Recommending StackOverflow Posts for Fixing Runtime Exceptions using Failure Scenario Matching

Anonymous Author(s)

ABSTRACT

Using online Q&A forums, such as StackOverflow (SO), for guidance to resolve program bugs, among other development issues, is commonplace in modern software development practice. Runtime exceptions (RE) is one such important class of bugs that is actively discussed on SO. In this work we present a technique and prototype tool called MAESTRO that can automatically recommend an SO post that is most relevant to a given Java RE in a developer's code. MAESTRO compares the exception-generating program scenario in the developer's code with that discussed in an SO post and returns the post with the closest match. To extract and compare the exception scenario effectively, MAESTRO first uses the answer code snippets in a post to implicate a subset of lines in the post's question code snippet as responsible for the exception and then compares these lines with the developer's code in terms of their respective Abstract Program Graph (APG) representations. The APG is a simplified and abstracted derivative of an abstract syntax tree, proposed in this work, that allows an effective comparison of the functionality embodied in the high-level program structure, while discarding many of the low-level syntactic or semantic differences. We evaluate MAESTRO on a benchmark of 78 instances of Java REs extracted from the top 500 Java projects on GitHub and show that MAESTRO can return either a highly relevant or somewhat relevant SO post corresponding to the exception instance in 71% of the cases, compared to relevant posts returned in only 8% - 44% instances, by four competitor tools based on state-of-the-art techniques. We also conduct a user experience study of MAESTRO with 10 Java developers, where the participants judge MAESTRO reporting a highly relevant or somewhat relevant post in 80% of the instances. In some cases the post is judged to be even better than the one manually found by the participant.

1 INTRODUCTION

Software developers regularly refer to online Q&A forums for a wide variety of development tasks, from system design and configuration, to code completion, to software debugging and patching [2, 4, 32, 44]. StackOverflow (SO), the most popular such forum, with over 28 million answers to 18 million questions, sees nearly 260 million views per month [38]. Software debugging and patching is a resource-intensive development activity, consuming up to 50% of developers' time [41]. In particular, runtime exceptions (REs) are an important class of bugs that have been recognized as having a severe impact on system availability and crashes [47]. Realizing their importance, researchers have proposed automated debugging, repair, and recovery techniques specifically addressing runtime errors in general and REs in particular [7, 8, 10, 20, 21, 36, 45]. In fact, of the nearly 2.65 million posts on SO tagged with `Java` and/or `Android` nearly 193,186 [1] – a remarkable 7% – are related just to

Java REs, showing that developers are actively discussing Java REs on SO, evidently to resolve such errors in their own code.

In this paper we present a technique and prototype tool MAESTRO (**M**ine and **A**nalyz**E ST**ackoverflow to fix **R**untime excepti**O**ns) that can automatically find an SO post that is *most relevant* to a given RE in a developer's (Java) code. We define the most relevant post (or posts, since there could be several) as one discussing the same runtime scenario, exciting the same type of RE as the developer's code. Such a tool can save the time it would take the developer to understand the high-level scenario producing the exception, create a search query based on it, search SO with the query and manually browse the search results, one discussion post at a time, to identify the most relevant post. These steps may need to be repeated several times till an acceptable post is found. Once found the user could potentially use one of the suggested answers to fix her bug. This fixing could be done manually or assisted by a tool like EXAMPLESTACK [46].

One solution to the above problem is to use SO post recommendation techniques like PROMPTER [28–30], which search for a post most relevant to the user's code context, i.e., the *function* the user's code is trying to implement. However, as shown in Section 4, these techniques do not work for our problem. The reason is that what we seek is *not* a post discussing the overall function the user's code was implementing but rather one addressing the specific sequence of program state manipulations that raised the exception. The overall function of the method within which these manipulations occur is somewhat irrelevant. Another idea would be to use traditional code clone detection techniques, such as [6, 12, 13, 35], or code-to-code search techniques, such as [14, 22] to check correspondence between the user's code and the question code snippet in a given SO post. However, this approach would not work either (see evaluation in Section 4), in part because of the above reason – it is not apparent what portions to try to match between the exception-throwing developer code and the SO code snippet. Further, even the relevant lines of the SO post snippet, if identified, could differ significantly from their counterparts in the developer code in terms of not only variable names (which code clone detectors easily handle) but also data types and program constructs (e.g., a `while` vs. a `for` loop), while instantiating the same core exception-causing scenario. Thus, this problem seems outside the realm of traditional code clone detection or code search, since the matching criterion is the cause of the exception, rather than the function of the containing method.

Our approach is designed around three key insights. First, in most SO discussion posts the question includes a code snippet exemplifying the scenario being discussed. In particular, for posts addressing REs, the question code snippet naturally includes a *structured* description of the exception-raising scenario. Thus, our approach exclusively uses this question code snippet and compares it to the exception-throwing user code to decide the relevancy of the post. Second, the question code snippet may include code to

---

```
211 public void dropColumn(String databaseName, String tableName, String columnName)
212 {
213     verifyCanDropColumn(this, databaseName, tableName, columnName);
214     org.apache.hadoop.hive.metastore.api.Table table = delegate.getTable(databaseName, tableName)
215         .orElseThrow(() -> new TableNotFoundException(new SchemaTableName(databaseName, tableName)));
216     for (FieldSchema fieldSchema : table.getSd().getCols()) {    ← exception thrown here
217         if (fieldSchema.getName().equals(columnName)) {
218             table.getSd().getCols().remove(fieldSchema);}
219         }
220     }
221     alterTable(databaseName, tableName, table);
222 }
```

(a) Buggy source code (BridgingHiveMetastore.java)

```
1 public static void main(String[] args) {
2     User user = new User("user1","user1",1l);
3     User user1 = new User("user2","user2",2l);
4     User user2 = new User("user3","user3",3l);
5
6     List<User> list = new ArrayList<User>();
7     list.add(user);
8     list.add(user1);
9     list.add(user2);
10
11     for (User user3 : list)  {
12         System.out.println(user3.getName());
13         if (user3.getName().equals("user1"))  {
14             list.remove(user3);
15         }
16     }
17 }                                             Q
```

(b) StackOverflow post #21973342 question code snippet

```
1 Iterator<User> it = list.iterator();
2 while (it.hasNext())  {
3     User user = it.next();
4     if (user.getName().equals("user1"))  {
5         it.remove();
6     }
7 }                                             A
```

(c) StackOverflow post #21973342 answer code snippet

```
1 Collection<T> myCollection; ///assume it is initialized and filled
2 for (Iterator<?> index = myCollection.iterator(); index.hasNext();) {
3     Object item = index.next();
4     myCollection.remove(item);
5 }                                             Q
```

(d) StackOverflow post #2054189 question code snippet

Figure 1: Example from Presto (https://github.com/prestodb/presto) throwing ConcurrentModificationException (issue #9733)

make the snippet functionally or syntactically complete, but otherwise irrelevant to the exception scenario. However, the answers in the post also include code snippets, suggesting solutions to the discussed problems. In our case these answer code snippets often suggest patches for the RE and often more directly address the failure producing lines. Thus, we use the answer code snippets to identify the lines in the question code snippet relevant to the exception scenario and discard the rest. We term this *Failure Scenario Localization (FSL)*. Third, as mentioned above, the failure producing lines in the developer code could have substantial syntactic differences from those the SO code snippet. To facilitate a meaningful comparison we develop a representation called an *Abstract Program Graph (APG)*. Conceptually, an APG is a simplified, abstracted, and (partially) canonicalized derivative of the Abstract Syntax Tree (AST) for a piece of code. Thus, we compare the developer code and question code snippet (after performing FSL) in terms of their respective APGs. We also use the APG representation when aligning answer code snippets in a post with the question code snippet for the purpose of Failure Scenario Localization.

We develop a technique and prototype tool, MAESTRO based on the above insights and conduct an internal evaluation on a benchmark of 78 instances of Java REs, spanning 19 exception types, extracted from the repositories of the top 500 Java projects on GitHub. The evaluation shows that MAESTRO returns either a highly relevant or somewhat relevant SO post corresponding to the exception instance for 71% of the instances, compared to relevant posts returned in only 8% - 44% instances, by four competitor tools based on state-of-the-art techniques. Further, a comparison with three different baseline variations of MAESTRO shows that each of the key design features of MAESTRO are essential to its overall

performance. We also conduct a user experience study of MAESTRO with 10 Java developers, where the participants judge MAESTRO as reporting a highly relevant or somewhat relevant post in 80% of the instances. In some cases the post is judged to be even better than the one manually found by the participant.

