JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2015; **00**:1–29
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr

A Bug Reproduction Approach Based on Directed Model Checking and Crash Traces

Mathieu Nayrolles [*, †, 1], Abdelwahab Hamou-Lhadj[1], Sofiène Tahar[2] and Alf Larsson[3]

[1] *SBA Research Lab, ECE, Concordia University, Montréal, Canada*
[2] *HVG Research Lab,ECE Department, Concordia University, Montréal, Canada*
[3] *PLF System Management, Ericsson, R & D, Stockholm, Sweden*

SUMMARY

Reproducing a bug that caused a system to crash is an important task for uncovering the causes of the crash and providing appropriate fixes. In this paper, we propose a novel crash reproduction approach that combines directed model checking and backward slicing to identify the program statements needed to reproduce a crash. Our approach, named JCHARMING (Java CrasH Automatic Reproduction by directed Model checkING), uses information found in crash traces combined with static program slices to guide a model checking engine in an optimal way. We show that JCHARMING is efficient in reproducing bugs from ten different open source systems. Overall, JCHARMING is able to reproduce 80% of the bugs used in this study in an average time of 19 minutes.

Received . . .

KEY WORDS: Automatic Bug Reproduction, Dynamic Analysis, Model Checking, Software Maintenance.

1. INTRODUCTION

The first (and perhaps main) step in understanding the cause of a field crash is to reproduce the bug that caused the system to fail. A survey conducted with developers of major open source software systems such as Apache, Mozilla and Eclipse revealed that one of the most valuable piece of information that can help locate and fix the cause of a crash is the one that can help reproduce it [1].

Bug reproduction is, however, a challenging task because of the limited amount of information provided by the end users. There exist several bug reproduction techniques. They can be grouped into two categories: (a) On-field record and in-house replay [2–4], and (b) In-house crash

*Correspondence to: Mathieu Nayrolles. SBA Research Lab, ECE, Concordia University. Sir George Williams Campus 1455 De Maisonneuve Blvd. W. Montreal, Quebec, Canada H3G 1M8
†Email: m_nayrol@encs.concordia.ca

explanation [5, 6]. The first category relies on instrumenting the system in order to capture objects and other system components at run-time. When a faulty behavior occurs in the field, the stored objects, as well as the entire heap, are sent to the developers along with the faulty methods to reproduce the crash. These techniques tend to be simple to implement and yield good results, but they suffer from two main limitations. First, code instrumentation comes with a non-negligible overhead on the system. The second limitation is that the collected objects may contain sensitive information causing customer privacy issues. The second category is composed of tools leveraging proprietary data in order to provide hints on potential causes. While these techniques are efficient in improving our comprehension of the bugs, they are not designed with the purpose of reproducing them.

In previous work, we proposed a hybrid approach called JCHARMING (Java CrasH Automatic Reproduction by directed Model checkING) [7] that uses a combination of crash traces and model checking to automatically reproduce bugs that caused field failures. Unlike existing techniques, JCHARMING does not require instrumentation of the code. It does not need access to the content of the heap either. Instead, JCHARMING uses the list of functions that are output when an uncaught exception in Java occurs (i.e., the crash trace) to guide a model checking engine to uncover the statements that caused the crash.

Model checking (also known as property checking) is a formal technique for automatically verifying a set of properties of finite-state systems [8]. More specifically, this technique builds a graph where each node represents one state of the program and the set of properties that needs to be verified in each state. For real-world programs, model checking is often computationally impracticable because of the state explosion problem [8]. To address this challenge and apply model checking on large programs, we direct the model checking engine towards the crash using program slicing and the content of the crash trace, and hence, reduce the search space. When applied to reproducing bugs of seven open source systems, JCHARMING achieved 85% accuracy.

This paper extends our previous work [7] with the following main contributions: An extensive validation of JCHARMING by conducting experiments with three additional systems: Mahout, Hadoop, and ActiveMQ, bringing the total number of subject systems used to validate our approach to 10 and the number of bugs to 30. It should be noted that the new systems are larger compared to Those used in [7]. The accuracy of JCHARMING when applied to the 30 bugs is 80%, which is a slight decrease compared to the previous study. This is explained by the fact that the new systems used in this study are relatively larger compared to those in [7]. Another important contribution of this paper is a completely redesigned JUnit test case generation process. This extended version of the paper also includes a detailed analysis of additional bugs, an elaborate discussion of the results, and an updated related work section.

The remainder of this paper is organized as follows: In Section 2, we present related work on crash reproduction. In Section 3, we provide some background information on model checking. JCHARMING is the topic of Section 5. Section 5 is dedicated to the case study, followed by threats to validity. We conclude the paper and sketch future directions in Section 8.

DOI: 10.1002/smr

2. RELATED WORKS

In this section, we put the emphasis on how crash traces are used in crash reproduction tasks. Existing studies can be divided into two distinct categories: (A) on-field record and in-house replay techniques [2, 3, 9, 10], and (B) on-house crash understanding [7, 11–14].

These two categories yield varying results depending on the selected approach and are mainly differentiated by the need for instrumentation. The first category of techniques oversees – by means of instrumentation – the execution of the target system on the field in order to reproduce the crashes in-house, whereas tools and approaches belonging to the second category only use data produced by the crash such as the crash stack or the core dump at crash time. In the first category, tools record different types of data such as the invoked methods [2], try-catch exceptions [15], or objects [4]. In the second category, existing tools and approaches are aimed towards understanding the causes of a crash, using data produced by the crash itself, such as a crash stack [14], previous – and controlled – execution [13], etc.

Tools and approaches that rely on instrumentation face common limitations such as the need to instrument the source code in order to introduce logging mechanisms [2–4], which is known to slow down the subject system. In addition, recording system behavior by means of instrumentation may yield privacy concerns. Tools and approaches that only use data about a crash – such as core dump or exception stack crashes – face a different set of limitations. They have to reconstruct the timeline of events that have led to the crash [7, 14]. Computing all the paths from the initial state of the software to the crash point is a NP-complete problem, and may cause state space explosion [14, 16].

In order to overcome these limitations, some researchers have proposed to use various SMT (satisfiability modulo theories) solvers [17] and model checking techniques [18]. However, these techniques require knowledge that goes beyond traditional software engineering, which hinders their adoption [19].

It is worth mentioning that both categories share a common limitation. It is possible for the required condition to reproduce a crash to be purely external, i.e., the reading of a file that is only present on the hard drive of the customer or the reception of a faulty network packet [7, 14]. Without this input, it may be challenging to reproduce the bug.

2.1. *On-field Record and In-house Replay*

Jaygarl *et al.* created OCAT (Object Capture based Automated Testing) [4]. The authors' approach starts by capturing objects created by the program when it runs on-field in order to provide them to an automated test process. The coverage of automated tests is often low due to lack of correctly constructed objects. Also, the objects can be mutated by means of evolutionary algorithms. These mutations target primitive fields in order to create even more objects and, therefore, improve the code coverage. While not directly targeting the reproduction of a bug, OCAT is an approach that was used as the main mechanism for bug reproduction systems.

