Botsing, a Search-based Crash Reproduction Framework for Java

Pouria Derakhshanfar
p.derakhshanfar@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Xavier Devroey
x.d.m.devroey@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Annibale Panichella
a.panichella@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Andy Zaidman
a.e.zaidman@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Arie van Deursen
arie.vandeursen@tudelft.nl
Delft University of Technology
Delft, The Netherlands

ABSTRACT

Approaches for automatic crash reproduction aim to generate test cases that reproduce crashes starting from the crash stack traces. These tests help developers during their debugging practices. One of the most promising techniques in this research field leverages search-based software testing techniques for generating crash reproducing test cases. In this paper, we introduce Botsing, an open-source search-based crash reproduction framework for Java. Botsing implements state-of-the-art and novel approaches for crash reproduction. The well-documented architecture of Botsing makes it an easy-to-extend framework, and can hence be used for implementing new approaches to improve crash reproduction. We have applied Botsing to a wide range of crashes collected from open source systems. Furthermore, we conducted a qualitative assessment of the crash-reproducing test cases with our industrial partners. In both cases, Botsing could reproduce a notable amount of the given stack traces.

Demo. video: https://www.youtube.com/watch?v=k6XaQjHqe48
Botsing website: https://stamp-project.github.io/botsing/

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Search-based software engineering**.

KEYWORDS

Search-based software testing, crash reproduction, Botsing

ACM Reference Format:
Pouria Derakhshanfar, Xavier Devroey, Annibale Panichella, Andy Zaidman, and Arie van Deursen. 2020. Botsing, a Search-based Crash Reproduction Framework for Java. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3324884.3415299

ASE '20, September 21–25, 2020, Virtual Event, Australia

1 INTRODUCTION

Crashes are usually reported to developers through an issue tracker. For Java programs, most of the time, these reports contain a stack trace, which provides information such as the *type of the exception* causing the crash and the stack of method calls (the *frames*) through which the exception has occurred. A developer relies on this stack trace to understand the root cause of the crash, debug, and fix the software. Zeller describes how writing a test able to reproduce a reported crash is one of the helpful practices to ease debugging [19]. Similarly, Soltani *et al.* [16] indicate that crash reproducing tests help developers to fix bugs faster. Eventually, such tests can be adapted and added to the test suite to prevent future regressions. However, reproducing a crash using its reported stack trace is a laborious and time-intensive task [16]. Also, we observed that manually reproducing a crash requires an experienced developer who has the proper amount of knowledge about the software.

Many automated approaches for crash reproduction have been introduced in the literature to ease the debugging process [2, 11, 13, 15, 16, 18]. These approaches either use *runtime data* or the *stack trace* to perform crash reproduction. For the former, the accuracy depends on the amount of data considered. However, such data are not always available due to the overhead induced by the data collection, or privacy violation concerns. In contrast, the latter approaches solely rely on stack traces, collected from issue reports or execution logs.

Among the different stack-trace-based crash reproduction approaches, Rößler *et al.* [13] and Soltani *et al.* [16] rely on Search-Based Software Testing (SBST) to automatically generate a *crash reproducing test*. Soltani *et al.* [16] empirically showed that evolutionary approaches based on guided operators outperform other existing approaches and confirmed the usefulness of the generated tests for debugging purposes.

In this paper, we present Botsing: an open-source, extendable search-based crash reproduction framework. Botsing implements search-based crash reproduction approaches introduced in previous studies [13, 15, 16]. The tool takes as input a stack trace and software under test. Then, it starts a single-objective or multi-objective search process to generate a test reproducing the crash.

Botsing has been designed as an extendable framework for implementing new features and search algorithms for crash reproduction. For example, in our recent study on the impact of various *seeding* strategies on crash reproduction, we have implemented

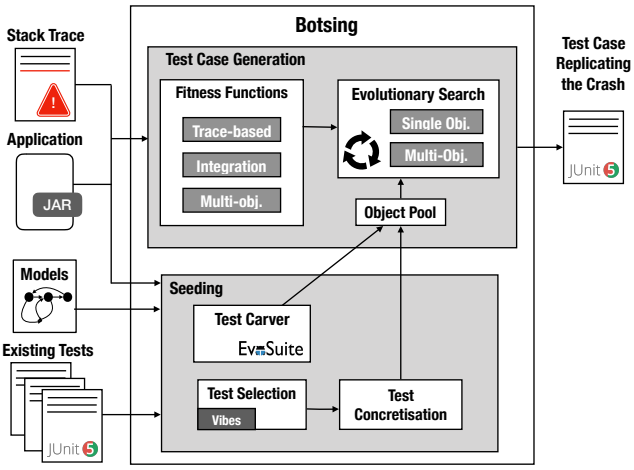

Figure 1: Botsing architecture overview

```
0  java.lang.ArrayIndexOutOfBoundsException: 4
1    at [...].FastDateParser.toArray(FastDateParser.java:413)
2    at [...].FastDateParser.getDisplayNames([...]:381)
3    at [...].FastDateParser$TextStrategy.addRegex([...]:664)
4    at [...].FastDateParser.init([...]:138)
5    at [...].FastDateParser.<init>([...]:108)
6    [...]
```

