Providing Real-time Assistance for Repairing Runtime Exceptions using Stack Overflow Posts

Sonal Mahajan
Fujitsu Research of America, Inc.
smahajan@fujitsu.com

Mukul R. Prasad
Fujitsu Research of America, Inc.
mukul@fujitsu.com

Abstract—**Runtime Exceptions (REs) are an important class of bugs that occur frequently during code development. Traditional Automatic Program Repair (APR) tools are of limited use in this "in-development" use case, since they require a test-suite to be available as a patching oracle. Thus, developers typically tend to manually resolve their in-development REs, often by referring to technical forums, such as Stack Overflow (SO). To automate this manual process we extend our previous work, MAESTRO, to provide real-time assistance to developers for repairing Java REs by recommending a relevant patch-suggesting SO post and synthesizing a repair patch from this post to fix the RE in the developer's code. MAESTRO exploits a library of Runtime Exception Patterns (REPs) semi-automatically mined from SO posts, through a relatively inexpensive, one-time, incremental process. An REP is an abstracted sequence of statements that triggers a given RE. REPs are used to index SO posts, retrieve a post most relevant to the RE instance exhibited by a developer's code and then mediate the process of extracting a concrete repair from the SO post, abstracting out post-specific details, and concretizing the repair to the developer's buggy code. We evaluate MAESTRO on a published RE benchmark comprised of 78 instances. MAESTRO is able to generate a correct repair patch at the top position in 27% of the cases, within the top-3 in 40% of the cases and overall return a useful artifact in 81% of the cases. Further, the use of REPs proves instrumental to all aspects of MAESTRO's performance, from ranking and searching of SO posts to synthesizing patches from a given post. In particular, 45% of correct patches generated by MAESTRO could not be produced by a baseline technique not using REPs, *even* when provided with MAESTRO's SO-post ranking. MAESTRO is also fast, needing around 1 second, on average, to generate its output. Overall, these results indicate that MAESTRO can provide effective real-time assistance to developers in repairing REs.**

Index Terms—**program repair, exceptions, Stack Overflow**

I. INTRODUCTION

Code search and re-use has long been recognized as a very natural part of software development [1], [2]. Technical discussion forums such as Stack Overflow (SO), a rich resource of succinct code artifacts embedded in explanatory text, provide an attractive option to developers for such code search and reuse. In fact, software developers frequently visit SO to resolve issues arising during software development, in particular those related to software debugging and patching [3], [4], [5].

Motivated by the above observation, in this work we propose a technique to provide *real-time* automated support for a developer to resolve a bug in her code by finding and adapting a suitable patch suggested in an SO post. In particular, we focus on resolving Java Runtime Exceptions (REs), a common

and important class of errors that have attracted significant research in the area of automated debugging and patching [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], and are also well represented in SO discussions [17]. Broadly, our technique shares the patch-generation goal of Automatic Program Repair (APR) techniques [18], [19], [20], [21], [22], [23]. However, traditional APR techniques, which operate on a *patch synthesis* paradigm and typically rely on a test suite as a patching oracle, involve long running times of tens of minutes or even hours. Hence they are not a good fit for our use case. Developers seeking to resolve an RE, encountered during development, on SO, are typically looking for real-time assistance, and may not have a test suite available.

SO-driven repair & Challenges. We posit that one solution lies in mimicking human developers' approach – first *finding* an SO post discussing a bug similar to theirs and *adapting* the proposed patching solution to their code (in contrast to the *patch synthesis* paradigm of current APR techniques). Such a *find-and-adapt* repair approach would necessarily need to solve three principal challenges highlighted in a recent study by Wu et al. [24], namely (C1) the mixed quality of SO code artifacts, (C2) the difficulty of comprehending code snippets, and (C3) the effort required to modify a chosen snippet to work in their code context. QACrashFix [25] proposes one such approach, for generating patches for Android-related crashes from SO posts. However, QACrashFix relies on descriptive platform-specific information from Android crashes. As we show in Section IV, QACrashFix's approach does not work well for general REs.

Insight. The guiding insight of our proposed approach is that *bug scenarios* (exception-triggering scenarios here) showcased in SO post questions, also the root causes for the developers' (exception) bugs, can form the basis for all key steps of a find-and-adapt repair approach – indexing SO posts and retrieving a post best matching the bug in a given developer's code (C1), extracting a generalized patch from question and answers in the post (C2), and adapting this patch to the developer's code context (C3). Note that by contrast, current APR techniques are organized around the mining and exploitation of *repair patterns* rather than bug patterns. Our previous work [17], automatically extracts *approximate* bug patterns, termed Exception Scenario Patterns (ESPs), by applying a predetermined set of abstractions to program statements appearing

in both question and answer code snippets of a post. These ESPs are then used as a basis for matching a given RE-generating developer code to a relevant SO post, to be then *manually* exploited by the developer. However, while such approximate bug patterns are adequate for searching relevant posts, as shown in Section IV, they are not *accurate* enough to support automatic patch generation, which is the focus of this work.

Approach. We extend our prior work, MAESTRO (**M**ine and **A**nalyz**E ST**ackoverflow to fix **R**untime excepti**O**ns) [17], to provide real-time patch-generation support to fix (Java) REs in developers' code. Our patching technique is built on top of a library of specifications called Runtime Exception Patterns (REPs), semi-automatically mined from Stack Overflow (SO), through a relatively inexpensive, one-time, incremental process. Each REP represents an abstract exception-triggering pattern for a specific RE, discussed in one or more SO posts.

Given a developer's code, crashing with an RE, MAESTRO first finds a REP from the library best describing the developer's error and using it finds the best SO post mirroring that error pattern. Then MAESTRO uses the identified REP to mediate the creation of a complete patch for the developer's buggy code by appropriately re-purposing an answer code snippet in the identified best SO post. To amplify the efficacy and scope of its patch generation MAESTRO also uses the REP library to rank the corpus of SO posts and a novel rule-based re-writing technique to make unparsable SO snippets parsable. We evaluate MAESTRO on an existing benchmark of 78 RE instances spanning 19 prominent REs. Our evaluation shows that MAESTRO is able to generate a correct repair patch at the top position in 27% of the cases, within the top-3 in 40% of the cases, and overall return a useful artifact (a correct patch, an almost correct patch, or simply a relevant post) in 81% of the cases. Further, MAESTRO only required 1 second, on average, per subject, validating its suitability for providing real-time assistance to developers in repairing REs. The evaluation also shows that the use of REPs is key to all aspects of MAESTRO's performance from ranking and searching of SO posts to synthesizing patches from a given post. In particular, 45% of correct patches generated by MAESTRO could not be produced by a baseline technique not using REPs, *even* when provided with MAESTRO's SO-post ranking.

This paper makes the following contributions:

- **REP Library.** A library of 158 REPs spanning 19 prominent RE types, systematically mined from Stack Overflow (SO), that can be used to characterize the exception-triggering patterns in SO posts, and hence to index, search and analyze those posts.
- **Patching Technique.** An automated technique and tool MAESTRO, that employs this REPs library to find an SO post mirroring the error pattern of the RE in a developer's code and then automatically creates a patch, derived from the post and instantiated for the developer's code, to resolve the developer's error.
- **Evaluation.** An evaluation of MAESTRO on an existing

benchmark of 78 RE instances, and against 4 baselines.
- **Artifacts.** The public release of our complete dataset, including the REPs library and all the patches synthesized and posts found by MAESTRO and each of the four baselines. https://doi.org/10.6084/m9.figshare.14518407

II. ILLUSTRATIVE EXAMPLE

We illustrate our technique using the example shown in Figure 1. The example is extracted from the JD-GUI project [26], which is a popular interface for viewing Java ".class" files. The buggy code shown in Figure 1e throws the common RE, ClassCastException, at line 40 because the `toArray()` method returns an `Object[]` which cannot be cast to a `URL[]`.

To find a repair for fixing this RE, the developer may refer to SO, to find a post discussing the same exception in the same scenario, and then adapt the fix suggested in the answer of the post to their own buggy code. To find such a relevant post, MAESTRO first searches for a REP best describing the developer's exception scenario (Figure 1c) from our systematically compiled library of REPs. Then, this REP is used to find the best SO post instantiating it (Figure 1a).

Consider the REP shown in Figure 1c. It represents an abstract exception-triggering pattern describing the `toArray()` failure scenario of ClassCastException. `toArray()` is an API defined for the Collections framework, hence the REP abstracts out the concrete data type of variable $v1$, allowing it to match with any of the Collection classes (e.g.,`ArrayList` and `Set`) specified on line 1. Similarly, the specific data type of $v2$ and the casting array is immaterial with reference to the RE. Therefore, the REP represents both these data types with the same "*wildcard*" notation encoding an inherent mapping between them. The wildcard can be instantiated with any data type, such as `URL[]` or `String[]`. This REP is identified by MAESTRO as the best match ($\mathcal{P}_{best}$) as it is perfectly instantiated by the developer's buggy code.