Narayanasamy *et al.* [2] proposed BugNet, a tool that continuously records program execution for deterministic replay debugging. According to the authors, the size of the recorded data needed to reproduce a bug with high accuracy is around 10MB. This recording is then sent to the developers and allows the deterministic replay of a bug. The authors argued that with nowadays Internet bandwidth the size of the recording is not an issue during the transmission of the recorded data.

DOI: 10.1002/smr

Another approach in this category was proposed by Clause *et al.* [16]. The approach records the execution of the program on the client side and compresses the generated data. Moreover, the approach keeps traces of all accessed documents in the operating system and also compresses. Overall, the approach is able to reproduce on-field bugs using a file that are 70kb in average. The minimal execution paths triggering the failure are then sent to the developer who can replay the execution on a sandbox simulating the client's environment. This special feature of the approach proposed by Clause *et al.* addresses the limitation where crashes are caused by external causes. While the authors broaden the scope of reproducible bugs, their approach records a lot of data that may be deemed private such as files used for the proper operation of the operating system.

Timelapse [20] also addresses the problem of reproducing bugs using external data. The tool focuses on web applications and allows developers to browse and visualize the execution traces recorded by Dolos. Dolos captures and reuses user inputs and network responses to deterministically replay a field crash. Also, both Timelapse and Dolos allow developers to use conventional tools such as breakpoints and classical debuggers. Similarly, private data are recorded without obfuscation of any sort.

Another approach was proposed by Artzi *et al.* and named ReCrash. ReCrash records the object states of the targeted programs [3]. The authors use an in-memory stack, which contains every argument and object clone of the real execution in order to reproduce a crash via the automatic generation of unit test cases. Unit test cases are used to provide hints to the developers about the buggy code. This approach particularly suffers from the limitation related to slowing down the execution. The overhead for full monitoring is considerably high (between 13% and 64% in some cases). The authors propose an alternative solution in which they record only the methods surrounding the crash. For this to work, the crash has to occur at least once so they could use the information causing the crash to identify the methods surrounding it. ReCrash was able to 100% (11/11) of the submitted bugs.

Similarly to ReCrash, JRapture [9] is a capture/replay tool for observation-based testing. The tool captures execution of Java programs to replay it in-house. To capture the execution of a Java program, the creators of JRapture used their own version of the Java Virtual Machine (JVM) and a lightweight, transparent capture process. Using a customized JVM allows capturing any interactions between a Java program and the system including GUI, files, and console inputs. These interactions can be replayed later with exactly the same input sequence as seen during the capture phase. However, using a custom JVM is not a practical solution. This is because, the authors' approach requires from users to install a JVM that might have some discrepancies with the original one and yield bugs if used with other software applications. In our view, JRapture fails to address the limitations caused by instrumentation because it imposes the installation of another JVM that can also monitor other software systems than the intended ones. RECORE (REconstructing CORE dumps) is a tool proposed by Robler *et al.*. The tool instruments Java byte code to wrap every method in a try-catch block while keeping a quasi-null overhead [15]. RECORE starts from the core dump and tries (with evolutionary algorithms) to reproduce the same dump by executing the subject program many times. When the generated dump matches the collected one, the approach has found the set of inputs responsible for the failure and was able to reproduce 85% (6/7) of the submitted bugs.

DOI: 10.1002/smr

The approaches presented at this point operate at the code level. There exist also techniques that focus on recording user-GUI interactions [10, 21]. Roehm *et al.* extract the recorder data using delta debugging [22], sequential pattern mining, and their combination to reproduce between 75% and 90% of the submitted bugs while pruning 93% of the actions.

Among the approaches presented here, only the ones proposed by Clause *et al.* and Burg *et al.* address the limitations incurred due to the need for external data at the cost, however, of privacy. To address the limitations caused by instrumentation, the RECORE approach proposes to let users choose where to put the bar between the speed of the subject program, privacy, and bug reproduction efficiency. As an example, users can choose to contribute or not to improving the software – policy employed by many major players such as Microsoft in Visual Studio or Mozilla in Firefox – and propose different types of monitoring where the cost in terms of speed, privacy leaks, and efficiency for reproducing the bug is clearly explained.

2.2. *On-house Crash Explanation*

On the other side of the picture, we have tools and approaches belonging to the on-house crash explanation (or understanding), which are fewer but newer than on-field record and replaying tools.

Jin *et al.* proposed BugRedux for reproducing field failures for in-house debugging [11]. The tool aims to synthesize in-house executions that mimic field failures. To do so, the authors use several types of data collected in the field such as stack traces, crash stack at points of failure, and call sequences. The data that successfully reproduced the field crash is sent to software developers to fix the bug. BugRedux relies on several in-house executions that are synthesized so as to narrow down the search scope, find the crash location, and finally reproduce the bug. However, these in-house executions have to be conducted prior the work on the bug really begins. Also, the in-house executions suffer from the same limitation as unit testing, *i.e.*, the executions are based on the developer's knowledge and ability to develop exceptional scenarios in addition to the normal ones. Based on the success of BugRedux, the authors built F3 (Fault localization for Field Failures) [12] and MIMIC [13]. F3 performs many executions of a program on top of BugRedux in order to cover different paths leading to the fault. It then generates many "pass" and "fail" paths, which can lead to a better understanding of the bug. They also use grouping, profiling and filtering, to improve the fault localization process. MIMIC further extends F3 by comparing a model of correct behavior to failing executions and identifying violations of the model as potential explanations for failures.

While being close to our approach, BugRedux and F3 may require the call sequence and/or the complete execution trace in order to achieve bug reproduction. When using only the crash traces (referred to as call stack at crash time in their paper), the success rate of BugRedux significantly drops to 37.5% (6/16). The call sequence and the complete execution trace required to reach 100% accuracy can only be obtained through instrumentation and with an overhead ranging from 10% to 1066%. Chronicle [23] is an approach that supports remote debugging by capturing inputs in the application through code instrumentation. The approach seems to have a low overhead on the instrumented application, around 10%.

Likewise, Zamfir *et al.* proposed ESD [24], an execution synthesis approach that automatically synthesizes failure execution using only the stack trace information. However, this stack trace is extracted from the core dump and may not always contain the components that caused the crash.

DOI: 10.1002/smr

To the best of our knowledge, the most complete work in this category is the one of Chen in his Ph.D thesis [14]. Chen proposed an approach named STAR (Stack Trace based Automatic crash Reproduction). Using only the crash stack, STAR starts from the crash line and goes backward towards the entry point of the program. During the backward process, STAR computes the required condition using an SMT solver named Yices [17]. The objects that satisfy the required conditions are generated and orchestrated inside a JUnit test case. The test is run and the resulting crash stack is compared to the original one. If both match, the bug is said to be reproduced. STAR aims to tackle the state explosion problem of reproducing a bug by reconstructing the events in a backward fashion and therefore saving numerous states to explore. STAR was able to reproduce 64% of the submitted bug (38/64). Also, STAR is relatively easy to implement as it uses Yices [17] and potentially Z3 [25] (stated in their future work) that are well-known and well-supported SMT solvers.