Figure 2: LANG-9b crash stack trace [10, 14]

```
@Test(timeout = 4000)
public void test0()  throws Throwable  {
  Locale locale0 = FastDateParser.JAPANESE_IMPERIAL;
  TimeZone timeZone0 = TimeZone.getDefault();
  FastDateParser fastDateParser0 = null;
  fastDateParser0 = new FastDateParser("GMTJST", timeZone0,
       locale0);
  }
```

Figure 3: LANG-9b crash reproducing test [14]

multiple seeding strategies in Botsing [4]. These seeding strategies leverage existing knowledge about the software under test to ease the test generation process. To evaluate these strategies, we used 124 real-world crashes from JCrashPack [14], a benchmark for Java crash reproduction tools. The results show that Botsing can reproduce 66 (out of 124) crashes without seeding. This number is improved to 70 by using model seeding.

From an industrial perspective, Botsing is used by our partners in the STAMP project.[1] They confirmed the relevance of Botsing for debugging and fixing application crashes [1]. The feedback —as well as the crash reproducing test cases— from our partners using Botsing are openly available in the STAMP GitHub repository.[2]

2 USAGE

Botsing is available as an open-source tool at https://github.com/STAMP-project/botsing. A user can download the jar file and run the tool using the command line according to the existing documentations [17]. For the example in Figure 2, we can call Botsing with the following command: `java -jar botsing-reproduction.jar -project_cp <path> -crash_log LANG-9b.log -target_frame 5`, where `<path>` is the directory with the jar files of *Apache commons lang*. At the end of the search process, Botsing generates the test case in Figure 3. In general, a developer needs to provide a log file containing the stack trace and the classpath to the jar file and all the dependencies of the software under test. To activate seeding strategies, she also needs to provide the compiled version of test cases (for test seeding) or the behavioral models generated by the model inference module.

3 BOTSING

The goal of Botsing is finding a test case that reproduces a crash, according to the corresponding stack trace. In this section, we first present the general workflow, and, next, describe the seeding strategies implemented in Botsing.

Figure 1 presents the general architecture of Botsing. The main component is the *Test Case Generation* module, which takes as input the binaries of the application and a Java stack trace. A stack trace contains two parts: the first line, indicating the type of the exception, followed by a list of *frames*. Each frame points to a specific line of code in the software under test. For instance, the stack trace of Figure 2, caused by a bug in the Apache *commons-lang* library [10, 14], indicates that an `ArrayIndexOutOfBoundsException` is thrown (at line 0) and propagated through different frames (from line 1 to line 6), indexed from 1 (at line 1) to the total number of frames in the stack trace.

To generate a unit test, Botsing requires to set the *target frame* and its associated `target class` for which the unit test will be generated. For instance, when setting the target frame to 5 for the stack trace in Figure 2, Botsing generates a unit test for the target class `FastDateParser`, presented in Figure 3. The last statement of the test case (calling the *target method* `FastDateParser.<init>`) triggers a crash, generating the same stack trace as in Figure 2.

Once the input is provided, Botsing starts an *Evolutionary Search* to generate a test case that triggers the target crash. To guide the search, at each iteration, the *Fitness Function* measures the adequacy level of the current set of generated tests *w.r.t.* their ability to reproduce the crash. By default, the tests in this search process are generated randomly to promote exploration of the search space. Each generated test uses objects in a random manner through random method sequences. However, there are no guarantees that these random usages of the objects are correct *w.r.t.* the explicit or implicit specification of the classes, which can lead to misguiding the search process. To alleviate such a limitation, Botsing includes a *seeding* mechanism that can be activated to generate objects and method calls closer to real-world scenarios, based on the knowledge of the software. The seeding process can rely on two types of knowledge: (i) the existing manually-written tests, or (ii) the models (*i.e.*, transition systems) abstracting usages of each class.

In the remaining part of this section, we describe the test case generation and the seeding mechanism, including the model generation process.

[1] Available at http://stamp-project.eu/
[2] Available at https://github.com/STAMP-project/botsing-usecases-output.

