Available online at www.sciencedirect.com

ScienceDirect

ELSEVIER

The Journal of Systems and Software 00 (2018) 1–21

JSS

Mining Repair Model for Exception-Related Bug

Hao Zhong, Hong Mei

Department of Computer Science and Engineering, Shanghai Jiao Tong University, China

Abstract

It has long been a hot research topic to fix bugs. As a common practice, researchers propose approaches for specific bugs, and their approaches typically are limited in handling the variety among bugs. Recently, researchers start to explore *automatic program repair*. With predefined repair operators and test cases, test-based repair approaches use search algorithms to generate patches for a bug, until a patch passes all the test cases. To improve the effectiveness to generate patches, Martinez and Monperrus [43] proposed an approach that mines repair models from past fixes. Although their approach produces positive results, we argue that it can be feasible to further improve their approach, if we mine repair models for bug categories, instead of all bugs. However, the benefits are still unclear, since existing benchmarks do not classify bug into categories and existing approaches cannot mine repair models for bug categories. In this paper, we implement a tool, called ExFi, that classifies bugs into categories based on their related exceptions. With its support, we construct a benchmark, in which bug categories are marked. Furthermore, we propose an approach, called MiMo, that mines a repair model for each exception. We compare the general repair model with our mined repair models. Our results show that our mined models are all significantly different from the general model. Outside of the projects where our models are mined, we selected 59 real bugs. For each bug, we used our models and the general model to generate correct repair shapes for these bugs. The results show that for 43 out of 59 bugs, our models found faster a correct shape than the general repair model [43], and for 5 bugs, our models were able to find correct shapes that were not found by the compared model.

Keywords: repair models; exception-related bugs; benchmark

1. Introduction

It has long been a hot research topic to detect and fix bugs automatically, since it is time-consuming and error-prone to fix bugs. As a common practice, researchers analyze the nature of their target bugs, and then propose corresponding treating techniques. The research in this line has difficulty in handling the variety in bugs, and fails to detect bugs if they are different from analyzed bugs. Instead, another research line is to explore automatic approaches that fix all bugs, referred to as *automatic program repair*. In this research line, under the guidance of search algorithms, a typical approach (*e.g.*, [77]) mutates the faulty code with predefined repair operators, until a mutation passes all the test cases. This research line produces promising results (*e.g.*, [28]), and a recent study [93] shows that at least twenty percents of bugs can be fixed in this way. However, the positive results also attract criticisms (*e.g.*, [58, 48]), partially due to the overfitting problem when validating correct patches with test cases.

Martinez and Monperrus [43] propose an approach that mines repair models from past bug fixes. When mining repair models, they first use ChangeDistiller [15] to extract a set of repair actions for each bug fix, and then calculate the probability distribution of repair actions. From 62,179 commits of 14 projects, they mined two repair models such as the CT (Change Type) and the CTET (Change Type Entity Type) models, with different granularity. For example, a part of the CTET model is as follows:

```
inserting method invocations   6.9%
inserting if statements        6.6%
updating method invocations    6.4%
...
```

According to the above model, the probability of inserting method invocations is 6.9%. Martinez and Monperrus [43] use repair models to guide generating repair shapes. A repair shape is an unordered tuple of repair actions, and is a similar concept of repair hints that are mined by Kaleeswaran *et al.* [23].

Martinez and Monperrus [43] shows that their mined repair models are useful to reduce the effort of generating repair shapes. Despite of their positive results, we argue that it can be more promising to mine a repair model for each category of bugs, due to the following three considerations. First, it is easier to repair a specific type of bugs than to repair general bugs. Second, Liang *et al.* [31] show that different programming rules can exist in the same repository. Mining repair models for bug categories can reduce the interference of violating different programming rules. Finally, it is easier to build the connection between the symptoms of bugs and their fixes. With such connections, researchers may design more effective guidance algorithms for fixing bugs. However, to compare such models, we have to overcome the following challenges:
Challenge 1. It is challenging to collect bug categories for analysis. Although researchers [22, 30, 69] have proposed several benchmarks, these benchmarks do not present bug categories. It can take much effort, if we manually identify bug categories from benchmarks.
Challenge 2. It is challenging to compare the general repair model with repair models for bug categories. Martinez and Monperrus [43] admit that their generation algorithm is a guided random search. As a random-based approach can produce different results in executions, it is not conclusive to compare two models with only several executions.

Our contributions. In this paper, we implement ExFi that automatically classifies more than one thousand bug fixes into categories (Section 2), and implement MiMo that mines repair models for each category (Section 4). As we resolved the above challenges, we conduct the first empirical study that compares the general repair model with repair models that are mined for bug categories. This paper makes the following contributions:

- The first benchmark for bug categories. Researchers [22, 30, 69] have proposed several benchmarks for automatic program repair, which are widely used in the research community (*e.g.*, [41]). We implement ExFi that builds mappings between bugs and their related exceptions. With its support, we construct the first benchmark, in which bugs are classified into categories. Comparing with existing benchmarks, our benchmark is marked with bug categories, and it is large scale, since it contains more than one thousand bug fixes.
- A tool, called MiMo (**Mi**ning repair **Mo**dels for exception-related bugs), that mines a repair model for each category of bugs, *i.e.*, bugs whose reports mention the same exception. MiMo first builds the links between exceptions and their modifications, and then mines repair models for each exception. In total, MiMo mines 21 repair models from our benchmark.
- Two comparisons between the existing general repair model and the top eight mined repair models. The result of the first comparison shows that the action probabilities of all our mined repair models are significantly different from the action probabilities of the general repair model. The result of the second comparison shows that our repair models are more effective in generating correct repair shapes than the baseline general repair model, and our models spend less time, for 43 out of 59 real bugs.

The rest is organized as follows. Section 2 introduces our benchmark. Section 3 introduces the general model of Martinez and Monperrus [43]. Section 4 presents MiMo that mines models for exception-related bugs. Section 5 presents the general model and our mined models. Section 6 presents our comparison results. Section 7 discusses issues of our study. Section 8 presents related work. Section 9 concludes this paper.

2. Benchmark

We consider that a bug is related to an exception, if its bug report mentions the exception, and we consider bugs that are related to the same exception as a category of bugs. In this paper, we select exception-related bugs to construct our benchmark, since (1) exception-related bugs cover many bug categories (*e.g.*, IO bugs in `IOException`, memory bugs in `NullPointerException`, and network bugs in `SocketException`); (2) the previous

Table 1: Importance of Exception-Related Bugs

Name	**Exception-related Bug**					**Bug**					**%**
	B	**C**	**Ma**	**Mi**	**T**	**B**	**C**	**Ma**	**Mi**	**T**	
Aries	2	5	85	8	2	5	18	403	63	8	20.5%
Cassandra	8	25	344	237	35	29	94	1138	1034	232	25.7%
Derby	9	27	463	164	14	27	80	1426	790	110	27.8%
Lucene/Solr	18	18	315	119	15	101	104	1861	911	168	15.4%
Mahout	1	4	54	40	6	5	12	262	139	39	23.0%
Total	38	79	1261	568	72	167	308	5090	2937	557	22.3%
Percentage	1.9%	3.9%	62.5%	28.1%	3.6%	1.8%	3.4%	56.2%	32.4%	6.1%	

B: Blocker; C: Critical; Ma: Major; Mi: Minor; T: Trivial.

work has used exceptions for bug clustering [10] and to guide fault localization [65]; and (3) it needs much expertise to handle the variety in fixing bugs, even if the same exception is thrown. As many factors can lead to a thrown exception, wrapping methods with `try-catch` statements is typically insufficient to repair exception-related bugs. In literature, researchers have proposed many approaches that detect and repair exception-related bugs. For example, to reduce `OutOfMemoryError`, Thummalapenta and Xie [71] propose an approach that cleans up resources in `catch` statements. Here, we agree that it is feasible to classify bugs into categories based on other criteria, and researchers can construct their benchmarks based on their own criteria other than exceptions.

2.1. Exception-Related Bug

In this paper, we refer to exception-related bugs as bugs whose reports mention exceptions. To analyze the importance of exception-related bugs, we queried the issue trackers of six popular Apache Java projects in February 2014, and Table 1 shows the results. Column "Exception-related Bug" lists the number of exception-related bugs with respect to their priorities. We queried the projects with the keywords such as "Exception" and "Error", since Java implements three types of exceptions such as unchecked exceptions (*e.g.*, `NumberFormatException`), checked exceptions (*e.g.*, `ClassNotFoundException`), and errors (*e.g.*, `OutOfMemoryError`) [8]. Here, we required the first capitalized letters to reduce irrelevant bug reports, and counted only fixed bugs to reduce superficial bugs, since a bug (*e.g.*, CASSANDRA-8296[1]) can be resolved as not a problem. Column "Bug" lists the number of total fixed bugs with respect to their priorities. Column "%" lists percentage of exception-related bugs among all the fixed bugs. The result shows that exception-related bugs are many, and account for more than 20% of the fixed bugs in total. As a comparison, although concurrency bugs are important, Lin *et al.* [33] show that only about five percents of bugs are related to concurrency issues. Row "Percentage" lists percentage of the corresponding bugs over the total bugs. In total, exception-related bugs are important, since more exception-related bugs are marked as blocker (1.9% vs 1.8%), critical (3.9% vs 3.4%), or major (62.5% vs 56.2%) than all the bugs.

Exception-related bugs can be difficult to be repaired. First, programming languages may define many exceptions. For example, as shown in the J2SE's document[2], `Exception` has more than seventy direct subclasses. Second, programming languages may allow customized exceptions. Finally, the symptoms of exceptions can be quite different.