MAESTRO recommends the SO post in Figure 1a, as it perfectly matches $\mathcal{P}_{best}$. Question code snippets from SO posts typically include several lines that make the snippets functionally or syntactically complete, but otherwise irrelevant to the exception scenario (shown by . . . in Figure 1a). This "noise" may lead to inaccuracies in the patch being extracted from the post, and consequently to an incorrect patch synthesized for the developer's code. To address this problem, MAESTRO prunes the question code snippet to the relevant lines implicated by $\mathcal{P}_{best}$. MAESTRO then captures the fix suggested by the answer code snippet in a concrete edit script, $\mathbb{S}$ (Figure 1b), by comparing this pruned Q and A code snippets represented by their Abstract Program Graphs (APGs) – a simplified and abstracted derivative of the Abstract Syntax Tree (AST) (see Section III-A). The edit script $\mathbb{S}$ is comprised of a series of three non-trivial changes to the faulty line: (1) *deleting* class cast (`String[]`), (2) *adding* a new argument of array instantiation `new String[...]` to `toArray()`, and (3) *adding* memory allocation to the newly instantiated array as `image_urls.size()`.

```
1  ...
2  listofurls = (String[]) image_urls.toArray();
3  ...                                                        Q
```
```
1  listofurls = image_urls.toArray(new String[image_urls.size()]);   A
```

(a) SO post #15264182 question and answer

DELETE class cast "**(String[])**" on **line 3**
ADD argument "**new String[...]**" to "toArray()" on **line 3**
ADD argument "**image_urls.size()**" to "new String[...]" on **line 3**

(b) Concrete edit script $\mathbb{S}$ to fix Q using A

```
1  @Abstract(name="_ABSTRACT_1", val="ArrayList, Set, ...")
2  public void pattern() {
3      _ABSTRACT_1 $v1;
4      _WILDCARD_1[] $v2 = (_WILDCARD_1[]) $v1.toArray();
5  }
```

(c) REP for ClassCastException

String[] ⟺ **_WILDCARD_1[]**, **image_urls** ⟺ **$v1**, **line 3** ⟺ **line 4**

DELETE class cast "**(_WILDCARD_1[])**" on **line 4**
ADD argument "**new _WILDCARD_1[...]**" to "toArray()" on **line 4**
ADD argument "**$v1.size()**" to "new _WILDCARD_1[...]" on **line 4**

(d) Generalized edit script $\mathbb{S}'$

```
34  ...
35  ArrayList<URL> urls;
... ...
40  - URL[] array = (URL[]) urls.toArray();   ← RE thrown here
40  + URL[] array = urls.toArray(new URL[urls.size()]);
... ...
```

(e) Buggy code and developer's patch (ExtensionService.java)

_WILDCARD_1[] ⟺ **URL[]**, **$v1** ⟺ **urls**, **line 4** ⟺ **line 40**

DELETE class cast "**(URL[])**" on **line 40**
ADD argument "**new URL[...]**" to "toArray()" on **line 40**
ADD argument "**urls.size()**" to "new URL[...]" on **line 40**

(f) Concretized edit script $\mathbb{S}''$ to developer's buggy code

Fig. 1: Example from JD-GUI (https://github.com/java-decompiler/jd-gui) throwing ClassCastException

(a) Q_{APG}: APG of Figure 1a
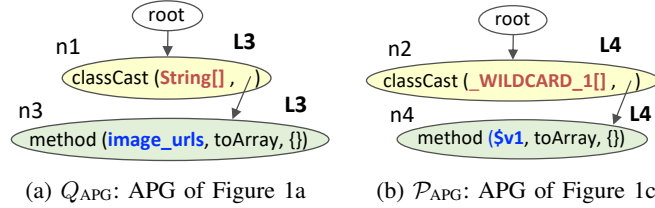
(b) $\mathcal{P}_{APG}$: APG of Figure 1c

Fig. 2: REPs of SO question code snippet and REP

The next step of MAESTRO is to abstract out post-specific details from the concrete edit script $\mathbb{S}$ to get a generalized edit script $\mathbb{S}'$. It is challenging to directly generalize the edit script, since points of abstraction vary with different RE types, as well as with different exception patterns in a specific RE type. Encoding rules for such abstractions is a cumbersome task. MAESTRO addresses this challenge by leveraging the abstractions specified in the REP. To establish correspondence, Q and $\mathcal{P}_{best}$ are structurally aligned using their APG representations as shown in Figure 2. The aligned nodes are shown with yellow and green color coding. Further, a fine-grained correspondence between the components of each matched pair of nodes is also established. For example, String[] from the classCast node in Figure 2a maps to _WILDCARD_1[] from Figure 2b. Similarly, image_urls is matched with $v1. This set of mappings as also shown in the gray box of Figure 1d are used to adapt $\mathbb{S}$ suitably to produce $\mathbb{S}'$.

Lastly, MAESTRO concretizes the generalized edit script $\mathbb{S}'$ to the developer's buggy code to get $\mathbb{S}''$ as shown in Figure 1f following the same correspondence and adaptation procedure as described above. As a final step, MAESTRO applies $\mathbb{S}''$ to developer's buggy code to synthesize a repair patch, which is identical to the developer's patch shown in Figure 1e.

The mediation of REPs is key to the success of MAESTRO in generating the correct patch. Without the REP, it would be challenging to determine that the String data type in the SO post needs to be migrated to the URL data type in the developer's code, leading to a incorrect patch. The entire process of finding the right post and generating the correct patch is performed completely automatically by MAESTRO, using our pre-compiled library of REPs.

III. APPROACH

MAESTRO's repair strategy is novel and fundamentally different than traditional APR techniques in several ways. First, MAESTRO uses *bug patterns* (REPs) to facilitate the generation of repairs, while other APR tools (e.g., PAR [19] and Getafix [11]) use a small set of predefined *fix patterns*. Second, indexing SO with REPs allows MAESTRO to leverage the diverse, but reasonably small space of exemplary concrete repair strategies from SO posts, while other APR tools are required to navigate the huge search space of scenario-agnostic concrete repairs. Lastly, a heavily pruned search space means that MAESTRO does not require a strong oracle (e.g., test cases) to validate patches, allowing MAESTRO to avoid overfitting and, moreover, achieve a real-time performance.

Our approach is designed around a library of REPs (*bug patterns*), semi-automatically mined from SO posts. Specifically, a REP is an abstracted sequence of Java statements representing an exception-triggering scenario. The REPs play a central role in mediating all aspects of MAESTRO's operation, from ranking and searching of SO posts, to synthesizing patches from a given post to fixing an RE in a developer's code. Compiling the REP library is a *low-cost*, *incremental*, *one-time effort*. Low-cost because only basic Java knowledge is required to read a post and write its REP. One-time because once the REPs are written, their benefit can be harvested thereafter without any extra cost. Incremental because new REPs can be gradually added to the library for new SO posts.
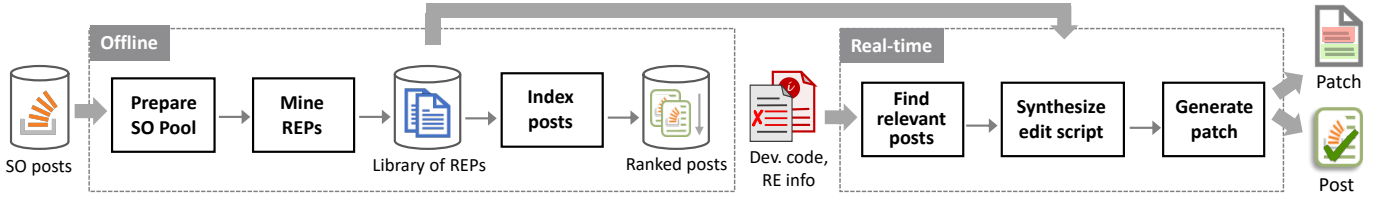
Fig. 3: Overview of the approach

Figure 3 shows an overview of MAESTRO's repair approach, consisting of four main stages: *preparing* the SO pool for analysis, *mining* REPs from SO posts, *indexing* of SO posts, and *fixing* REs in a developer's buggy code. The preparing, mining, and indexing stages are performed offline, while the fixing stage is performed in real-time.