Except for STAR, existing approaches that target the reproduction of field crashes require the instrumentation of the code or the running platform in order to save the stack call or the objects to successfully reproduce bugs. As we discussed earlier, such approaches yield good results 37.5% to 100% but the instrumentation can cause a massive overhead (1% to 1066%) while running the system. In addition, the data generated at run-time using instrumentation may contain sensitive information.

JCHARMING's directed model checking overcomes the state explosion problem of classical model checking techniques and allows the generation of JUnit test cases in a reasonable amount of time. JCHARMING is also easy to deploy. It does not require instrumentation, and hence does not require access to data that may potentially be considered confidential. Moreover, JCHARMING offers better results than approaches that only uses bug report materials. Indeed, our approach was able to reproduce 80% (24/30) of submitted bugs and outperforms STAR (64%, 38/64) and BugRedux (37.5%, 6/16).

3. PRELIMINARIES

Model checking (also known as property checking) will, given a formally defined system (that could be software [18] or hardware based [26]), check if the system meets a specification by testing exhaustively all the states of the system under test (SUT), which can be represented by a Kripke [27] structure:

$$SUT =< S, S_0, T, L > \tag{1}$$

where S is a set of states, S_0 the set of initial states, T the transitions relations between states and L the labeling function which labels a state with a set of atomic properties. Figure 1 presents a system with four steps S_0 to S_3 which have five atomic properties p_1 to p_5. The labeling function L gives us the properties that are true in a state: $L(S_0) = \{p_1\}, L(S_1) = \{p_1, p_2\}, L(S_2) = \{p_3\}, L(S_3) = \{p_4, p_5\}$.

The SUT is said to satisfy a property p at a given time where there exist a sequence of states x leading to a state where p holds. This can be written as:

DOI: 10.1002/smr

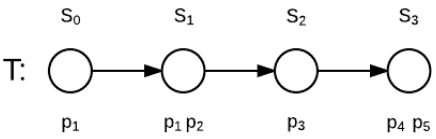

Figure 1. System with four steps S_0 to S_3 which have five atomic properties p_1 to p_5

$$(SUT, x) \models p \tag{2}$$

For the SUT of Figure 1, we can write $(SUT, \{S_0, S_1, S_2\}) \models p_3$ because the sequence of states $\{S_0, S_1, S_2\}$ will lead to a state S_2 where p_3 holds. However, $(SUT, \{S_0, S_1, S_2\}) \models p_3$ only ensures that $\exists x$ such that p is reached at some point in the execution of the program and not that p_3 holds for $\forall x$.

In JCHARMING, we assume that SUTs must not crash under a typical environment. In the framework of this study, we consider a typical environment as any environment where the transitions between the states represent the functionalities offered by the program. For example, in a fair environment, the program heap or other memory spaces cannot be modified. Without this fairness constraint, all programs could be tagged as buggy since we could, for example, destroy objects in memory while the program continues its execution. As we are interested in verifying the absence of unhandled exceptions in the SUT, we aim to verify that for all possible combinations of states and transitions there is no path leading towards a crash. That is:

$$\forall x.(SUT, x) \models \neg c \tag{3}$$

If there exists contradicting path (i.e., $\exists x$ such that $(SUT, x) \models c$) then the model checker engine will output the path x (known as the counter-example), which can then be executed. The resulting Java exception crash trace is compared with the original crash trace to assess if the bug is reproduced. While being accurate and exhaustive in finding counter-examples, model checking suffers from the state explosion problem, which hinders its applicability to large software systems.

To show the contrast between testing and model checking, we use the hypothetical example of Figures 2, 3 and 4 and sketch the possible results of each approach. These figures depict a toy program where from the entry point, unknown calls are made (dotted points) and, at some points, two methods are called. These methods, called `Foo.Bar` and `Bar.Foo`, implement a for `loop` from 0 to `loopCount`. The only difference between these two methods is that the `Bar.Foo` method throws an exception if i becomes larger than two. Hereafter, we denote this property as $p_{i>2}$.

Figure 2 shows the program statements that could be covered using testing approaches. Testing software is a demanding task where a set of techniques is used to test the SUT according to some input. Software testing depends on how well the tester understands the SUT in order to write relevant test cases that are likely to find errors in the program. Program testing is usually insufficient because it is not exhaustive. In our case, using testing will mean that the tester knows what to look for in order to detect the causes of the failure. We do not assume this knowledge in JCHARMING.

DOI: 10.1002/smr

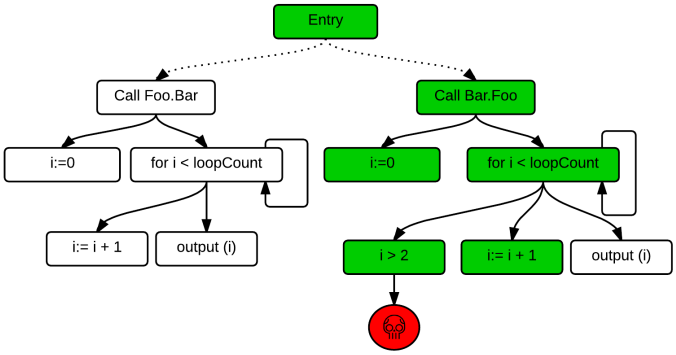

Figure 2. A toy program under testing

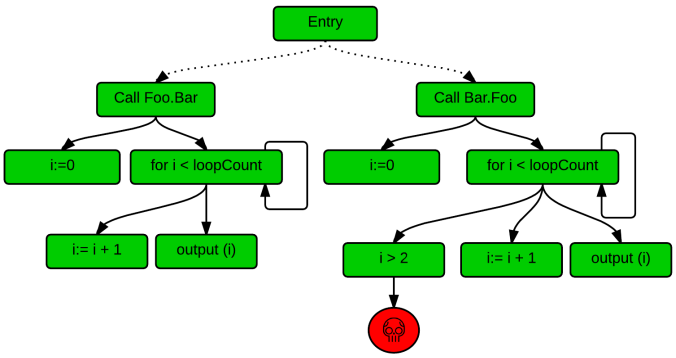

Figure 3. A toy program under model checking

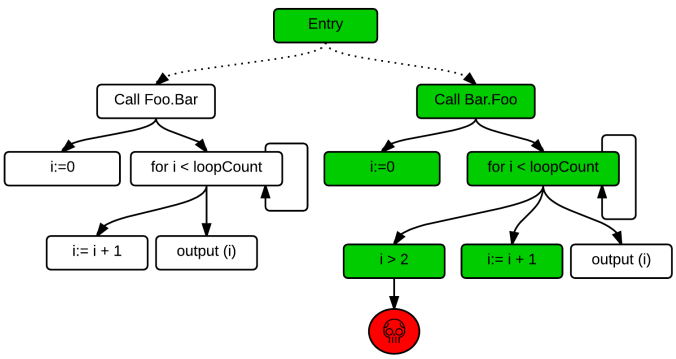

Figure 4. A toy program under directed model checking

Model checking, on the other hand, explores each and every state of the program (Figure 3), which makes it complete, but impractical for real-world and large systems. To overcome the state explosion problem of model checking, directed (or guided) model checking has been introduced [28, 29]. Directed model checking uses insights — generally heuristics — about the SUT in order to reduce the number of states that need to be examined. Figure 4 explores only the states that may lead to a specific location, in our case, the location of the fault. The challenge, however, is to design