3.1 Test Case Generation

In the first step of the search process, Botsing generates a *random initial population* of test cases such that: (i) each individual in this population is a test case containing a sequence of calls to methods of the target class; and (ii) the last method called in the sequence is the target method. After generating the random initial population, Botsing starts an evolutionary search process to refine the test cases until one can reproduce the stack trace. At each iteration, Botsing will select the best individuals in the population to build the next generation of test cases using crossover and mutation.

To select the best test case, Botsing relies on a fitness function to compute a distance measuring how close the execution of a test case is to reproducing the crash. Three fitness functions are available in Botsing. The default fitness function is the *single objective fitness function* [16]. This fitness function combines three conditions in one single measure (*i.e.,* one objective): (i) if the generated test reaches the line of the target frame, (ii) if it throws the same type of exception, and (iii) if the occurred stack trace in the generated test is similar to the given one. As an alternative, Botsing can use multi-objectivization approaches (to improve diversity), which splits the single-objective fitness function in three independent sub-objectives [15], or add two helper objectives (method-sequence diversity and test length minimization) [6]. The last fitness function, introduced by Rößler *et al.* [13], checks if the generated test covers each frame one by one and, after covering all of the frames, checks the type of the thrown exception.

After selecting the fittest test cases, Botsing uses the two standard evolutionary operations [9] to produce the next generation of test cases: *single-pointed crossover* and *mutation*. After the application of each operator, there is a risk that the test case no longer contains the target method. To prevent such test cases from being included in the next generation, we use an additional operator to repair the evolved chromosomes (if needed) [16]. Botsing continues the search until it produces a test case able to reproduce the given stack trace or until it exceeds its given time budget (timeout).

3.2 Seeding

One of the challenges in search-based crash reproduction is reaching the state that throws the given exception [14]. In some cases, it is hardly feasible using only random test generation. Seeding addresses that problem by providing additional information to the search process, based on the knowledge of the system. Botsing implements two seeding strategies: (i) the *Test Seeding* strategy introduced by Rojas *et al.* [12] and suitably adapted for crash reproduction, and (ii) a novel seeding strategy called *Behavioral Model Seeding*, introduced in our previous study [4].

Test Seeding. Instead of random generation, test seeding uses the existing test cases, manually-written by developers, and executes them to observe the usages of the different objects created during the execution. During this process, called *carving*, the objects are added to an object pool, used later during the test generation. We adapted EvoSuite's implementation of test seeding to crash reproduction: *i.e.,* we implemented an additional step to check that the carved objects of the target class indeed call the target method (i.e., the methods in the crash stack frames).

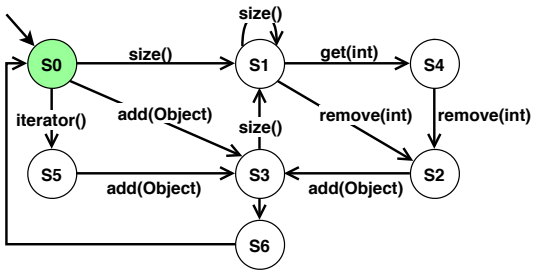

Figure 4: Transition system with the usages of `linkedLists`

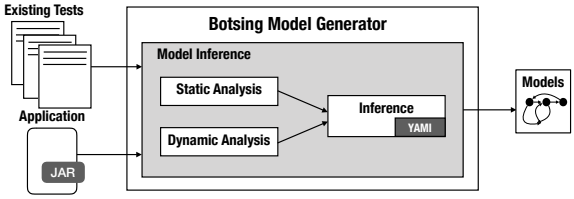

Figure 5: Model Inference architecture overview

3.2.1 Behavioral Model Seeding. The second seeding strategy implemented in Botsing is *Behavioral Model Seeding* [4]. This seeding strategy gets a set of transition systems, representing usages of a set of classes, as input. These models are generated by the `Model Inference` module, described hereafter. Each transition system models the potential method call sequences for a class. For instance, Figure 4 shows the transition system for Java `LinkedLists`. Each transition is a method call, and each path is an *abstract behavior* denoting a potential sequence of method calls. Botsing relies on *VIBeS* [8], a model-based testing tool, to select the most dissimilar paths (*i.e.,* abstract behavior) in a model. Next, it concretizes the abstract behavior to a concrete object with method sequence, and adds it to the object pool. To mimic realistic usages of a class, the objects in the pool are later used during the search process to craft test cases.