A. Abstract Program Graph (APG)

We use the APG representation proposed in our prior work [17] for analyzing code snippets. Briefly, the APG is a simplified and abstracted derivative of the AST. It captures the structural relationships between program statements, while normalizing low-level syntactic details to facilitate a meaningful comparison between code snippets with significant differences in variable names, data types, and program constructs (e.g., while vs. for loop). For example, Figure 2a shows the APG for the code snippet in Figure 1a. The similarity score between two APGs is computed by aligning them using the APTED tree edit distance algorithm [27], [28], and cumulatively counting the normalized number of matching components in a pair of corresponding nodes. For example, in Figure 2, the similarity score is 1.0 as the component `String[]` matches with the wildcard in $\langle n1, n2 \rangle$. Similarly, the components, type of method caller (`ArrayList`) and method name (`toArray`), in $\langle n3, n4 \rangle$ match perfectly.

B. Stage 1: Preparing the Stack Overflow (SO) pool

In this stage, MAESTRO selects RE-related posts and groups them by RE type. An SO post is selected if it has: (1) RE type in the title, (2) "java" or "android" tags, (3) at least one answer, and (4) at least one *parsable* question code snippet.

A snippet is considered parsable if it is syntactically well-formed and can be parsed using any off-the-shelf Java parser (e.g., Eclipse JDT [29]). However, a large number of code snippets in SO are found to be malformed with missing or extraneous syntactic characters (e.g., parentheses) and/or undefined tokens (e.g., an ellipsis (...) in place of actual code), rendering them unusable by our approach [30], [31].

Addressing this concern, we propose an error-driven iterative approach to automatically repair unparsable snippets to make them parsable. For a given code snippet, our approach picks a parsing error $e \in \mathbb{E}$, applies a fix for e, and continues this process as long as the number of parsing errors are reducing ($|\mathbb{E}| < |\mathbb{E}_{prev}|$) or the snippet is now parsable ($\mathbb{E} = \emptyset$). We use Eclipse JDT for populating $\mathbb{E}$. For an error e, defined by the problematic element $\mathbb{T}$ at location $\mathcal{L}$, our approach applies a fix based on the rules shown in Table I. For example, for the invalid token `"..."` in snippet `List`

Error	Fixing rule
Invalid token $\mathbb{T}$ at $\mathcal{L}$	$\mathcal{L} \in$ Expression $\longrightarrow$ replace $\mathbb{T}$ with $\mathbb{T}_{valid}$ at $\mathcal{L}$
	$\mathcal{L} =$ Statement $\longrightarrow$ delete $\mathbb{T}$ at $\mathcal{L}$
Missing $\mathbb{T}$ at $\mathcal{L}$	$\mathbb{T} \in$ Symbol $\longrightarrow$ insert $\mathbb{T}$ at $\mathcal{L}$
	$\mathbb{T} \in$ Grammar Rule $\longrightarrow$ insert instantiation of $\mathbb{T}$ at $\mathcal{L}$
Extra token $\mathbb{T}$ at $\mathcal{L}$	delete $\mathbb{T}$ at $\mathcal{L}$

TABLE I: Rules for fixing parsing errors in code snippets

```
⟨pattern⟩    ::= (⟨Java statement⟩)+
⟨identifier⟩ ::= ⟨Java identifier⟩
             |  '_ABSTRACT_'[1-9]+ | '_WILDCARD_'[1-9]+
⟨annotation⟩ ::= '@Abstract (name=_ABSTRACT_' [1-9]+ ',' 'val='
                 {set of permissible values} ')'
```

Fig. 4: Extensions to Java grammar for writing REP

`x = ...;`, our approach corrects it with the fixing rule of applying a suitable initialization expression: `List x = new ArrayList<>()`. Similarly, other examples include inserting a missing semi-colon (`;`) at the end of a statement, inserting a missing `catch` block after a `try`, or removing an extra brace (`}`) from the end of a method declaration.

C. Representation for Runtime Exception Patterns (REPs)

Our REP conforms to Java grammar rules, with a couple of straightforward extensions to support generalization, shown in Figure 4. We define two types of generalizations in representing identifiers, such as method names and data types. The first is *wildcard*, to represent zero or more program elements. For example, the data type of the casting array in Figure 1c is specified with a wildcard since it could be instantiated with any of the Java or user-defined types, such as `String[]` or `URL[]`. The second generalization is called *abstract semantics*, which allows specifying a set of permissible semantic equivalent values using Java annotations. For example, the REP shown in Figure 1c represents the data type of "$v1" as a semantic abstraction, since it can only be instantiated with a Collection class, such as `List`, `ArrayList`, or `Set`.

D. Stage 2: Mining Runtime Exception Patterns (REPs)

Our approach is predicated on *precisely* extracting the exception-triggering bug patterns, i.e., REPs represented in SO posts. Our previous work [17] highlighted the challenges of doing so fully automatically, particularly because of the poor quality of Q&A snippets. On the other hand manually writing REPs for each of thousands of SO posts is impractical and also redundant, since several SO posts exhibit the same REP. Therefore, we adopt a semi-automated approach, shown in Figure 5, that optimally combines human knowledge and automation. Specifically, we iteratively ask the human to *manually* author a REP based on a single *representative* SO post
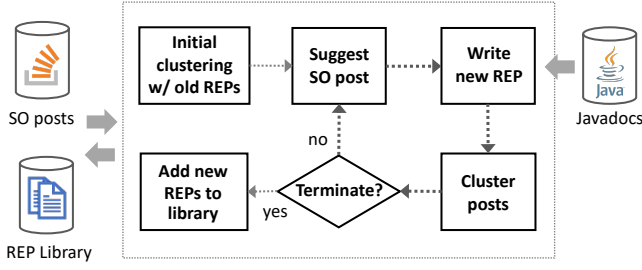
Fig. 5: Overview of the REP mining process

and then *automatically* cluster all other posts instantiating the same REP, removing them from the pool of human inspection. Key features of our mining algorithm include *inexpensive*, *one-time* effort of writing REPs, with support for *incrementally* expanding the library of REPs for new SO posts.

1) MAESTRO *suggests an SO post:* In this step MAESTRO identifies an unvisited SO post that exemplifies a commonly occurring REP. To this end, MAESTRO picks the top post after ordering unvisited SO posts (for an RE type) by user votes, breaking ties using mean distance of the candidate post from the current pool of REPs. The distance is the inverse of similarity score (Section III-A) between a REP and a post's question code snippet.

2) Human writes a REP: This step involves reviewing two specific sources, namely the suggested SO post and relevant Javadocs, to write the REP. We expect that following the below procedure a person reasonably experienced in Java, need not be an expert, should be able to write the REPs. First, the human studies the suggested SO post, particularly focusing on the question text and code snippet, to extract the post-specific exception scenario. To assist in this process, our approach highlights the potential relevant lines in the question code snippets that are referenced in the answer. To further generalize the post-specific exception scenario, the human may refer to the official Java documentation [32] of the RE as well as the different classes or APIs used in the post code.

For example, consider the SO post shown in Figure 1a. We now explain how the human can write the REP shown in Figure 1c from this post. First, the human can capture the post-specific scenario as shown in line 2. To abstract out the post-specific details, the human could then refer to the Javadocs shown in Figure 6. For instance, the human could visit the definition of ClassCastException (Figure 6a) to know that casting to `String[]` in our example is the problem. Then, the human could explore the Javadocs of `ArrayList`, which is the data type of the `image_urls` variable, specifically focusing on the `toArray()` API (Figure 6b). Upon studying this documentation, the human can figure out two generalizations: (1) `Object[]` returned by the `toArray()` API when cast to any other array type would cause the RE, thus, the concrete casting type `String[]` could be safely generalized to a *wildcard*, and (2) `toArray()` API is common for all implementations of the `Collection` framework (highlighted in Figure 6b), thus, the data type of the `image_urls` variable could be represented using *abstract semantics*. The set of values for this could be extracted from the Javadoc of the

`Collection` framework (Figure 6c), which comprehensively lists all concrete instantiations (e.g., List, Set, and ArrayList). Finally, the human could normalize the REP by renaming the variables to generic names (e.g., `image_urls` to `$v1`) to get the generalized REP shown in Figure 1c.

Alternatively, the human can simply mark the suggested post as *unviable* if a unique or meaningful REP cannot be extracted from the it. Another post is suggested in this case.

3) MAESTRO *clusters posts by REP:* The goal is to group together SO posts that constitute instances of a newly authored REP, i.e., question code snippets of posts having a perfect similarity score (1.0) with the REP. (Similarity score is computed as described in Section III-A.) Clustered posts are then marked as visited. This smartly eliminates redundancies in the work performed by the human. For *incremental* compilation of the REP library, an *initial clustering* is performed to group new SO posts instantiating existing REP library, so that the human is suggested only truly "unseen" posts.