DOI: 10.1002/smr

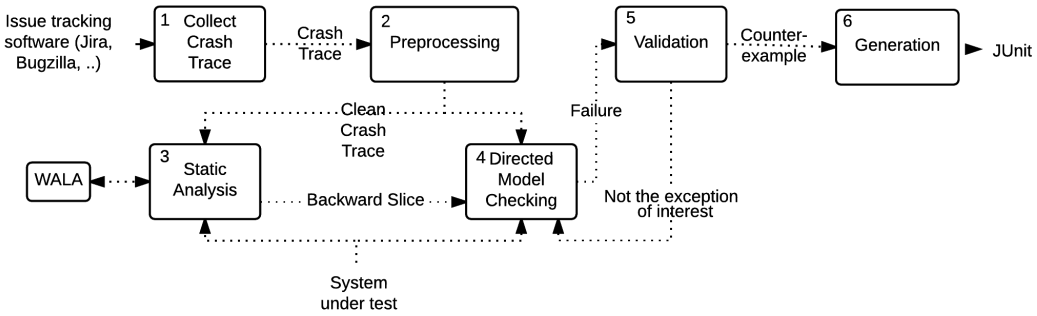

Figure 5. Overview of JCHARMING.

techniques that can guide the model checking engine. As we will describe in the next section, we use crash traces and program slicing to overcome this challenge.

As powerful as directed model checking is, it loses the advantage of model checking over testing: completeness. Indeed, model checking ensures that a system is absolutely correct. With directed model checking, we are not ensuring the absolute correctness of the program, but hunting for a bug inside a subset of the said program.

4. THE JCHARMING APPROACH

Figure 5 shows an overview of JCHARMING. The first step consists of collecting crash traces (also called stack traces), which contain raw lines displayed to the standard output when an uncaught exception in Java occurs. In the second step, the crash traces are preprocessed by removing noise (mainly calls to Java standard library methods). The next step is to apply backward slicing using static analysis to expand the information contained in the crash trace while reducing the search space. The resulting slice along with the crash trace is given as input to the model checking engine. The model checker executes statements along the paths from the main function to the first line of the crash trace (i.e., the last method executed at crash time, also called the crash location point). Once the model checker finds inconsistencies in the program leading to a crash, we take the crash stack generated by the model checker and compare it to the original crash trace (after preprocessing). The last step is to build a JUnit test, to be used by software engineers to reproduce the bug in a deterministic way.

4.1. Collecting Crash Traces

The first step of JCHARMING is to collect the crash trace caused by an uncaught exception. Crash traces are usually included in crash reports and can therefore be automatically retrieved using a simple regular expression. Figure 6 shows an example of a crash trace that contains the exception thrown when executing the program depicted in Figures 2 to 4. More formally, we define a Java crash trace T of size N as an ordered sequence of frames $T = f_0, f_1, f_2, ..., f_N$. A frame represents either a method call preceded with its location in the source code, an exception, or a wrapped exception. In Figure 6, the frame f_0 represents an exception, the frame f_7 a method call (`jsep.Foo.buggy`) in a file (`Foo.java` line 17) leading to the exception, and f_6 represents a wrapped exception.

Prepared using smrauth.cls
DOI: 10.1002/smr

```
1.javax.activity.InvalidActivityException:loopTimes
should be < 3
2. at Foo.bar(Foo.java:10)
3. at GUI.buttonActionPerformed(GUI.java:88)
4. at GUI.access$0(GUI.java:85)
5. at GUI$1.actionPerformed(GUI.java:57)
6. caused by java.lang.IndexOutOfBoundsException : 3
7. at jsep.Foo.buggy(Foo.java:17)
8. and 4 more ...
```

Figure 6. Java InvalidActivityException is thrown in the Bar.Foo loop if the control variable is greater than 2.

It is common in Java to have crash traces that contain wrapped exceptions. Such crash traces are incomplete in the sense that they do not show all the method calls that are invoked from the entry point of the program to the crash point. According to the Java documentation [30], line 8 of Figure 6 should be interpreted as follows: *"This line indicates that the remainder of the stack trace for this exception matches the indicated number of frames from the bottom of the stack trace of the exception that was caused by this exception (the "enclosing exception"). This shorthand can greatly reduce the length of the output in the common case where a wrapped exception is thrown from the same method as the "causative exception" is caught.*"

We are likely to find shortened traces in bug repositories as they are what the user sees without any possibility to expand their content.

In more details, Figure 6 contains a call to the `Bar.foo()` method – the crash location point – and calls to Java standard library functions (in this case, GUI methods because the program was launched using a GUI). As shown in Figure 6, we can see that the first line (referred to as frame f_0, subsequently the next line is called frame f_1, etc.) does not represent the real crash point but it is only the last exception of a chain of exceptions. Indeed, the `InvalidActivity` has been triggered by an `IndexOutOfBoundsException` in `jsep.Foo.buggy`. This crash trace shows also an example of nested try-catch blocks.

4.2. Preprocessing

In the preprocessing step, we first reconstruct and reorganize the crash trace in order to address the problem of nested exceptions. Nested exception refers to the following structure in Java.

```
1 java.io.IOException: Spill failed
...
14 Caused by: java.lang.IllegalArgumentException
...
28 Caused by: java.lang.NullPointerException
```

In such a case, we want to reproduce the root exception (line 28) that led to the other two (lines 14 and 1). This said, we remove the lines 1 to 14. Then, with the aim to guide the directed model checking engine in an optimal way, we remove frames that are beyond our control. These refer to Java library methods and third party libraries. In Figure 6, we can see that Java GUI and event

