.

4.3 Research Questions

We investigate the following research questions:

- **RQ1: What's the quality of the knowledge graph?** We assess the accuracy of the triples extracted from the API documentation and evaluate the quality of the knowledge graph constructed.
- **RQ2: What's the overall performance of Kat?** We utilize a close-world dataset to compare the overall performance of Kat with other SBST approaches in finding API-related bugs.
- **RQ3: What's the effectiveness of Kat's different design choices?** We compare the performance of Kat with its variants on the close-world dataset to evaluate the effectiveness of our design choices.
- **RQ4: What's the generalizability of Kat on an open-world dataset?** We quantitatively and qualitatively analyze the performance of Kat and other baselines on an open-world dataset.

5 EVALUATION

5.1 Quality of Knowledge Graph (RQ1)

5.1.1 *Evaluation Strategy & Metric.*
The API exception handling knowledge graph is used to guide generating test oracles in this work, so its quality can directly affect the performance of Kat in detecting bugs violating API specifications. We need to evaluate whether the knowledge graph is highly qualified to be used. As we consider four types of exception triggers in the knowledge graph, we adopt a statistical sampling method [58] to examine the quality of the minimum number (MIN) of triggers in each type (see section 3.2.2), concerning their classified types and extracted triples. This sampling method ensures that the estimated accuracy is in a certain error margin at a certain confidence level. This MIN is calculated by $MIN = n_0/(1 + (n_0 - 1)/populationsize)$. In the equation, n_0 depends on the selected confidence level and the desired error margin: $n_0 = (Z^2 * 0.25)/e^2$, where Z is a confidence level's Z-score and e is the error margin. We use the error margin 0.05 at the 95% confidence level in our evaluation. Given a large number of triggers, MIN is approximately 384 in this statistical setting. Two authors are invited to independently evaluate the accuracy (i.e., $\frac{number\ of\ correct\ triples}{number\ of\ sampled\ triples}$) for the sampled triggers. They judge two aspects: whether or not the triggers are classified into the appropriate types, and whether the triples extracted from corresponding triggers are correct. We compute Cohen's Kappa [35] to evaluate the inter-rater agreement. If there is a disagreement between the above two judgments, authors must discuss and come to a consensus. Based on the consensus annotations, we evaluate the quality of each type of extracted information.

5.1.2 *Results of RQ1 - Quality of knowledge graph.*
Table 1 shows the statistics of triggers and their corresponding triples extracted. Table 2 reports the accuracy of all sampled triggers evaluated by us. The columns $Acc.1$ and $Acc.2$ show the accuracy determined by two annotators independently, and the column $Acc.F$ is the final accuracy after resolving disagreements. Column Kappa shows the inter-rater agreement.

The statistics of triggers show that we collect a total of 40,132 triggers with four types, covering 17,039 Java SE&JDK APIs. For the accuracy of different trigger types in Table 2, the final accuracy values are all above 94.00%, and the Cohen's Kappa values are over 0.77. These results demonstrate substantial agreement between the annotators and highlight the high quality of the extracted triples. This, in turn, reflects the quality of our API exception handling knowledge graph. Note that *T4* has the most significant number of triggers and *T3* has the least, and both of them show relevant low accuracy and Kappa. The reason is different from *T1* and *T2*, which can be identified by explicit patterns (e.g., "null", "less than" etc.), we cannot extract patterns to identify *T3* and *T4*. We identify them by computing sentence similarity when linking entities in triples (see section 3.2.2 for details). Thus, we sacrifice the quantity of *T3* for the high quality of the knowledge graph. Although the accuracy of *T3* is the lowest, it still achieves high performance with a final accuracy of 94%.

Answer to RQ1: Kat is supported by sufficient high-quality knowledge, covering a wide range of APIs, which can be reliably used in detecting potential bugs, generating and validating test cases.

5.2 Performance of Kat (RQ2/3/4)

5.2.1 *Evaluation Strategy & Metric.*
For **RQ2** and **RQ3**, we apply Kat and other baselines to generate test cases and validate them with ground truth for all the buggy classes in the close-world dataset. For **RQ4**, we apply Kat, Catcher and EvoSuite in the open-world dataset and compare how many bugs they can detect, respectively.