Model Inference. Figure 5 shows the architecture of the Model Inference module. This module observes how specific classes are used in the source code and the manually-written test cases to learn a behavioral model. For source code, the module applies *static analysis* and collects all of the sequences of method calls for the classes under analysis. For existing test cases, it performs *dynamic analysis* and executes all the tests to collect sequences of methods (effectively) called on the different objects during test execution. After collecting the call sequences, Botsing abstracts them in transition systems (one transition system *per* class) using *YAMI* [7], a 2-gram model inference tool. Practically, for one system, model inference is a one-time process: *i.e.,* the generated models can be used for different executions of Botsing.

3.3 Implementation

Botsing relies on EvoSuite for code instrumentation, test case manipulation and execution. Concretely, we use `evosuite-client` as a dependency. During the implementation of Botsing, we extend the existing classes in EvoSuite to adapt them to the crash

reproduction problem. For instance, Botsing needs to instrument the classes appearing in the given stack trace. In contrast, EvoSuite is a unit testing tool and it instruments only one class. Hence, for implementing fitness functions for crash reproduction we need to extend the instrumentation of EvoSuite. To make some classes extendable in EvoSuite, we had to change the visibility of some classes in EvoSuite. Hence, we change the access level of some methods in some classes to make sure that we can extend those classes. These changes on EvoSuite are available in our fork from the main repository.[3]

Botsing's architecture is designed to be extendable. For this purpose, most of the classes related to different parts of the genetic algorithm (*e.g.,* fitness functions, genetic algorithm, *etc.*) are designed as factory classes. We also reported the architecture and a contribution guide in the documentation [17].

4 EVALUATION

We use JCrashPack [3, 14], a crash benchmark for evaluating crash reproduction approaches, to assess Botsing. This benchmark contains real-world crashes collected from seven well-known projects, namely *Closure compiler*, *Apache commons-lang*, *Apache commons-math*, *Mockito*, *Joda-Time*, *XWiki*, and *ElasticSearch*. Moreover, to ease benchmarking using JCrashPack, we developed a bash-based execution runner, openly available on GitHub.[4] This experiment runner runs different instances of a crash reproduction tool (here, Botsing) in parallel processes and collects relevant information about the execution in a CSV file.

In our evaluation of the impact of test and model seeding for search-based crash reproduction [4], we ran Botsing without seeding and with each implemented seeding strategy on JCrashPack. Due to the involved randomness in the search process, we repeat each execution 30 times. We observe that Botsing can reproduce 66 (out of 124) crashes without any seeding strategy in a majority of the executions. The implemented seeding strategies help the crash reproduction process to reproduce four additional crashes in the majority of executions.

In total, we run 186,560 independent Botsing runs, distributed among two clusters with 20 CPU-cores, 384 GB memory, and 482 GB hard drive. The results and replication package of this study are openly available on Zenodo [5]. The relevance of Botsing for crash reproduction has been confirmed by our industrial partners [1]. Botsing reproduced 25%, 20%, and 30% of crashes in *TellU*, *XWiki*, and *OW2* projects, respectively. Compared to the complexity of the used projects (for instance, XWiki has an average of 177K non-commenting statements), the reproduction ratios are noteworthy.

5 CONCLUSION

In this paper, we introduced Botsing, an open-source search-based crash reproduction framework, which contains the implementation of the best-performing approaches. It also contains the adapted version of the *Test Seeding* strategy, which was originally introduced for search-based software testing, but which we adapted to the crash replication context. Additionally, Botsing provides a novel seeding strategy, called *Behavioral Model Seeding*.

[3]https://github.com/STAMP-project/evosuite-ramp
[4]https://github.com/STAMP-project/ExRunner-bash

The Botsing framework is developed with extensibility in mind. So, it can be used for implementing new features and genetic algorithms for the crash reproduction problem. We also provide an open-source evaluation infrastructure to ease the assessment process of the new approaches.

In our evaluation, Botsing can reproduce 66 crashes out of 124 hard-to-reproduce crashes, partially or entirely, in the majority of executions. This number increases to 70 crashes when using Behavioral Model Seeding. Also, Botsing has been used by our industrial partners. They managed to reproduce some of their crashes using Botsing and argued that Botsing is helpful for their debugging practices.

ACKNOWLEDGMENTS

This research was partially funded by the EU Project STAMP ICT-16-10 No.731529.

REFERENCES

[1] Mael Audren, Mohamed Boussaa, Lars Thomas Boye, Pierre-Yves Gibello, Jesús Gorroñogoitia, Vincent Massol, Fernando Mendez, Assad Montasser, and Pedro Velho. 2019. STAMP WP5 - D5.7 - Use Cases Validation Report V3. https://www.stamp-project.eu/view/main/deliverables.