4) Termination condition: The mining process terminates if all posts are exhausted or when U consecutive posts are marked as unviable by the human. The latter indicates a point of diminishing returns, conceivably because most distinct, popular REPs have already been extracted.

E. Stage 3: Indexing of SO Posts

Analyzing an SO post to generate a repair patch is computationally expensive. Hence, effective indexing of SO posts plays an important role in the success of our approach in achieving the real-time use case. For each cluster of SO posts from the mining step (Section III-D), our indexing ranks the posts based on their relevance to the cluster's representative REP. Particularly, we compute the similarity score between a REP and each answer code snippet in a post to up-rank posts with answers that suggest a complete fix with regards to the REP, while down-ranking posts with partial or irrelevant answers. Thus, for our illustrative example (Figure 1), post #1524182 gets a high rank since it matches closely to the REP. Post #16656384 gets a lower rank since it gives only a partial answer with missing assignment operation. Post #46201465 gets an even lower rank, since its answer is completely irrelevant to the REP. For posts with multiple answers, the highest scoring answer is used for the ranking.

F. Stage 4: Fixing RE in Developer's Code

Algorithm 1 shows the overall algorithm. Three sets of input are required. First is from the developer's failure: buggy code, $\mathcal{B}$, and RE information, E. Second is from MAESTRO's offline analysis: the library of mined REPs, $\mathcal{P}$, and the library of indexed SO posts and their Q&A pairs, $\mathcal{I}$. Third is a set of configurable parameters: number of patches to generate, $\mathcal{K}$, and number of SO posts to analyze, $\mathcal{Z}$. The output of the approach is a list $\mathcal{O}$ of $\langle r, s \rangle$ pairs, where r is a repair patch synthesized from an SO post s.

Step 1. Find Relevant SO Posts. The initial part of the algorithm (lines 3 and 4) finds the REP, $\mathcal{P}_{best}$, best matching the RE-throwing developer code, and fetches relevant SO posts

(a) Explanation of ClassCastException (b) ArrayList Javadoc for `toArray()` API (c) Specification of `Collection<E>`

Fig. 6: JavaDocs helpful in writing the REP shown in Figure 1c from SO post shown in Figure 1a

---

Algorithm 1 Algorithm for Fixing RE in Developer's Code

Input: $\mathcal{B}$: Developer's buggy code
  E: Exception information (RE type and failing line number)
  $\mathcal{P} = \{p_1, ..., p_n\}$: Library of REPs
  $\mathcal{I} = \{s_1, ..., s_n\}$: Library of indexed SO posts and their Q&A pairs
  $\mathcal{K}$: Number of repair patches to generate
  $\mathcal{Z}$: Number of SO posts to evaluate
Output: $\mathcal{O} = \{\langle r_1, s_1 \rangle, ..., \langle r_k, s_k \rangle\}$: List of k patch-post pairs
begin
1: $\mathcal{O} \leftarrow \{\}$
2: /* Step 1. Find relevant SO posts */
3: $\mathcal{P}_{best} \leftarrow$ findBestPattern($\mathcal{P}$, $\mathcal{B}$, E)
4: $\mathcal{S} \leftarrow$ getRankedPosts($\mathcal{I}$, $\mathcal{P}_{best}$)
5: $P_{APG} \leftarrow$ buildAPG($\mathcal{P}_{best}$)
6: postCount $\leftarrow 0$
7: **for each** $s \in \mathcal{S}$ **do**
8:    postCount $\leftarrow$ postCount + 1
9:    $qaPairs \leftarrow$ getQAPairs($\mathcal{I}$, s)
10:   **for each** $\langle Q, A \rangle \in qaPairs$ **do**
11:       /* Step 2. Clean Q and A code snippets */
12:       $Q_{APG} \leftarrow$ buildAPG(Q)
13:       $A_{APG} \leftarrow$ buildAPG(A)
14:       $T \leftarrow$ triangulate(Q_{APG}, A_{APG}, P_{APG})
15:       $Q_{APG} \leftarrow$ pruneAPG(Q_{APG}, T.quesRelevantLines)
16:       $A_{APG} \leftarrow$ pruneAPG(A_{APG}, T.ansRelevantLines)
17:       /* Step 3. Synthesize Generalized Edit Script */
18:       $\mathbb{S} \leftarrow$ getEditScript(Q_{APG}, A_{APG})
19:       $\mathbb{S}' \leftarrow$ adaptEditScript($\mathbb{S}$, Q_{APG}, P_{APG})
20:       /* Step 4. Generate repair patch */
21:       $\mathbb{S}'' \leftarrow$ adaptEditScript($\mathbb{S}'$, P_{APG}, $\mathcal{B}_{APG}$)
22:       $\mathcal{B}'_{APG} \leftarrow$ applyChanges($\mathcal{B}_{APG}$, $\mathbb{S}''$)
23:       **if** isValid($\mathcal{B}'_{APG}$) = *true* **then**
24:          $r \leftarrow$ convertAPGToJavaPatch($\mathcal{B}'_{APG}$)
25:          **if** isParsable(r) = *true* **then**
26:             $\mathcal{O} \leftarrow \mathcal{O} \cup \langle r, s \rangle$
27:          **end if**
28:       **end if**
29:       /* Step 5. Check termination criteria */
30:       **if** $|\mathcal{O}| \geq \mathcal{K}$ **or** postCount $\geq \mathcal{Z}$ **then**
31:          **if** $|\mathcal{O}| = 0$ **then**
32:             $\mathcal{O} \leftarrow \langle null, \mathcal{S}[1] \rangle$
33:          **end if**
34:          **return** $\mathcal{O}$
35:       **end if**
36:    **end for**
37: **end for**
38: **return** $\mathcal{O}$
end

for $\mathcal{P}_{best}$ from the indexed library. The algorithm then iterates over each SO post and its Q&A pairs to perform steps 2–5.

Step 2. Clean Q and A code snippets. Q and A snippets may include code to make the snippet functionally or syntactically complete, but otherwise irrelevant to the exception scenario and its repair. If used as such, the repair patch extracted from such snippets would likely be noisy and incorrect. Therefore, the second step of the algorithm (lines 12–16) aims to prune the snippets to relevant lines by triangulating Q, A, and $\mathcal{P}_{best}$, and retaining any newly inserted fix lines (e.g., null check for NullPointerException).

Step 3. Synthesize Generalized Edit Script. The goal is to derive a script, $\mathbb{S}'$, that is post-agnostic. We use the

---

Algorithm 2 Algorithm for Adapting Edit Script

Input: $\mathcal{T}$: Edit script to be adapted
  X: APG w.r.t. $\mathcal{T}$ (source APG)
  Y: APG to be used for adaptation (target APG)
Output: $\mathcal{T}'$: Adapted edit script
begin
1: $\mathcal{T}' \leftarrow \{\}$
2: $\mathcal{N} \leftarrow$ computeMatchedNodes(X, Y)
3: $\mathcal{C} \leftarrow$ computeCorrespondingComponents($\mathcal{N}$)
4: **for each** $op \in \mathcal{T}$ **do**
5:    $n' \leftarrow$ getMatchedNode($\mathcal{N}$, $op.n$)
6:    **if** $op.type$ = add **or** $op.type$ = update **or** $op.type$ = replace **then**
7:       $m' \leftarrow$ updateCorrespondingComponents($op.m$, $\mathcal{C}$)
8:    **end if**
9:    $op' \leftarrow$ buildEditOperation($op.type$, n', m', $op.pos$)
10:   **if** isValid(op', Y) = *true* **then**
11:       $\mathcal{T}' \leftarrow \mathcal{T}' \cup op'$
12:   **end if**
13:   **return** $\mathcal{T}'$
14: **end for**
end

APTED [27], [28] tree-edit distance algorithm to first compute the concrete edit script, $\mathbb{S}$ (line 18), which suggests edit operations for fixing the RE in Q as prescribed by A. Each entry in our edit script is comprised of one of the following four edit operations:

- add(n, m, pos): Insert a new node m in the APG at position pos (parent or child) with reference to node n
- delete(n): Delete node n from the APG
- update(n, m): Update value of n with the value of m
- replace(n, m): Replace the subtree rooted at node n with the subtree given by new node m

For generalizing $\mathbb{S}$ (line 19), our insight is to adapt it in the context of $\mathcal{P}_{best}$, since, by definition, REPs exemplify an *abstracted* description of the exception-raising scenario. We use the adaptation algorithm shown in Algorithm 2 (and discussed below) to obtain the generalized edit script, $\mathbb{S}'$.