In the same category of bugs, the symptom is straightforward, and we can infer corresponding fixes. For example, as shown in its document[3], `NullPointerException` is thrown when a program attempts to use a `null` object. Based on this domain knowledge, it is feasible to fix related bugs by checking whether objects are `null`. In particular, Figures 1a and 1b show two fixes from Lucene[4] and Cassandra[5]. Although the two bugs are from different projects and are fixed by different programmers, programmers use the same pattern to fix the two bugs, *i.e.*, adding an `if` statement to check a value, and a `return` statement to exit the method, if such a value is `null`. Their repair shapes are similar, and they share two repair actions, *i.e.*, inserting an `if` statement and inserting a `return` statement. Here,

[1]`https://issues.apache.org/jira/browse/CASSANDRA-8296`
[2]`http://docs.oracle.com/javase/7/docs/api/java/lang/Exception.html`
[3]`http://docs.oracle.com/javase/7/docs/api/java/lang/NullPointerException.html`
[4]`https://issues.apache.org/jira/browse/LUCENE-1510`
[5]`https://issues.apache.org/jira/browse/CASSANDRA-2377`

```
+if (norms == null) {
+  return new byte[0]; //to do...
+}
```

(a) A patch in Lucene

```
+if (session == null){
+   logger.warn(...);
+   return;
+}
```

(b) A patch in Cassandra

```
-} else if (!clause.getField().equals(field)) {
+} else if (clause.getField() != null &&
           !clause.getField().equals(field)) {
```

(c) The other way to fix bugs

Figure 1: The different ways to fix bugs that throw `NullPointerException`

Martinez and Monperrus [43] define a repair action as a modification on a code element. Still, due the complexity of fixing bugs, a pattern is insufficient to describe all bug fixes. For example, Figure 1c shows the other way to fix buggy code that throws the same exception[6]. The condition of an `if` statement is modified to avoid the exception. Its repair shape is updating an `if` statement. Furthermore, different from above two patterns, Cornu *et al.* [7] propose another fix pattern that catches unanticipated exceptions. The observations show that it needs different strategies to fix even a simple exception, which highlights the importance of automatic program repair.

2.2. ExFi

Many projects provide source code repositories that record all the commits (*e.g.*, bug fixes, improvements, and new features), and a commit consists of a message and a set of modified files. The message of a commit often includes an issue key for tracking issues. For example, in the code repository of Aries, the message of a commit is "ARIES-1269 Add blueprint maven plugin". In this message, the issue key is "ARIES-1269". In some projects, the practice is strictly complied. Existing studies [80, 3] show that most commits in Apache projects follow the practice, so their approach achieves similar precisions and recalls with the issue key heuristic, when they rebuild the links between bug reports and commits for Apache projects. For each exception-related bug report, ExFi uses its issue key to retrieve related commits. It is a simple heuristic, but it works well when the practice is followed. For those projects that do not follow this practice, Tian *et al.* [72] and Wu *et al.* [80] propose approaches that uses various features (*e.g.*, the number of added `if` statements) to identify bug fixes from commits. It is feasible to collect more bug fixes for analysis with the support of their approaches.

It is difficult to identify specific bug fixes (*e.g.*, exception-related bugs). Tufano *et al.* [73] analyzed commits of 100 Apache projects, and they found that only 38% of the total commits are compilable. Tufona *et al.* [73] released their analysis results on their website[7]. In particular, 58% of such cases are caused by the resolution of dependencies. Without such dependencies, commits are not compilable, even if their code does not have syntax errors. As bug fixes are commits that repair bugs, many bug fixes are not compilable. Although researchers (*e.g.*, [76, 56]) proposed approaches that can determine whether a bug is caused by exceptions, it is expensive to adapt such techniques to analyze bug fixes, since many bug fixes are not compilable. Although a recent approach [94] enables static analysis on partial programs, it needs dynamic analysis to determine related exceptions. Hassan *et al.* [18] repair configuration files to fix complication errors in commits, but can fix only half of commits.

Instead of commit source code, ExFi analyzes bug reports to identify bug categories. A bug report can be superficial, since programmers can resolve the report as other types of issues (*e.g.*, "Improvement"). To filter superficial bug reports, ExFi selects only fixed bugs. ExFi then queries fixed bug reports with the keywords such as "Error" and "Exception". We notice that bug reports can mention exceptions or errors, but do not describe their names. It is difficult to

[6] `https://issues.apache.org/jira/browse/LUCENE-5450`
[7] `http://www.cs.wm.edu/semeru/data/breaking-changes`

Table 2: Exception-related Bugs.

Name	Identified bug	Exception-related bug	Percent	Exception	Unique exception	Average file
Aries	102	102	100.0%	159	62	1.45
Cassandra	282	649	43.5%	461	84	1.55
Derby	662	677	97.8%	1,004	115	1.45
Lucene/Solr	343	485	70.7%	465	81	1.92
Mahout	94	105	89.5%	125	39	1.26
Total	1,483	2,018	73.5%	2,214	257	1.57

C: exception-related bugs whose commits are identified; E: exception-related bugs; Ex.: mentioned exceptions; U.Ex: unique mentioned exceptions.

link such bug reports to specific exceptions or errors. To filter these bug reports, ExFi requires that the first letters of mentioned exceptions/errors shall be capitalized. For example, if a bug report mentions `OutOfMemoryError`, ExFi considers this bug report, since it mentions an exception name whose first letter is capitalized. Furthermore, ExFi links this bug report to `OutOfMemoryError`.

2.3. Dataset

Table 2 lists our dataset. Column "Name" lists names of projects. Aries is an OSGi application programming tool. Cassandra is a distributed database management system. Derby is a relational database. Lucene is an information retrieval library, and Solr is an enterprise search platform that is built on Lucene. Lucene and Solr share the same source code repository, so we have to put their results into a row. Mahout is a machine learning library. All the six projects are from Apache and in Java. We collected their bug reports and fixes in February 2014. Column "Identified bug" lists the number of exception-related bugs whose commits are identified by ExFi. Column "Exception-related bug" lists the total number of the exception-related bugs. Column "Percent" lists the percents of identified bugs. The result shows that ExFi identified bug fixes for most exception-related bugs. The remaining bug fixes are not identified, since their issue number is not found in commit messages. Here, Cassandra changed its source code repository from SVN[8] to Git[9] in December 2011. ExFi retrieved commits from only SVN repositories, so the commits after December 2011 are not extracted. If a bug report is reported after December 2011, it is infeasible to identify its corresponding bug fix from our extracted commits. As a result, the ratio of Cassandra is low. Column "Exception" lists the number of mentioned exceptions. Among them, column "Unique exception" lists the number of unique ones. Its total number is smaller than its sum, since some exceptions are mentioned in multiple projects. Column "Average file" lists the averages of modified source files per bug. More details of the benchmark are presented in its website[10].

3. The General Repair Model

Martinez and Monperrus [43] define a repair action as a modification on a code element, and a repair model as a set of repair actions. For example, the repair model of Weimer *et al.* [77] has three repair actions such as deleting a statement, inserting a statement taken from the software, and swapping two statements. The repair model of an approach determines its repair capability, and it needs much expertise to design effective repair models. Martinez and Monperrus [43] propose an approach that mines repair models from past manual fixes. From 14 open source Java projects, they collected 62,179 commits that fix bugs. From each commit, they extracted repair actions with the support of ChangeDistiller [15], and calculated two repair models such as CT and CTET with different granularity (see Section 5.1 for details). Here, their mined repair model is the probability distribution of repair actions. The appendix of Martinez and Monperrus [42] provides the complete distributions of the CT and CTET models. In their evaluation, they use mined repair models to guide generating repair shapes. Martinez and Monperrus [43] define a repair shape as an unordered tuple of repair actions, and the shape space as all possible combinations of repair actions. A generated repair shape is more abstract than a patch and cannot be executed. As a result, it is infeasible to guide the

[8]`https://svn.apache.org/repos/asf/cassandra/`
[9]`https://git-wip-us.apache.org/repos/asf?p=cassandra.git`
[10]`https://github.com/drzhonghao/exception.bugs`

search with fitness functions. Martinez and Monperrus [43] use weighted random search for repair actions, and the weights are based on the probabilities of repair actions dictated by their models (see Section 7 for more discussions). With repair models, automatic program repair can generate fixes more effectively.

4. MiMo

MiMo has two major steps such as identifying exception-method mappings (Section 4.1), and mining a repair model for each exception (Section 4.2).

4.1. Identifying Exception-method Mapping

The first step of MiMo is to build the mappings between mentioned exceptions and modified methods. Some bug reports have stack traces [78]. From such bug reports, MiMo extracts stack traces by parsing their contents. Zhong and Su [92] detect errors in API documents, and their approach includes a component that extracts code samples from API documents. MiMo reuses the component, and employs a natural language checker [47] to detect language errors (*e.g.*, spelling errors, grammar errors, style errors, and semantic errors). The error ratio is defined as follows:

$$error\_ratio = \frac{|errors|}{|words|} \tag{1}$$

As stack traces do not follow the grammars of natural languages, they have much higher error ratios than natural language sentences. MiMo uses the error ratio to identify stack traces. Here, although code samples can be identified as stack traces, it is easy to distinguish them, since stack traces follow a different format from code samples. As a result, we do not need a more advanced technique (*e.g.*, [2]) to identify stack traces. In a stack trace, an exception is followed by a sequence of methods. For example, a bug report of Cassandra[11] mentions three exceptions, and the reporter presents the stack trace as follows:

```
Looks related to ...
java.util.concurrent.ExecutionException:...
at ...FutureTask$Sync.innerGet(...)
...
Caused by: java.lang.RuntimeException...
at ...WrappedRunnable.run(...)
...2 more
Caused by: java.lang.AssertionError...
at ...CommitLog.discardCompletedSegmentsInternal(...)
...1 more
```

MiMo links a thrown exception to its follow-up methods, if such methods are modified. For example, in the above stack trace, `AssertionError` has a follow-up method, `discardCompletedSegmentsInternal()`. This bug report has an attached patch[12]. The patch shows that ten methods are modified, and one of the them is the `discardCompletedSegmentsInternal()` method. In particular, a line of the method is deleted:

```
- assert context.position >= context.getSegment();
```

The fixed code does not throw the exception any more, since in the method, the `assert` statement is deleted. As only this method appears after `AssertionError`, MiMo links the `discardCompletedSegmentsInternal()` method to `AssertionError`. We inspected the other nine modified methods, and we found that all the methods are related to refactoring. In this paper, we use $\mathcal{M}_{st}$ to denote mappings that are identified by stack traces.