This main contributions of this paper are as follows:

- **Technique:** An automated technique, MAESTRO for recommending relevant StackOverflow posts to Java developers which could assist them in diagnosing and fixing REs in their code.
- **Tool:** A prototype implementation of the MAESTRO technique along with four different baseline variations of it, to evaluate its design features.
- **Evaluation:** An evaluation of MAESTRO, its three baseline variants, and four competitor tools, on a benchmark of 78 RE instances extracted from the top 500 Java projects on GitHub.
- **User experience study:** A user study with 10 Java developers to qualitatively evaluate the performance of MAESTRO.

2 ILLUSTRATIVE EXAMPLE

In this section, we illustrate our technique using the example shown in Figure 1. Consider the developer's buggy source code given in Figure 1a extracted from the BridgingHiveMetastore.java file of the Presto project [31], a popular distributed SQL query engine. The buggy source code throws "ConcurrentModifcationException", a common runtime exception (RE) that may be thrown by methods detecting concurrent modification on an object, when such modification is not permitted. In this example, the exception is thrown because the Collection object returned by "table.getSd().getCols()" is modified at line 218 while being iterated over at line 216.

(a) APG_B: APG of buggy code shown in Figure 1a

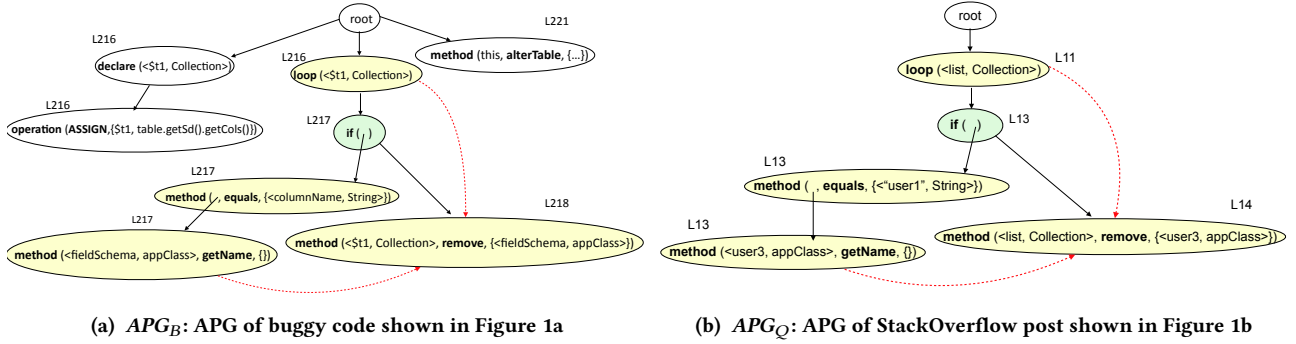(b) APG_Q: APG of StackOverflow post shown in Figure 1b

Figure 2: APGs of buggy code and StackOverflow post

In order to resolve the encountered RE, the developer may refer to StackOverflow (SO) to find a post discussing the exception in the same context. To facilitate this, Maestro collects SO posts discussing ConcurrentModificationException and compares code snippets in the question section of each post with the developer's buggy code. Figures 1b and 1d shows question code snippets extracted from two SO posts discussing this RE.

Let us consider the snippet shown in Figure 1b. Comparing the snippets from Figure 1a and Figure 1b directly is challenging since their function is different. Figure 1a's function is to drop a database column, while Figure 1b's function is to delete a user. The code snippets also have lines that are irrelevant to the exception scenario. In particular, only lines 216-218 from Figure 1a and lines 11, 13 and 14 from Figure 1b are pertinent to the RE being thrown. To address this challenge, our approach is to extract the failure scenario on the SO side and then check if it also instantiated in the buggy code.

Maestro automatically localizes the failure scenario in the question snippet (Figure 1b) by leveraging the answer code snippets, such as the one shown in Figure 1c. The answer snippet in Figure 1c suggests a way for fixing the ConcurrentModificationException by using "Iterator" to remove the user object. The answer snippet clearly points to lines constituting the exception scenario in the question snippet. However, it is challenging to establish this correspondence since the question and answer snippets may include syntactic differences. In this case, the question iterates using a for loop over the List object, while the answer iterates using a while loop over an Iterator object. To address this challenge, Maestro abstracts and canonicalizes the snippets in the Abstract Program Graph (APG), and then structurally aligns the two APGs to find the corresponding lines automatically. The color-coding in Figure 1b and Figure 1c show the corresponding lines obtained from the APG alignment. Figure 2b shows the APG representing the matched lines extracted from the question snippet. This APG forms the Exception Scenario Pattern (ESP) for ConcurrentModificationException as per this post. Let us call this pattern APG_Q.

To find if this post matches the exception scenario in the developer's code, Maestro first represents the buggy code in APG form (Figure 2a), let us call it APG_B. Now the next step is to compare APG_Q to APG_B. For this, Maestro aligns the two APGs, by computing a 1-to-1 node mapping, and computes a similarity score. The similarity score calculation is comprised of analyzing the equivalence of programming "constructs" used in a matched pair and

the equivalence between the involved data "types". For both construct equivalence and type equivalence, Maestro applies some level of abstraction depending on program constructs. For example, Maestro identifies user-defined classes as "appClass". Hence, The methods remove in lines 218 and 14 of Figure 1a and Figure 1b, respectively, are equivalent at both type and construct level since both can be abstracted as Collection.remove(appClass). The results for APG alignment is shown in Figure 2a and Figure 2b. The nodes with type similarity are shown in yellow and the nodes with construct similarity are shown in green. The similarity score is calculated as a weighted sum of the matched nodes with construct similarity and type similarity. Thus, the similarity score for this post (Figure 1b) is 9, assuming the weights for construct and type similarity to be 1 and 2, respectively, since there are four type matches and one construct match. In addition to types similarity and construct similarity, Maestro measures the similarity between the uses of variables appearing in the equivalent nodes of APG_Q and APG_B. For example, both variables $\$t1$ and $list$ in the matched loop nodes from Figure 2a and Figure 2b, respectively, are used in the matched remove nodes as well. The var-use relationships are shown with red arrows in Figure 2a and Figure 2b.

Figure 1d shows a code snippet from another StackOverflow post. Assuming the same weights as above, the similarity score for this is 2, since there is no data matches and there are only two construct matches for loop and remove at lines 2 and 5, respectively. Therefore, Maestro chooses the first post (Figure 1b) with score 9 as the most relevant post for the developer's buggy code, over the second post (Figure 1d) with score 2. This result is correct since the first post very closely mimics the exception scenario of the developer than the second post.

3 APPROACH

The goal of our approach is to automatically find the most relevant StackOverflow (SO) post for the runtime exception (RE) encountered by the developer. Our key insight in finding the most relevant post is that if the developer's code exhibits the *same exception scenario* as presented in the code snippet provided in the SO question, then it is very likely that the post will contain answers to fix the exception in the very context of the developer.

Finding a relevant post based on exception scenario similarity is complicated by several challenges. The first challenge is in capturing the exception scenario in the SO post. Code snippets in the
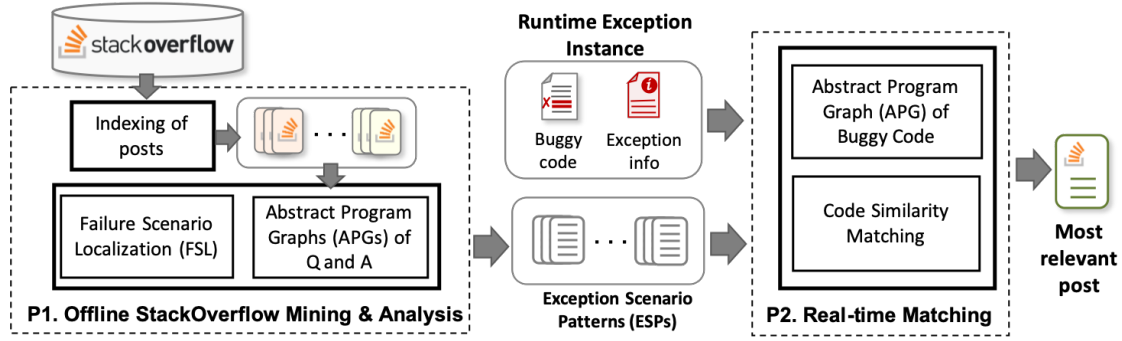
Figure 3: Overview of the approach

question section of SO posts can be arbitrarily large, while the statements capturing the *exception scenario*, i.e., the lines that are responsible for exciting the RE are typically small. For example, the question code in Figure 1b is of 17 lines, while the lines capturing the scenario of `ConcurrentModificationException` are only three (lines 11, 12, and 13). This challenge requires that our solution correctly identifies the lines relevant to the RE under consideration. The second challenge is that developer's code and SO code may have significant syntactic and semantic differences, such as in the structure of the code (e.g., interleaving of the exception scenario with added/removed lines), syntactic constructs (e.g., `for` vs. `while`), and identifier names. This challenge requires that our solution tolerates the various syntactic and semantic differences between the developer's code and SO code while establishing similarity.