Prepared using smrauth.cls
DOI: 10.1002/smr

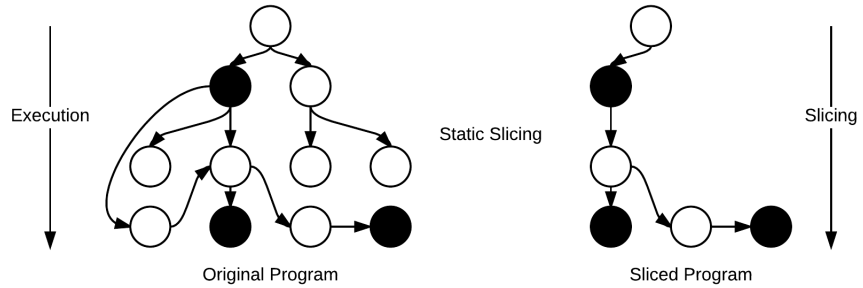

Figure 7. Hypothetical example of static program slicing

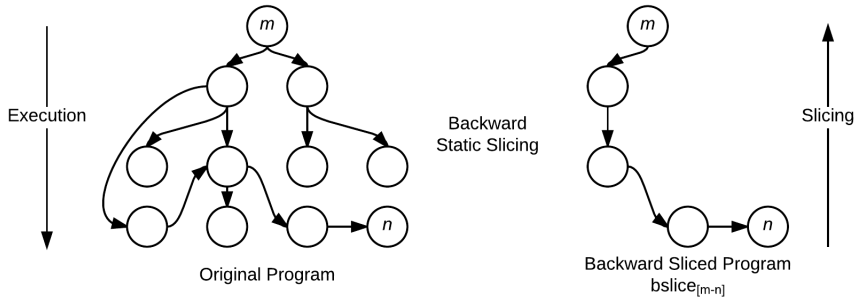

Figure 8. Hypothetical example of backward static program slicing

management components appear in the crash trace. We assume that these methods are not the cause of the crash; otherwise, it means that there is something wrong with the JDK itself. If this is the case, we will not be able to reproduce the crash. Note that removing these unneeded frames will also reduce the search space of the model checker.

4.3. Building the Backward Static Slice

Static analysis of programs consists of analyzing programs without executing them or making assumptions about the inputs of the program. There exist many techniques to perform static analysis (data flow analysis, control flow analysis, theorem proving, etc.) that can be used to improve programs in terms of performance, understandability, or to debug them. One of the products of static analysis is the static slice which takes a program and slicing properties in order to create a smaller program with respect to the slicing properties. This is particularly interesting for JCHARMING as the sliced program, being smaller, will have a smaller state space. In Figure 7, the black states are states that are impacted by the slicing properties. After the slicing, the final static slice contains the black states and the states that are needed to reach them.

In JCHARMING, we use a particular static slicing known as backward static slicing [31]. A backward slice contains all possible branches that may lead to a point *n* in the program from a point *m* as well as the definition of the variables that control these branches [31]. In other words, the slice of a program point *n* is the program subset that may influence the reachability of point *n* starting from point *m*. The backward slice containing the branches and the definition of the variables leading to *n* from *m* is noted as $bslice_{[m \leftarrow n]}$. Figure 8 presents an example of a backward static slice.

Prepared using smrauth.cls
DOI: 10.1002/smr

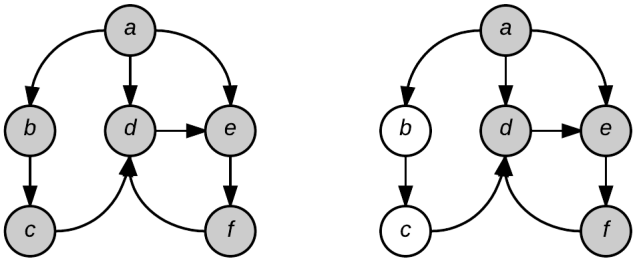

Figure 9. Hypothetical example representing $bslice_{[a \leftarrow d]}$ (left) Vs. $\cup_{i=d}^{a} bslice_{[f_{i+1} \leftarrow f_i]}$ (right) for a crash trace $T = \{d, f, e\}$.

We perform backward static slicing between each frame to compensate for possible missing information in the crash trace. More formally, the final backward slice is represented as follows:

$$bslice_{[f_n \leftarrow f_0]} = bslice_{[f_1 \leftarrow f_0]} \cup bslice_{[f_2 \leftarrow f_1]} \cup ... \cup bslice_{[f_n \leftarrow f_{n-1}]} \tag{4}$$

Note that the union of the slices computed between each pair of frames must be a subset of the final slice between f_0 and the entry point of the program f_n. More formally:

$$\bigcup_{i=0}^{n-1} bslice_{[f_{i+1} \leftarrow f_i]} \subseteq bslice_{[f_n \leftarrow f_0]} \tag{5}$$

Figure 9, presents an example that allow us to grasp Equation 5. In this example, a toy program composed of six methods $a...f$ crashes in d with a crash trace $T = e, f, d$. The backward static slice from the crash point d to the entry point a without using T will be $\{a, b, c, d, e, f\}$ as we do not assume to know which path was taken to reach d. However, if we use T to compute the backward static slice, then, some method can be disregarded as we know which path was taken to reach d. The slice directed by T is $\{a, e, f, d\}$ which is a subset of $\{a, b, c, d, e, f\}$.

In the worst case scenario where there exists one and only one transition between each two frames (this is very unlikely for real and complex systems) $bslice_{[entry \leftarrow f_0]}$ and $\cup_{i=0}^{n} bslice_{[f_{i+1} \leftarrow f_i]}$ yield the same set of states with a comparable computational cost since the number of branches to explore will be the same in both cases.

Algorithm 1 is a high-level representation of how we compute the backward slice between each frame. The algorithm takes as input the pre-processed crash trace $T = f_0, f_1, f_2, ..., f_N$, the byte code of the SUT, and the entry point. From line 1 to line 4 we initialize the different variables used by the algorithm. Namely, our algorithm needs to have the crash trace as an array of frames (line 1), the size N of the crash trace (line 2), a `null` backward static slice (line 4, and an offset set to 1 (line 3). The main loop of the algorithm begins at line 5 and ends at line 13. In this loop, we compute the static slice between the current frame and the next one. If the slice is not empty then we update the final backward slice with the newly computed slice. If the computed slice is empty, it means that Frame `i + 1` was corrupted then we move to Frame `i + 2`, and so on. At the end of

DOI: 10.1002/smr

the algorithm, we compute the slice between the last frame and the entry point of the program, and update the final slice.