On the close-world experiments, we compute precision, recall, and F1-score to evaluate the performance of finding bugs among our Kat, the other two baselines (i.e., Catcher and EvoSuite to answer **RQ2**), and the variants of Kat (i.e., Kat$_{static}$ and Kat$_{dynamic}$ to answer **RQ3**). Precision ($\frac{TP}{TP+FP}$) represents the proportion of bugs that are correctly classified as bugs among all bugs detected by the methods. Recall ($\frac{TP}{TP+FN}$) represents the proportion of all bugs

Table 1: Four types of exception triples in Knowledge Graph

Trigger Type	# of triggers	# of covered API	Trigger Example	triple
T1	3,525	526	Method.getAnnotation(annotationClass) throw NullPointerException if the given annotation class is null.	NullPointerException: [annotationClass, ==, null]
T2	6,488	772	List.remove(index) throw IndexOutOfBoundsException if the index is out of range (index < 0 || index >= size())	IndexOutOfBoundsException: *OR* ([index, <, 0], [index, >=, List.size()])
T3	1,357	1,188	SortedSet.first() throw NoSuchElementException if this set is empty	NoSuchElementException: [util.Set.isEmpty(), ==, true]
T4	28,762	14,553	Certificate.getEncoded() throw CertificateEncodingException if an encoding error occurs	CertificateEncodingException: [Certificate.getEncoded(), try, catch]

Table 2: Quality of API Exception Handling Knowledge Graph

Trigger Type	MIN	Acc.1	Acc.2	Acc.F	Kappa
T1	346	99.72%	98.97%	99.25%	0.86
T2	363	97.40%	96.81%	97.15%	0.84
T3	300	95.28%	94.40%	94.00%	0.77
T4	379	96.33%	95.26%	95.42%	0.81
Average	347	97.18%	96.36%	96.46%	0.82

that are correctly classified as bugs. F1-score ($\frac{2 \times precision \times recall}{precision + recall}$) is the harmonic mean of precision and recall, which is a balanced metric of them. The four statistics appear above are as follows: FP (false positive) represents the number of bug-free programs that are classified as bugs; FN (false negative) represents the number of bugs that are classified as bug-free programs; TP (true positive) represents the number of bugs that are correctly classified as bugs; and TN (true negative) represents the number of bug-free programs classified as bug-free. On the open-world experiments to answer **RQ4**, we follow the evaluation metrics of Catcher (i.e., counting the total number of real bugs detected by KAT, Catcher and EvoSuite) to demonstrate the generalizability of KAT.

For experiments on the two datasets with SBST methods (i.e., KAT, Catcher, EvoSuite, KAT$_{dynamic}$), we repeat each of the experiments 25 times (the repeat times is determined by [33]) since such approaches employ optimization strategies to search. We allocate 200 seconds for each class to generate test cases in individual execution, and run each execution on gnu/Linux system (Ubuntu 18.04.6 LTS) with 5.4.0-128-generic Linux kernel, 28-core 2.60GHz Intel(R) Xeon(R) Gold 6348 CPU and 1TB RAM. Note that the variant KAT$_{static}$ does not require repeated experiments as it is based on static code analysis.

5.2.2 *Results of RQ2 - Overall Performance of KAT.*

Table 3 presents the comprehensive results obtained from the close-world dataset, highlighting the effectiveness of different approaches in identifying API-related bugs. The table consists of three columns: Catcher, EvoSuite, and KAT, representing the outcomes of two baselines and our proposed method, respectively. Each column indicates the ability of the corresponding method to detect bugs and the number of times it correctly identified the ground truth out of 25 executions. For instance, 7/25 (see the first row of column Catcher) signifies that Catcher accurately detected the bug on seven occasions during 25 executions. The Total row represents the total number of bugs identified among all the bugs present in the close-world dataset. For example, 5/12 (see the last row of column Catcher) means that Catcher successfully identified five bugs out of the total 12 bugs in the close-world dataset.

Table 3: Overall Performance of Bug Detection on the Close-world Dataset

Source	Type	Bug ID	Project Name	Catcher	EvoSuite (KAT$_{dynamic}$)	KAT$_{static}$	KAT
Defects4j	B1	CSV5	Commons-csv	7/25	6/25	8/1	21/25
	B4	Lang13	Commons-lang	-	-	1/1	-
		Lang37	Commons-lang	-	-	12/1	7/25
Github	B1	#02f519e	Commons-cli	-	-	3/1	25/25
		#48f0e90	Hibernate-ORM	6/25	-	8/1	25/25
	B2	#899f2f4	Rumble	-	-	6/1	25/25
		#2d0436e	sphinx4	3/25	15/25	5/1	25/25
		#d88d7a1	jMeter	-	13/25	8/1	25/25
	B3	#5787fe0	jStyleParser	3/25	-	4/1	25/25
		#7314516	jMeter	-	-	8/1	25/25
		#9ed8de3	msgraph-sdk	10/25	12/25	4/1	25/25
		#586d1ed	tabula	-	-	5/1	25/25
Total	-	-	-	5/12	4/12	72/12	11/12

From the total row of Catcher, EvoSuite and KAT, we can observe an overall better performance of KAT in detecting bugs violating API specifications. Specifically, KAT can find **six** $(11 - 5)$ **more bugs than Catcher** and **seven** $(11 - 4)$ **more bugs than EvoSuite** after repeating 25 times experiments. Such a significant achievement can be attributed to the fact that KAT can specifically point out where to test based on the (in)consistencies between the API knowledge graph and the API usage in the PUT. KAT then transforms the (in)consistencies into programs and instruments them into the PUT as test goals. Both Catcher and EvoSuite show poor performance, especially when no test goals are found because they can only search randomly for test goals in this case.