Two insights into these challenges guide the design of our approach. The first insight is that it is possible to automatically identify the potentially relevant lines contributing to the exception scenario in SO question code snippets by referring to the answers of that question. Code snippets in SO answers tend to be more pointed in discussing the problem in the question code while suggesting ways of fixing it. Leveraging this observation, we designed a *Failure Scenario Localization (FSL)* technique that overlays answer code on question code to identify the relevant lines (Section 3.4). The second insight is that it is possible to abstract out the syntactic and semantic differences allowing for a normalized structural code comparison between developer's code and SO code. For this, we designed the *Abstract Program Graph (APG)* that captures the structural and data relationships between program statements, while abstracting out low-level syntactic details (Section 3.2). Further, our *matching* function calculates a weighted similarity score(Section 3.5).

Our approach can be roughly broken down into two distinct phases, *offline mining and analysis* of SO posts to index them by Exception Scenario Patterns (ESPs) and *real-time matching* of the mined SO posts and developer's buggy code to find the most relevant post. These are shown in Figure 3. The first phase indexes SO posts by filtering in only the RE related posts. For each of the indexed post, it then performs FSL to identify the lines of code relevant to the RE type being discussed in the post by using answer pointers. This involves representing the SO post's question and answer code snippets in APG formats and then structurally aligning (Section 3.3) them to find the overlapping nodes. The APG of the question is then cleaned by removing the non-matching nodes and saved in persistent storage for use in the second phase. We

refer to the cleaned APG as the *ESP*. The second phase takes three inputs: RE information (name and failing line available from the stacktrace), buggy source code containing the failing line, and the ESPs extracted from the first phase. The second phase begins by representing the buggy code in the APG format. It then performs a *code similarity matching* between the APG of the buggy code and the ESPs grouped under the given RE type. The SO post mapping to the ESP receiving the highest similarity score is then returned as the output. We now explain the different parts of our approach.

3.1 Indexing of SO Posts

In this step we index SO posts by retaining only the posts that are related to fixing REs, and discard the rest. Our filtering criteria was that the post has: (1) at least one answer, (2) "java" or "android" tags, (3) RE type in the title, and (4) at least one parseable code snippet in the question. The filtered SO posts were then grouped by RE types.

3.2 Abstract Program Graph (APG)

The APG is a model capturing the structural and data relationships among the program statements of a given code snippet. Compared to traditional data structures, such as AST and program dependence graph, the APG focuses on simplifying, abstracting, and canonicalizing the low-level syntactic details. This is a more suitable data structure for our purpose, since it allows our approach to compare code despite the frequent differences in their syntax and semantics.

3.2.1 APG Definition. Formally, we define the APG as a directed graph of the form $\langle N, R_s, R_d, M \rangle$. Here $n \in N$ is a node in the graph that corresponds to a statement in the code. Figure 4 gives the grammar used for defining the nodes in the APG. A node can be of type `method`, `loop`, `variable`, etc. Node type `root` denotes the starting point of the APG. Different node types further have different embodiments, such as `method` node is defined by three features, "`caller`", which can be another node type (e.g., variable or method), "`name`", which is the method name, and a list of arguments denoted by "`args`", which can be a list of other nodes (e.g., variable or literal). For each node type, its natural-language embodiments are denoted on the right-end after the "//" marker. $R_s \subseteq N \times N$ is a set of directed edges, such that for each pair of nodes $\langle n_1, n_2 \rangle \in R_s$, there exists a structural relationship between n_1 and n_2. Similarly, R_d is a set of directed edges capturing the data relationship between n_1 and n_2. M is a function $M : R_d \mapsto 2^C$ that maps each edge in R_d to a set of tuples of the form $C : \langle v, \varphi \rangle$, where v is a variable used in n_1

and n_2 and φ is the type of v. We define the following methods for a node n in the APG.

Definition 3.1. construct (n): returns the node type of n, such as `loop`, `if`, and `variable`. The only exception is for node types `declare`, `method` and `operation`, for which node type appended with the variable type, method name, and operator, respectively, are returned to make the constructs more specific.

Definition 3.2. types (n): returns the list of types for node n. For example, the node "method (Integer, reverse, {x})" returns the list {Integer, Integer}, where x is an int variable.

Figure 2a shows an excerpt of the APG for the buggy code snippet in Figure 1a. The solid black arrows show the structural relationships and the dashed red arrows show the data relationships. To keep the APG visually simple, we have omitted the tuples for the data edges. The line numbers corresponding to the snippet are shown above the nodes.

3.2.2 Building the APG. The process of building the APG can be broken down into three steps: *creation* of nodes and edges, *attribution* of missing types, and *abstraction* of constructs and types.

Creation: To build the APG for a given code snippet, the approach first parses the snippet into an AST. Nodes of the APG are then created by traversing over the AST and extracting groups of AST nodes that can be simplified and summarized into one node of the APG. For example, the sub-tree corresponding to the method call expression is summarized into the APG node type "method" with information about the method's caller, name, and arguments. The APG retains the structural edges between the groups of summarized nodes obtained from the AST. Data edges are then added to the APG by tracking uses of the different "variable" nodes in the APG.

Attribution: After the APG has been created, the attribution step augments the APG by adding missing information about types. This step is necessary as the code snippets under consideration may not be complete, especially the ones coming from SO. Our approach is comprised of three kinds of attributions: (1) Use constructs to decide missing types of variables. For example, if the construct being used is `for-each`, then the type of the variable used in its iteration can be assigned as "Collection", since the `for-each` loop typically operates over a collection object. Similarly, the type of a variable used to specify the conditions in the `if`, `while`, `do-while`, and `for` constructs can be assigned to be "Boolean". (2) The caller of method invocations without explicit specifications can be presumed to be local methods, with the caller attributed as "this". (3) Variables without explicit declaration of types, but containing assignments from literals, constructors, or arrays, are attributed with the types of the kinds of assignment. For example, type of x in the code snippet `x = 1;` is inferred to be "Integer".

Abstraction: This step abstracts and canonicalizes the information in the APG to allow code comparison in the later steps to focus on high-level structure of the code rather than small syntactic differences. With this goal in mind, we designed the abstraction step to operate at different levels. First, canonicalize semantically equivalent constructs. We identify `for`, `for-each`, `while`, and `do-while` as semantic equivalents of each other, since they perform the same function despite the syntactic differences. For these looping constructs, we canonicalize their node type as "loop". For example,

$\langle node \rangle$	$\rightarrow$	$\langle root \rangle \mid \langle variable \rangle \mid \langle method \rangle \mid \langle if \rangle \mid \langle loop \rangle \mid \langle arrayAccess \rangle \mid \langle cast \rangle \mid$
		$\langle constructor \rangle \mid \langle declare \rangle \mid \langle fieldAccess \rangle \mid \langle literal \rangle \mid \langle operation \rangle \mid \langle instanceof \rangle \mid$
		$\langle sync \rangle$

