2016 International Conference on Software Analysis, Testing and Evolution

Automatic Reproducible Crash Detection

Yongfeng Gu Jifeng Xuan* Tieyun Qian
State Key Lab of Software Engineering
Wuhan University
Wuhan, China
{yongfenggu, jxuan, qty}@whu.edu.cn

***Abstract*—Crash reproduction, which spends much time of developers in reading and understanding source code, is a crucial yet time-consuming task in program debugging. To reduce the time and resource cost, automatic techniques of test generation have been proposed. These techniques aim to automatically generate test cases to reproduce the scenario of a crashed project. Unfortunately, due to the lack of a detailed comprehension of the source code, a generated test case may fail in reproducing an expected crash. In this paper, we propose an automatic approach to reproducible bug detection. This approach predicts whether a crash is difficult to reproduce or not via training a classifier based on historical reproducible crash data. If a crash is difficult to reproduce, it is better to assign the crash to a developer, instead of using an automatic technique of test generation. Our work can help to prioritize crashes and to save the cost of developers. Preliminary experiments show that our approach effectively detects reproducible crashes via evaluating 45 crashes.**

***Index Terms*—Crash reproduction, machine learning, test generation**

I. Introduction

Software testing and debugging spends a large amount of human labor during software development [1]. Crash reproduction is crucial yet time-consuming process in program debugging since it is hard to fix a bug without being able to reproduce it. In practice, a common way is that a developer writes a test case to trigger the crash via reading the failure trace and buggy source code; then the developer executes this test case and manually analyzes the reproduced scenario [2]. The way of manually reproducing a crash consumes much time cost of software development. This cost increases rapidly with the growing of the project scale [3]. In a large project, simple bugs may even spend a company unbelievable time and resource cost. For instance, de Simone [4] has reported an Apple developer has written a bug of two continuous `goto fail` statements. This bug makes the program always jump to the `fail` label, no matter what condition the first `goto` has. Manual efforts of reproducing such bugs results requires carefully checking and/or understanding the source code.

Many automatic test generation approaches [5], [6], [7], [8], [9] have been proposed in recent years to reduce the manual cost of crash reproduction. By recording the full or partial executive information of a test case, we can mimic the original situation when a crash occurs. Then the generated test case can trigger the expected crash without the intervention of developers. Unfortunately, despite the promising progress of test generation techniques, automatic crash reproduction may fail due to the complexity of software. Without understanding the buggy source code and the failure trace (also called the *stack trace*, which records the executive states and exceptions while a crash occurs), automatic techniques may generate many useless or meaningless test cases, which cannot correctly trigger the crash scenario.

In this paper, we propose an automatic approach to reproducible crash detection. This approach predicts whether a crash is difficult to be reproduced by learning historical crash data. If a crash is difficult to reproduce, it is better to assign the crash to a developer, instead of using an automatic technique of test generation. Prioritizing crashes with our approach can improve the schedule of bug fixing by assigning difficult crashes to human developers and assigning easy crashes to automatic test generation tools.

In our work, to build a predictive model, we extract 23 features from either the buggy source code or the failure trace. These extracted features capture general characteristics of the crashed program, such as the number of related files, the LoC (Lines of Code) of the crashed method, the type of the exception. During extracting features, we assume the difficulty of crash production correlates to the characteristics of the program to some extend. In our preliminary work, we evaluate our approach on 45 crashes from the Defects4J dataset [10]. Defects4J is a Java bug database, consisting of 357 real-world bugs from five open source projects. Each bug is equipped with at least one test case to reproduce it. We conduct the evaluation with five-fold cross validation on 45 selected bugs from three out of five projects. Experiments demonstrate that our approach effectively can predict the difficulty of crash reproduction with the prediction accuracy of 0.644.

The main contributions of this paper are as follows.

1. We first address the problem of reproducible crash detection, namely detecting the difficulty of crash reproduction for a crashed project. The predicted difficulty can be used to prioritize crashes by assign difficult crashes to human developers and assigning easy crashes to automatic test generation tools.

2. We propose an automatic approach to the reproducible crash detection by learning from historical crash data. The detective model is trained by extracting 23 features from the source code and the failure trace of the targeted project.

*Corresponding author.