Adapting edit script. Algorithm 2 takes as input the edit script to be adapted, $\mathcal{T}$, and the source and target APGs, X and Y, to output the adapted edit script, $\mathcal{T}'$. The algorithm begins by computing a function $M : X \mapsto Y$ that aligns X and Y to get pairs of matching nodes ($\mathcal{N}$) and pairs of corresponding components ($\mathcal{C}$) for $\mathcal{N}$ (lines 2 and 3). Then for each edit operation, $op \in \mathcal{T}$, the algorithm uses this equivalence information to suitably adapt op (lines 5–8). First, the anchor node n is updated with the matched node n'. Next, components, such as variable names, types, and method names, in node m are updated with corresponding values from Y to get m'. The adapted edit operation, op', is then checked for correctness in the context of Y with two checks: (1) n' is not empty and (2) identifiers in m' are from the namespace of Y (line 10). If the operation op' is found to pass these two

Rating scale for judging repair patches	
Correct	Patch is identical or semantically equivalent to developer patch
Almost correct	Patch has one-token difference from correct patch
No patch	Patch could not be synthesized by the tool
Incorrect	Patch is incapable of fixing the RE

Rating scale for judging SO posts	
Perfect	Post suggests an accurate repair for the RE scenario
Helpful	Post is informative, but no direct repair offered
No post	Post was not recommended by the tool
Irrelevant	Post is misleading for repairing the RE

TABLE II: Rating scales for patches and posts

checks, then it is added to the list of adapted script $\mathcal{T}'$.

Step 4. Generate Repair Patch. The goal of this step is to concretize the generalized edit script to developer's buggy code and generate a repair patch (lines 21–28). First, the same modular "adaptEditScript()" algorithm (Algorithm 2) is used to obtain the concretized script, $\mathbb{S}''$. Then, the fixes (edit operations) in $\mathbb{S}''$ are applied to developer's buggy code APG to get a modified version, $\mathcal{B}'_{\text{APG}}$. This APG is then validated for well-formedness with checks, such as no cycles. The $\mathcal{B}'_{\text{APG}}$ is then translated to a Java repair patch r. The patch r is then validated for parsability by running it through a Java parser, such as Eclipse JDT (line 25). If parsable, r is then added to the output list along with the SO post, s.

Step 5. Check Termination Criteria. The algorithm terminates if: (1) $\mathcal{K}$ repair patches are generated or (2) $\mathcal{Z}$ SO posts have been analyzed. Upon termination, output $\mathcal{O}$ is returned with up to $\mathcal{K}$ patch-post pairs, or only the top-post if a patch could not be generated (lines 30-35).

IV. EVALUATION

Our evaluation addresses the following research questions:

RQ1: How effective is MAESTRO in assisting fixing of REs?

RQ2: How does MAESTRO perform against other techniques?

RQ3: How effective are the key contributions in MAESTRO?

RQ4: What is the cost of maintaining the REP library?

A. Implementation

We implemented our approach in Java as a prototype tool named MAESTRO (**M**ine and **A**nalyz**E** **ST**ackoverflow to fix **R**untime excepti**O**ns). We used Eclipse JDT [29] to verify parsability of code snippets and build them into ASTs. We used the APTED tree edit distance algorithm [27], [28], [33] to compute the preliminary Q&A edit script, which we augmented with the "update" and "replace" edit operations. For the REP *mining* process, we empirically selected the termination criteria value as $U = 3$ (Section III-D) to stop the algorithm when it starts to repeatedly suggest low-quality SO posts. For the *fixing* algorithm discussed in Section III-F, we set the following values. The number of SO posts to be evaluated is set as $\mathcal{Z} = 15$. We chose this value with the real-time use case in mind, since analyzing an SO post for repair is an expensive operation. The value for the number of patches to generate is set to $\mathcal{K} = 3$ with the rationale that developers often only inspect a few top patches from a ranked list [34].

B. Datasets

SO Pool. We used the SO data dump released in released in March 2019 [35]. Based on the selection criteria discussed in Section III-B, this gave us a pool of 24,343 usable SO posts. The number of posts per RE type ranged from 3 to 13,415, with an average of 1281 posts and a median of 128 posts. Out of the total 115,009 code snippets evaluated, about 60% were found to be readily parsable. With our algorithm of converting unparsable snippets to parsable, we were able harvest a significant 16,799 more snippets, thereby improving the overall number of parsable snippets to 75%.

Library of REPs. Our REP library is comprised of 158 REPs, clustering 10,143 posts. MAESTRO, on average, suggested 17 SO posts per RE type to the human for processing, REPs were written for roughly half and the other half were marked as unviable. On average, 8 REPs per RE type resulted in the clustering of 533 posts, implying that the human is required to write only one REP per 67 SO posts. From the perspective of SO pool, writing REPs for 0.75% of the SO posts gives a coverage of over 50%, on average. This indicates that a small amount of human effort can yield large number of REPs.

Benchmark. For our experiments, we use our publicly released benchmark [17], [36]. It is an RE-specific benchmark, comprising 78 instances spanning 19 RE types collected from the top-500 Java projects on GitHub.

Defects4J [37], [38] is a popular benchmark used by state of the art APR techniques. However, it is not suitable for our use case for two reasons. First, Defects4J is a dataset for general-purpose bugs, with only a few instances (less than 10%) related to RE failures. Furthermore, many of these instances also require non-RE fixes to completely resolve the bug, which is out of scope for our technique. Second, the usable instances from Defects4J are limited in diversity, covering only a few common RE types, such as NullPointerException.

C. Evaluation Methodology

We evaluate MAESTRO and its baselines in terms of the number of useful artifacts that they produce, i.e., the ability to recommend a relevant SO post and correctness of the synthesized patch. We use manual examination for patch validation, which is a recommended protocol in the APR community [21], [39], [6], [20], [40], [41], [42], [43].

Participants and Protocol. To avoid any bias in the evaluation, we recruited two external participants to judge the artifacts. Our participants are software professionals with over 10 years of Java experience. For each of the five tools (MAESTRO and 4 baselines), the participants were presented with up to 3 patch-post pairs for each instance and asked to provide a rating for the best pair based on the metrics discussed below. To reduce bias in the experiment, we presented the results in randomized order with the tool names anonymized. The participants rated all of the 390 results (78 instances $\times$ 5 tools) independently. We then measured the inter-rater reliability using Cohen's Kappa [44]. The Kappa coefficient was $\kappa = 0.813$, indicating an *almost perfect agreement* between

RE type	#inst	C$\mathcal{P}$ R$\mathbb{P}$		A$\mathcal{P}$ R$\mathbb{P}$		I$\mathcal{P}$ R$\mathbb{P}$	I$\mathcal{P}$ I$\mathbb{P}$
		Top-1	*Top-3*	*Top-1*	*Top-3*		
ClassCastException	8	5	6	0	0	2	0
ConcurrentModificationException	8	0	0	6	6	2	0
IllegalArgumentException	8	1	1	0	0	2	5
IllegalStateException	8	2	2	0	0	0	6
IndexOutOfBoundsException	8	1	5	0	1	2	0
NullPointerException	8	3	5	0	0	3	0
ArithmeticException	4	4	4	0	0	0	0
NoSuchElementException	4	1	2	0	0	0	2
RejectedExecutionException	4	0	0	0	0	4	0
SecurityException	4	0	0	0	0	3	1
UnsupportedOperationException	4	1	1	0	0	2	1
EmptyStackException	2	1	2	0	0	0	0
NegativeArraySizeException	2	2	2	0	0	0	0
ArrayStoreException	1	0	0	0	0	1	0
BufferOverflowException	1	0	0	0	0	1	0
BufferUnderflowException	1	0	0	0	0	1	0
CMMException	1	0	0	0	0	1	0
IllegalMonitorStateException	1	0	1	0	0	0	0
MissingResourceException	1	0	0	0	0	1	0
Total	**78**	21	31	6	7	25	15

C$\mathcal{P}$: Correct patch, A$\mathcal{P}$: Almost Correct patch, I$\mathcal{P}$: No/Incorrect patch
R$\mathbb{P}$: Perfect/Helpful post, I$\mathbb{P}$: No/Irrelevant post, **#inst**: no. of instances
Average runtime per instance = 1 second (median = 0.6 sec)

TABLE III: Effectiveness Results of MAESTRO

the participants (ref. [45]: $\kappa > 0.81$). In cases of disagreement, the participants discussed the results with each other to reconcile the differences with one of the authors mediating the process [2], [46].