$\langle method \rangle$	$\rightarrow$	method ($\langle node \rangle \mid \langle type \rangle$, $\langle name \rangle$, {$\langle node\text{-}list \rangle$})	// method (caller, name, {args})
$\langle if \rangle$	$\rightarrow$	if ($\langle node \rangle$)	// if (condition)
$\langle loop \rangle$	$\rightarrow$	loop ($\langle node \rangle$)	// loop (condition)
$\langle arrayAccess \rangle$	$\rightarrow$	arrayAccess ($\langle node \rangle$, $\langle node \rangle$)	// arrayAccess (var, index)
$\langle cast \rangle$	$\rightarrow$	cast ($\langle type \rangle$, $\langle node \rangle$)	// cast (type, expr)
$\langle constructor \rangle$	$\rightarrow$	constructor ($\langle type \rangle$, {$\langle node\text{-}list \rangle$})	// constructor (class name, {args})
$\langle declare \rangle$	$\rightarrow$	declare ($\langle variable \rangle$)	// declare (var)
$\langle fieldAccess \rangle$	$\rightarrow$	fieldAccess ($\langle type \rangle$, $\langle name \rangle$)	// fieldAccess (caller, name)
$\langle literal \rangle$	$\rightarrow$	literal ($\langle type \rangle$, $\langle value \rangle$)	// literal (type, value)
$\langle operation \rangle$	$\rightarrow$	operation ($\langle operator \rangle$, {$\langle node\text{-}list \rangle$})	// operation (operator, {operands})
$\langle instanceof \rangle$	$\rightarrow$	instanceof ($\langle node \rangle$, $\langle type \rangle$)	// instanceof (expr, type)
$\langle sync \rangle$	$\rightarrow$	synchronized ($\langle node \rangle$)	// synchronized (expr)
$\langle variable \rangle$	$\rightarrow$	$\langle$ $\langle name \rangle$, $\langle type \rangle$ $\rangle$	
$\langle node\text{-}list \rangle$	$\rightarrow$	$\langle node \rangle$, $\langle node\text{-}list \rangle \mid \langle node \rangle$	

$\langle root \rangle$	$\rightarrow$	'root'
$\langle name \rangle$	$\rightarrow$	variable name | method name | field name
$\langle type \rangle$	$\rightarrow$	primitive | class name | 'this' | MISSING
$\langle value \rangle$	$\rightarrow$	value
$\langle operator \rangle$	$\rightarrow$	assign | binary | unary

Figure 4: Grammar of nodes in the APG

`for-each` at L216 in Figure 1a is translated to `loop` node in Figure 2a. Similarly, `if`, ternary operator (?:), and `switch-case` constructs are assigned a canonicalized representation of "if" node. All binary operations (e.g.,+ and -), unary operations (e.g.,++), and assignment of values to variables are generalized to an "operation" node with the operator field specifying the operation type, such as PLUS, MINUS, and ASSIGN. Second, the abstraction step processes the data types of variables. Our approach first converts all primitive types into their corresponding wrapper classes, such as `int` is normalized to `Integer`. Then, the approach identifies user-defined or non-Java framework classes and re-attributes them as "appClass" (type of fieldSchema is changed from FieldSchema to appClass in Figure 2a). Lastly, the approach abstracts all collection class/interface types, such as `List` and `HashMap` into a common type "Collection" (type of list object in Figure 2b is changed from ArrayList to Collection). Normalization of the collection classes is driven by the observation that the root cause for the RE across all different Collection classes is the same. Canonicalizing all collection classes allows our approach to focus on the exception causing scenario and find a relevant SO post addressing the root cause, rather than being tied up in type differences. Finally, the abstraction step scans for duplicate sub-trees within the APG, creates and assigns them to new variables, and refactors their use to point to the new variables. This refactoring is done to reduce the overall size of the APG. For example, the method chain, `table.getSd().getCols()`, appears at lines 216 ans 218, hence, it is refactored into a new variable, `$t1`, as shown in Figure 2a.

3.3 APG Structural Alignment

We now describe our alignment component that is used in the FSL and code similarity matching steps. The goal of this component is to find the structural correspondence between two APGs. The APGs are first converted into *structural trees* by removing their data edges. We then apply the APTED [26, 27] tree-edit distance algorithm to find the corresponding nodes. The output is a set of tuples of the form $\langle n_1, n_2, M \rangle$, where n_1 and n_2 is a pair of matched nodes and M indicates the type of match; full or partial.

Given a cost model, the APTED algorithm produces as output an optimal (minimal-cost) set of edit operations required for transforming structural tree T_1 into tree T_2. The different possible operations are: *match*, *delete*, *insert*, and *update*. We define the cost model as follows. The *match* operation is designed to perform a leveled equivalence check between two nodes, $n_1 \in T_1$ and $n_2 \in T_2$. If both, the constructs (Definition 3.1) and types (Definition 3.2), of n_1 and n_2 are identical, then it is considered as a *full match*, and has no cost, implying the most preferred operation. If only the constructs match, then it is considered to be a *partial match*, and the cost is 0.5. If nothing matches, then a unit cost is returned. The *delete* operation generally has a unit cost, unless the node to be deleted is same as the node at the failing line, then it returns an ∞ cost. The intuition behind preempting the deletion of the node at the failing line is to prevent false-positive alignment at locations in the code that are not related to the exception scenario. *insert* and *update* operations simply return a unit cost.

3.4 Failure Scenario Localization (FSL)

Given a SO post, the goal of this step is to identify lines in the question code snippet that are relevant to the exception scenario. To perform the FSL, our insight is to use the answer code snippets to identify the relevant lines and discard the remaining redundant lines. The answer code snippets often suggest patches for the RE being discussed in the question and often more directly address the failure producing lines. Our idea of using answer code snippets is analogous to spectrum-based fault localization, which use failing test cases to locate the faulty lines.

The FSL step takes as input a SO post. For each question and answer code snippet pair, the approach begins by transforming them into APG_Q and APG_A, respectively. It then aligns APG_Q and APG_A using the algorithm discussed in Section 3.3 to obtain a set of matched nodes. Each matched node $n_Q \in APG_Q$, is annotated as relevant. After all of the answer code snippets in the post have been processed, the approach prunes the APG_Q by deleting the non-annotated nodes. This pruned APG_Q, called the *Exception Scenario Pattern (ESP)*, is then produced as the output of the FSL step.

3.5 Code Similarity Matching

The goal of this step is find the degree of similarity between developer's buggy code snippet and a SO post's question code snippet to determine the relevancy of the post for the developer. To determine relevance, our insight is to compare the two code snippets based on the similarity between their exception scenarios.

The code similarity matching step takes three inputs: developer's buggy code, exception information (failing line and exception type), and ESPs obtained from the FSL of SO posts during the offline phase. The approach first converts the buggy code into APG_B. It then extracts a set, P, of ESPs corresponding to the given exception type. $p \in P$ is aligned with APG_B using the algorithm described in Section 3.3. The approach then computes the similarity score (discussed below) for p using the set of matched nodes given by the alignment function and analyzing APG_B and p. The approach orders the SO posts corresponding to the ESPs in P in descending order of similarity. Posts with same similarity score are ordered based on the number of user votes (high to low), which is an approximate

indicator of the popularity of the post. Finally, the topmost SO post is returned as the output of the approach.

$$similarity\_score = w_1 \times C + w_2 \times \mathcal{T} + w_3 \times \mathcal{V} + w_4 \times \mathcal{S} \quad (1)$$

Equation (1) shows the similarity score, which is a function of four weighted heuristics. The heuristics cover different aspects of similarity: structural (construct and type), data (var-use), and size (ESP). Each of the four heuristics are normalized to report a value in the range [0, 1]. We now describe the heuristics.

Construct similarity (C) is the sum of *partially* matched nodes divided by the total number of matched nodes. The partially matched nodes are obtained by scanning the set of tuples returned by the alignment algorithm where $M = partial$.

Types similarity ($\mathcal{T}$) is the sum of *fully* matched nodes divided by the total number of matched nodes. Similar to above, fully matched nodes are obtained by scanning the tuples for $M = full$.

Var-use similarity ($\mathcal{V}$) measures the similarity between the uses of variables appearing in the matched nodes of p and APG_B. The intuition behind this heuristic is that related exception scenarios should respect the same data relationships between their nodes as well. For a matched node pair, $\langle n_p, n_B \rangle$, the approach extracts the variables nodes at n_p and n_B, respectively. For each variable, v, it then collects the set of nodes (U_v) where the variable is being used by traversing the data edges in the APG under consideration. The approach then computes the Jaccard similarity between the U_v of p and that of APG_B. For example, for the matched loop (...) nodes from Figure 2a and Figure 2b, $U_{\$t1} = U_{list} = \{loop, remove\}$. Thus, their Jaccard similarity score is 1.0. Var-use similarity score is calculated as the sum of all Jaccard similarities of all variables in the matched nodes divided by the total number of variables.