978-1-5090-4517-4/16 $31.00 © 2016 IEEE
DOI 10.1109/SATE.2016.15

II. Background

Software crashes are unavoidable in practice. It is necessary to reproduce a crash before fixing its root cause [11]. In this section, we describe the background of crash production and its automatic solutions.

A. Crash reproduction

In this paper, a *failure trace* denotes the logged information of exceptions once a crash is triggered; a *direct cause method* denotes the first method (by ignoring the methods of the programming language, such as the JDK APIs in Java) in a failure trace, which relates to the direct reason of a crash. Then a direct cause method can be directly extracted from a failure trace. Note that a direct cause method may be not the root cause of an crash.

We take Bug 747 from the Apache Commons Lang project as an example.[1] This bug is also Bug 1 in the Defects4J dataset. The failure trace and code snippets of source code are shown in Listings 1, 2, and 3. In Listing 1, we notice that the crash is caused by a number format exception and the input data that triggers the crash is an input string of "80000000". In other words, this exception occurs when the program attempts to convert a string "80000000" into an integer.

```
java.lang.NumberFormatException: For input string:
    ``80000000''
  at java.lang.NumberFormatException.forInputString(
      NumberFormatException.java:65)
  at java.lang.Integer.parseInt(Integer.java:495)
  at java.lang.Integer.valueOf(Integer.java:556)
  at java.lang.Integer.decode(Integer.java:984)
  at org.apache.commons.lang3.math.NumberUtils.
      createInteger(NumberUtils.java:684)
  at org.apache.commons.lang3.math.NumberUtils.
      createNumber(NumberUtils.java:474)
  at org.apache.commons.lang3.math.NumberUtilsTest.
      testLang747(NumberUtilsTest.java:256)
```

Listing 1. Failure trace of Bug 747 in the Lang project.

From the trace in Listing 1, a method `createInteger()` is identified as the direct cause method since its the first method in the failure trace except Java JDK. That is, the bug is directly caused by a method `createInteger()`, which is called by another method `createNumber()`. The location of source code that throws the exception is when `decode()` is invoked in Listing 2. The root cause of the bug is in the method `createNumber()` in Listing 3 when `hexDigits > 8`.

```
public Integer createInteger(String str) {
  if (str == null) {
      return null;
  }  //decode() handles 0xAABD and 0777
  return Integer.decode(str);
}
```

Listing 2. Direct cause method in the failure trace.

[1]Lang Bug 747, http://issues.apache.org/jira/browse/LANG-747.

```
public Number createNumber(String str) { ...
  if (pfxLen > 0) { // For a hex number
    final int hexDigits = str.length() - pfxLen;
    if (hexDigits > 16)
      return createBigInteger(str);
    if (hexDigits > 8)   // BUG, loss of conditions
      return createLong(str);
    return createInteger(str);
  } ...
}
```

Listing 3. Root cause: loss of conditions when `hexDigits > 8`.

Listing 4 shows one test case that can reveal the bug in Listing 3. This test case is manually written by developers to ensure the reproduction of the bug. Among four assertions in Listing 4, the last assertion makes the program crash and its previous three assertions cannot trigger the crash. During daily development, such manually-written test case is hard to be created because software is complex and understanding source code is tedious and time-consuming.

```
public void testLang747() {
  assertEquals(0x8000,       NumberUtils.createNumber
      (``0x8000''));
  assertEquals(0x8000000,    NumberUtils.createNumber
      (``0x8000000''));
  assertEquals(0x7FFFFFFF,   NumberUtils.createNumber
      (``0x7FFFFFFF''));
  assertEquals(0x80000000L, NumberUtils.createNumber
      (``0x80000000''));
}
```

Listing 4. A test case to reproduce Bug 747.

B. Automatic test generation

To reduce the manual effort by developers on crash reproduction, several automatic approaches based on test generation have been proposed [5], [6], [7], [8], [9]. Rößler et al. [7] have proposed a genetic-algorithm based method to trigger a specific path for a given core dump. Chen and Kim [8] propose an automatic method based on symbolic execution to reproduce 31 real-world Java crashes. Xuan et al. [9] mutate existing test cases to generate new test cases to trigger hard-to-reproduce bugs.