Schroter *et al.* [62] show that in about 40% of bug reports with stack traces, programmers do not modify any methods that appear in stack traces to fix bugs. For example, we notice that some fixes do not modify any source files (see Section 5.2 for examples). In this paper, we use $\mathcal{M}_{ns}$ to denote mappings where no Java source files are modified to fix bugs. Some remaining mappings are one-to-one, since only an exception is mentioned, and only a method is modified. For example, a bug report of Aries[13] is related to `NullPointerException`, and programmers modified a method to fix the thrown exception:

[11]https://issues.apache.org/jira/browse/CASSANDRA-1330
[12]https://issues.apache.org/jira/secure/attachment/12456141/1330.txt
[13]https://issues.apache.org/jira/browse/DERBY-4306

```
private synchronized void unregisterMBean(...){
+        //Has this service been shut down?
+        if (registeredMbeans == null)
+            return;
```

For one-to-one mappings, MiMo links exceptions with their corresponding modified methods. In this paper, we use $\mathcal{M}_{1-1}$ to denote one-to-one mappings.

From the other bug reports, MiMo extracts two special mappings, *i.e.*, 1-to-n mapping ($\mathcal{M}_{1-n}$) and n-to-1 mapping ($\mathcal{M}_{n-1}$). An $\mathcal{M}_{1-n}$ mapping indicates that a bug report mentions an exception, and multiple methods are modified to fix the bug. MiMo links the exception to all the modified methods. An $\mathcal{M}_{n-1}$ mapping indicates that a bug report mentions multiple exceptions, and only a method is modified to fix the bug. MiMo links all the exceptions to the modified method. The remaining bugs fall into the $\mathcal{M}_{n-m}$ category, where a bug report mentions multiple exceptions, and programmers modify multiple methods to fix the bug. In summary, MiMo identifies six types of mappings such as $\mathcal{M}_{ns}$, $\mathcal{M}_{st}$, $\mathcal{M}_{1-1}$, $\mathcal{M}_{1-n}$, $\mathcal{M}_{n-1}$, and $\mathcal{M}_{n-m}$.

4.2. Mining Repair Model for Exception

When they repair a bug, programmers often modify its test code to determine whether the bug is correctly fixed. As a result, the modifications on test code are more like implementing new features than repairing bugs. As we mine repair models for fixing bugs, MiMo ignores changes on test code. For the remaining source code, MiMo extracts a repair shape from each pair of modified methods, with the support of ChangeDistiller [15]. A repair shape (s) is a pair $\langle n, A\rangle$, where n is a client code method, and A is the set of repair actions on n. Based on the exception-method mappings of each exception, MiMo mines a repair model, $\langle e, \mathcal{H}, \mathcal{X}\rangle$, where:

- e is the thrown exception.
- $\mathcal{H} = \{h_1, \ldots, h_n\}$ is the set of known repair shapes that fix e-related bugs. In our approach, $\mathcal{H}$ is extracted from linked methods of e, and it includes mappings such as $\mathcal{M}_{1-1}$, $\mathcal{M}_{st}$, $\mathcal{M}_{n-1}$, and $\mathcal{M}_{1-n}$.
- $\mathcal{X} = \{x_1, \ldots, x_m\}$ is the probability distribution that is built from $\mathcal{H}$, and x_i is a pair, $\langle a_i, \rho_i\rangle$. Here, a_i denotes a type of repair actions, and ρ_i is calculated as $\frac{|a_i|}{\sum_1^m |a_i|}$ where $|a_i|$ denotes the frequency of a_i.

Zhong and Su [93] show that automatic program repair is less effective to fix $\mathcal{M}_{1-n}$ and $\mathcal{M}_{n-m}$ bugs. To improve the state of the art, we focus on the other four types of mappings such as $\mathcal{M}_{1-1}$, $\mathcal{M}_{st}$, $\mathcal{M}_{1-n}$, and $\mathcal{M}_{n-1}$. We use $\langle E, M\rangle$ to denote a mapping where E is a set of exceptions, and M is the set of mapped methods. When calculating the repair model $\langle e, \mathcal{H}, \mathcal{X}\rangle$ for the e exception, if $\langle E, M\rangle$ is a mapping and $e \in E$, MiMo extracts a repair shape (h) from each method in M, and adds h to $\mathcal{H}$. In addition, MiMo extracts repair actions A in M, and counts A to $\mathcal{X}$.

5. Repair Models

5.1. The General Model

As introduced in Section 1, we choose the repair models of Martinez and Monperrus [43] as the baseline for comparison. Martinez and Monperrus [43] mine two models such as the CT model and the CTET model. The two repair models define the probability distributions of repair actions. From 14 open source Java projects, they collected 62,179 commits that fix bugs. From each commit, they extracted repair actions with the support of ChangeDistiller [15], and refined the repair actions on different granularity. In particular, the CT model counts at a coarser granularity, and the CTET model counts at a finer granularity. For example, when ChangeDistiller identifies an `Insert:IF_STATEMENT` action and an `Insert:FOR_STATEMENT` action, the CT model counts the two actions as two `Insert:STATEMENT` actions. In contrast, when ChangeDistiller identifies an `Update:MODIFIER` action, the CTET model refines it into multiple categories (*e.g.*, increasing or decreasing the accessibility of a modifier).

Typically, a finer repair model leads to a larger search space and a lower probability for generating correct repair shapes. For example, CTET is finer than CT, and Martinez and Monperrus [43] show that it is able to generate longer correct repair shapes with the CT model than with the CTET model. Despite of a lower capability of generating correct repair shapes, a finer repair model generates more detailed repair actions, which make it easier to synthesize patches.

Table 3: Mined Repair Models.

CM		NPE		AIOOBE		IAE		IOE		FNFE		IOOBE		CCE		NCDFE	
U:MI	8.1%	I:IF	13.6%	I:IF	11.8%	U:VDS	8.1%	U:IF	8.5%	U:IF	10.6%	I:MI	13.8%	I:MI	13.6%	M:MI	22.0%
I:MI	6.9%	I:VD	8.5%	I:VD	10.8%	I:VD	8.1%	I:MI	6.4%	I:MI	10.6%	I:VD	8.6%	I:IF	12.3%	I:IF	11.0%
I:IF	6.6%	M:MI	7.5%	U:VD	8.8%	M:AS	8.1%	I:IF	6.4%	M:CO	9.1%	M:MI	6.9%	I:VD	9.3%	U:IF	9.3%
D:MI	5.5%	U:IF	6.7%	U:IF	6.9%	I:MI	7.5%	M:IF	6.4%	D:MI	7.6%	I:IF	6.9%	U:VD	8.6%	I:MI	6.8%
D:IF	5.0%	I:AS	6.3%	M:AS	6.9%	I:AS	6.8%	U:MI	5.3%	D:VD	6.1%	U:VD	5.2%	I:AS	5.6%	M:VD	5.9%
I:VD	4.6%	U:MI	5.5%	I:MI	5.9%	I:IF	5.6%	M:MI	5.3%	D:IF	4.5%	U:IF	5.2%	I:EL	4.3%	I:VD	5.9%
U:IF	4.4%	I:MI	5.3%	U:RS	4.9%	M:IF	5.0%	U:VD	5.3%	M:VD	4.5%	M:VD	5.2%	I:TR	3.7%	M:AS	4.2%
U:VD	4.3%	U:VD	4.4%	U:AS	4.9%	M:MI	4.3%	U:AS	4.3%	D:CO	3.0%	I:TR	5.2%	U:TS	3.7%	D:MI	4.2%
I:AS	4.1%	I:RS	4.1%	I:AS	4.9%	U:AS	4.3%	I:CC	4.3%	D:TS	3.0%	I:AE	3.4%	D:MI	3.7%	I:RS	4.2%
I:ME	4.1%	M:VD	4.1%	U:MI	4.9%	D:TS	3.7%	I:VD	4.3%	D:CC	3.0%	U:MI	3.4%	U:MI	3.1%	U:VD	3.4%

Models. CM: The general repair model that is calculated from CTET; NPE: NullPointerException; AIOOBE: ArrayIndexOutOfBoundsException; IAE: IllegalArgumentException; IOE: IOException; FNFE: FileNotFoundException; IOOBE: IndexOutOfBoundsException; CCE: ClassCastException; NCDFE: NoClassDefFoundError.

Edit action. U: Update; I: Insert; M: Move; D: Delete.

Code element. MI: METHOD_INVOCATION; IF: IF_STATEMENT; VD: VARIABLE_DECLARATION_STATEMENT; ME: METHOD; AS: ASSIGNMENT; RS: RETURN_STATEMENT; TS: THROW_STATEMENT; CO: CONTINUE_STATEMENT; TR: TRY_STATEMENT; CC: CATCH_CLAUSE; AE: ASSERT_STATEMENT; EL: ELSE_BRANCH.

As introduced in Section 4, when MIMO mines repair models, it does not refine the granularity of ChangeDistiller. As a result, CT, CTET, and our mined models are at different granularity. The granularity becomes an independent variable of a controlled experiment. Even if our models achieve better results than CT and CTET, we cannot determine whether our different repair probabilities or different granularity lead to the better results. To eliminate the impacts from granularity, we have to recount a repair model at the same granularity of ChangeDistiller. It is infeasible to recount the model from CT, since the details are missing. In the above example, CT counts two `Insert:STATEMENT` actions, but it does not record the two actions are an `Insert:IF_STATEMENT` action and an `Insert:FOR_STATEMENT` action. However, it is feasible to recount the model from the CTET model. In the above example, we can count all increasing or decreasing the accessibility of a modifier to `Update:MODIFIER` actions of ChangeDistiller.