ESP size similarity ($\mathcal{S}$) is given by $min\ (|p|, |p_{ideal}|)$ divided by $max\ (|p|, |p_{ideal}|)$, where $|p|$ is the size (total number of nodes) of p and $|p_{ideal}|$ is the size of the ideal ESP for that exception type. This allows our approach to penalize rather small or large ESPs, which may contain too little or superfluous information that may lead to false positive matches. A small ESP can be caused if the answer code snippets used in the FSL step do not fully capture the exception scenario, while a large ESP can be caused by answer snippets that contain redundant matches with the question code snippet. Since knowing the ideal ESP size for each RE context is not feasible, for a given exception type, we use *median* size derived from all ESPs of the exception type as a proxy for the ideal size.

4 EVALUATION

To assess the effectiveness of our approach, we conducted an empirical evaluation to address the following research questions:

RQ1: How effective is MAESTRO in recommending relevant SO posts for fixing REs?

RQ2: How effective are the key contributions, *localization, matching*, and *program abstraction*, in finding a relevant post?

RQ3: How relevant are the SO posts suggested by MAESTRO compared to other state-of-the-art techniques?

4.1 Implementation

We implemented our technique as a Java prototype tool named Maestro. We used JavaParser [11] to create ASTs for code snippets. We leveraged the implementation of APTED algorithm [25–27] to compute the structural alignment between APGs. The four weights, w_{1-4}, used in Equation (1) were set to 1, 2, 1, and 1.4, respectively. These weight values were selected by performing an empirical analysis on a subset of the dataset.

4.2 Datasets

We instantiated Maestro on the SO data dump released in March 2019 [37]. For evaluation, we used 78 instances selected from the top 500 Java repositories on GitHub. Each instance comprises of the buggy Java file throwing the RE and the failing line number. The top 500 GitHub repositories represent popular, large, and well-maintained projects. To select our evaluation instances, we first scanned the commit messages of the top 500 GitHub repositories for the mention of at least one of the 53 Java RE types in a fixing context. We established the fixing context by considering keywords such as *fix*, *resolve*, *repair*, *avoid*, and *prevent*. We then filtered out duplicates, commits that did not contain any Java change files, and commits that were consisted of simply throwing the RE.

After the filtration, we were left with 1,724 candidate patches across 19 unique RE types. We then categorized these candidates, by exception type, into four groups: *high*, *medium*, *low*, and *very low*, based on the frequency of their corresponding RE types. High category comprised of RE types having more than 100 candidates. Similarly, medium with 10–100 candidates, low with 2–10, and very low with only one candidate. This resulted in the four categories including 6, 5, 2, and 6 RE types, respectively. This distribution approximately represents the popularity of the different REs in the real-world. Finally, since our evaluation metrics involve manual inspection (Section 4.3), we chose to select a small but representative sample from each of the RE types. Thus to mimic the geometric progression observed across the four RE categories, our methodology was to randomly select 8, 4, 2, and 1 candidates from the four categories, respectively, for each of the corresponding RE types. To collect the instance from each selected candidate, we downloaded the buggy Java file (i.e., one version before the candidate's commit ID) and determined the failing line by analyzing bug reports, commit messages, and/or based on our domain knowledge. Table 1 shows a summary of our evaluation dataset under the columns "Category", "RE type", and "# inst". Instance specifics are at [1].

4.3 Evaluation Metrics

The goal of our evaluation is to measure the relevancy of the SO posts recommended by Maestro (RQ1), its variants (RQ2), and its competitors (RQ3) for fixing REs. However, there exists no "ground truth" for calculating such a relevancy. This relevancy has to be derived through a manual inspection that is necessarily subjective.

For this, we recruited two external participants to manually evaluate and provide the relevancy ratings for RQ1–3. Our participants were software professionals with a Java experience of 5–10 years. To further reduce bias, for each of the 78 instances, the participants were provided with eight SO posts produced by the different tools (1 by Maestro, 3 by its variants, and 4 by its competitors) in a *randomized* and *anonymized* fashion. The participants independently analyzed all of the instances, and then sat down together to resolve differences with one of the authors serving as a mediator to reach consensus [22, 23]. Before resolving the differences, we measured inter-rater reliability using Cohen's Kappa [5], which gives a score of the degree of agreement existing between two raters. Across the 624 ratings provided by each participant, the value of Kappa (κ) was found to be 0.63, implying substantial agreement between the raters [17].

The participants were asked to rate each SO post in one of the following four categories: **Instrumental (I)**: The participant feels confident that the SO post captures the RE scenario precisely, and offers a highly effective repair in the context of the instance. **Helpful (H)**: The participant finds the SO post informative, i.e., offers insight into the RE, but does not provide a direct solution for fixing the RE in the context of the instance. **Unavailable (U)**: No SO post was recommended, requiring developers to perform their own manual search. **Misleading (M)**: The participant finds the SO post highly irrelevant to the instance.

This rating scale broadly follows the approach of Zimmermann et al. [3, 19]. We did not include a "Don't Know" category following the advice of Kitchenham et al. [15], as our participants were well-experienced to always make an informed decision.

To characterize the overall effectiveness in recommending a relevant SO post, we used the following metrics, again following prior research [3, 19], where they have been shown to be successful in avoiding scale violations [15].

I-score: Percentage of *perfect* SO posts, i.e., rated Instrumental	$I\text{-}score = \frac{I}{I+H+U+M}$
IH-score: Percentage of *relevant* SO posts, i.e., rated Instrumental or Helpful	$IH\text{-}score = \frac{I+H}{I+H+U+M}$
M-score: Percentage of *irrelevant* SO posts, i.e., rated Misleading	$M\text{-}score = \frac{M}{I+H+U+M}$

Details of the participant ratings for the 78 instances across all eight tools (RQ1–3) are available at [1]

4.4 RQ1: Effectiveness of Maestro

Table 1 shows the results grouped by RE types. Maestro reported an overall IH-score of 71%, i.e., SO posts for 55 out of 78 instances were rated *relevant* (Instrumental or Helpful). Of these, 31 posts were rated Instrumental (I-score = 40%). This shows that Maestro was effective in recommending relevant SO posts for fixing REs.

A perfect IH-score (100%) was reported for 10 out of 19 RE types. One prominent pattern that we observed among these successful cases was that the exception scenario was typically well-defined with a specific sequence of actions leading to the RE. An example of such a pattern is described in Section 2 for ConcurrentModification. On the contrary, RE types such as NullPointerException showcase an overly generic pattern, viz. dereferencing a null object, which can have a wide range of manifestations. This makes it challenging for Maestro to anchor upon specific program elements that can help find the best post. Figure 5 shows an example of this case. The RE is thrown at line 2 in the buggy code because "get()" is invoked

Table 1: Effectiveness Results of MAESTRO

Category	RE type	# inst	I-score	IH-score	M-score
High [100, ∞]	ClassCastException	8	0.63	1.00	0.00
	ConcurrentModificationException	8	0.75	1.00	0.00
	IllegalArgumentException	8	0.38	0.50	0.38
	IllegalStateException	8	0.25	0.50	0.25
	IndexOutOfBoundsException	8	0.38	0.63	0.38
	NullPointerException	8	0.00	0.38	0.63
Medium [10, 99]	ArithmeticException	4	1.00	1.00	0.00
	NoSuchElementException	4	0.25	0.25	0.75
	RejectedExecutionException	4	0.75	1.00	0.00
	SecurityException	4	0.25	0.75	0.25
	UnsupportedOperationException	4	0.00	0.75	0.25
Low [2, 9]	EmptyStackException	2	0.50	1.00	0.00
	NegativeArraySizeException	2	0.50	0.50	0.50
Very Low [1, 1]	ArrayStoreException	1	0.00	1.00	0.00
	BufferOverflowException	1	0.00	1.00	0.00
	BufferUnderflowException	1	0.00	0.00	1.00
	CMMException	1	0.00	1.00	0.00
	IllegalMonitorStateException	1	0.00	1.00	0.00
	MissingResourceException	1	1.00	1.00	0.00
	Overall		**0.40**	**0.71**	**0.26**

```
1 ...
2 while (!isShutdown.get())  ← RE
3 ...
4   if (this.metricUploader != null)
5 ...
```

```
1 ...
2 for (Square[] s:gameBoard)
3   for (Square ss : s)
4     ss = Square.EMPTY;
5 ...
```

(a) Buggy Code from alibaba/jstorm   **(b) SO post: #21819264**

**Figure 5: Irrelevant post for NullPointerException**

on "isShutdown", which is null. Since the program elements are not specifically pertinent to the RE, MAESTRO finds an arbitrary post that matches at irrelevant constructs, such as loop (line 2) and field access (line 4).