However, automatically reproducing a crash is hard due to the complexity of software projects. To date, existing approaches are far away from fully automatic crash reproduction. In this paper, we predict whether a crash is difficult to reproduce or not, instead of directly reproducing the crash. Based on our work, developers can assign a hard crash to a human developer or assign an easy crash to an automatic test generation tool. Such assignment can assist the schedule of bug fixing by avoiding assigning all the crashes to developers.

III. Overall Framework

Fig. 1 illustrates our framework of reproducible crash reproduction. The goal of this framework is to build a predictive model to detect whether a crash is difficult or easy to be reproduced.

As shown in Fig.1, the framework consists of two main phases: the training phase and the deployment phase. The

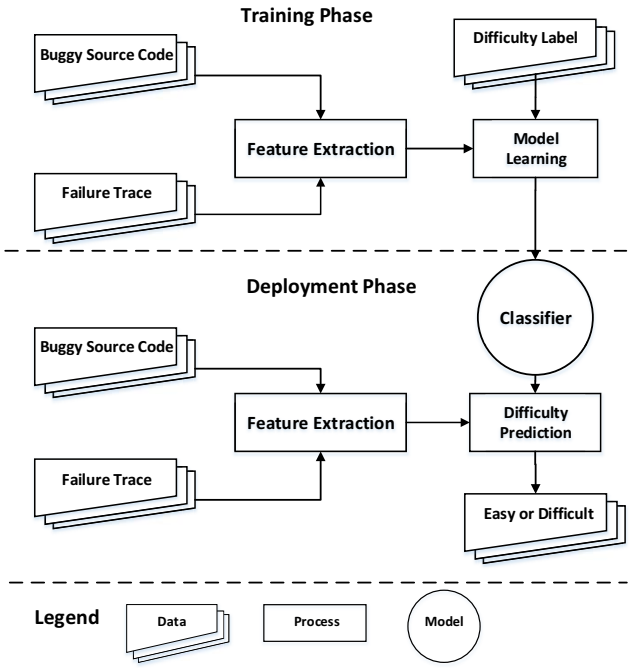

Fig. 1. Overall framework of reproducible crash detection.

training phase extracts features from both buggy source code and the failure trace; historical crash data with these features as well as the difficulty labels (difficult to be reproduced or not) are used to train a classifier, such as Naive Bayes. The deployment phase detects whether a newly-coming crash is difficult based on the trained classifier.

A. Training Phase

To detect whether a crash is difficult to be reproduced, we view each crash as a feature vector with 23 features and its difficulty label (difficult or easy). Each of the 23 features can be directly extracted from a given crash. In the training phase, we train a classifier with the historical crash data, i.e., known crashes with their difficulty labels.

Reproducible crash detection is a binary classification problem. We extract features from a crash (from both source code and the failure trace) based on an assumption that these features correlates the difficulty of crashes. For instance, the LoC (Lines of Code) is widely-used to measure the complexity of programs. Then we use the LoC to measure the complexity of the crash-related code. Details of 23 features in our work are shown in Section III-C.

B. Deployment Phase

For a newly-coming crash, our work aims to predict whether this crash is difficult to be reproduced. Given a recorded crash, the same 23 features as in Section III-A are extracted. Based on the trained classifier, we label the newly-coming crash as a difficult one or an easy one. Then the user of our approach can decide to assign the crash to a human developer since it is difficult or to assign the crash to an automatic tool of test generation. This assignment helps to improve the schedule of solving crashes and to reduce the human labor of crash reproduction of developers.

TABLE I
LIST OF 23 FEATURES FROM SOURCE CODE AND FAILURE TRACE

ID	Description
Features from the buggy source code (16 features)	
BS1	Number of Java files in the whole project
BS2	Number of classes in the whole project
BS3	Number of variables in the exception class
BS4	Number of methods in the exception class
BS5	Number of import statements in the exception class
BS6	Whether the exception class is inherited from others
BS7	LoC of comments in the exception class
BS8	LoC of methods in the direct cause method
BS9	Number of parameters in the direct cause method
BS10	Number of local variables in the direct cause method
BS11	Number of if-statements in the direct cause method
BS12	Number of loops in the direct cause method
BS13	Number of try-catch blocks in the direct cause method
BS14	Number of assignments in the direct cause method
BS15	Number of invocations in the direct cause method
BS16	Whether has a return value in the direct cause method
Features from the failure Trace (7 features)	
FT1	Type of the exception in the failure trace
FT2	Number of lines of the failure trace, without JUnit logs
FT3	Number of classes in the failure trace
FT4	Number of methods in the failure trace
FT5	Whether overloaded methods exist in the failure trace
FT6	Length of the name of the exception class
FT7	Length of the name of the exception method