Figure presents a step by step graphical representation of Algorithm 1. In this figure, an hypothetical program composed of eleven states ($a...k$) crashes at point k. The produced crash trace is composed of six frames $T = f_0, f_1, f_2, f_3, f_4, f_5$. The frames represent k, i, h, d, b and a, respectively. In the crash trace, f_3 is a corrupt frame and no longer matches a location inside the SUT. This can be the result of a copy-paste error or a deliberate intervention of the reported as shown in Section 6. In such a situation, Algorithm 1 will begin by computing the backward static slice between f_0 (k) and f_1 (i), then between f_1 (i) and f_2 (h). At this point, we passed through the for loop (line 5 to 12) two times, and in both cases the backward static slice was not empty. Consequently, the if statement was equal to $true$ and we combined both backward static slice in the $bSlice$ variable. $bSlice$ is equal to $\{k, j, i, h\}$. Then, we want to compute the backward static slice between f_2 (h) and f_3 (d). Unfortunately, f_3 is corrupted and does not point towards a valid location in the SUT. As a result, the slice between f_2 (h) and f_3 (d) will be empty, and we will go to the $else$ statement (line 10). Here, we simply increment $offset$ by one in order to compute the backward static slice from f_2 (h) and f_4 (b) instead of f_2 (h) and f_3 (d). f_4 is valid and the backward static slice from f_2 (h) and f_4 (b) can be computed and merged to $bSlice$. Finally, we compute the last slice between f_4 (b) and f_5 (a). The final backward static slice is k, i, h, d, b and a.

```
   Data: Crash Stack, BCode, Entry Point
   Result: BSolve
 1 Frame[] frames ← extract frames from crash stack;
 2 Int n ← size of frames;
 3 Int offset ← 1;
 4 Bslice bSlice ← ∅;
 5 for i ← 0; (i < n && offset < n - 1); i++ do
 6 |   BSlice currentBSlice ← backward slice from frames[i] to frames[i + offset];
 7 |   if currentBSlice ≠ ∅ then
 8 |   |   bSlice ← bSlice ∪ currentBSlice;
 9 |   |   offset ← 1;
10 |   else
11 |   |   offset ← offset +1;
12 |   end
13 end
```

Algorithm 1: High level algorithm computing the union of the slices

Our algorithm, given an uncorrupted stack trace will be able to compute an optimum backward static slice based on the frames (Figure 9) and compensate for frames corruption if needs be by `offsetting` the corrupted frame (Figure 10). The compensation of corrupted frames, obviously, comes at the cost of a sub-optimum backward static slice. In our previous example, the transition between b and h could have been omitted if f_3 was not corrupted.

In the rare cases where the final slice is empty (this may happen in situations where the content of the crash trace is seriously corrupted) JCHARMING would simply proceed with non-directed model checking.

DOI: 10.1002/smr

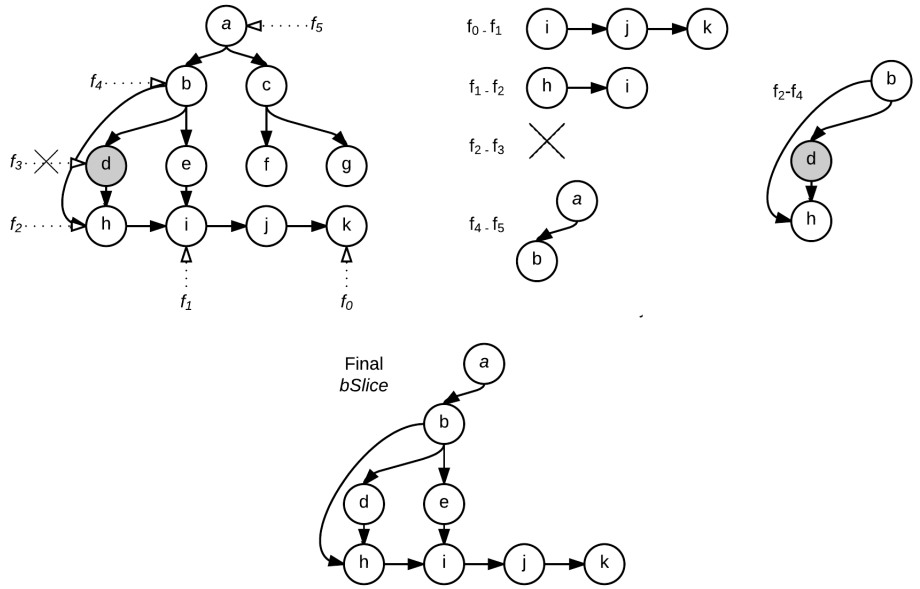

Figure 10. Steps of Algorithm 1 where f_3 is corrupted.

Note that while we allow in JCHARMING the possibility to resort to non-directed model checking, none of the 30 bugs used in this study contained crash traces that were damaged enough for this fall back mechanism to be used.

In order to compute the backward slice, we implement our algorithm as an add-on to the T. J. Watson Libraries for Analysis (WALA) [32], which provide static analysis capabilities for Java Byte code and JavaScript. WALA offers a very comprehensive API to perform static backward slicing on Java Byte code from a specific point to another. We did not need to extend WALA to perform our analysis.

At first sight, it may appear that static slicing alone can be used to reproduce the bug since it contains all the functions that lead to the first user-defined method of the crash trace. The problem is that a static slice may also contain many other parts of the program that are not relevant to the failure. A quick solution to this is to prune out the content of the resulting static slice by eliminating the unwanted statements. This can be achieved by defining specific inputs. The result will be a dynamic slice. This assumes that we know in advance which input we need to provide to the program in order to reach the crash, which defeats the purpose of bug reproduction in the first place.

Using backward slicing, the search space of the model checker that processes the example of Figures 2 to 4, where a crash happens when $i > 2$, is given by the following expression:

$$Sliced_{SUT} = \begin{pmatrix} \bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]} \subseteq SUT, \\ S_0, \\ T. \bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]} \subseteq T.SUT, \\ L \end{pmatrix} \tag{6}$$

Where $\bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]} \subseteq SUT$ is the subset of states that can be reached in the computed backward slice, S_0 the set of initial states, $T. \bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]}$ the subset of transitions relations

DOI: 10.1002/smr

between states that exist in the computed backward slice and L the labeling function which labels a state with a set of atomic properties. Then, in the sliced SUT, we try to find:

$$(Sliced_{SUT}, x) \models p_{i>2} \tag{7}$$

That is, there exists a sequence of state transitions x that satisfies $p_{i>2}$. The only frame that needs to be untouched for the backward static slice to be meaningful is f_0. In Figure 6, f_0 is $at\ Foo.bar(Foo.java:10)$. If this line of the crash trace is corrupt then JCHARMING cannot perform the slicing because it does not know where to start the static backward slicing from. The result is a non-directed model checking which is likely to fail.

4.4. Directed Model Checking

The model checking engine we use in this paper is called JPF (Java PathFinder) [19], which is an extensible JVM for Java byte code verification. This tool was first created as a front-end for the SPIN model checker [33] in 1999 before being open-sourced in 2005. The JPF model checker can execute all the byte code instructions through a custom JVM — known as JVM^{JPF}. Furthermore, JPF is an explicit state model checker, very much like SPIN [33]. This is contrasted with a symbolic model checker based on binary decision diagrams [34]. JPF designers have opted for a depth-first traversal with backtracking because of its ability to check for temporal liveness properties.