Metrics. We define two rating scales to evaluate the patch and the post, as shown in Table II. Our rating scales are largely inspired from the approach of Zimmermann et al. [47], [48], and follow the advice of Kitchenham et al. [49] to define a balanced scale and to exclude a "Don't Know" category if the participants are experts in the field.

The patch and post ratings implicate four possible outcomes of MAESTRO and its baselines: (1) a correct patch derived from a perfect post, (2) an almost correct patch from a relevant (perfect/helpful) post, (3) no/incorrect patch but a relevant post, and (4) no/incorrect patch and no/irrelevant post. The first three constitute a *useful artifact* in assisting developers for fixing REs, with (1) being the most desirable output.

D. RQ1: Effectiveness of MAESTRO

Table III shows the results of RQ1. Column "#inst" shows the number of instances per RE type. The remaining columns show details of the four possible outcomes of MAESTRO as

```
1 for (String str : new ArrayList<String>(listOfStr)) {    ✔
2     listOfStr.remove(/* object reference or index */);    ✘
3 }
```
(a) Stack Overflow post answer #11201224

```
1 - for (Order order : orders) {
2 + for (Order order : new ArrayList<Order>orders) {    ✔
3       orders.remove(order);
4 -     orders.remove(order);
4 +     orders.remove();    ✘
```
(b) MAESTRO patch for swagger-api/swagger-core

Fig. 7: Almost Correct Patch for ConcurrentModification

discussed in Section IV-C. For example, C$\mathcal{P}$, R$\mathbb{P}$, implies the first outcome: correct patch derived from a relevant post.

MAESTRO returned a correct repair patch at top-1 position in 27% of the cases, within the top-3 in 40% of the cases, and produced an overall useful artifact in 81% instances. MAESTRO was fast, requiring an average of 1 second (median = 0.6 sec) end-to-end on a 6-core MacOS laptop. Thus, MAESTRO can be effective in providing real-time assistance to developers for fixing REs.

In a diverse 14 out of 19 RE types, MAESTRO returned at least one useful artifact for every instance. MAESTRO was successful on REs, such as ClassCastException and IndexOutOfBoundsException, that comprised of commonly occurring exception scenarios and SO posts that recommended an accurate repair. REs, such as IllegalArgumentException and IllegalStateException, proved problematic since they are comprised of numerous scenarios that are not a part of our REP library, or scenarios that are very rare or application specific, and hence do not have representation on SO.

We investigated the results to understand why MAESTRO was not able to successfully generate a correct repair patch in all useful artifact cases. The first reason is that MAESTRO produced *almost correct patches* if the fix suggested SO posts was inadvertently malformed. An example of such a case is shown in Figure 7 for ConcurrentModificationException which is thrown when a Collection object (e.g., `List`) is structurally modified (e.g., `remove()`) during iteration. The SO post suggests a fix of creating a temporary copy of the list and using this new list for iteration (line 1 of Figure 7a). However on line 2, the SO answer inadvertently omits the argument of `remove()`. Therefore, when this fix is translated to the developer's buggy code (Figure 7b), it produces a patch that is one token away from the correct patch (i.e., missing argument `order` in `remove()`).

The second reason is when the fix in SO posts is insufficient in capturing all of the necessary changes to repair the developer's RE. For example, consider a NullPointerException thrown at line `v.m1().m2()`. The correct fix is to guard the failing line with two chained null checks: `if(v != null && v.m1() != null) {...}`. However, the SO post only suggests the first null check since its question code snippet discusses the failure of the kind `t.foo()`. Such incomplete patches are judged incorrect in our rating, since they are more than one token edit from the complete patch.

> **RQ1:** MAESTRO *demonstrates strong potential for providing real-time assistance in RE repair – it generates a correct repair patch at the top in 27% instances, within the top-3 in 40% of the cases, and some useful artifact in 81% of the cases, in only 1 second, on average.*

E. RQ2: Comparison with State of the Art

A direct comparison with the current state-of-the-art technique, QACrashFix [25], for generating repair patches using Stack Overflow is unfortunately not possible as QACrashFix is designed to generate patches for Android-related crashes using
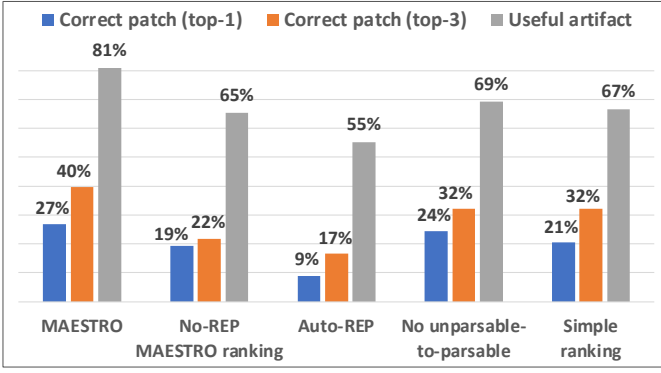
Fig. 8: MAESTRO vs. its baselines

```
1 - URL[] array = (URL[])urls.toArray();
2 + URL[] array = urls.toArray(new String[urls.size()]);
```

Fig. 9: Wrong patch by NO-REP-MAESTRO-RANKING for example in Figure 1 (Yellow shows problem in the patch)

Android-specific information, while our use case is of general-purpose REs. Instead, we implemented the core algorithm of QACrashFix in our use case, which is to find relevant posts using failure description and transform an SO fix directly to developer's code without the mediation of a REP.

We made the following changes to MAESTRO to create this version: (1) removed the mediation of REP from our fixing algorithm, (2) modified the function to find relevant SO posts to query a web search engine (e.g., Google) comprising of the RE type and failing line from developer's buggy code, since general REs do not give a Android-like description of the crash. However, in our experiments we found that this search for relevant posts performed rather poorly, ultimately generating a correct repair patch in only 5 instances. Hence, to give this baseline a starting advantage, we provided it with the ranked list of relevant SO posts from MAESTRO's indexing. We call this baseline "NO-REP-MAESTRO-RANKING".

Figure 8 shows the results for RQ2. NO-REP-MAESTRO-RANKING returns a correct patch at top-1 position in 19% of the cases, within top-3 in 22% cases, and an overall useful artifact in 65% cases. Thus, it falls significantly short of MAESTRO on all metrics, producing 45% fewer correct patches in the top-3 than MAESTRO. The primary reason for the performance deficit of NO-REP-MAESTRO-RANKING is that, even with the benefit of MAESTRO's post ranking, without the mediation of the REP, it is challenging to precisely map program elements from SO post's question code snippet to the developer's buggy code. For example, consider the patch generated by NO-REP-MAESTRO-RANKING shown in Figure 9 for our illustrative example (Section II). NO-REP-MAESTRO-RANKING simply transfers `new String[...]` from the SO post without any adaptation, leading to an imperfect patch.

> **RQ2:** *A baseline representing state-of-the-art SO-based patch synthesis produces 45% fewer correct patches than* MAESTRO, *even when provided with* MAESTRO*'s ranking.*
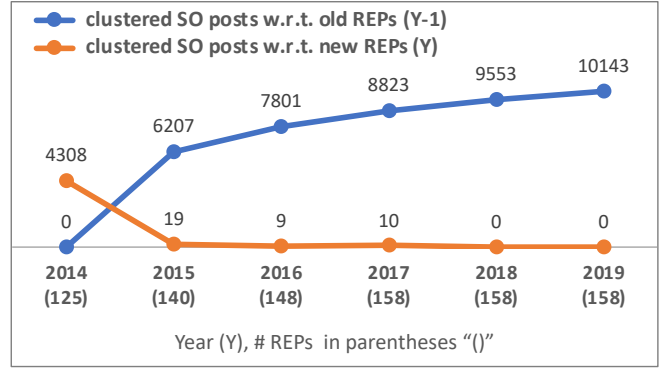

Fig. 10: Overhead of maintaining REP library over 5 years

F. RQ3: Key Contributions of MAESTRO

We create three baselines of MAESTRO. The first baseline, MAESTRO-NOUNPARSABLETOPARSABLE, measures the impact of our unparsable-to-parsable algorithm (Section III-B) by only using SO posts with readily available parsable code snippets. The second baseline, MAESTRO-SIMPLERANKING, assesses the importance of our REP-based indexing algorithm (Section III-E) by replacing it with a naive user-votes based ranking of SO posts and their answers. Finally, our third baseline, MAESTRO-AUTOREP, evaluates the importance of our hand-written REP-library by replacing it with the patterns auto-extracted using Q&A localization in MAESTRO's prior version [17].