C. Feature Extraction

In our proposed framework, we extract 23 features to model the characteristics of crashes, including 16 features from the buggy source code and 7 features from the failure trace. The detailed description of the extracted features is listed in Table I.

1) Features from the buggy source code: We extract 16 features from the buggy source code, i.e., BS1 to BS16.

BS1 and BS2 represent the general information about the whole project, including the number of Java files and the number of classes. These features assume that a Java class with inner classes or anonymous classes may make the crash reproduction difficult.

BS3 to BS7 capture the details of the exception class. An *exception class* in our work denotes the class, which the direct cause method of the failure trace belongs to. BS3, BS4, and BS5 record the number of local variable, contained method, and import statements in the exception class, respectively. Among these features, BS3 only records the general methods in the exception class and ignores the class constructors; BS6 indicates whether there exist the inheritance in the exception class; BS7 records the LoC of the exception class.

BS8 to BS16 extract vital features from the direct cause method. BS8 counts the LOC of the direct cause method. Then

BS9, BS10, BS11, BS12, and BS13 represent the number of parameters, local variables, if-statements, loops, and try-catch blocks in the method, respectively. In BS12, the number of loops is the sum of the number of for-loops, while-loops, do-while-loops, and foreach-loops. BS14 captures assignments, which indicate implicit and explicit data dependencies of variables in the method. BS15 represents the number of invocations. We design this feature by assuming that the difficulty of crash reproduction increases with the growth of method invocations BS16 counts the number of return values in the method.

2) Features from the failure trace: We extract 7 features from the failure trace, i.e., FT1 to FT7.

FT1 shows the type of exception. In our preliminary experiment, we collect 13 kinds of exceptions, including the null pointer, the index of boundaries, the number format, etc. FT2 counts the number of lines of the failure trace. FT3 and FT4 record the number of exception classes and methods in the failure trace, respectively. FT5 checks whether there exist overloaded methods in the failure trace. In addition, the length of the name of the exception class name and the direct cause method are recorded in FT6 and FT7, respectively. We assume that there may exist implicit relevance between these two features and the difficulty of crash reproduction.

IV. Experiments

In this section, we present the dataset under evaluation, the labeling of hard and easy crash reproduction, the evaluation setup, the preliminary results, and the threats to the validity.

A. Dataset

In our preliminary experiment, we choose 45 real-world crashes from the Defects4J dataset [10]. Defects4J is a Java bug database, consisting of five open source projects with 357 reproducible real-world bugs. Due to the advantage of reproducibility of bugs and the unified structure in organization, many experiments are conducted on the Defects4J database. For instance, Durieux et al.[12] have investigated results on three typical repair methods on Defects4J. We choose three out of five projects: Apache Commons Lang, Joda Time, and JFreeChart. Each bug in these projects is organized with the source code of programs and test cases; test cases are organized in JUnit with at least one failing test case that can trigger the bug.

Bugs in the Defects4J is not originally extracted for crash reproduction. To facilitate the study of reproducible crashes, we only extract bugs, which throw explicit exceptions. In details, the selection criterion is that the type of exceptions and the direct root method (see Section II-A) are already recorded in the failure trace. For instance, if a crash is caused by the violation of an assertion in JUnit, the direct root method is ignored in the failure trace. We filter out such crashes with assertion violation. Among all bugs in these three projects, we select 45 crashes as our evaluation dataset.