Such randomness is also shown by the hit times in 25 executions. The bugs detected by Catcher and EvoSuite often have a few hit times in all 25 executions, such as three times hitting the bug for sphinx4. However, as long as the bug is a little more complicated, it can be detected by neither Catcher nor EvoSuite. For an example of the bug #586d1ed in tabula, which is a *B3* bug about checking `Iterator.hasNext()` before `Iterator.next()`, it can be detected by neither Catcher nor EvoSuite. Unlike Catcher and EvoSuite, KAT can add test goals to the PUT by adding additional branches. In this case, KAT can avoid randomness and always hit the bugs, and more complex bugs can be detected. The PUT in Fig. 1 is a complicated example of missing test goals. Both Catcher and EvoSuite cannot detect it, because it is difficult for them to generate a null object to trigger the bug randomly.

Moreover, Table 4 shows the performance of each method from the aspects of different bug types by measuring precision, recall, and F1-score. The bug types are corresponding to the types of exception triggers. The results illustrate that *B2/B3/B4* (i.e., missing null check, missing range checking and missing call) show better performances than *B4* (i.e., missing try/catch). It is because the corresponding

Table 4: Results of Precision, Recall and F1-score on the Close-World Dataset

Bug Type	Catcher			EvoSuite (Kat$_{dynamic}$)			Kat$_{static}$			Kat			F1 Improvement		
	Prec.	*Rec.*	*F1*	*Prec.*	*Rec.*	*F1*	*Prec.*	*Rec.*	*F1*	*Prec.*	*Rec.*	*F1*	*Catcher*	*EvoSuite*	*Kat$_{static}$*
B1	0.40	0.67	0.50	0.33	0.33	0.33	0.16	1.00	0.27	0.50	0.67	0.57	14.00%	72.72%	111.11%
B2	0.20	0.33	0.25	0.20	0.33	0.25	0.16	1.00	0.27	0.38	1.00	0.55	120.00%	120.00%	103.70%
B3	0.22	0.50	0.31	0.13	0.25	0.17	0.19	1.00	0.32	0.40	1.00	0.57	83.87%	235.29%	78.13%
B4	-	-	-	-	-	-	0.15	1.00	0.27	0.50	0.50	0.50	-	-	85.18%
Total	0.26	0.36	0.30	0.20	0.29	0.24	0.16	1.00	0.28	0.42	0.79	0.55	83.33%	129.17%	96.43%

triggers, (i.e., *T1/T2/T3*) have more specific API constraints, which can be easily converted into test goals to help search algorithms optimize with branch coverage. In contrast, *T4* provides limited knowledge to guide test case generation because it is vague. Kat just instruments `try{[Target]} catch([Exception] e)` into the PUT, considering T4 triggers.

Answer to RQ2: Kat can find six and seven more API-related bugs than Catcher and EvoSuite, respectively. Moreover, Kat shows great performance in detecting *missing null check*/*missing range*/*missing call* bugs.

5.2.3 *Results of RQ3 - Effectiveness of Kat.*

To evaluate the effectiveness of Kat, we compare it with its two variants (i.e., Kat$_{static}$ and Kat$_{dynamic}$). Table 3 and Table 4 report the performance of Kat and its variants (i.e., Kat$_{static}$ and Kat$_{dynamic}$) in the close-world dataset.

In Table 3, we present the performance of Kat and its variants, Kat$_{static}$ and Kat$_{dynamic}$, on the close-world dataset. The column labeled Kat$_{dynamic}$ (EvoSuite) provides information on whether the bug can be detected by Kat$_{dynamic}$ and how many times it successfully detects bugs out of 25 executions. Column Kat$_{static}$ reports whether the bug can be detected by Kat$_{static}$ and how many bugs in total it can detect in the programs. For example, 8/1 (see the first row of column Kat$_{static}$) means the static approach (i.e., Kat$_{static}$) reports eight potential bugs and contains the ground truth bug. 72/12 (see the last row of column Kat$_{static}$) means Kat$_{static}$ reports a total of 72 potential bugs, which covers all the 12 ground truth bugs. In Table 4, we measure the precision, recall, and F1-score of Kat and its variants. Note that once the bug is detected among 25 executions, the result is denoted as positive; otherwise negative. Results of the close-world experiment in Table 3 show, although Kat$_{static}$ can find all bugs in the close-world dataset, it suffers from an extremely high false positive rate. Because Kat$_{static}$ uses static analysis to detect bugs, and naively assumes that programs violating constraints in knowledge graphs are bugs. Thus, bugs detected by Kat$_{static}$ require manual verification, which cannot effectively help in bug detection, and even increases the burden of verification. For Kat$_{dynamic}$, as discussed in section 5.2.2, it uses a random search strategy to generate test cases for bug detection automatically. The strategy makes it with acceptable results in the quality of the bug detected, but poor performance in the number of bugs in the close-world dataset.