The appendix of Martinez and Monperrus [42] lists all the occurrences of repair actions. Based on their appendix, we align the repair actions of CTET to the repair actions of ChangeDistiller, and recount a CM model. In Table 3, column "CM" shows the recounted model. Although the CM model is calculated from the CTET model, the CM model is different from the original CTET model, since some repair actions of CTET are merged. For example, as shown in Section 1, the first repair action of the CTET model is inserting method invocations (6.9%), but the first repair action of the CM model is updating method invocations (8.1%).

5.2. *Our Mined Models*

Figure 2 shows bugs that are related to exceptions. Its vertical axis lists the number of bugs with regards to their relations, and its horizontal axis lists exceptions in the descending order of total corresponding bugs. Due to space limit, we do not show exceptions whose linked bugs are fewer than twenty. Comparing the number of unique exceptions in Table 2 with the number of bugs in Figure 2, we found that only about ten percents of exceptions have more than twenty related bugs. Most exceptions are a special type of API classes, and we found that the long tail

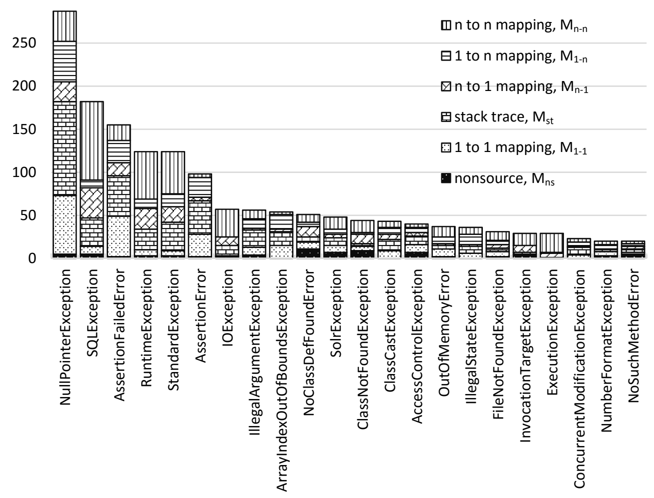

Figure 2: The bugs that are related to each exception.

effect of exceptions is consistent with other API elements. For example, Thummalapenta and Xie [70] found that even in popular API libraries, some API classes are rarely used. As another example, Parnin *et al.* [54] found that only 36.9% of J2SE classes are frequently discussed on StackOverflow, and 22.7% of J2SE classes are never mentioned. The long tail effect of APIs does not indicate that those rarely mentioned exceptions are useless. In contrast, it takes much more effort to fix a rare exception, since programmers are unfamiliar with the exception, but the effect allows us to mine repair models for those popular exceptions.

Yin *et al.* [87] show that programmers may introduce configuration bugs into their code. We found that the $\mathcal{M}_{ns}$ category contains configuration bugs, and these bugs are associated with several exceptions (*e.g.*, `ClassNotFound-Exception` and `AccessControlException`). For example, to fix a `AccessControlException`-related bug[14], programmers modified a configuration file to allow a required permission:

```
+grant codeBase "${derbyTesting.codejar}/derbynet.jar" {
...
+  permission java.io.FilePermission "${derby...","write";
+};
```

Besides configuration errors, we notice that some bugs in the $\mathcal{M}_{ns}$ category are related to source files in programming languages other than Java. For example, to fix a `ClassNotFoundException`-related bug[15], programmers modified a batch file:

```
-if NOT DEFINED CASSANDRA_HOME set CASSANDRA_HOME=%CD%
+if NOT DEFINED CASSANDRA_HOME set CASSANDRA_HOME=%~dp0..
```

Moreover, we find documentation errors that throw exceptions. For example, a bug report of Lucene[16] says that the online user guide threw `SAXParseException`, and the modified line is as follows:

```
-<!DOCTYPE..."../xhtml1/DTD/xhtml1-transitional.dtd">
+<!DOCTYPE..."../html4/loose.dtd">
```

Figure 2 shows that some exceptions (*e.g.*, `SQLException`) have more $\mathcal{M}_{n-m}$ relations than others. In other words, these exceptions are often mentioned with other exceptions, and multiple methods have to be modified to fix such exceptions.

[14]https://issues.apache.org/jira/browse/DERBY-1334
[15]https://issues.apache.org/jira/browse/CASSANDRA-1713
[16]https://issues.apache.org/jira/browse/LUCENE-4302

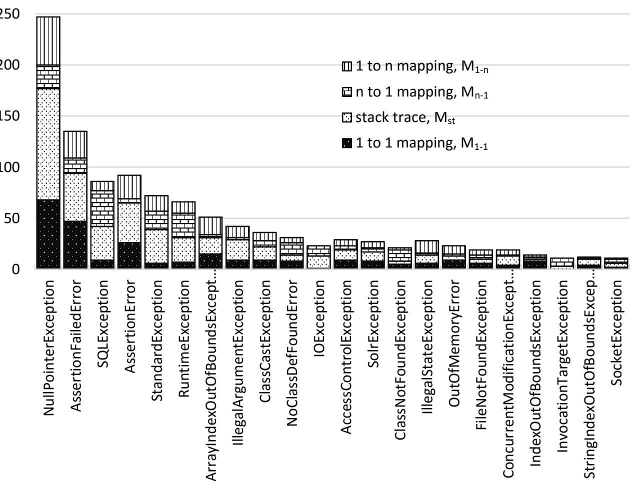

Figure 3: The bugs that are used to mine repair models.

MIMO mines repair models from bug fixes of $\mathcal{M}_{1-1}$, $\mathcal{M}_{st}$, $\mathcal{M}_{1-n}$, and $\mathcal{M}_{n-1}$. Figure 3 shows bugs that belong to the four relations. The vertical axis lists the number of bugs with regards to their relations. The horizontal axis lists exceptions in the descending order of total bugs. MIMO further builds repair models for all the twenty-one exceptions in Figure 3. Table 3 shows $\mathcal{X}$ of the top nine mined repair models, and the general model (the CM model). For a column, the first subcolumn lists repair actions (a_i), and the second subcolumn lists probability distributions (ρ_i). In total, MIMO mined 21 repair models. Due to space limit, Figure 3 ignores exceptions whose related bugs are fewer than ten, and Table 3 presents only the top ten repair actions of nine repair models. We select the nine models, since we will use the nine models in the next evaluation. In addition, Table 3 does not present $\mathcal{H}$ either, but Section 6.2.2 presents an example. Our results show that MIMO successfully mined repair models for popular exceptions.

6. Empirical Comparison

We conducted two empirical comparisons to address the following two research questions:

(RQ1) To what degree are our mined repair models different from the general repair model (Section 6.1)?

(RQ2) To what degree do our mined repair models improve the general model (Section 6.2)?

RQ1 mainly concerns the differences among repair models, and RQ2 mainly concerns the improvement of our repair models over the general repair model of Martinez and Monperrus [43]. In Section 6.1, our results show that our mined repair models are all significantly different from the general repair model of Martinez and Monperrus [43], and it may be feasible to mine a repair model for several exceptions. The difference does not indicate effectiveness, which we explored in Section 6.2. In Section 6.2, our results show that in 43 out of 59 real bugs, our repair models significantly improve the general repair model of Martinez and Monperrus [43].

6.1. RQ1. The Differences

6.1.1. Hypotheses

The motivation of our work is that instead of a general repair model, we need different repair models to fix specific types of bugs. The motivation can be verified by the differences between the general repair model of Martinez and Monperrus [43] and our mined repair models. The null hypothesis is as follows:

(H_0) The difference between the general repair model of Martinez and Monperrus [43] and our mined repair models is not statistically significant.

Table 4: The Differences among Repair Models.

	NPE	AIOOBE	IAE	IOE	FNFE	IOOBE	CCE	NCDFE
CM	0.0032	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001	<0.0001
NPE		0.0002	0.0247	0.0121	0.0003	<0.0001	0.0063	0.0001
AIOOBE			0.261	0.2624	0.9002	0.7915	0.4807	0.7529
IAE				0.8296	0.3319	0.1761	0.5159	0.1868
IOE					0.4679	0.265	0.8164	0.208
FNFE						0.8957	0.5225	0.7958
IOOBE							0.3185	0.9491
CCE								0.3598

```
StringBuilder builder = builders[currentField];
-if(builder.length()>0){
+if(builder.length()>0&&builder.length()<maxLength){
   builder.append(      ); // ...
}
if(builder.length()+value.length()>maxLength){
```

(a) A patch that fixes an IOOBE-related bug

```
-ColumnDescriptor cd;
+ColumnDescriptor cd = null;
 if (tcl == null) {
        cd = ttd.getColumnDescriptor(index + 1);
 }
-else
+else \textbf{if (index < tcl.size())}
{
        ResultColumn trc = (ResultColumn) tcl.elementAt(index);
```

(b) A patch that fixes an AIOOBE-related bug

Figure 4: Similar patch

In addition, it is interesting to investigate whether different repair models are similar. The null hypothesis is as follows:

(H_1) The difference between our mined repair models is not statistically significant.

Although Arcuri and Briand [1] recommend the Mann-Whitney U test to compare random-based approaches, the test is not limited to only this application. As introduced by McKnight and Najab [44], the Mann-Whitney U test is more general than many other tests, since it does not require specific distributions. For example, the independent samples t-test requires that two groups shall be normally distributed. Before we test the hypotheses, we run the Shapiro-Wilk test [49] on all the repair models, and the results show that the repair models are not normally distributed. As a result, it is proper to use the Mann-Whitney U test to compare mined repair models. We reject a null hypothesis only when p is less than 0.05.

6.1.2. Results

Result 1. Our repair models are all significantly different from the general model. The first row of Table 4 shows the results. In Table 4, the column and the row of a cell denote two compared repair models. The abbreviations are consistent with the abbreviations in Table 3. For each cell, the grey background indicates that a hypothesis is rejected. The results show that our repair models are all statistically different from the general repair model, since all the null hypotheses in this row are rejected. The results confirm our motivation of mining repair models for specific bugs, and highlight the importance of the divide-and-conquer strategy.