Table II shows the basic statistics on the chosen projects. Column "#Selected crashes" shows the number of selected crashes; Column "#Total bugs" shows the original number of bugs in each project; Columns "KLoc", "#Test cases", and "Test KLoC" show the basic statistics for the first crash in each project. Column *KLoC* measures the code with 1,000 lines of Code. Apache Commons Lang is a library of java language extension utilities;[2] Joda-Time is a library for the application of time or date processing;[3] and JFreeChart is a library for drawing professional charts in Java applications.[4]

TABLE II
Selected crashes from the Defects4J dataset

Project	#Selected crashes	#Total bugs	KLoc	#Test cases	Test KLoc
Lang	25	65	22	2,245	6
Joda-Time	10	27	28	4,130	53
JFreeChart	10	26	96	2,205	50
Total	45	118	146	8,580	109

B. Labeling difficult or easy crash reproduction

In this paper, we aim to detect whether reproducing a crash is difficult or easy. In practice, there are many ways of distinguishing a hard one from an easy one. For instance, a hard crash reproduction could be a crash that is reproduced over ten days; or a hard one could be a crash that is reproduced by over two developers. In our work, we do not discuss which way is the best to label a hard or an easy crash reproduction. Instead, we use the following heuristic to simplify the labeling process.

To label the difficulty of crash reproduction, we directly count the dependency inside a test case, which triggers the crash. The *dependency number* of a test case is defined as how many dependent method invocations are used to trigger one crash. To identify the dependency inside test cases, we use a Java program slicing tool, JavaSlicer.[5] JavaSlicer [13] is an open source dynamic slicing tool developed by Saarland University. It can trace the Java program executions and then compute dynamic backward slices. By using JavaSlicer, we can get exact the dynamic dependency number of each test case. A test case after executing the slicing keeps the dependency number inside the test case. Fig. 2 shows the distribution of the crashes based on the dependency number. To keep the balance between the difficult crash reproduction and the easy one, we heuristically choose four as the boundary. That is, crashes with 1, 2, 3, or 4 dependencies are referred to as easy ones while crashes with 5 and more dependencies are referred to as difficult one. Among the 45 crash in our experiment, we find that 27 crashes are labeled with easy ones while the other 18 crashes are labeled with difficult ones.

C. Evaluation Setup and Implementation

We evaluate our predictive model in terms of precision, recall, F-measure, and accuracy. These metrics are defined as follows,

[2]Apache Commons Lang, http://commons.apache.org/lang/.
[3]Joda Time, http://joda.org/joda-time/.
[4]JFreeChart, http://jfree.org/jfreechart/.
[5]JavaSlicer, http://github.com/hammacher/javaslicer/.

$$Precision(X) = \frac{\text{\#correctly predicted crashes with } X \text{ label}}{\text{\#crashes that are predicted as } X}$$

$$Recall(X) = \frac{\text{\#correctly predicted crashes with } X \text{ label}}{\text{\#crashes with } X \text{ label}}$$

$$F\text{-}measure(X) = \frac{2 \times Precision(X) \times Recall(X)}{Precision(X) + Recall(X)}$$

$$Accuracy = \frac{\text{\#correctly predicted crashes}}{\text{\#total crashes}}$$

where X denotes either the difficult label or the easy label of crashes.

We count the evaluation based on k-fold cross validation, which is a basic validation method in data mining. This validation randomly divides the dataset into k folds with equal number of instances; in each round, one fold is used in the deployment phase while the other $k-1$ folds are used in the training phase. Then the metric values in k rounds are collected and the average values are calculated. In our work, we use 5-fold cross validation to evaluate our experiment.

We implement our tool with Java JDK 1.7. The feature extraction in Section III-C is based on Spoon [14], a Java program analysis tool; the labeling of crash reproduction in Section IV-B is based on JavaSlicer [13], a dynamic slicing tool for Java; the training phase and the deployment phase in Section III are based on Weka.[6] Weka [15] is a data mining tool suite, which consists of many existing machining learning algorithms.

In our work, without loss of generality, we use five typical classifiers, including Naive Bayes (the multi-nomial implementation of Naive Bayes), Random Forest (a state-of-the-art decision tree), C4.5 (a typical decision tree), SMO (Sequential minimal optimization), and KStar (a lazy learning classifier).

D. Results

We use five typical classifiers to evaluate our approach. Table III shows the metric values of precision, recall, F-measure, and accuracy, respectively.