Table 4 shows that the design of Kat is very effective, outperforming both variants with the improvement of F1-score by 129.17 % and 96.43%, respectively. Kat is able to achieve better performance because it considers the characteristics of both variants.

Table 5: Performance on the Open-world Dataset (Total reports the number of all unique bugs detected from the open-world dataset.)

ID	Project Name	Version	Kat	Catcher	EvoSuite
BCEL	bcel	6.2	8	**9**	7
CLI	commons-cli	1.4	0	0	0
CODEC	commons-codec	1.12	**12**	9	9
COLL	commons-collections	4.2	**15**	10	9
COMP	commons-compress	1.17	33	**34**	18
LANG	commons-lang	3.7	28	**32**	23
MATH	commons-math	3.6.1	**16**	12	9
EASY	easymock	3.6	**18**	16	11
GSON	gson	2.8.5	16	16	8
HAMC	hamcrest-core	1.3	0	0	0
JACK	jackson-databind	2.9.6	**7**	6	5
JAVS	javassist	3.23.1	**6**	4	2
JCOM	jcommander	1.71	2	2	1
JFCH	jfreechart	1.5.0	**28**	27	23
JODA	joda-time	2.10	23	**25**	11
JOPT	jopt-simple	5.0.4	**5**	3	3
NATT	natty	0.13	**5**	4	4
NEO4	neo4j-java-driver	1.6.2	**21**	17	8
SHIRO	shiro-core	1.3.2	**12**	7	6
XJOB	xwiki-commons-job	10.6	10	**10**	9
XTEX	xwiki-commons-text	10.6	0	0	0
Total	-	-	265	243	166

Kat designs an effective strategy to integrate both variants' advantages by converting constraints in the knowledge graph to program constraints and instrumenting them into the PUT as additional branches.

Answer to RQ3: The design choices can contribute to the effectiveness of Kat, which improve the the F1-score from 96.43% ~ 129.17%.

5.2.4 *Results of RQ4 - Generalizability of Kat.*

Table 5 shows the total number of bugs detected by Kat and the other two baselines (i.e., Catcher and EvoSuite) with 25 executions of experiments in the open-world dataset. Overall, Kat exhibits remarkable performance in bug detection, surpassing both Catcher and EvoSuite. With a total of 265 unique bugs identified across all 25 executions, Kat showcases superior effectiveness and remarkable generalizability in the large-scale open-world dataset. In more specific terms, Kat outperforms Catcher by discovering 22 additional bugs (265 − 243) and surpasses EvoSuite by detecting 99 additional bugs (265 − 166). These findings further emphasize the significant advantage and robustness of Kat over the baselines.

Table 6: Overlap Between KAT and Catcher Regarding the Unique Bugs Detected Across 25 Executions.

Porject ID	KAT ∩ Catcher	KAT \ Catcher	Catcher \ KAT
BCEL	8	-	1
CLI	-	-	-
CODEC	9	3	-
COLL	10	5	-
COMP	33	-	1
LANG	28	-	4
MATH	11	5	1
EASY	16	2	-
GSON	16	-	-
HAMC	-	-	-
JACK	6	1	-
JAVS	4	2	-
JCOM	2	-	-
JFCH	27	1	-
JODA	23	-	2
JOPT	3	2	-
NATT	4	1	-
NEO4	17	4	-
SHIRO	7	5	-
XJOB	10	-	-
XTEX	-	-	-
Total	234	31	9

Table 6 presents a comprehensive summary of bug detection results for both KAT and Catcher. The table highlights the number of bugs detected by both approaches, as well as the distinct set of bugs identified exclusively by each tool. The analysis reveals that 234 unique bugs were detected by both KAT and Catcher, indicating a considerable overlap in their bug detection capabilities. Additionally, KAT identified 31 unique bugs that were not detected by Catcher, demonstrating its ability to uncover specific bugs that might have been overlooked by the other tool. Conversely, Catcher discovered nine unique bugs that were not detected by KAT, showcasing its effectiveness in identifying a subset of bugs missed by the alternative approach. Compared with EvoSuite, both KAT and Catcher take absolute advantages, because both methods utilize external exception information from documentation/source code to identify bugs. Catcher first detects potential bugs and then optimizes searching objectives by combining four coverage (i.e., branch, line, input, and output). Differently, KAT transforms API exception handling knowledge into program constraints and instruments them into PUT as additional branches. As a result, it can guide search algorithms to cover more branches and detect more bugs. For example, Fig. 1 is a bug from the class `SignatureReader` in EASY project that can only be detected by KAT. The bug happens in the method `acceptType` in line 57 of PUT, because it can trigger the exception of `String.charAt(index)` in line 63 of the method `parseType`. This case shows a bug without test goals, and Catcher cannot easily synthesize a test case to trigger the exception because there is no guidance such as branch distance to synthesize the two specific Java objects.