Result 2. H_1 does not always hold on our repair models. In Table 4, the rows except the first row show the results. For NPE, we reject all the hypotheses. When fixing bugs, programmers often simply avoid thrown NPEs by guiding `null` pointers. However, other exceptions are more complicated, and need much different repairs to be fixed.

For the other exceptions, we do not reject any hypotheses. That is to say, we cannot tell whether these repair models are significantly different. We found that inheritance relations may lead to less difference. For example, IOOBE[17] has only two subclasses, and AIOOBE is a subclass of IOOBE. The description of IOOBE says "Thrown to indicate that an index of some sort (such as to an array, to a string, or to a vector) is out of range", and the description of AIOOBE says "Thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array." As the similar descriptions indicate, their fixes may be similar. For example, Figure 4a shows a patch that fixes an IOOBE-related bug[18], and Figure 4b shows a patch that fixes an AIOOBE-related bug[19]. The above two patches show the same good practice to avoid IOOBE or AIOOBE, *i.e.*, checking whether an index is smaller than the size of an array before accessing the array. The long tail effect of APIs makes it infeasible to mine repair models for rarely used exceptions, but our results indicate that it may be feasible to merge similar exceptions to mine a repair model for multiple exceptions. Still, our results show that the inheritance relation does not guarantee the similarity between two repair models. For example, FNFE is a direct subclass of IOE, but the p value between their repair models is relatively low. As shown in the API document of IOE[20], IOE has more than thirty direct subclasses, and its direct subclasses can have even more subclasses. For example, `SocketException` is a direct subclass of IOE, and as shown in its API document[21], it has four direct subclasses. The similarity between IOE and FNFE is relatively low, since IOE has many subclasses and FNFE is only one of its subclasses.

In summary, MiMo mined twenty-one repair models, and all our tested repair models are significantly different from the general model of Martinez and Monperrus [43]. The result highlights the importance of an approach like MiMo. The long tail effect of APIs places a barrier to mine repair models for rare exceptions, but our results indicate that we can overcome the barrier, since it may be feasible to merge bugs of similar exceptions to mine a repair model for multiple exceptions.

6.2. RQ2: Generating Repair Shape

6.2.1. Setup

In this section, we select another project called FLEX[22], since the project is large and actively maintained. Table 5 shows the selected bugs. We use the names of the exceptions to query the issue tracker of FLEX. From the returned bugs, we selected $\mathcal{M}_{1-1}$, $\mathcal{M}_{st}$, and $\mathcal{M}_{n-1}$ ones. Column "Issue key" lists issue keys of all the selected bugs. The size is comparable to the sizes in existing studies (*e.g.*, [57]). In Table 5, Column "Exception" lists the names of exceptions. Column "Priority" lists the priorities of the bugs. Column "Size" lists the size of golden repair shapes.

6.2.2. Comparison Criteria

Martinez and Monperrus [43] use generated correct repair shapes to evaluate the effectiveness of their approach, since generating correct repair shapes is an essential step in automatic repairs and researchers (*e.g.*, [23]) believe such shapes are even useful for manual repairs. As a generated repair shape is more abstract than a patch and cannot be executed, it is infeasible to guide the search, since its fitness function requires execution results but generated repair shapes cannot be executed. As a result, for each bug, we also use our corresponding repair model and the general model (the CM model in Table 3) to guide the random search for correct repair shapes.

Martinez and Monperrus [43] use repair shapes of manual fixes as the golden standard. To compare with their approach, we use the same golden standard when we explore this research question. In particular, for each selected fix, we use ChangeDistiller to compare its buggy code and fixed code, and use the extracted repair shape as the oracle. Based on the golden standard, we define two criteria:

1. Success rate. For each bug, we execute with the two compared models for 1,000 times. In each execution, we stop the search when the correct repair shape is generated, or the number of attempts reaches the maximum value of 1,000,000. If the correct repair shape is generated in an execution, we consider the execution as a success. In addition, before using our model ($\langle e, \mathcal{H}, \mathcal{X}\rangle$) to generate repair shapes, we compare the oracle repair shape with $\mathcal{H}$. In Columns

[17] `http://docs.oracle.com/javase/8/docs/api/java/lang/IndexOutOfBoundsException.html`
[18] `https://issues.apache.org/jira/browse/LUCENE-5032`
[19] `https://issues.apache.org/jira/browse/DERBY-4449`
[20] `http://docs.oracle.com/javase/8/docs/api/java/io/IOException.html`
[21] `http://docs.oracle.com/javase/8/docs/api/java/net/SocketException.html`
[22] `http://flex.apache.org`

"Mean" and "Median", *h* denotes that the oracle is found in $\mathcal{H}$. When this happens, we consider that the MiMo model achieves better results than the CM model for two considerations. First, the size of $\mathcal{H}$ is small as shown in Figure 3.

Table 5: The Results of Generating Repair Shapes.

Issue key	Exception	Priority	Size	Model	Mean	Median	Rate	Result
FELIX-143	NPE	Major	1	MiMo	*h*	*h*	100.0%	✓
				CM	12	9	100.0%	
FELIX-1195	NPE	Major	10	MiMo	794,454	1,000,001	38.8%	✓
				CM	1,000,001	1,000,001	0.0%	
FELIX-1238	NPE	Minor	3	MiMo	1,763	1,235	100.0%	✓
				CM	32,971	22,778	100.0%	
FELIX-1496	NPE	Major	10	MiMo	979,650	1,000,001	4.1%	✓
				CM	1,000,001	1,000,001	0.0%	
FELIX-1566	NPE	Major	6	MiMo	4,723	3,417	100.0%	✓
				CM	989,759	1,000,001	2.0%	
FELIX-1629	NPE	Minor	2	MiMo	*h*	*h*	100.0%	✓
				CM	63	44	100.0%	
FELIX-1846	NPE	Blocker	4	MiMo	1,603	1,151	100.0%	✓
				CM	2,206	1,608	100.0%	
FELIX-1867	NPE	Major	2	MiMo	*h*	*h*	100.0%	✓
				CM	701	515	100.0%	
FELIX-1872	NPE	Major	2	MiMo	*h*	*h*	100.0%	✓
				CM	187	131	100.0%	
FELIX-1961	NPE	Major	1	MiMo	*h*	*h*	100.0%	✓
				CM	44	29	100.0%	
FELIX-2143	NPE	Minor	1	MiMo	*h*	*h*	100.0%	✓
				CM	43	29	100.0%	
FELIX-2159	NPE	Minor	1	MiMo	*h*	*h*	100.0%	✓
				CM	39	29	100.0%	
FELIX-2213	NPE	Major	7	MiMo	708,241	918,791	53.4%	✓
				CM	1,000,001	1,000,001	0.0%	
FELIX-2230	NPE	Minor	7	MiMo	174,687	125,250	99.7%	✓
				CM	723,187	1,000,001	48.8%	
FELIX-2326	NPE	Minor	7	MiMo	998,119	1,000,001	0.4%	—
				CM	997,940	1,000,001	0.3%	
FELIX-2432	NPE	Major	1	MiMo	*h*	*h*	100.0%	✓
				CM	44	30	100.0%	
FELIX-2574	NPE	Minor	2	MiMo	*h*	*h*	100.0%	✓
				CM	752	507	100.0%	
FELIX-2596	NPE	Minor	2	MiMo	*h*	*h*	100.0%	✓
				CM	702	465	100.0%	
FELIX-2796	NPE	Minor	2	MiMo	*h*	*h*	100.0%	✓
				CM	758	552	100.0%	
FELIX-3117	NPE	Major	5	MiMo	46,001	33,041	100.0%	✓
				CM	997,109	1,000,001	0.7%	
FELIX-780	AIOOBE	Major	1	MiMo	*h*	*h*	100.0%	✓
				CM	39	27	100.0%	
FELIX-1164	AIOOBE	Major	1	MiMo	*h*	*h*	100.0%	✓
				CM	38	25	100.0%	
FELIX-1580	AIOOBE	Critical	3	MiMo	4,494	3,034	100.0%	✓
				CM	11,258	8,022	100.0%	
FELIX-2922	AIOOBE	Minor	15	MiMo	1,000,001	1,000,001	0.0%	—
				CM	1,000,001	1,000,001	0.0%	
FELIX-3463	AIOOBE	Major	3	MiMo	1,207	854	100.0%	✓