First, we can find that Naive Bayes has the highest accuracy of 0.644 while the other classifiers are not higher than 0.600. SMO has the lowest accuracy of 0.533 among the five classifiers.

Second, Naive Bayes has the highest precision for both two classes (difficult and easy ones) of 0.762 and 0.542, respectively. We notice that Naive Bayes reaches the highest recall of the difficulty class and does not perform well in the recall of the easy class, which is 0.593 and the lowest among the five. The similar result occurs in F-measure: Naive Bayes reaches the highest F-measure of 0.619 in the difficult class but the second highest in the easy class. Random Forest does have the highest recall and F-measure of the easy class (0.741 and 0.678, respectively) while those of the difficult class are low (0.333 and 0.387, respectively). Detecting reproducible crashes with Random Forest may cause the untrusted results.

[6]Weka, http://www.cs.waikato.ac.nz/ml/weka/.

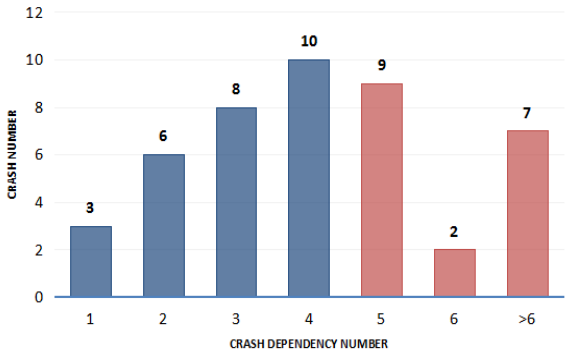

Fig. 2. Distribution of the crashes by counting the number of invocations that trigger the crash.

Third, Naive Bayes has the best stability among precision, recall and F-measure when comparing with the other four classifiers. For instance, the easy class by SMO can reach the recall of 0.704, but the difficult class just reaches 0.278.

Generally speaking, our approach based on the Naive Bayes classifier performs well, even with our small dataset of 45 crashes. The other four classifiers can work but have not achieved better results than the Naive Bayes. Since our experiment is preliminary, the empirical results can be improved in the future. The threats to the validity of our work are shown in Section IV-E.

E. Threats to Validity

In this work, we mainly consider two kinds of threats to the validity: internal validity and external validity.

Internal validity. In our preliminary experiment, we use JavaSlicer to find the dependency inside a manually-written test case. To our knowledge, all program slicing tools cannot guarantee an exactly accurate result when slicing real-world programs. Hence, the number of invocations that are used to trigger a crash may be slightly disturbed. In our work, we distinguish the difficulty of crash reproduction by finding the mean number of invocations of all crash data. The boundary between the difficult crash reproduction and the easy one is decided by empirical data. It is possible that this boundary changes when we move to another dataset. In our work, we only use the boundary to show that it is feasible to learn a classifier to detect difficult or easy crash production.

External Validity. The dataset of crash data in our preliminary work only contains 45 crashes. These crash data are collected from the real-world bugs in the Defects4J dataset. However, the number of crashes is small. Many classification techniques, including the ones in our experiment, are limited by the number of data. We will add more crashes in our future work to enlarge the data under evaluation.

V. Related Work

To reduce the manual work in crash reproduction, many automatic approaches to crash reproduction have been proposed. Artzi et al. [5] create ReCrash, which reproduces

TABLE III
METRIC VALUES OF PRECISION, RECALL, F-MEASURE, AND ACCURACY FOR FIVE CLASSIFIERS, BASED ON 5-FOLD CROSS VALIDATION

Classifier	Precision		Recall		F-measure		Accuracy
	Easy	Difficult	Easy	Difficult	Easy	Difficult	
Naive Bayes	**0.762**	**0.542**	0.593	**0.722**	0.667	**0.619**	**0.644**
Random Forrest	0.625	0.462	**0.741**	0.333	**0.678**	0.387	0.578
C4.5	0.667	0.500	0.667	0.500	0.667	0.500	0.600
SMO	0.594	0.385	0.704	0.278	0.644	0.323	0.533
KStar	0.643	0.471	0.667	0.444	0.655	0.457	0.578

crashes via storing partial copies of method arguments in the memory. Their paper is one of the pioneering works in test case generation based crash reproduction. Jin and Orso [6] propose BugRedux, which preserves bug reports that save the executive information to replay the failure. Rößler et al. [7] design ReCore, which first use evolutionary test generation to reconstruct the input data from saved core dumps. A state-of-the-art approach is Star, presented by Chen and Kim [8]. This approach uses symbolic execution to synthesize test cases for crash reproduction; 31 out of 55 Java crashes can be successfully reproduced. Different from the above methods that generate test case from scratch, Xuan et al. [9] recently propose an approach to crash reproduction via test case mutation (i.e., updating existing test case to generate new test case to trigger the crash). This method leverages existing test cases and target the hard-to-reproduce crashes.