When considering the bugs exclusively detected by Catcher, our observations indicate that these cases primarily fall into the category of *B4* bugs, which are triggered by *T4*. The reason behind Catcher's success in detecting these bugs lies in its utilization of various coverage techniques, allowing it to address the challenge posed by such transformations effectively. On the other hand, KAT handles this transformation by introducing try/catch as an additional branch. However, covering this branch within a reasonable branch distance can pose challenges for search algorithms. Nonetheless, Catcher's approach proves effective in overcoming these challenges.

Answer to RQ4: KAT can detect 22 more and 99 more bugs than Catcher and EvoSuite in the open-world dataset, which indicates better generalizability.

6 DISCUSSION

6.1 Effectiveness and Efficiency of KAT

6.1.1 Effectiveness. With extensive experiments, our KAT can outperform static analysis and other dynamic SBST approaches in finding bugs. Compared to static analysis (i.e., KAT_{static}), KAT demonstrates significant improvements in addressing false positive API-related bugs and achieving automated bug detection. By incorporating search strategies in Phase III (as depicted in Fig. 2), KAT effectively generates test cases to verify the existence of bugs reported by static analysis. This capability leads to a considerable reduction of false positive instances, for example, a decrease of 61 cases (72 − 11). Furthermore, when compared to existing SBST approaches (i.e., Catcher and EvoSuite/$\text{KAT}_{dynamic}$), KAT exhibits the ability to discover a greater number of bugs within a limited search time. Specifically, KAT can detect 22 more and 99 more bugs than Catcher and EvoSuite, respectively. This advantage stems from the utilization of an API exception handling knowledge graph, which enables KAT to identify potential bugs and transform the corresponding violation constraints into instrumented branches. Consequently, KAT effectively improves branch coverage and enhances the bug detection process. While the advancements achieved by KAT may be considered relatively modest compared to the state-of-the-art SBST approaches, it is important to recognize that even modest progress can have significant implications. Such advancements contribute to the cumulative progress within the SBST field and lay the groundwork for further developments and improvements.

6.1.2 Efficiency. When optimizing searching strategies, SBST requires setting a time budget. KAT allocates a maximum of 200 seconds for each class under test, which is the same time set in Catcher. For the same class under test, all of Catcher, EvoSuite and KAT are allocated the same time (i.e., 200s), but KAT can find more bugs.

Note, for KAT, we do not consider the time consumption of constructing the API exception handling knowledge graph, static bug detection and instrumentation. First, the construction of the knowledge graph occurs offline, meaning it is not performed in real time during the bug detection. As a result, any time consumed during this process does not impact the overall efficiency of KAT. Second, regarding the static bug detection and instrumentation for each class, it is crucial to note that these operations are highly efficient and completed within a very short duration, typically within one second. When considering the overall time budget allocated for the evaluation or execution of KAT, which is around 200 seconds, the negligible time consumed by static bug detection and instrumentation becomes inconsequential. Therefore, it is appropriate to disregard this time when comparing the performance of KAT within the allocated time budget.

6.2 Threats to Validity

6.2.1 Internal Validity. The internal threats to validity refer to the quality of the API exception handling knowledge graph, which plays a pivotal role in our bug detection approach. To reduce personal bias when evaluating the knowledge graph quality, we employed a method that involved two authors independently annotating the data instances. To assess the agreement between the annotators and ensure reliability, we computed Cohen's kappa coefficient, which indicates a substantial or almost perfect agreement between the two annotators.

6.2.2 External Validity. The external threats to validity arise from two aspects: limited knowledge from the API documentation and the small-scale close-world dataset. First, considering the quality of API knowledge is critical to the performance of KAT, we prefer selecting the high-quality Java official documentation to construct a knowledge graph inspired by the previous studies [36, 50]. Although the official documentation provides a limited number of APIs, we design our API exception handling knowledge graph and open information extraction pipeline to be adaptable and extendable, allowing us to incorporate additional knowledge from diverse sources. By continuously expanding our knowledge graph, we aim to enhance the comprehensiveness of the information available. Second, the small scale of our close-world dataset can be attributed to two key factors: 1) our research primarily focuses on a specific subset of API knowledge (i.e., Java SE&JDK API knowledge). Consequently, the number of bugs directly associated with these APIs is naturally limited. 2) Our adherence to the rigorous strategy employed by Defects4j [32] has led us to conduct an exhaustive search for high-quality bugs. This meticulous approach is time-consuming and demands significant effort and resources. Nevertheless, we remain dedicated to ongoing efforts aimed at expanding and enriching our close-world dataset. This commitment entails actively collecting more bugs in the future. Hence, we aim to broaden the scope of our dataset, enhance its representativeness, and strengthen the validity and reliability of our research findings.