Figure 8 shows the results for RQ3. The three baselines return a useful artifact in 55–69% of the cases (vs. 81% for MAESTRO), with a correct patch reported in top-3 in only 17–32% of the cases (vs. 40% for MAESTRO). Effectively, both the MAESTRO-NOUNPARSABLETOPARSABLE and MAESTRO-SIMPLERANKING baselines produce 20% fewer top-3 correct patches than MAESTRO, showing that unparsable code snippets or low-voted posts may at times exclusively contain the correct fixes. The MAESTRO-AUTOREP baseline performs the worst. This is because these auto-REPs are post-specific, and may even be sub-optimal if the answer code snippets are lengthy and/or non-specific. Such approximate patterns may be adequate for searching relevant posts (the target of [17]), but not for patch generation.

> **RQ3:** *Each of the three components contributes meaningfully to boosting the overall performance of* MAESTRO.

G. RQ4: Maintenance Cost for the REP Library

Figure 10 shows the potential cost of compiling and maintaining a REP library over a span of 5 years. For estimating this, we projected back from our REP library curated from the 2019 SO snapshot by tracking the chronology of the posts, and assuming an average writing time of 5 minutes per REP, calculated based on our 2019 effort. Based on this proposition, compiling the library of 125 REPs for the first time in 2014 is a seemingly *inexpensive task*, requiring roughly 10 person-hours. In 2015, almost 2,000 posts could be automatically clustered with the REPs from 2014, as they exhibited the same

9

exception-triggering patterns. This demonstrates that writing the REPs is indeed a *one-time investment*, with no extra cost thereafter. An *incremental addition* of 15 new REPs, clustering 19 SO posts, came at a significantly low overhead of only 1 person-hour. In subsequent years, the cost-effectiveness of REPs is evident from the increased clustering of new posts with old REPs, and significantly diminished human effort, with no new REP added after 2017, conceivably because most distinct, popular REPs had already been written.

> **RQ4:** *Compiling and maintaining the REP library is a low-cost, incremental, and one-time undertaking.*

H. Limitations

Repair scope. MAESTRO's current implementation can only generate fixes limited to intra-procedural exception scenarios. This design choice is motivated by the observation that RE scenarios generally tend to be concise and local in nature. However, to expand the scope, we plan to extend MAESTRO to handle inter-procedural analysis in the future.

Usefulness of artifacts. Judging repair patches and SO posts is a subjective task that is performed manually, posing a threat to construct validity. Manual judgement for correctness of patches is an establishednorm in the APR community (e.g., [6], [21], [43]). Nonetheless, to minimize this threat, we clearly defined the criteria for our rating scales, consistent with prior work [47], [48], [49], [17]. Further, to further reduce bias, we recruited two *external participants* to evaluate independently. We then used Cohen's Kappa [44] to measure inter-rater reliability, which showed *almost perfect agreement* among the participants. Disagreements were reconciled via discussion – consistent with previous work [46], [2], [17].

Mining REPs MAESTRO employs a semi-automated process for extracting REPs from SO posts. As we show through experiments in RQ3, automatically extracting REPs is challenged by the sub-optimal Q&A localization. To make the mining process efficient, MAESTRO only suggests a small number of SO posts to the human that are likely to represent diverse REPs, and also aids the writing task by presenting an auto-extracted approximate REP. Further, compiling a library of REPs through our mining process is a one-time undertaking that could be performed incrementally.

V. RELATED WORK

APR using SO. QACrashFix [25] is closely related to our work. QACrashFix is a generate-and-validate APR technique for fixing Android-related crash bugs. As shown in Section IV, QACrashFix does not work well for our use case of fixing general-purpose REs for two reasons. First, QACrashFix is limited in finding relevant SO posts for general REs, since they often do not have a crash description that QACrashFix uses to do the search. Second, even when provided with MAESTRO's SO posts ranking, it fails to accurately generate a correct patch in 45% of the cases. Different from our use case of generating repairs from SO posts, another APR technique, SOFix [50], manually extracts a set of repair templates from SO.

Assisting debugging of REs. FuzzyCatch [7] recommends Android-related exception handling (`try-catch` blocks). Several techniques provide tailored repair solutions for only the common REs, such as NullPointerException (NPE). Sinha et al. [8] use stack traces to locate and create a patch, NPEFix [9] uses NPE-specific heuristics, VFix [10] uses data and control-flow analysis to prune the NPE repair space, Getafix [11] and Genesis [12] learn fix patterns from human-written patches, and Droix [13] uses search-based algorithms. Another research isolates and recovers from runtime errors [14], [15], yet another focuses on the automatic generation of test oracles for REs [16]. By contrast, our work presents a general-purpose repair approach that applies to a diverse set of REs by leveraging crowd intelligence encoded in forums (e.g., SO).

Automatic Program Repair (APR). We share the patch-generation goal of traditional APR techniques. Generate-and-validate APR techniques, such as SketchFix [51], Kali [40], SPR [42], Elixir [6], Hercules [52], Angelix [20], and Gen-Prog [53], explore a search space of manually crafted repair transformations that are tried in succession until a plausible repair is found. Another group of techniques, such as CoCoNut [21], Sequencer [54], and DLFix [55], use deep learning to auto-learn the repair transformations. Yet other techniques, such as Phoenix [39] and Refazer [56], use programming by example to learn repair strategies. Unlike us, such traditional APR techniques typically rely on a dynamic patching oracle (e.g., test suite) and thereby involve long running times, making them unfit for our use case of real-time assistance when patching oracles may not even be available.

Mining developer forums (e.g., SO). The previous version of MAESTRO [17], [57] recommends relevant SO posts for manual fixing of REs, while Prompter [58], [59] and Libra [60] suggest posts to assist during implementation. ExampleStack [61] shows examples of SO adaptations. FaCoY [62] performs code-to-code search. Nagy et al. [63] mine common SQL error patterns. AnswerBot [64] and Crokage [65] summarize SO answers. Chen et al. [3] use SO to fault localize code and suggest posts. SEQUER [66] reformulates user queries to find posts. CSnippEx [31] makes non-compilable SO code snippets compilable. By contrast, such techniques are not related to our use case of SO-based patching.

VI. CONCLUSION

We extend our prior work, MAESTRO, by adding real-time patching support for fixing REs using SO posts. MAESTRO exploits a library of REPs semi-automatically mined from SO posts through a one-time, incremental process. A REP represents an RE pattern and is used to mediate each of the key steps: indexing SO posts, retrieving a relevant post matching the RE scenario exhibited by the developer's code, and finally adapting the post-suggested answer to fix developer's buggy code. An evaluation on a published benchmark of 78 RE instances showed that MAESTRO generated a correct repair patch at top-1 in 27% of the cases, within the top-3 in 40% cases, and an overall useful artifact in 81% cases. Further, MAESTRO only required 1 second, on average, per instance.

REFERENCES

[1] C. Sadowski, K. T. Stolee, and S. Elbaum, "How developers search for code: A case study," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 191–201. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786855

[2] S. Luan, D. Yang, C. Barnaby, K. Sen, and S. Chandra, "Aroma: Code recommendation via structural code search," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019.

[3] F. Chen and S. Kim, "Crowd debugging," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. Association for Computing Machinery, 2015, p. 320–332.

[4] R. Abdalkareem, E. Shihab, and J. Rilling, "What do developers use the crowd for? a study using stack overflow," *IEEE Software*, vol. 34, no. 2, pp. 53–60, 2017.

[5] S. Beyer, C. Macho, M. Di Penta, and M. Pinzger, "What kind of questions do developers ask on stack overflow? a comparison of automated approaches to classify posts into question categories," *Empirical Software Engineering*, vol. 25, 05 2020.

[6] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object oriented program repair," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. IEEE Press, 2017, p. 648–659.