To the best of our knowledge, our work is the first work to predict whether a crash is difficult to be reproduced. Researchers have proposed several works to identify whether one task can succeed. Le and Lo [16] build a predictive model, based on support vector machine, to detect whether the fault localization can be trust or not. Xia et al. [17] propose a novel approach TIE, which updates the classification model by new data to recommend suitable reviewers to examine code changes.

VI. CONCLUSION AND FUTURE WORK

In this study, we propose an automatic approach to reproducible crash detection. This approach builds a predictive model to detect the difficulty of reproducing a given crash. Based on this detection, project managers can assign a difficult crash reproduction task to human developers as well as assign an easy task to automatic test generation tools; such assignment can dramatically save the time cost of developers.

In future work, we plan to enlarge the dataset of our experiment. In this paper, our preliminary result only consists of 45 crashes in Defects4J; we believe that more training data will improve the predictive result by our approach. We also would like to design more features that can capture the characteristics of crashes, which lead to a better model. This work may need manual effort of reading and comprehending crashes.

ACKNOWLEDGMENT

This work is partly supported by the National Natural Science Foundation of China (under grants 61502345, 61272275).

REFERENCES

[1] B. Beizer, *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., 1990.
[2] J. Xuan, B. Cornu, M. Martinez, B. Baudry, L. Seinturier, and M. Monperrus, "B-refactoring: Automatic test code refactoring to improve dynamic analysis," *Information & Software Technology*, vol. 76, pp. 65–80, 2016.
[3] B. R. S. Pressman, "Third edition), software engineering: A practitioner's approach," 2010.
[4] S. De Simone, "Lessons learned from apple's gotofail bug," 2014, http://www.infoq.com/news/2014/02/apple_gotofail_lessons.
[5] S. Artzi, S. Kim, and M. D. Ernst, "Recrash: Making software failures reproducible by preserving object states." in *ECOOP*, 2008, pp. 542–565.
[6] W. Jin and A. Orso, "Bugredux: Reproducing field failures for in-house debugging," in *International Conference on Software Engineering*, 2012, pp. 474–484.
[7] J. Rößler, A. Zeller, G. Fraser, C. Zamfir, and G. Candea, "Reconstructing core dumps," in *IEEE Sixth International Conference on Software Testing, Verification and Validation*, 2013, pp. 114–123.
[8] N. Chen and S. Kim, "Star: Stack trace based automatic crash reproduction via symbolic execution," *IEEE Transactions on Software Engineering*, vol. 41, no. 2, pp. 198–220, 2015.
[9] J. Xuan, X. Xie, and M. Monperrus, "Crash reproduction via test case mutation: let existing test cases help," in *Joint Meeting on Foundations of Software Engineering*, 2015, pp. 910–913.
[10] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.
[11] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. L. Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2016.
[12] 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
[13] C. Hammacher, K. Streit, S. Hack, and A. Zeller, "Profiling java programs for parallelism," in *The Workshop on Multicore Software Engineering*, 2009, pp. 49–55.
[14] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A library for implementing analyses and transformations of java source code," *Software Practice & Experience*, 2015.
[15] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *Acm Sigkdd Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, 2010.
[16] T. D. B. Le and D. Lo, "Will fault localization work for these failures? an automated approach to predict effectiveness of fault localization tools," in *IEEE International Conference on Software Maintenance*, 2013, pp. 310–319.
[17] X. Xia, D. Lo, X. Wang, and X. Yang, "Who should review this change?: Putting text and file location analyses together for more accurate recommendations," in *IEEE International Conference on Software Maintenance and Evolution*, 2015, pp. 261–270.