7 RELATED WORK

7.1 Search-based Software Testing

Search-based software testing (SBST) is a powerful technique for automating test generation, showing promising results [2, 27, 38, 40]. SBST relies on specific heuristics for different coverage criteria, guiding the generation of test cases to achieve higher coverage.

Initially, the single-target approach aimed to satisfy one coverage target (e.g., one branch) at a time through multiple search iterations (e.g., genetic algorithms) [12, 17, 18, 54]. However, multi-target approaches have emerged as superior, using many-objective search techniques to achieve higher code coverage [31, 41, 45, 56, 71].

Catcher [33], related to our work, employs a two-step approach. It first uses exception propagation to identify potential API misuses in the program under test statically (PUT), narrowing the search space. Then, it uses coverage-based heuristics to guide a many-objective search to cover the identified API call sites and expose propagated exceptions, primarily focusing on API-related bugs.

In contrast, our approach, called KAT, goes beyond static detection of API misuses. We leverage API exception-handling knowledge gathered from API usage specifications to create test oracles, enhancing KAT's effectiveness by incorporating domain-specific API exception handling into the process.

7.2 Test Oracle Automation

To effectively detect real-world software faults, we must improve generated tests with accurate test oracles [4]. These oracles significantly impact testing quality and software system reliability [22, 28]. Constructing test oracles automatically is a challenging task, often considered a bottleneck in automated software testing [8]. Recent efforts have approached this problem from three angles: implicit oracles, derived oracles, and specified oracles [7, 13, 55, 67].

Implicit oracles rely on domain expert knowledge to determine test pass/fail. For example, segment faults are typical errors, and frame rate drops in game applications may indicate performance issues [8]. Derived oracles use existing information to distinguish correct from incorrect software behavior. Techniques include metamorphic relations, version comparisons, and metadata analysis [7, 13, 55, 67]. Specified oracles require formal specifications or contracts to define expected behavior [16, 48] or contracts [14, 64]. The quality and completeness of these specifications are critical [8, 24].

In contrast, our approach focuses on converting exception handling documentation knowledge into test oracles.

8 CONCLUSION

Search-based software testing (SBST) has proved its effectiveness in generating test cases to achieve its defined test goals such as branch and data-dependency coverage. However, to detect more program faults in an effective way, pre-defined goals can hardly be adaptive in diversified projects. We propose KAT, a knowledge extraction-based approach to generate on-demand assertions in the PUT, validating the internal execution states when interacting with API frameworks. With evaluations on both close-world and open-world datasets, we illustrate KAT performs better than other baselines (i.e., Catcher and EvoSuite) in finding more bugs violating API specifications. Additionally, KAT also shows good generalizability in a large-scale dataset. In the future, we will extend our work in the following two aspects. First, we will extend the knowledge graph to include more knowledge to guide test case generation. Second, we will further develop KAT and apply the idea to program repair.

9 DATA AVAILABILITY

All Experimental data are available at our GitHub repository.

ACKNOWLEDGEMENTS

This research was partially supported by the Fundamental Research Funds for the Central Universities (226-2022-00064) and Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14206921 of the General Research Fund).

REFERENCES

[1] KK Aggarwal, Yogesh Singh, Arvinder Kaur, and OP Sangwan. 2004. A neural net based approach to test oracle. *ACM SIGSOFT Software Engineering Notes* 29, 3 (2004), 1–6.

[2] Shaukat Ali, Lionel C Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. 2009. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering* 36, 6 (2009), 742–762.

[3] M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. 2017. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 263–272.

[4] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, et al. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 86, 8 (2013), 1978–2001.

[5] Andrea Arcuri, José Campos, and Gordon Fraser. 2016. Unit test generation during software development: Evosuite plugins for maven, intellij and jenkins. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 401–408.

[6] Andrea Arcuri and Gordon Fraser. 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering* 18, 3 (2013), 594–623.

[7] Jon Ayerdi, Pablo Valle, Sergio Segura, Aitor Arrieta, Goiuria Sagardui, and Maite Arratibel. 2022. Performance-driven metamorphic testing of cyber-physical systems. *IEEE Transactions on Reliability* (2022).

[8] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2014), 507–525.

[9] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 242–253.

[10] Javassist: Java bytecode engineering toolkit. [n. d.]. https://www.javassist.org/. ([n. d.]).

[11] José Campos, Andrea Arcuri, Gordon Fraser, and Rui Abreu. 2014. Continuous test generation: Enhancing continuous integration with automated test generation. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 55–66.