More specifically, JPF's core checks for defects that can be checked without defining any property. These defects are called *non functional properties* in JPF and cover deadlock, unhandled exceptions and `assert` expression. In JCHARMING, we leverage the *non functional properties* of JPF as we want to compare the crash trace produced by unhandled exceptions in order to compare them to the bug at hand. Consequently, we do not need to define any property ourselves. However, in JPF, one can define its own properties by implementing `listerners` — very much like what we did for Section 4.6 — that can monitor all actions taken by JPF. which enables the verification of temporal properties for sequential and concurrency Java programs. One of the popular `listener` of JPF is `jpf-ltl`. This listener supports the verification of method invocations or local and global program variables. jpf-ltl can verify temporal properties of method call sequences, linear relations between program variables and the combination of both. In a near future, we might use `jpf-ltl` and the LTL logic to check multi-threaded related crashes, as described in Section 8.

JPF is organized around five simple operations: (i) *generate states*, (ii) *forward*, (iii) *backtrack*, (iv) *restore state* and (v) *check*. In the forward operation, the model checking engine generates the next state s_{t+1}. Each state consists of three distinct components:

- The information of each thread. More specifically, a stack of frames corresponding to methods' calls.
- The static variables of a given class.
- The instance variables of a given object.

If s_{t+1} has successors then it is saved in a backtrack table to be restored later. The backtrack operation consists of restoring the last state in the backtrack table. The restore operation allows restoring any state. It can also be used to restore the entire program as it was the last time we chose between two branches. After each forward, backtrack and restore state operation the check properties operation is triggered.

DOI: 10.1002/smr

In order to direct JPF, we have to modify the *generate states* and the *forward* steps. The *generate states* is populated with the states in $\bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]} \subset SUT$ and we adjust the *forward step* to explore a state if the target state $s_i + 1$ and the transition x to pass from the current state s_i to s_{i+1} are in $\bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]} \subset SUT$ and $x.\bigcup_{i=0}^{entry} bslice_{[f_{i+1} \leftarrow f_i]} \subset x.SUT$.

In other words, JPF does not generate each possible state for the system under test. Instead, JPF generates and explores only the states that fall inside the backward static slice we computed in the previous step. As shown in figure 9, our backward static slice can greatly reduces the space search and is able to compensate for corrupted frames. In short, the idea is that instead of going through all the states of the program as a complete model checker would do, the backward slice directs the model checker to explore only the states that might reach the targeted frame.

4.5. Validation

To validate the result of directed model checking, we modify the *check properties* step that checks if the current sequence of state transition x satisfies a set of properties. If the current state transition x can throw an exception, we execute x and compare the exception thrown to the original crash trace (after preprocessing). If the two exceptions match, we conclude that the conditions needed to trigger the failure have been met and the bug is reproduced.

However, as argued by Kim *et al.* in [35], the same failure can be reached from different paths of the program. Although the states executed to reach the defect are not exactly the same, they might be useful to enhance the understanding of the bug by software developers, and speed up the deployment of a fix. Therefore, in this paper, we consider a defect to be partially reproduced if the crash trace generated from the model checker matches the original crash trace by a factor of t, where t is a threshold specified by the user. Thus, t is the percentage of identical frames between both crash traces.

For our experiments (see Section 6), we set the value of t to 80%. The choice of t should be guided by the need to find the best trade-off between the reproducibility of the bug and the relevance of the generated test cases (the tests should help reproduce the on-field crash). To determine the best t, we made several incremental attempts, starting from $t = 10\%$. For each attempt, we increased t with a factor of 5% and observed the number of bugs reproduced and the quality of the generated tests. Having $t = 80\%$ provided the best trade-off. This said, we anticipate that the tool that implements our technique should allow software engineers to vary t depending on the bugs and the systems under study. Based on our observations, we recommend, however, to set t to 80% as a baseline. It should also be noted that we deliberately prevented JCHARMING to perform directed model checking with a threshold below 50%. This is because the tests generated with such a low threshold during our experiments did not yield qualitative results.

4.6. Generating Test Cases for Bug Reproduction

To help software developers reproduce the crash in a lab environment, we automatically produce the JUnit test cases necessary to run the SUT to cause the bug.

To build a test suite that reproduces a defect, we need to create a set of objects used as arguments for the methods that will enable us to travel from the entry point of the program to the defect location. JPF has the ability to keep track of what happens during model checking in the form of

DOI: 10.1002/smr

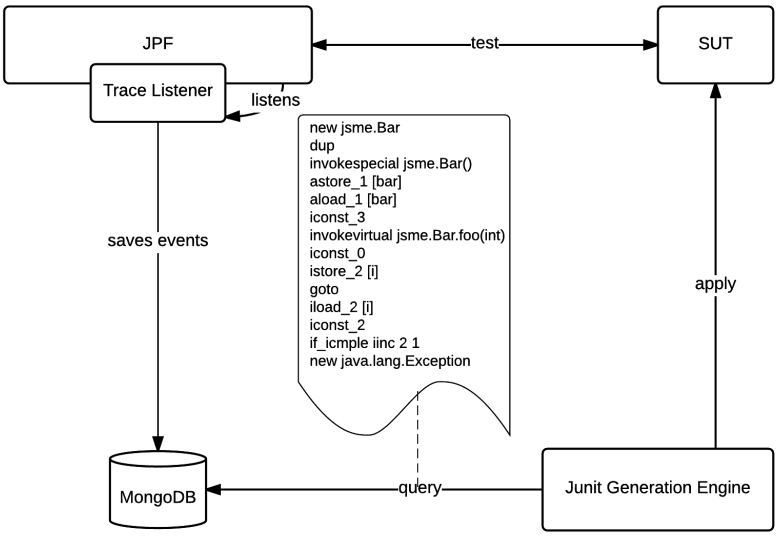

Figure 11. High level architecture of the Junit test case generation

traces containing the visited states and the value of the variables. We leverage this capability to create the required objects and call the methods leading to the failure location.

During the testing of the SUT, JPF emits a trace that is composed of the executed instructions. For large systems, this trace can contain millions of instructions. This trace is stored in memory and therefore can be queried while JPF is running. However, accessing the trace during the JPF execution considerably slows down the checking process as both the querying mechanism and the JPF engine compete with each other for resources. In order to allow JPF to use 100% of the available resources and still be able to query the executed instructions, we implemented a `listener` that listens to the JPF trace emitter. Each time a new instruction is processed by JPF, our listener catches it and then saves it to a MongoDB database to be queried in a post-mortem fashion. Figure 11 presents a high-level architecture of the components of the JUnit test generation process.