[2] Ning Chen and Sunghun Kim. 2015. STAR: Stack trace based automatic crash reproduction via symbolic execution. *IEEE Trans. on Software Engineering* 41, 2 (2015), 198–220. https://doi.org/10.1109/TSE.2014.2363469

[3] Pouria Derakhshanfar and Xavier Devroey. 2020. *JCrashPack: A Java Crash Reproduction Benchmark*. Zenodo. https://doi.org/10.5281/zenodo.3766689

[4] Pouria Derakhshanfar, Xavier Devroey, Gilles Perrouin, Andy Zaidman, and Arie Deursen. 2020. Search-based crash reproduction using behavioural model seeding. *STVR* 30, 3 (may 2020), e1733. https://doi.org/10.1002/stvr.1733

[5] Pouria Derakhshanfar, Xavier Devroey, Gilles Perrouin, Andy Zaidman, and Arie van Deursen. 2019. *Replication package of "Search-based Crash Reproduction using Behavioral Model Seeding"*. https://doi.org/10.5281/zenodo.3673916

[6] Pouria Derakhshanfar, Xavier Devroey, Andy Zaidman, Arie van Deursen, and Annibale Panichella. 2020. Crash Reproduction Using Helper Objectives. In *Genetic and Evolutionary Computation Conference Companion (GECCO '20 Companion)*. ACM, Cancún, Mexico. https://doi.org/10.1145/3377929.3390077

[7] Xavier Devroey, Gilles Perrouin, Maxime Cordy, Hamza Samih, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. 2017. Statistical prioritization for software product line testing: an experience report. *Software & Systems Modeling* 16, 1 (feb 2017), 153–171. https://doi.org/10.1007/s10270-015-0479-8

[8] Xavier Devroey, Gilles Perrouin, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. 2016. Search-based Similarity-driven Behavioural SPL Testing. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems - VaMoS '16*. ACM Press, Salvador, Brazil, 89–96. https://doi.org/10.1145/2866614.2866627

[9] Gordon Fraser and Andrea Arcuri. 2011. Evolutionary Generation of Whole Test Suites. In *Proc. Int'l Conf. on Quality Software (QSIC)*. IEEE, 31–40.

[10] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*. ACM Press, San Jose, CA, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[11] Mathieu Nayrolles, Abdelwahab Hamou-Lhadj, Sofiene Tahar, and Alf Larsson. 2015. JCHARMING: A bug reproduction approach using crash traces and directed model checking. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 101–110. https://doi.org/10.1109/SANER.2015.7081820

[12] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. 2016. Seeding strategies in search-based unit test generation. *Softw. Test. Verif. Reliab.* 26, 5 (2016), 366–401. https://doi.org/10.1002/stvr.1601

[13] Jeremias Rößler, Andreas Zeller, Gordon Fraser, Cristian Zamfir, and George Candea. 2013. Reconstructing core dumps. In *Proc. International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 114–123. https://doi.org/10.1109/ICST.2013.18

[14] Mozhan Soltani, Pouria Derakhshanfar, Xavier Devroey, and Arie van Deursen. 2020. A benchmark-based evaluation of search-based crash reproduction. *Empirical Software Engineering* 25, 1 (jan 2020), 96–138. https://doi.org/10.1007/s10664-019-09762-1

[15] Mozhan Soltani, Pouria Derakhshanfar, Annibale Panichella, Xavier Devroey, Andy Zaidman, and Arie van Deursen. 2018. Single-objective Versus Multi-objectivized Optimization for Evolutionary Crash Reproduction. In *Symposium*

on Search-Based Software Engineering. SSBSE 2018. (LNCS), Thelma Elita Colanzi and Phil McMinn (Eds.), Vol. 11036. Springer, Montpellier, France, 325–340. https://doi.org/10.1007/978-3-319-99241-9_18

[16] Mozhan Soltani, Annibale Panichella, and Arie Van Deursen. 2018. Search-Based Crash Reproduction and Its Impact on Debugging. *IEEE Transactions on Software Engineering* (2018). https://doi.org/10.1109/TSE.2018.2877664

[17] STAMP. 2019. Botsing documentation. https://stamp-project.github.io/botsing/

[18] Jifeng Xuan, Xiaoyuan Xie, and Martin Monperrus. 2015. Crash reproduction via test case mutation: Let existing test cases help. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. ACM Press, New York, New York, USA, 910–913. https://doi.org/10.1145/2786805.2803206

[19] Andreas Zeller. 2009. *Why Programs Fail, Second Edition: A Guide to Systematic Debugging* (2nd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.