[12] Kai H Chang, JAMES H CROSS II, W Homer Carlisle, and Shih-Sung Liao. 1996. A performance evaluation of heuristics-based test case generation methods for software branch coverage. *International Journal of Software Engineering and Knowledge Engineering* 6, 04 (1996), 585–608.

[13] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 2020. Metamorphic testing: a new approach for generating next test cases. *arXiv preprint arXiv:2002.12543* (2020).

[14] Yoonsik Cheon and Gary T Leavens. 2002. A simple and practical approach to unit testing: The JML and JUnit way. In *ECOOP 2002—Object-Oriented Programming: 16th European Conference Málaga, Spain, June 10–14, 2002 Proceedings 16*. Springer, 231–255.

[15] Myra B Cohen. 2019. The maturation of search-based software testing: successes and challenges. In *2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 13–14.

[16] Xin Feng, David Lorge Parnas, TH Tse, and Tony O'Callaghan. 2011. A comparison of tabular expression-based testing strategies. *IEEE Transactions on Software Engineering* 37, 5 (2011), 616–634.

[17] Roger Ferguson and Bogdan Korel. 1996. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 5, 1 (1996), 63–86.

[18] Gordon Fraser and Andrea Arcuri. 2011. Evolutionary generation of whole test suites. In *2011 11th International Conference on Quality Software*. IEEE, 31–40.

[19] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[20] Gordon Fraser and Andrea Arcuri. 2012. Whole test suite generation. *IEEE Transactions on Software Engineering* 39, 2 (2012), 276–291.

[21] Gordon Fraser and Andrea Arcuri. 2014. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 2 (2014), 1–42.

[22] Gordon Fraser and Andrea Arcuri. 2015. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empirical software engineering* 20, 3 (2015), 611–639.

[23] Gordon Fraser and Andreas Zeller. 2011. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering* 38, 2 (2011), 278–292.

[24] Marie-Claude Gaudel. 2001. Testing from formal specifications, a generic approach. In *Reliable SoftwareTechnologies—Ada-Europe 2001: 6th Ada-Europe International Conference on Reliable Software Technologies Leuven, Belgium, May 14–18, 2001 Proceedings*. Springer, 35–48.

[25] Gensim. [n. d.]. https://pypi.org/project/gensim/. ([n. d.]).

[26] Alberto Goffi, Alessandra Gorla, Michael D Ernst, and Mauro Pezzè. 2016. Automatic generation of oracles for exceptional behaviors. In *Proceedings of the 25th international symposium on software testing and analysis*. 213–224.

[27] Mark Harman, Yue Jia, and Yuanyuan Zhang. 2015. Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–12.

[28] Mark Harman, Sung Gon Kim, Kiran Lakhotia, Phil McMinn, and Shin Yoo. 2010. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 182–191.

[29] Johannes Henkel, Christoph Reichenbach, and Amer Diwan. 2007. Discovering documentation for Java container classes. *IEEE Transactions on Software Engineering* 33, 8 (2007), 526–543.

[30] Standard Edition & Java Development Kit Version 17 API Specification Java Platform. [n. d.]. https://docs.oracle.com/en/java/javase/17/docs/api. ([n. d.]).

[31] Yue Jia. 2015. Hyperheuristic search for sbst. In *2015 IEEE/ACM 8th International Workshop on Search-Based Software Testing*. IEEE, 15–16.

[32] 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 2014 International Symposium on Software Testing and Analysis*. 437–440.

[33] Maria Kechagia, Xavier Devroey, Annibale Panichella, Georgios Gousios, and Arie van Deursen. 2019. Effective and efficient API misuse detection via exception propagation and search-based testing. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*. 192–203.

[34] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. 2015. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*. IEEE, 560–564.

[35] J Richard Landis and Gary G Koch. 1977. An application of hierarchical kappa-type statistics in the assessment of majority agreement among multiple observers. *Biometrics* (1977), 363–374.

[36] Hongwei Li, Sirui Li, Jiamou Sun, Zhenchang Xing, Xin Peng, Mingwei Liu, and Xuejiao Zhao. 2018. Improving api caveats accessibility by mining api caveats knowledge graph. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 183–193.

[37] Nan Li and Jeff Offutt. 2016. Test oracle strategies for model-based testing. *IEEE Transactions on Software Engineering* 43, 4 (2016), 372–395.

[38] 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 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1068–1080.

[39] Mingwei Liu, Xin Peng, Andrian Marcus, Zhenchang Xing, Wenkai Xie, Shuangshuang Xing, and Yang Liu. 2019. Generating query-specific class API summaries. In *Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 120–130.

[40] Phil McMinn. 2011. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 153–163.

[41] Mahshid Helali Moghadam, Markus Borg, and Seyed Jalaleddin Mousavirad. 2021. Deeper at the sbst 2021 tool competition: ADAS testing using multi-objective search. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 40–41.