Table 5 – continued from previous page

Issue key	Exception	Priority	Size	Model	Mean	Median	Rate	Result
				CM	6,535	4,587	100.0%	
FELIX-3548	AIOOBE	Major	6	MiMo	1,000,001	1,000,001	0.0%	×
				CM	988,283	1,000,001	2.3%	
FELIX-3977	AIOOBE	Major	2	MiMo	*h*	*h*	100.0%	✓
				CM	41	29	100.0%	
FELIX-4565	AIOOBE	Major	7	MiMo	1,000,001	1,000,001	0.0%	—
				CM	1,000,001	1,000,001	0.0%	
FELIX-2610	IAE	Major	14	MiMo	1,000,001	1,000,001	0.0%	—
				CM	1,000,001	1,000,001	0.0%	
FELIX-2375	IAE	Minor	7	MiMo	878,615	1,000,001	23.6%	✓
				CM	1,000,001	1,000,001	0.0%	
FELIX-2387	IAE	Major	2	MiMo	515	374	100.0%	✓
				CM	754	537	100.0%	
FELIX-2672	IAE	Major	10	MiMo	999,887	1,000,001	0.1%	—
				CM	1,000,001	1,000,001	0.0%	
FELIX-3086	IAE	Major	1	MiMo	*h*	*h*	100.0%	✓
				CM	42	28	100.0%	
FELIX-3567	IAE	Major	7	MiMo	149,136	106,681	99.9%	×
				CM	134,058	94,461	99.8%	
FELIX-3670	IAE	Major	1	MiMo	*h*	*h*	100.0%	✓
				CM	39	28	100.0%	
FELIX-4444	IAE	Major	5	MiMo	19,006	13,624	100.0%	✓
				CM	195,048	135,349	99.3%	
FELIX-4616	IAE	Major	4	MiMo	2,004	1,731	100.0%	✓
				CM	258,356	176,386	97.7%	
FELIX-547	IAE	Major	3	MiMo	2,864	1,994	100.0%	✓
				CM	27,091	17,841	100.0%	
FELIX-414	IOE	Major	5	MiMo	235,813	166,509	98.6%	✓
				CM	925,487	1,000,001	15.0%	
FELIX-1188	IOE	Major	4	MiMo	3,369	2,346	100.0%	✓
				CM	95,333	65,472	100.0%	
FELIX-1517	FNFE	Major	6	MiMo	998,565	1,000,001	0.2%	—
				CM	998,942	1,000,001	0.3%	
FELIX-2912	FNFE	Major	7	MiMo	996,182	1,000,001	0.8%	—
				CM	995,310	1,000,001	1.1%	
FELIX-4628	FNFE	Major	3	MiMo	1,225	827	100.0%	×
				CM	104	73	100.0%	
FELIX-1580	IOOBE	Critical	3	MiMo	4,525	3,157	100.0%	✓
				CM	13,923	10,196	100.0%	
FELIX-2120	IOOBE	Major	1	MiMo	65	46	100.0%	×
				CM	44	29	100.0%	
FELIX-2615	IOOBE	Major	2	MiMo	451	307	100.0%	✓
				CM	924	652	100.0%	
FELIX-3402	IOOBE	Minor	12	MiMo	1,000,001	1,000,001	0.0%	—
				CM	1,000,001	1,000,001	0.0%	
FELIX-4524	IOOBE	Major	6	MiMo	240,146	171,505	98.5%	✓
				CM	969,506	1,000,001	5.5%	
FELIX-411	CCE	Major	1	MiMo	*h*	*h*	100.0%	✓
				CM	42	30	100.0%	
FELIX-1197	CCE	Major	29	MiMo	1,000,001	1,000,001	0.0%	—
				CM	1,000,001	1,000,001	0.0%	
FELIX-1216	CCE	Major	2	MiMo	130	91	100.0%	✓

Table 5 – continued from previous page

Issue key	Exception	Priority	Size	Model	Mean	Median	Rate	Result
				CM	969	632	100.0%	
FELIX-2306	CCE	Minor	4	MiMo	18,799	13,353	100.0%	✓
				CM	45,515	33,058	100.0%	
FELIX-3001	CCE	Minor	2	MiMo	*h*	*h*	100.0%	✓
				CM	793	512	100.0%	
FELIX-3323	CCE	Major	3	MiMo	345	231	100.0%	✓
				CM	986	678	100.0%	
FELIX-3960	CCE	Blocker	4	MiMo	9,179	6,194	100.0%	✓
				CM	23,716	16,927	100.0%	
FELIX-819	NCDFE	Major	11	MiMo	1,000,001	1,000,001	0.0%	—
				CM	1,000,001	1,000,001	0.0%	
FELIX-1516	NCDFE	Major	7	MiMo	999,756	1,000,001	0.1%	—
				CM	999,480	1,000,001	0.1%	
FELIX-2492	NCDFE	Major	2	MiMo	2,502	1,766	100.0%	×
				CM	2,188	1,569	100.0%	
FELIX-3153	NCDFE	Major	4	MiMo	21,033	14,613	100.0%	✓
				CM	1,000,001	1,000,001	0.0%	

Second, a found match in $\mathcal{H}$ provides valuable hints to synthesize patches. For example, Figure 5a shows the fault code of FELIX-1872[23]. With our repair model, we found a match, LUCENE-3032[24], and Figure 6 shows the patch for this bug. This patch indicates that programmers can place a buggy line inside an `If` statement, and check whether its values are `null` to avoid NPE. Although the programmers of FLEX are unlikely to read LUCENE-3032 before they fix FELIX-1872, they follow the same way to fix this bug, as shown in Figure 5c.

The success rate is calculated as successes over the total times of executions (it is 1,000 in our evaluation).

2. Tried times. In each execution, if both compared models successfully generate a correct repair shape, we compare their tried times. As our generation is a guided random search, we follow the guidance of Arcuri and Briand [1]. Our null hypothesis is as follows:

(H_2) The difference between the the CM model and our mined repair model, as far as times of attempts are concerned, is not statistically significant.

We reject the null hypothesis, when the p value is smaller than 0.05. When it is rejected, if the mean times of our model is smaller than the mean times of the CM model, we consider that our model synthesizes a correct repair shape faster than the general mode. That is to say, we consider a model (t_1) is better than another model (t_2), only when t_1 needs significantly fewer tried times to generate the golden standard than t_2 does.

6.2.3. Results

Table 5 shows the overall results. Column "Model" lists compared repair models. In each row, "MiMo" denotes our repair models, and "CM" denotes the general repair model. Column "Success rate" lists success rates. Based on this column, we come to the following results:

Result 1. The success rates of the CM model are consistent with the prediction of Martinez and Monperrus [43]. The evaluation results of Martinez and Monperrus [43] show that when generating correct repair shapes, the maximum lengths for the CT model and the CTET model are eight and four, respectively. As the CM model is finer than the CT model and is coarser than the CTET model, the maximum length for CM shall be between four and eight. In Table 5, the maximum length for the CM model is seven (FELIX-2230, FELIX-2326, FELIX-2912, and FELIX-3567), with low success rates (*e.g.*, 1.1%). We estimate that seven is near to the maximum value. Our result indicates

[23]`https://issues.apache.org/jira/browse/FELIX-1872`
[24]`https://issues.apache.org/jira/browse/LUCENE-3032`

```
public void setAttribute(String name, Object value){
    this.attributes.put(name, value);
}
```

(a) The buggy code of FELIX-1872

```
-deleteSlice.apply(pendingDeletes, numDocsInRAM);
+if(deleteSlice != null) {
+    deleteSlice.apply(pendingDeletes, numDocsInRAM);
+}
```

(b) The patch of LUCENE-3032

```
public void setAttribute(String name, Object value){
   if ((name != null) && (value != null)) {
     this.attributes.put(name, value); }}
```

(c) The fixed code of FELIX-1872

Figure 5: The usefulness of $\mathcal{H}$

that the CM model and our random generation are correct, since the capability meets the prediction of Martinez and Monperrus [43] (seven is between four and eight).

Result 2. Our models have higher success rates than the CM model. In Table 5, Row "MiMo" shows the results of our models. The maximum length for our models is ten (FELIX-1195, FELIX-1496, and FELIX-2672), which is higher than the CM model (seven). In particular, for the six bugs such as FELIX-1195, FELIX-1496, FELIX-2213, FELIX-3117, FELIX-2375, and FELIX-3153, our repair models are able to generate correct repair shapes, but the CM model fails to generate any correct repair shapes. The results show that the divide-and-conquer strategy improves the state of the art. However, we notice that some bug fixes (*e.g.*, FELIX-2922) are too complicated for all the repair models. This could be a limitation of the random search, and we further discuss this issue in Section 7.

In Table 5, Columns "Mean" and "Median" list the mean and median values of attempts when generating the first correct repair shape, respectively. Based on the test, we come to the following result:

Result 3. Our models have fewer tried times than the CM model for most bugs. Column "Result" lists comparison results. For each row of this column, "✓" denotes that the mean values of MiMo attempts are significantly smaller than the mean values of CM attempts; "×" denotes that the mean values of CM attempts are significantly smaller than the mean values of MiMo attempts; and "—" denotes that the difference is not significant. The results show that our repair models are more effective in 72.8% of bugs; are less effective in 8.5% of bugs; and are not significantly different in the remaining bugs. For the three "×" bugs such as FELIX-4628, FELIX-2120, and FELIX-2492, our models also achieve 100% success rates, despite of more tried times.

As shown in Table 2, our repository includes only 6 projects for mining repair models, and as shown in Figure 3, even the most popular exception, NPE, has only about two hundred bugs. In contrast, the repair model of Martinez and Monperrus [43] is built on 89,993 commits from 14 projects. Our much smaller pool of commits can reduce the effectiveness of our repair models. Our results are inspiring, since our repair models still achieve better results.

6.2.4. More Findings

Finding 1. The priorities of bug reports are not correlated with exception names nor repair sizes. In Table 5, Column "Priority" lists priorities of bugs. The distribution of priorities shows the importance of exception-related bugs. Column "Size" lists sizes of oracle repair shapes. It is interesting to know whether exception names and repair sizes are correlated with priorities, and we run the Pearson correlation analysis [21] on the above three columns. The *sig* value between exception names and priorities is 0.591, and the *sig* value between repair sizes and priorities is 0.556. The results show that priorities are not correlated with exception names nor repair sizes. That is to say, a bug can be important, even if the bug throws a simple exception and it does not involve many repair actions.

The result is inspiring for automatic program repair. Although Monperrus [48] criticises that existing approaches can fix only limited bugs, our result shows that simple bugs may still be critical. For example, in Table 5, FELIX-

```
-final String targetFilter=(String)properties.get(...);
-if((getTarget()==null&&targetFilter==null)
   ||getTarget().equals(targetFilter))
+final String newTarget=(String)properties.get(...);
+final String currentTarget = getTarget();
+if((currentTarget==null&&newTarget==null)
+   ||(currentTarget!=null&&currentTarget.equals(newTarget)))
...
-ServiceReference[] refs = ...(targetFilter);
+ServiceReference[] refs = ...(newTarget);
```

Figure 6: The patch that fixes FELIX-1846

1846[25] is a blocker bug (the highest priority in an Apache issue tracker). Its description says that NPE is thrown, when a target filter is undefined, and Figure 6 shows its patch. When a target filter is undefined, `getTarget()` returns a `null` value, and the `null` value leads to NPE, when its member method, `equals()`, is called. The fixed code solves the problem, since the return value is checked whether it is `null`, before its member method is called. The fix is simple, and it takes only several hours to fix the bug. However, the reporter still believes that it is a critical bug.
Finding 2. Small fixes are similar, but are often not identical. Nguyen *et al.* [52] show that similar small bug fixes frequently recur. Our results are consistent with their results, since all the found matches are small bug fixes (marked as *h* in Table 5). Although Nguyen *et al.* [52] show that many bug fixes are similar, our results show that most bug fixes are not identical. The results highlight the importance of automatic program repair, since this direction deals with the variety in fixing bugs.