Our investigations revealed two other reasons when MAESTRO reported irrelevant posts. First, inaccuracies in the FSL step lead to under or over specific ESPs. This happens when the answers do not capture the failure scenario succinctly *and* with a sufficient code context. As a consequence of this MAESTRO finds redundant matches, resulting in arbitrary posts. The second reason is when the exception scenario in the buggy code is very rare or app specific. In such situations, MAESTRO may not report any post, which happened for 3 out of the 78 instances, or it may report irrelevant posts based on peripheral matches.

In our experiments, we found that MAESTRO takes a median 2.6 sec (average = 76 sec) end-to-end to find the most relevant SO post on a 8-core desktop machine. We believe this makes MAESTRO effective for real-time use.

4.5 RQ2: Key contributions of MAESTRO

4.5.1 Design. In this experiment, we evaluate the importance of the three main contributions of our work by creating three baseline variants of MAESTRO. The first variant, MAESTRO-NoLoc, measures the impact of the FSL step (Section 3.4) by removing it from the workflow. The second variant, MAESTRO-SIMPLEMATCH, assesses the importance of the heuristics-based weighted similarity score computation (Section 3.5) by replacing it with a simple sum of the matched nodes (full and partial), i.e., the post with the highest number of matches is returned as the most relevant post. Lastly, the third variant, MAESTRO-AST, evaluates the importance of the APG representation (Section 3.2) by replacing it with AST. For this variant,
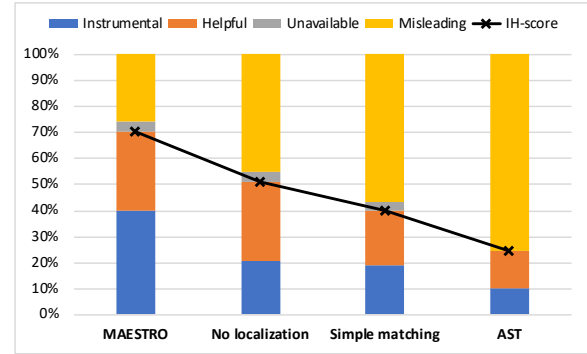
Figure 6: MAESTRO vs. its variants
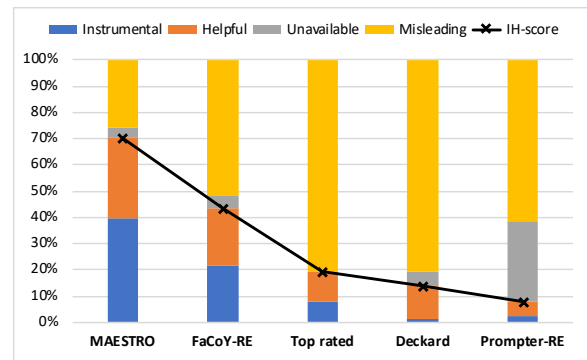
Figure 7: MAESTRO vs. other techniques

we implemented the cost function for structural alignment (Section 3.3) as follows: A strict syntactic match between method calls, class names, data types, etc. constitutes a *full* match, while matches between rule nodes, such as `IfStatement` and `MethodCallExpr`, constitute a *partial* match.

4.5.2 Results. Figure 6 shows the distribution of the participant ratings for MAESTRO and its variants. The IH-scores for the three variants show a significant almost-linear drop compared to MAESTRO: 28%, 44%, and 66%, respectively. The primary reason for this decrease is that the variants tend to match with the lines in the SO snippet that are not related to the exception scenario. Concretely, MAESTRO-NoLoc matches entire SO snippets with the instance. This falsely causes the snippets to match peripherally to the core exception scenario, leading them to be ranked higher. Conversely, while MAESTRO-SIMPLEMATCH employs FSL, it gives preference to ESPs that match maximally with the instance, albeit redundantly. MAESTRO overcomes this problem by penalizing inflated ESPs by comparing them with the ideal ESP size, as discussed in Section 3.5. On the other hand, while MAESTRO-AST employs both FSL and the heuristics-based similarity scoring, it performs poorly because the AST representation alignment results in very few *full* matches, while a high number of false positive *partial* matches. Overall, this experiment demonstrates the strengths of the different components in MAESTRO, which in synergy contribute to facilitating a meaningful comparison to report relevant SO posts.

4.6 RQ3: Comparison with other techniques

4.6.1 Design. Maestro does not have any direct competitors, since there exist no techniques targeting its exact use case. However, state-of-the-art (1) SO recommendation techniques, (2) code similarity matching techniques, and (3) native SO search engine can serve as viable alternatives. Representing the first group is Prompter [29], the only existing technique that searches and suggests SO posts for a given context, albeit from a code completion perspective. To adapt to our use case, we created **Prompter-RE**, which prefixes Prompter's default search query with the RE name [9]. To provide sufficient code context, Prompter-RE is given the entire method from the instance containing the failing line, as its input. For the second group, we used state-of-the-art syntactic and semantic code clone detectors to match input buggy code with SO code snippets. For the syntactic code clone detector, we selected **Deckard** [12] which is based on comparing ASTs. We made this choice by comparing its performance with current-generation token-based detector, SourcererCC [35]. Details of this experiment can be found at [1]. Aroma's [22] light-weight search could also have served as a syntactic code clone detector alternative. However, since principally Deckard and Aroma both capture and compare syntactic and structural relationships in code, we selected Deckard as a representative. As our semantic code clone detector, we used FaCoY [14]. We compare Maestro with the first phase of FaCoY, which produces a ranked list of SO posts by matching structural code tokens. To suit our use case, we created **FaCoY-RE** by changing its default implementation to compare input with SO *question* snippets, rather than answers. We instantiated both, Deckard and FaCoY-RE, on the same SO pool as Maestro, and provided as input a fragment of the instance encompassing the failing line and its immediate surrounding context, ±3 lines [33]. Lastly, we compare with the native SO search engine [39], which for a given RE type keyword, returns a list of **top rated** posts, ranked by relevance.

4.6.2 Results. Figure 7 shows the distribution of participant ratings for the 78 instances across different competitors. The IH-scores of FaCoY-RE, Top rated, Deckard, and Prompter-RE are 44%, 19%, 14%, and 8%, respectively. Maestro outperforms all the techniques with a substantial margin, showing 38%–89% improvement in IH-score.

FaCoY-RE finds SO posts using tf-idf and Cosine similarity, which calculates the highest overlap between less frequent code tokens. This does not work in cases where the code elements are commonly occurring ones, such as in IndexOutOfBoundsException, or in cases with extremely rare (e.g., app specific) tokens that may not have any representation on SO. Deckard demonstrated a similar problem in such instances, where finding syntactic matches between buggy code and SO snippets resulted in irrelevant or no matches. This underscores the strength of Maestro's APG representation which is able to sufficiently abstract out the syntactic details to compare at a high-level. The main reason behind low quality posts reported by Prompter-RE is that its technique is based on a bag of words algorithm that tries to find a general-purpose post discussing the overall function of the buggy code, rather than focusing on the exception scenario. We also observed that although we had augmented the search query, the added RE name was in some cases removed by the search engine (e.g., Google), likely due to the close relevancy among the other words in the query. In such situations,

the posts returned by Prompter-RE did not even pertain to the RE. Lastly, although the SO top-rated post for a given RE type contains rich and informative content, it is unable to provide satisfactory resolution for *all* different exception scenarios of the RE.

This experiment shows that state-of-the-art techniques, applied in a straight-forward manner, may not be well-suited for Maestro's use case of finding most relevant SO posts for fixing REs.

4.7 User Experience Case Study

We conducted a controlled study (results at [1]) to understand developers' experience using Maestro. We presented 10 instances from our dataset to 10 Java developers. For a given instance, the participants were asked to manually search and report a relevant SO post. Then they were shown and asked to rate the post recommended by Maestro, and provide feedback about their experience. We decided not to measure time saving, as other research has shown that manual time varies greatly by experience level and has little correlation with task complexity [22].