[42] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. 2012. What should developers be aware of? An empirical study on the directives of API documentation. *Empirical Software Engineering* 17, 6 (2012), 703–737.

[43] neuralcoref. [n. d.]. https://spacy.io/universe/project/neuralcoref. ([n. d.]).

[44] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.

[45] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2015. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE, 1–10.

[46] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2017. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering* 44, 2 (2017), 122–158.

[47] Anjana Perera, Aldeida Aleti, Marcel Böhme, and Burak Turhan. 2020. Defect prediction guided search-based software testing. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 448–460.

[48] Dennis K Peters and David Lorge Parnas. 2002. Requirements-based monitors for real-time systems. *IEEE Transactions on Software Engineering* 28, 2 (2002), 146–158.

[49] Z3 Prover. [n. d.]. https://github.com/Z3Prover/z3. ([n. d.]).

[50] Xiaoxue Ren, Zhenchang Xing, Xin Xia, Guoqiang Li, and Jianling Sun. 2019. Discovering, explaining and summarizing controversial discussions in community q&a sites. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 151–162.

[51] Xiaoxue Ren, Xinyuan Ye, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Jianling Sun. 2020. API-misuse detection driven by fine-grained API-constraint knowledge graph. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 461–472.

[52] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. 2016. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability* 26, 5 (2016), 366–401.

[53] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. 2017. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering* 22, 2 (2017), 852–893.

[54] Simone Scalabrino, Giovanni Grano, Dario Di Nucci, Rocco Oliveto, and Andrea De Lucia. 2016. Search-based testing of procedural programs: Iterative single-target or multi-target approach?. In *International symposium on search based software engineering*. Springer, 64–79.

[55] Sergio Segura, Gordon Fraser, Ana B Sanchez, and Antonio Ruiz-Cortés. 2016. A survey on metamorphic testing. *IEEE Transactions on software engineering* 42, 9 (2016), 805–824.

[56] Mohammad Mehdi Dejam Shahabi, S Parsa Badiei, S Ehsan Beheshtian, Reza Akbari, and S Mohammad Reza Moosavi. 2017. EVOTLBO: A TLBO based Method for Automatic Test Data Generation in EvoSuite. *International Journal of Advanced Computer Science and Applications* 8, 6 (2017).

[57] Seyed Reza Shahamiri, Wan Wan-Kadir, Suhaimi Ibrahim, and Siti Zaiton Mohd Hashim. 2012. Artificial neural networks as multi-networks automated test oracle. *Automated Software Engineering* 19, 3 (2012), 303–334.

[58] Ravindra Singh and Naurang Singh Mangat. 2013. *Elements of survey sampling*. Vol. 15. Springer Science & Business Media.

[59] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo A. Maia. 2018. Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J. In *Proceedings of SANER*.

[60] Soot. [n. d.]. http://soot-oss.github.io/soot/. ([n. d.]).

[61] Spacy. [n. d.]. https://spacy.io/. ([n. d.]).

[62] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Zhendong Su. 2011. Synthesizing method sequences for high-coverage testing. *ACM SIGPLAN Notices* 46, 10 (2011), 189–206.

[63] Raja Vallee-Rai and Laurie J Hendren. 1998. Jimple: Simplifying Java bytecode for analyses and transformations. (1998).

[64] Yi Wei, Carlo A Furia, Nikolay Kazmin, and Bertrand Meyer. 2011. Inferring better contracts. In *Proceedings of the 33rd International Conference on Software Engineering*. 191–200.

[65] Word2vec. [n. d.]. https://code.google.com/archive/p/word2vec/. ([n. d.]).

[66] Tao Xie. 2006. Augmenting automatically generated unit-test suites with regression oracle checking. In *ECOOP 2006–Object-Oriented Programming: 20th European Conference, Nantes, France, July 3-7, 2006. Proceedings 20*. Springer, 380–403.

[67] Xiaoyuan Xie, Joshua WK Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. 2011. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software* 84, 4 (2011), 544–558.

[68] Juan Zhai, Jianjun Huang, Shiqing Ma, Xiangyu Zhang, Lin Tan, Jianhua Zhao, and Feng Qin. 2016. Automatic model generation from documentation for Java API functions. In *Proceedings of the 38th International Conference on Software Engineering*. 380–391.

[69] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. 2009. Inferring resource specifications from natural language API documentation. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 307–318.

[70] Yu Zhou, Changzhi Wang, Xin Yan, Taolue Chen, Sebastiano Panichella, and Harald Gall. 2018. Automatic detection and repair recommendation of directive defects in Java API documentation. *IEEE Transactions on Software Engineering* 46, 9 (2018), 1004–1023.

[71] Ziming Zhu, Xiong Xu, and Li Jiao. 2017. Improved evolutionary generation of test data for multiple paths in search-based software testing. In *2017 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 612–620.

Received 2023-02-02; accepted 2023-07-27