6.3. Threats to Validity

The threat to internal validity includes the possible errors in the recounted model. To make a fair comparison, we have to recount the CM model from the CTET model, and a miscalculation could lead to a wrong model. To reduce the threat, we read the CTET model carefully, when we build the CM model. As the CTET model does not explain its repair actions, we can misunderstand some repair actions, and miscalculate the probabilities of some merged repair actions. However, our results show that the effectiveness of the CM model is in line with the predication of Martinez and Monperrus [43], which indicates that the CM model is correctly counted. The threat to external validity includes that selected open source projects may not fully reflect the nature of commercial projects. However, Ma *et al.* [39] show that many commercial companies involve in open source development, which weakens the boundary between open source projects and commercial projects. The threat to external validity also includes the selected oracle bugs. Although we conducted our study on 59 real bugs, our approach is evaluated on limited bugs. In addition, we notice that programmers can hide exceptions, even if they do not fully understand the causes of thrown exceptions. In some cases, hiding exceptions does not repair bugs, but makes the debugging process even more difficult. As a result, our golden standard is not fully reliable. The threat could be further reduced by introducing more bugs and more manual inspection in future work.

7. Discussion and Future Work

Mining repair models for more bug categories. We notice that many bugs do not throw exceptions. For example, concurrency bugs can hang a system without any exceptions. For these bugs, it is feasible to determine their categories with other sources (*e.g.*, runtime behaviors [55, 32]). Furthermore, researchers have conducted various empirical studies on bugs (*e.g.*, [67]). Their results are useful to identify more categories of bugs for analysis.
Mining synthesis models for bug categories. Kaleeswaran *et al.* [23] show that an approach is useful, even if it provides only hints on fixing bugs but does not fix bugs. Still, we understand the importance of reducing the gap between repair shapes and patches. It can be feasible to build the mappings between bug symptoms and their treatments. For example, when we fix NPE-related bugs, we often check variables against `null` values. In addition, as pointed out by Le Goues *et al.* [29], researchers proposed approaches that mine specifications from code [95, 89],

[25]`https://issues.apache.org/jira/browse/FELIX-1846`

documents [96], and traces [13]. Zhong and Mei [90] conduct an empirical study on API usages. Their findings can be useful to repair API-related bugs. It may also be feasible to borrow ideas from related research fields (*e.g.*, behaviour model synthesis [74], service composition [40], and online healing [11]). In future work, we plan to leverage the preceding work to build a synthesis model for specific bugs.

Guiding the repair process. In the research context of automatic program repair, researchers assume that faults are already localized by existing spectra-based fault localization approaches [79]. However, Liu and Zhong [35] show that a located fault is often not the true location of a bug, and the effectiveness of an approach can be significantly reduced, if located bugs are not ranked as the top ones. Especially for exception-related bugs, simply trying to repair the location that throws the exception is a very dangerous action, since it potentially hides the failure behavior of the bug. In addition, although Qi *et al.* [57] show that the random search is more effective than the genetic algorithm, Martinez and Monperrus [43] and our work both show that it is feasible to fix only small bugs, without a better guidance algorithm. Murphy-Hill *et al.* [50] analyze various issues, when programmers fix bugs manually. Based on these issues, we plan to explore more advanced fault localization approaches and guiding algorithms in future work.

8. Related Work

Automatic program repair. Weimer *et al.* [77] is a pioneer in the automatic-program-repair research. Follow-up researchers extend the approach with additional operators [28, 25, 20, 75] and additional inputs such as forum discussions [16], past fixes [38] and code repositories [24]. Qi *et al.* [57] show that the random search is more effective than the genetic algorithm in guiding repairs. Liu and Zhong [35] mine more repair templates from StackOverflow. Qi *et al.* [58] show that automatic repair can lead to false fixes. Smith *et al.* [66] show that the random search is less effective than the genetic algorithm, and developers can also make similar false fixes. Martinez and Monperrus [43] mine repair models to reduce the search space. Xuan and Monperrus [83] purify test cases to better locate faults. Long and Rinard [36] focus the search on the most promising regions, and present a detailed analysis on the search space [37]. Sidiroglou *et al.* propose approaches that repairs buffer overflows [63] and internet worms [64]. Rolim *et al.* [60] learn repairs from existing code samples. Xiong *et al.* [81] and Xuan *et al.* [82] repair bugs in `if` conditions, with knowledge learnt from documents and code samples. Chen *et al.* [6] repair bugs with learnt contracts. Le *et al.* [27] synthesize patches based on examples. Saha *et al.* [61] introduce more repair templates and algorithms to rank patches. Liu and Zhong [35] mine StackOverflow for more repair templates. Instead of the genetic algorithm, Mechtaev *et al.* [45] use constraint solving to generate patches, and combine symbolic execution for improvement [46]. Tan and Roychoudhury [68] show that learning the nature of recurring bugs can lead to an effective repair approach for such bugs. Recent studies [85, 86] show that better test cases can lead to better synthesized patches. Zhong and Meng [91] analyze the potential of reusing past fixes. Liu *et al.* [34] compare test coverage to determinate correct patches. Hassan and Wang [19] repair build scripts with fixing histories. Yang *et al.* [84] show that the suspiciousness-first algorithm is better than the rank-first algorithm in parallel repair and patch diversity. We further analyze the divide-and-conquer strategy in a detailed depth, and our evaluation results show that our repair models improve the general repair model.

Repairing corrupt data. Researchers have proposed various approaches to repair errors in data. Demsky and Rinard [9] dynamically detect and repair data structures based on predefined constraints. Novark *et al.* [53] propose Exterminator that repairs heap-based memory errors. Elkarablieh *et al.* [12] repair data structures based on written `assert` statements. Carzaniga *et al.* [5] repair corrupt data based on recorded successful executions. It may be feasible to borrow ideas from automatic program repair to deal with the variety in repairing corrupt data, and our strategy shall work too.

Empirical studies on bug fixes and exceptions. Some empirical studies were conducted to understand bug fixes or exceptions. Yin *et al.* [88] show that bug fixes can introduce new bugs. Nguyen *et al.* [51] show that repetitiveness is common in small size bug fixes. Eyolfson *et al.* [14] show that the bugginess of a commit is correlated with the time to make the commit. Zhong and Su [93] estimate the potential of automatic program repair by comparing what programmers did in repairing bugs with what are implemented in existing repair approaches. Campos and Maia [4] replicate this study [93] with more bug fixes, and explored more issues such as which repair patterns are common. Baishakhi *et al.* [59] show that buggy code has higher entropies. Koopman and DeVale [26] analyze the exception handling mechanisms of different operation systems. Garcia *et al.* [17] survey the exception handling mechanisms of object-oriented software. Zhong and Meng [91] explore the potential of using hints from past fixes. Our study reveals the nature of fixing exception-related bugs, complementing the previous studies.

9. Conclusion

Although the general repair model reduces the search space of generating patches, we argue that it can improve the general model if we mine repair models for bug categories. To explore the hypothesis, we implement ExFi that classifies bugs based on their related exceptions. With its support, we construct the first benchmark in which bug categories are marked. In total, the benchmark contains 1,483 bugs. Furthermore, we implement MiMo that mines a repair model for each category of bugs that are related to an exception. In our study, we compare the general repair model with our mined repair models. Our results show that our repair models are significantly different from the general repair model, and are more effective than the general repair model in generating correct repair shapes.

Acknowledgments

We appreciate the anonymous reviewers for their constructive comments. This work is sponsored by the National Key Basic Research Program of China No. 2015CB352203, the National Nature Science Foundation of China No. 61572313, and the grant of Science and Technology Commission of Shanghai Municipality No. 15DZ1100305.