Maestro met with a high user approval with the IH-score of 80% (I-score: 50%). Comments left by the participants were very insightful. Three participants stated that Maestro found "*a better post than them*", while three others stated that Maestro was able to find "*the exact post that they found*" during their manual search. Participants also largely commented on the quality of the posts, e.g., "correct context and solution; same data structures being used and same method" and "the post can solve the problem perfectly". In the remaining 2 instances, where Maestro returned irrelevant posts, the comments resonated with our observations discussed in Section 4.4, such as "*the post has nothing to do with the bug*" and "*What can I say about null pointer exception? It's always there but finding the best post is hard!*". Overall, we found that the participants had a positive attitude towards Maestro, as it was able to find posts of comparable or even higher quality than them, for most instances.

4.8 Limitations

Mining SO: Developer forums like SO only discuss commonly occurring, generic development issues. Maestro inherits this limitation in that it cannot assist with very application-specific REs, e.g., an RE rooted in the semantics of application-specific APIs. It is also not very helpful with overly generic REs like NullPointerException where there is no common, yet sufficiently descriptive pattern to the exception, and a specific post discussing it.

Code-based search: Maestro's current search relies exclusively on an analysis of code snippets in the Q&A threads. Although this provides relevant matches in a significant fraction of instances, mining and incorporating information from the natural language text in the posts would be a valuable next step.

Scope of analysis: Our FSL step uses the structure of answer snippets to pin-point the failure-inducing statements. However, posts with lengthy and/or non-specific answer snippets may lead to sub-optimal localization and match results. In future work we propose to use other sources of information, including static or dynamic analysis of question and answer snippets or information in the surrounding text to improve the accuracy of localization. Another limitation is that currently Maestro's analysis can only handle intra-procedural failure scenarios. This design is driven by

the observation that the vast majority of RE scenarios tend to be succinct and local. However, extend MAESTRO to support to inter-procedural scenarios could further expand its scope.

Construct Validity: Judging the relevance of SO posts produced by any of the tools (Section 4: RQ1-RQ3) is an inherently subjective task, and hence a potential threat. We mitigated this threat by specifying clear criteria for each of a 4-valued rating scale, consistent with previous work [3, 15, 19]. Further, we used two *non-authors* to independently rank each output instance (post), to remove author-bias and get a plurality of opinions. Then we used a discussion process for the raters to reach consensus on instances of different opinion, also following previous work [22, 23]. Finally, we calculate and report Cohen's Kappa [5], a measure of inter-rater reliability, showing substantial agreement between the raters' original ratings.

5 RELATED WORK

Mining Q & A sites. The work most closely related to ours is PROMPTER [28, 29] which continuously searches and recommends relevant SO posts to a developer as she develops code in an IDE. PROMPTER models the user's code as a bag of words, creates a search query of rare tokens derived from it, and uses this query to search SO. Libra [30] augments this approach by including relevant terms from the user's recent browsing history. However, while these techniques are a good fit for the general "code completion" use-case that they target, as shown in Section 4, they do not work well for our use-case where more complete code and specific (RE) error information is available. Hence our approach performs a more structured comparison of the user's code with the SO code snippets, rather than using a bag-of-words model.

In recent work Zhang et al.[46] propose a tool EXAMPLESTACK, to guide developers in adapting code snippets from relevant SO posts to their own codebase. EXAMPLESTACK nicely complements our contribution of finding the relevant posts. CSNIPPEX [40] proposes an approach to make SO code snippets compilable by adding missing imports and variable, method, and class declarations. Such a technique can make a larger fraction of SO code analyzable by approaches such as ours. Nagy et al. [24] mine common error patterns in SQL queries for potential use by SQL developers. SOFix [18] mines SO to *manually* extract a set of repair schemas for use in a generate-and-validate automatic program repair (APR) tool. In both these works the aim is *offline* mining of SO for subsequent use in specialized use-cases. By contrast, our problem is real-time recommendation of relevant SO posts on an instance-specific basis. QACrashFix [8] uses error information in an Android-related crash bug to gather a population of relevant SO posts and then uses the posts' answer code snippets in a generate-and-validate APR approach to fix the bug. Unlike us, here the core contribution is on the use of information in SO posts for repair, rather than the search for relevant posts.

Code clone detection and code search. Syntactic code clone detection techniques detect syntactically similar code fragments (i.e., Type 1,2,3 clones) by matching tokens, such as CCFinder [13] and SOURCERERCC [35], or comparing ASTs, such as DECKARD [12], or using hybrid approaches such as NiCad [6]. However, as discussed in Sections 1,2 and empirically evaluated in Section 4 matching the developer's code with the SO code snippet is not a typical syntactic code clone detection problem because of the degree

of dissimilarity between the two. Semantic code clone detection (i.e., Type 4 clones) match syntactically dissimilar but semantically equivalent code fragments. Prominent representatives include the work by White et al. [43] based on deep learning, Oreo [34] which combines information retrieval, machine learning, and software metrics, CCAligner [42] which specializes in large-gapped clones, and FaCoY [14] which uses a novel *query alternation* strategy leveraging natural language descriptions of code. Our problem is not suitable for these techniques either, since our problem instances do in fact possess structural similarity, *provided* a technique can localize the exception-triggering segments and suitably abstract away syntactic differences. Our technique is tailored to do precisely this. Code-to-code search engines such as Aroma [22] and Krugle [16] form another related body of work. However, since their purpose is code recommendation for the purpose of code completion or code enhancement they seek to find extensions or modifications of the query code, rather than seeking to align the error-producing scenarios of the two code segments, as we do.

Debugging, patching, and recovery for runtime errors. Our work is inspired by the body of research on remediation of runtime errors. Sinha et al. [36] proposed one of the earliest techniques for fault localization and repair of Java runtime exceptions, NPEFix [7] proposes a generate-and-validate APR approach for null-pointer exceptions (NPE), VFix [45] uses data and control-flow analysis to prune the repair space for NPEs and generate more accurate repairs, and Genesis [20] automatically extracts repair patterns specific to various exception types to use in an APR approach. There is also interesting research on isolating and recovering from runtime errors [10, 21]. The above techniques use program analysis as the basis for their remediation. By contrast, our work facilitates the use of crowd-sourced knowledge available in online Q&A sites for this purpose, and is therefore complementary to the above.

6 CONCLUSION

In this work we presented a technique and prototype tool called MAESTRO to automatically recommend an SO post most relevant to a given Java RE in a developer's code. Specifically, MAESTRO returns the post best matching the exception-generating program scenario in the developer's code. To extract and compare the exception scenario, MAESTRO first uses the answer code snippets in a post to implicate relevant lines in the post's question code snippet and then compares these lines with the developer's code in terms of their respective Abstract Program Graph (APG) representations. The APG is a simplified and abstracted derivative of an AST that enables an effective high-level semantic comparison, while discarding low-level syntactic or semantic differences. An evaluation of MAESTRO on a benchmark of 78 instances of Java REs extracted from the top 500 Java GitHub projects showed that MAESTRO can return a relevant SO post for 71% of the exception instances, compared to relevant posts returned in only 8% - 44% instances, by four competitor tools based on state-of-the-art techniques. Further, in a user experience study of MAESTRO with 10 Java developers, the participants judged MAESTRO as reporting a highly relevant or somewhat relevant post in 80% of the instances, and in some cases, even better than the one manually found by the participant.

REFERENCES

[1] Authors. [n.d.]. Maestro Evaluation Data. Retrieved Mar 2020 from https://figshare.com/s/c7e3a7247d6b314cb5b0

[2] Sebastian Baltes and Stephan Diehl. 2018. Usage and Attribution of Stack Overflow Code Snippets in GitHub Projects. *CoRR* abs/1802.02938 (2018). arXiv:1802.02938 http://arxiv.org/abs/1802.02938

[3] Andrew Begel and Thomas Zimmermann. 2014. Analyze This! 145 Questions for Data Scientists in Software Engineering. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 12–23. https://doi.org/10.1145/2568225.2568233

[4] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Boston, MA, USA) *(CHI '09)*. ACM, New York, NY, USA, 1589–1598. https://doi.org/10.1145/1518701.1518944

[5] J. Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20, 1 (1960), 37.