[7] T. Nguyen, P. Vu, and T. Nguyen, "Code recommendation for exception handling," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1027–1038. [Online]. Available: https://doi.org/10.1145/3368089.3409690

[8] S. Sinha, H. Shah, C. Görg, S. Jiang, M. Kim, and M. J. Harrold, "Fault localization and repair for java runtime exceptions," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09. New York, NY, USA: ACM, 2009, pp. 153–164. [Online]. Available: http://doi.acm.org/10.1145/1572272.1572291

[9] B. Cornu, T. Durieux, L. Seinturier, and M. Monperrus, "Npefix: Automatic runtime repair of null pointer exceptions in java," Arxiv, Tech. Rep. 1512.07423, 2015. [Online]. Available: https://arxiv.org/pdf/1512.07423.pdf

[10] X. Xu, Y. Sui, H. Yan, and J. Xue, "Vfix: Value-flow-guided precise program repair for null pointer dereferences," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 512–523. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00063

[11] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360585

[12] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 727–739. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106253

[13] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, "Repairing crashes in android apps," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 187–198. [Online]. Available: https://doi.org/10.1145/3180155.3180243

[14] F. Long, S. Sidiroglou-Douskos, and M. Rinard, "Automatic runtime error repair and containment via recovery shepherding," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 227–238. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594337

[15] T. Gu, C. Sun, X. Ma, J. Lü, and Z. Su, "Automatic runtime recovery via error handler synthesis," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 684–695. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970360

[16] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè, "Automatic generation of oracles for exceptional behaviors," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 213–224. [Online]. Available: https://doi.org/10.1145/2931037.2931061

[17] S. Mahajan, N. Abolhassani, and M. R. Prasad, "Recommending stack overflow posts for fixing runtime exceptions using failure scenario matching," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1052–1064. [Online]. Available: https://doi.org/10.1145/3368089.3409764

[18] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," 2009, pp. 364–374.

[19] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," 2013, pp. 802–811.

[20] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 691–701. [Online]. Available: https://doi.org/10.1145/2884781.2884807

[21] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: Combining context-aware neural translation models using ensemble for program repair," ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 101–114. [Online]. Available: https://doi.org/10.1145/3395363.3397369

[22] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, p. 56–65, Nov. 2019.

[23] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Trans. Softw. Eng.*, vol. 45, no. 1, p. 34–67, Jan. 2019.

[24] Y. Wu, S. Wang, C.-P. Bezemer, and K. Inoue, "How do developers utilize source code from stack overflow?" *Empirical Software Engineering*, vol. 24, no. 2, pp. 637–673, Apr 2019. [Online]. Available: https://doi.org/10.1007/s10664-018-9634-5

[25] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei, "Fixing recurring crash bugs via analyzing q&a sites (t)," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 307–318. [Online]. Available: https://doi.org/10.1109/ASE.2015.81

[26] JD-GUI, "JD-GUI project at commit 0e504af," 2021. [Online]. Available: https://github.com/java-decompiler/jd-gui

[27] M. Pawlik and N. Augsten, "Efficient computation of the tree edit distance," *ACM Trans. Database Syst.*, vol. 40, no. 1, pp. 3:1–3:40, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2699485

[28] ——, "Tree edit distance: Robust and memory-efficient," *Information Systems*, vol. 56, pp. 157 – 173, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0306437915001611

[29] E. JDT, "Eclipse JDT," 2021. [Online]. Available: https://www.eclipse.org/jdt/

[30] D. Yang, A. Hussain, and C. V. Lopes, "From query to usable code: An analysis of stack overflow code snippets," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 391–402. [Online]. Available: https://doi.org/10.1145/2901739.2901767

[31] V. Terragni, Y. Liu, and S.-C. Cheung, "Csnippex: Automated synthesis of compilable code snippets from q&a sites," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 118–129. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931058

[32] Java, "JavaDocs." [Online]. Available: https://docs.oracle.com/javase/8/docs/api

[33] M. Pawlik and N. Augsten, "APTED algorithm for the Tree Edit Distance Implemenataion." [Online]. Available: https://github.com/DatabaseGroup/apted

[34] C. Parnin and A. Orso, "Are Automated Debugging Techniques Actually Helping Programmers?" in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2011)*, Toronto, Canada, July 2011, pp. 199–209.

[35] I. Stack Exchange, "Stack Overflow Dump," 2019. [Online]. Available: https://archive.org/details/stackexchange

[36] S. M. et al., "Maestro Evaluation Data." [Online]. Available: https://doi.org/10.6084/m9.figshare.11948619

[37] R. Just, D. Jalali, and M. D. Ernst, "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*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 437–440. [Online]. Available: https://doi.org/10.1145/2610384.2628055

[38] R. Just, "Defects4J – version 2.0.0." [Online]. Available: https://github.com/rjust/defects4j

[39] R. Bavishi, H. Yoshida, and M. R. Prasad, "Phoenix: Automated data-driven synthesis of repairs for static analysis violations," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 613–624. [Online]. Available: https://doi.org/10.1145/3338906.3338952

[40] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 24–36. [Online]. Available: https://doi.org/10.1145/2771783.2771791

[41] T. Durieux, M. Martinez, M. Monperrus, R. Sommerard, and J. Xuan, "Automatic repair of real bugs: An experience report on the defects4j dataset," *CoRR*, vol. abs/1505.07002, 2015. [Online]. Available: http://arxiv.org/abs/1505.07002

[42] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 166–178. [Online]. Available: https://doi.org/10.1145/2786805.2786811

[43] K. Liu, L. Li, A. Koyuncu, D. Kim, Z. Liu, J. Klein, and T. F. Bissyandé, "A critical review on the evaluation of automated program repair systems," *J. Syst. Softw.*, vol. 171, p. 110817, 2021. [Online]. Available: https://doi.org/10.1016/j.jss.2020.110817

[44] J. Cohen, "A Coefficient of Agreement for Nominal Scales," *Educational and Psychological Measurement*, vol. 20, no. 1, p. 37, 1960.

[45] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, 1977.

[46] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, Aug. 2017.

[47] D. Lo, N. Nagappan, and T. Zimmermann, "How practitioners perceive the relevance of software engineering research," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, pp. 415–425. [Online]. Available: https://doi.org/10.1145/2786805.2786809

[48] A. Begel and T. Zimmermann, "Analyze this! 145 questions for data scientists in software engineering," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 12–23. [Online]. Available: https://doi.org/10.1145/2568225.2568233

[49] B. A. Kitchenham and S. L. Pfleeger, "Personal opinion surveys," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. I. Sjøberg, Eds. Springer London, 2008, pp. 63–92.

[50] X. Liu and H. Zhong, "Mining stackoverflow for program repair," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2018, pp. 118–129.

[51] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Sketchfix: A tool for automated program repair approach using lazy candidate generation," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 888–891. [Online]. Available: https://doi.org/10.1145/3236024.3264600

[52] S. Saha, R. K. Saha, and M. R. Prasad, "Harnessing evolution for multi-hunk program repair," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 13–24. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00020

[53] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Press, 2012, p. 3–13.

[54] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. PP, pp. 1–1, 09 2019.

[55] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 602–614. [Online]. Available: https://doi.org/10.1145/3377811.3380345

[56] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning syntactic program transformations from examples," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. IEEE Press, 2017, p. 404–415. [Online]. Available: https://doi.org/10.1109/ICSE.2017.44

[57] Y. Kimura, T. Akazaki, S. Kikuchi, S. Mahajan, and M. R. Prasad, "Q&A MAESTRO: Q&A Post Recommendation for Fixing Java Runtime Exceptions," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 1227–1231.

[58] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Mining stackoverflow to turn the ide into a self-confident programming prompter," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 102–111. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597077

[59] L. Ponzanelli, A. Bacchelli, and M. Lanza, "Leveraging crowd knowledge for software comprehension and development," in *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, ser. CSMR '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 57–66. [Online]. Available: http://dx.doi.org/10.1109/CSMR.2013.16

[60] L. Ponzanelli, S. Scalabrino, G. Bavota, A. Mocci, R. Oliveto, M. Di Penta, and M. Lanza, "Supporting software developers with a holistic recommender system," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 94–105. [Online]. Available: https://doi.org/10.1109/ICSE.2017.17

[61] T. Zhang, D. Yang, C. Lopes, and M. Kim, "Analyzing and supporting adaptation of online code examples," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 316–327. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00046

[62] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. L. Traon, "Facoy: A code-to-code search engine," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 946–957. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180187

[63] C. Nagy and A. Cleve, "Mining stack overflow for discovering error patterns in sql queries," in *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ser. ICSME '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 516–520. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2015.7332505

[64] B. Xu, Z. Xing, X. Xia, and D. Lo, "Answerbot: Automated generation of answer summary to developersundefined technical questions," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. IEEE Press, 2017, p. 706–716.

[65] R. F. G. Silva, C. K. Roy, M. M. Rahman, K. A. Schneider, K. Paixao, and M. de Almeida Maia, "Recommending comprehensive solutions for programming tasks by mining crowd knowledge," in *Proceedings of the 27th International Conference on Program Comprehension*, ser. ICPC '19. IEEE Press, 2019, pp. 358–368. [Online]. Available: https://doi.org/10.1109/ICPC.2019.00054

[66] K. Cao, C. Chen, S. Baltes, C. Treude, and X. Chen, "Automated query reformulation for efficient search based on query logs from stack overflow," 2021.