Reference

[1] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proc. ICSE*, pages 1–10, 2011.
[2] A. Bacchelli, A. Cleve, M. Lanza, and A. Mocci. Extracting structured data from natural language documents with island parsing. In *In Proc. ASE*, pages 476–479, 2011.
[3] T. F. Bissyande, F. Thung, S. Wang, D. Lo, L. Jiang, and L. Reveillere. Empirical evaluation of bug linking. In *Proc. CSMR*, pages 89–98, 2013.
[4] E. C. Campos and M. A. Maia. Common bug-fix patterns: A large-scale observational study. In *Proc. ESEM*, page to appear, 2017.
[5] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezze. Automatic recovery from runtime failures. In *Proc. ICSE*, pages 782–791, 2013.
[6] L. Chen, Y. Pei, and C. A. Furia. Contract-based program repair without the contracts. In *Proc. ASE*, pages 637–647, 2017.
[7] B. Cornu, L. Seinturier, and M. Monperrus. Exception handling analysis and transformation using fault injection: Study of resilience against unanticipated exceptions. *Information and Software Technology*, 57:66–76, 2015.
[8] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.
[9] B. Demsky and M. C. Rinard. Automatic detection and repair of errors in data structures. In *Proc. OOPSLA*, pages 78–95, 2003.
[10] T. Dhaliwal, F. Khomh, and Y. Zou. Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox. In *Proc. ICSM*, pages 333–342, 2011.
[11] R. Ding, Q. Fu, J.-G. Lou, Q. Lin, D. Zhang, J. Shen, and T. Xie. Healing online service systems via mining historical issue repositories. In *Proc. ASE*, pages 318–321, 2012.
[12] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid. Assertion-based repair of complex data structures. In *Proc. ASE*, pages 64–73, 2007.
[13] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.
[14] J. Eyolfson, L. Tan, and P. Lam. Correlations between bugginess and time-based commit characteristics. *Empirical Software Engineering*, 19(4):1009–1039, 2014.
[15] B. Fluri, M. Wursch, M. PInzger, and H. C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.
[16] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei. Fixing recurring crash bugs via analyzing Q&A sites. In *Proc. ASE*, pages 307–318, 2015.
[17] A. F. Garcia, C. M. Rubira, A. Romanovsky, and J. Xu. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software*, 59(2):197–222, 2001.
[18] F. Hassan, S. Mostafa, E. Lam, and X. Wang. Automatic building of Java projects in software repositories: A study on feasibility and challenges. In *Proc. ESEM*, page to appear, 2017.
[19] F. Hassan and X. Wang. HireBuild: An automatic approach to history-driven repair of build scripts. In *Proc. ICSE*, 2018.
[20] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated concurrency-bug fixing. In *Proc. OSDI*, pages 221–236, 2012.
[21] R. A. Johnson, D. W. Wichern, and P. Education. *Applied multivariate statistical analysis*, volume 4. Prentice hall Englewood Cliffs, NJ, 1992.
[22] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proc. ISSTA*, pages 437–440, 2014.
[23] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso. Minthint: Automated synthesis of repair hints. In *Proc. ICSE*, pages 266–276, 2014.
[24] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun. Repairing programs with semantic code search. In *Proc. ASE*, pages 295–306, 2015.
[25] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proc. ICSE*, pages 802–811, 2013.
[26] P. Koopman and J. DeVale. The exception handling effectiveness of POSIX operating systems. *IEEE Transactions on Software Engineering*, 26(9):837–848, 2000.

[27] X. D. Le, D. Chu, D. Lo, C. Le Goues, and W. Visser. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proc. ESEC/FSE*, pages 593–604, 2017.

[28] 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 *Proc. ICSE*, pages 3–13, 2012.

[29] C. Le Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.

[30] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015.

[31] B. Liang, P. Bian, Y. Zhang, W. Shi, W. You, and Y. Cai. AntMiner: mining more bugs by reducing noise interference. In *Proc. ICSE*, pages 333–344, 2016.

[32] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proc. PLDI*, pages 15–26, 2005.

[33] Z. Lin, D. Marinov, H. Zhong, Y. Chen, and J. Zhao. JaConTeBe: A benchmark suite of real-world Java concurrency bugs. In *Proc. ASE*, pages 178–189, 2015.

[34] X. Liu, M. Zeng, Y. Xiong, L. Zhang, and G. Huang. Identifying patch correctness in test-based program repair. *Proc. ICSE*, 2018.

[35] X. Liu and H. Zhong. Mining stackoverflow for program repair. In *Proc. SANER*, page to appear, 2018.

[36] F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proc. ESEC/FSE*, page to appear, 2015.

[37] F. Long and M. Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *Proc. ICSE*, 2016.

[38] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *Proc. POPL*, pages 298–312, 2016.

[39] X. Ma, M. Zhou, and D. Riehle. How commercial involvement affects open source projects: three case studies on issue reporting. *Science China Information Sciences*, 56(8):1–13, 2013.

[40] Y. Ma, X. Lu, X. Liu, X. Wang, and M. B. Blake. Data-driven synthesis of multiple recommendation patterns to create situational Web mashups. *Science China Information Sciences*, 56(8):1–16, 2013.

[41] 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*, pages 1–29, 2016.

[42] M. Martinez and M. Monperrus. Appendix of "mining software repair models for reasoning on the search space of automated program fixing". Technical report, INRIA, 2013.

[43] M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2013.

[44] P. E. McKnight and J. Najab. Mann-Whitney U test. *Corsini Encyclopedia of Psychology*, 2010.

[45] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *Proc. ICSE*, pages 448–458, 2015.

[46] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proc. ICSE*, 2016.

[47] M. Miłkowski. Developing an open-source, rule-based proofreading tool. *Software: Practice and Experience*, 40(7):543–566, 2010.

[48] M. Monperrus. A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair. In *Proc. ICSE*, pages 234–242, 2014.

[49] G. S. Mudholkar, D. K. Srivastava, and C. Thomas Lin. Some p-variate adaptations of the shapiro-wilk test of normality. *Communications in Statistics-Theory and Methods*, 24(4):953–985, 1995.

[50] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan. The design of bug fixes. In *Proc. ICSE*, pages 332–341, 2013.

[51] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan. A study of repetitiveness of code changes in software evolution. In *Proc. ASE*, pages 180–190, 2013.

[52] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *Proc. ICSE*, pages 315–324, 2010.

[53] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: automatically correcting memory errors with high probability. In *Proc. PLDI*, pages 1–11, 2007.

[54] C. Parnin, C. Treude, L. Grammel, and M.-A. Storey. Crowd documentation: Exploring the coverage and the dynamics of API discussions on Stack Overflow. Technical report, Georgia Institute of Technology, 2012.

[55] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proc. ICSE*, pages 465–475, 2003.

[56] P. Prabhu, N. Maeda, G. Balakrishnan, F. Ivančić, and A. Gupta. Interprocedural exception analysis for C++. In *Proc. ECOOP*, pages 583–608, 2011.

[57] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *Proc. ICSE*, pages 254–265, 2014.

[58] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proc. ISSTA*, pages 24–36, 2015.

[59] B. Ray, V. Hellendoorn, Z. Tu, C. Nguyen, S. Godhane, A. Bacchelli, and P. Devanbu. On the "naturalness" of buggy code. In *Proc. ICSE*, 2016.

[60] 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 *Proc. ICSE*, pages 404–415, 2017.

[61] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad. ELIXIR: effective object oriented program repair. In *Proc. ASE*, pages 648–659, 2017.

[62] A. Schroter, N. Bettenburg, and R. Premraj. Do stack traces help developers fix bugs? In *Proc. MSR*, pages 118–121, 2010.

[63] S. Sidiroglou, G. Giovanidis, and A. D. Keromytis. A dynamic mechanism for recovering from buffer overflow attacks. In *Proc. ICISC*, pages 1–15, 2005.

[64] S. Sidiroglou and A. D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security & Privacy*, 3(6):41–49, 2005.

[65] S. Sinha, H. Shah, C. Görg, S. Jiang, M. Kim, and M. J. Harrold. Fault localization and repair for Java runtime exceptions. In *Proc. ISSTA*, pages 153–164, 2009.

[66] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proc.*

ESEC/FSE, pages 532–543, 2015.

[67] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, 19(6):1665–1705, 2014.

[68] S. H. Tan and A. Roychoudhury. relifix: Automated repair of software regressions. In *Proc. ICSE*, pages 913–923, 2015.

[69] S. H. Tan, J. Yi, S. Mechtaev, A. Roychoudhury, et al. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *Proc. ICSE*, pages 180–182, 2017.

[70] S. Thummalapenta and T. Xie. SpotWeb: Detecting framework hotspots and coldspots via mining open source code on the web. In *Proc. ASE*, pages 327–336, 2008.

[71] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *Proc. ICSE*, pages 496–506, 2009.

[72] Y. Tian, J. Lawall, and D. Lo. Identifying linux bug fixing patches. In *Proc. ICSE*, pages 386–396, 2012.

[73] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 29(4), 2017.

[74] S. Uchitel, J. Kramer, and J. Magee. Synthesis of behavioral models from scenarios. *IEEE Transactions on Software Engineering*, 29(2):99–115, 2003.

[75] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proc. ISSTA*, pages 61–72, 2010.

[76] W. Weimer and G. C. Necula. Exceptional situations and program reliability. *ACM Transactions on Programming Languages and Systems*, 30(2):8, 2008.

[77] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proc. ICSE*, pages 364–374, 2009.

[78] C. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *Proc. ICSME*, pages 181–190, 2014.

[79] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[80] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In *Proc. ESEC/FSE*, pages 15–25, 2011.

[81] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang. Precise condition synthesis for program repair. In *Proc. ICSE*, pages 416–426, 2017.

[82] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in Java programs. *IEEE Transactions on Software Engineering*, 43(1):34–55, 2017.

[83] J. Xuan and M. Monperrus. Test case purification for improving fault localization. In *Proc. FSE*, pages 52–63, 2014.

[84] D. Yang, Y. Qi, and X. Mao. An empirical study on the usage of fault localization in automated program repair. In *Proc. ICSME*, pages 504–508, 2017.

[85] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan. Better test cases for better automated program repair. In *Proc. ESEC/FSE*, pages 831–841, 2017.

[86] J. Yi, S. H. Tan, S. Mechtaev, M. Böhme, and A. Roychoudhury. A correlation study between automated program repair and test-suite metrics. *Empirical Software Engineering*, pages 1–32, 2017.

[87] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proc. SOSP*, pages 159–172, 2011.

[88] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proc. ESEC/FSE*, pages 26–36, 2011.

[89] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou. Automatic parameter recommendation for practical API usage. In *Proc. 34th ICSE*, pages 826–836, 2012.

[90] H. Zhong and H. Mei. An empirical study on API usages. *IEEE Transaction on software engineering*, 2018.

[91] H. Zhong and N. Meng. Towards reusing hints from past fixes -an exploratory study on thousands of real samples. *Empirical Software Engineering*, 2018.

[92] H. Zhong and Z. Su. Detecting API documentation errors. In *Proc. SPASH/OOPSLA*, pages 803–816, 2013.

[93] H. Zhong and Z. Su. An empirical study on real bug fixes. In *Proc. ICSE*, pages 913–923, 2015.

[94] H. Zhong and X. Wang. Boosting complete-code tool for partial program. In *Proc. ASE*, pages 671–681, 2017.

[95] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *Proc. ECOOP*, pages 318–343, 2009.

[96] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *Proc. ASE*, pages 307–318, 2009.