[6] James R. Cordy and Chanchal K. Roy. 2011. The NiCad Clone Detector. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension (ICPC '11)*. IEEE Computer Society, Washington, DC, USA, 219–220. https://doi.org/10.1109/ICPC.2011.26

[7] Benoit Cornu, Thomas Durieux, Lionel Seinturier, and Martin Monperrus. 2015. *NPEFix: Automatic Runtime Repair of Null Pointer Exceptions in Java*. Technical Report 1512.07423. Arxiv. https://arxiv.org/abs/1512.07423.pdf

[8] Qing Gao, Hansheng Zhang, Jie Wang, Yingfei Xiong, Lu Zhang, and Hong Mei. 2015. Fixing Recurring Crash Bugs via Analyzing Q&Amp;A Sites (T). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '15)*. IEEE Computer Society, Washington, DC, USA, 307–318. https://doi.org/10.1109/ASE.2015.81

[9] Google. 2019. Search word order matters. Retrieved Aug 2019 from https://edu.google.com/coursebuilder/courses/pswg/1.2/assets/notes/Lesson1.5/Lesson1.5Wordordermatters_Text_.html

[10] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Jian Lü, and Zhendong Su. 2016. Automatic Runtime Recovery via Error Handler Synthesis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. ACM, New York, NY, USA, 684–695. https://doi.org/10.1145/2970276.2970360

[11] JavaParser. 2019. JavaParser. Retrieved Aug 2019 from https://javaparser.org/

[12] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 96–105. https://doi.org/10.1109/ICSE.2007.30

[13] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Softw. Eng.* 28, 7 (July 2002), 654–670. https://doi.org/10.1109/TSE.2002.1019480

[14] Kisub Kim, Dongsun Kim, Tegawendé F. Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. 2018. FaCoY: A Code-to-code Search Engine. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 946–957. https://doi.org/10.1145/3180155.3180187

[15] Barbara A. Kitchenham and Shari L. Pfleeger. 2008. Personal Opinion Surveys. In *Guide to Advanced Empirical Software Engineering*, Forrest Shull, Janice Singer, and Dag I.K. Sjøberg (Eds.). Springer London, 63–92.

[16] Ken Krugler. 2013. *Krugle Code Search Architecture*. Springer New York, New York, NY, 103–120. https://doi.org/10.1007/978-1-4614-6596-6_6

[17] J. Richard Landis and Gary G. Koch. 1977. The Measurement of Observer Agreement for Categorical Data. *Biometrics* 33 (1977).

[18] X. Liu and H. Zhong. 2018. Mining stackoverflow for program repair. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 118–129. https://doi.org/10.1109/SANER.2018.8330202

[19] David Lo, Nachiappan Nagappan, and Thomas Zimmermann. 2015. How Practitioners Perceive the Relevance of Software Engineering Research. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 415–425. https://doi.org/10.1145/2786805.2786809

[20] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic Inference of Code Transforms for Patch Generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. ACM, New York, NY, USA, 727–739. https://doi.org/10.1145/3106237.3106253

[21] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. 2014. Automatic Runtime Error Repair and Containment via Recovery Shepherding. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 227–238. https://doi.org/10.1145/2594291.2594337

[22] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: Code Recommendation via Structural Code Search. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 152 (Oct. 2019), 28 pages.

[23] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic Repair of Real Bugs in Java: A Large-Scale Experiment on the Defects4j Dataset. *Empirical Software Engineering* 22, 4 (Aug. 2017), 1936–1964.

[24] Csaba Nagy and Anthony Cleve. 2015. Mining Stack Overflow for Discovering Error Patterns in SQL Queries. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME) (ICSME '15)*. IEEE Computer Society, Washington, DC, USA, 516–520. https://doi.org/10.1109/ICSM.2015.7332505

[25] Mateusz Pawlik and Nikolaus Augsten. [n.d.]. APTED algorithm for the Tree Edit Distance Implemenataion. Retrieved Aug 2019 from https://github.com/DatabaseGroup/apted

[26] Mateusz Pawlik and Nikolaus Augsten. 2015. Efficient Computation of the Tree Edit Distance. *ACM Trans. Database Syst.* 40, 1, Article 3 (March 2015), 40 pages. https://doi.org/10.1145/2699485

[27] Mateusz Pawlik and Nikolaus Augsten. 2016. Tree edit distance: Robust and memory-efficient. *Information Systems* 56 (2016), 157 – 173. https://doi.org/10.1016/j.is.2015.08.004

[28] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. 2013. Leveraging Crowd Knowledge for Software Comprehension and Development. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering (CSMR '13)*. IEEE Computer Society, Washington, DC, USA, 57–66. https://doi.org/10.1109/CSMR.2013.16

[29] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. 2014. Mining StackOverflow to Turn the IDE into a Self-confident Programming Prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. ACM, New York, NY, USA, 102–111. https://doi.org/10.1145/2597073.2597077

[30] Luca Ponzanelli, Simone Scalabrino, Gabriele Bavota, Andrea Mocci, Rocco Oliveto, Massimiliano Di Penta, and Michele Lanza. 2017. Supporting Software Developers with a Holistic Recommender System. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 94–105. https://doi.org/10.1109/ICSE.2017.17

[31] prestodb. 2019. Presto project at commit 2babbe3. Retrieved Aug 2019 from https://github.com/prestodb/presto

[32] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. 2015. How Developers Search for Code: A Case Study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. ACM, New York, NY, USA, 191–201. https://doi.org/10.1145/2786805.2786855

[33] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. ELIXIR: Effective Object Oriented Program Repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, 648â€Ş659.

[34] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V. Lopes. 2018. Oreo: Detection of Clones in the Twilight Zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. ACM, New York, NY, USA, 354–365. https://doi.org/10.1145/3236024.3236026

[35] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-code. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. ACM, New York, NY, USA, 1157–1168. https://doi.org/10.1145/2884781.2884877

[36] Saurabh Sinha, Hina Shah, Carsten Görg, Shujuan Jiang, Mijung Kim, and Mary Jean Harrold. 2009. Fault Localization and Repair for Java Runtime Exceptions. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (Chicago, IL, USA) *(ISSTA '09)*. ACM, New York, NY, USA, 153–164. https://doi.org/10.1145/1572272.1572291

[37] Inc. Stack Exchange. 2019. StackOverflow Dump. Retrieved March 2019 from https://archive.org/details/stackexchange

[38] Inc. Stack Exchange. 2019. StackOverflow Statistics. Retrieved August 2019 from https://stackexchange.com/sites#traffic

[39] StackOverflow. [n.d.]. StackOverflow Search. Retrieved Mar 2020 from https://stackoverflow.com/search

[40] Valerio Terragni, Yepang Liu, and Shing-Chi Cheung. 2016. CSNIPPEX: Automated Synthesis of Compilable Code Snippets from Q&A Sites. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. ACM, New York, NY, USA, 118–129. https://doi.org/10.1145/2931037.2931058

[41] Cambridge University. 2013. Cambridge University Study States Software Bugs Cost Economy $312 Billion Per Year. http://www.prweb.com/releases/2013/1/prweb10298185.htm.

[42] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K. Roy. 2018. CCAligner: A Token Based Large-gap Clone Detector. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 1066–1077. https://doi.org/10.1145/3180155.3180179

[43] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep Learning Code Fragments for Code Clone Detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. ACM, New York, NY, USA, 87–98. https://doi.org/10.1145/2970276.2970326

[44] Yuhao Wu, Shaowei Wang, Cor-Paul Bezemer, and Katsuro Inoue. 2019. How do developers utilize source code from stack overflow? *Empirical Software Engineering* 24, 2 (01 Apr 2019), 637–673. https://doi.org/10.1007/s10664-018-9634-5

[45] 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 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 512–523. https://doi.org/10.1109/ICSE.2019.00063

[46] Tianyi Zhang, Di Yang, Crista Lopes, and Miryung Kim. 2019. Analyzing and Supporting Adaptation of Online Code Examples. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 316–327. https://doi.org/10.1109/ICSE.2019.00046

[47] Xuanhui Wang Shan Lu Yuanyuan Zhou Zhenmin Li, Lin Tan and Chengxiang Zhai. 2006. Have Things Changed Now? An Empirical Study of Bug Characteristics in Modern Open Source Software. In *Proceedings of the Workshop on Architectural and System Support for Improving Software Dependability* (San Jose, California) *(ASID '06)*.