When the `validate` step triggers a crash stack with a similarity larger than a factor `t`, the JUnit generation engine queries the MongoDB database and fetches the sequence of instructions that led to the crash of interest. Figure 11 contains a hypothetical sequence of instructions related to the example of Figures 2, 3, 4, which reads : `new jsme.Bar`, `invokespecial jsme.Bar()`, `astore_1 [bar]`, `aload_1 [bar]`, `iconst_3`, `invokevirtual jsme.Bar.foo(int)`, `const_0`, `istore_2 [i]`, `goto`, `iload_2 [i]`, `iconst_2`, `if_icmple iinc 2 1`, `new java.lang.Exception`. From this sequence we know that, to reproduce to crash of interest, we have to (1) create a new object `jsme.Bar` (`new jsme.Bar`, `invokespecial jsme.Bar()`), (2) store the newly created object in a variable named `bar` (`astore_1 [bar]`), (3) invoke the method `jsme.Bar.foo(int)` of the `bar` object with `3` as value (`aload_1 [bar]`, `iconst_3`, `invokevirtual jsme.Bar.foo(int)`). Then, the `jsme.Bar.foo(int)` method will execute a $for-loop$ from $i = 0$ until $i = 3$ and throw an exception at `i` =

Prepared using smrauth.cls
DOI: 10.1002/smr

3 (`const_0`, `istore_2 [i]`, `goto`, `iload_2 [i]`, `iconst_2`, `if_icmple iinc 2 1`, `new java.lang.Exception`).

The generation of the JUnit test itself is based on templates and targets directly the system under test. Templates are excerpts of Java source code with well-defined tags that will be replaced by values. We use templates because the JUnit test cases have common structures. Figure 12 shows our template for generating JUnit test cases. In this figure, each `{% %}` will be dynamically replaced by the corresponding value when the JUnit test is generated. For example, `{% SUT %}` will be replaced by `Ant` if the SUT is Ant. First, we declare four variables that contain the `failure`, the `threshold` above which a given bug is said to be partially reproduced, the `differences` which count how many lines differ between the original failure and the failure produced by JCHARMING and a `StringTokenizer`. The `StringTokenizer` allows to break the original `failure` into tokens. Second, the test method where `{%SUT%}` is replaced by the name of the SUT and `{%STEPS%}` by the steps to make the SUT crash. Then, the crash trace related to the crash is received in the `catch` part of the `try-catch` block. In the `catch` part, we compute the number of lines that do not match the original exception and store it into `differences` [†]. Finally, the `assertTrue` call will assert that the crash traces from the induced and the original crash are at least `threshold` percent similar to each other.

```
public class {% SUT %}_{% BUG %} extends TestCase {

        private static final String failure = {% CRASH_STACK %};
        private static final int threshold = {% THRESHOLD %};
        private static int differences = Integer.MAX_VALUE;
        private static final StringTokenizer tokenizerFailure =
                new StringTokenizer(failure, "\n");

        @Test
        public test{% SUT %}() {
                try {
                        {% STEPS %}
                } catch (Exception e) {
                        // Count the differences
                }
                assertTrue(differences <= tokenizeOriginalFailure
                        .countTokens() / 100 * (threshold - 100));
        }
}
```

Figure 12. Simplified Unit Test template

[†]This code has been replaced by `//Count the differences` to ease the reading.

Prepared using smrauth.cls
DOI: 10.1002/smr

Table I. List of target systems in terms of Kilo line of code (KLoC), number of classes (NoC) and Bug # ID

SUT	KLOC	NoC	Bug #ID
Ant	265	1,233	38622, 41422
ArgoUML	58	1,922	2603, 2558, 311, 1786
dnsjava	33	182	38
jfreechart	310	990	434, 664, 916
Log4j	70	363	11570, 40212, 41186, 45335, 46271, 47912, 47957
MCT	203	1267	440ed48
pdfbox	201	957	1,412, 1,359
Hadoop	308	6,337	2893, 3093, 11878
Mahout	287	1,242	486, 1367, 1594, 1635
ActiveMQ	205	3,797	1054, 2880, 2880
Total	1,517	17,348	30

5. CASE STUDIES

In this section, we show the effectiveness of JCHARMING to reproduce bugs in ten open source systems [‡]. The aim of the case studies is to answer the following question: *Can we use crash traces and directed model checking to reproduce on- field bugs in a reasonable amount of time?*

5.1. Targeted Systems

Table I shows the systems and their characteristics in terms of Kilo Line of Code (KLoC) and Number of Classes (NoC).

Apache Ant [36] is a popular command-line tool to build Makefiles. While it is mainly known for Java applications, Apache Ant also allows building C and C++ applications. We choose to analyze Apache Ant because it has been used by other researchers in similar studies.

ArgoUML [37] is one of the major players among the open source UML modeling tools. It has many years of bug management and, similar to Apache Ant, it has been extensively used as a test subject in many studies.

Dnsjava [38] is a tool for the implementation of the DNS mechanisms in Java. This tool can be used for queries, zone transfers, and dynamic updates. It is not as large as the other two, but it still makes an interesting case subject because it has been well maintained for the past decade. Also, this tool is used in many other popular tools such as Aspirin, Muffin and Scarab.

JfreeChart [39] is a well-known library that enables the creation of professional charts. Similar to dnsjava, it has been maintained over a very long period of time. JfreeChart was created in 2005. It is a relatively large application.

Apache Log4j [40] is a logging library for Java. This is not a very large library, but it is extensively used by thousands of programs. As other Apache projects, this tool is well maintained by a strong open source community and allows developers to submit bugs. The bugs that are in the bug reporting system of Log4j are generally well documented. In addition, the majority of bugs contain crash traces, which makes Log4j a good candidate system for this study.

[‡]The bug reports used in this study and the result of JCHARMING are made available for download from https://research.mathieunayrolles.com/jcharming/

Prepared using smrauth.cls
DOI: 10.1002/smr

MCT [41] stands for Mission Control technologies and was developed by the NASA Ames Research Center (the creators of JPF) for use in spaceflight mission operation. This tool benefits from two years of history and targets a very critical domain, Spacial Mission Control. Therefore, this tool has to be particularly and carefully tested and, consequently, the remaining bugs should be hard to discover and reproduce.

PDFBox [42] is another tool supported by the Apache Software Foundation since 2009 and was created in 2008. PDFBox allows the creation of new PDF documents and the manipulation of existing documents.

Hadoop [43] is a framework for storing and processing large datasets in a distributed environment. It contains four main modules: *Common*, *HDFS*, *YARN* and *MapReduce*. In this paper, we study the *Common* module that contains the different libraries required by the other three modules.

Mahout [44] is a relatively new software application, built on top of Hadoop. We used Mahout version 0.11, which was released in August 2015. Mahout supports various machine learning algorithms with a focus on collaborative filtering, clustering, and classification.

Finally, ActiveMQ [45] is an open source messaging server that allows applications written in Java, C, C++, C#, Ruby, Perl or PHP to exchange messages using various protocols. ActiveMQ has been actively maintained since it became an Apache Software Foundation project in 2005.

5.2. *Bug Selection and Crash Traces*

In this study, we have selected the reproduced bugs randomly in order to avoid the introduction of any bias. We selected a random number of bugs ranging from 1 to 10 for each SUT containing the word "exception" and where the description of the bug contains a match to a regular expression designed to find the pattern of a Java exception.

6. RESULTS

Table II shows the results of JCHARMING in terms of Bug #ID, reproduction status, and execution time (in minutes) of directed model checking (DMC), length of the counter-example (statements in the JUnit test) and execution time (in minutes) for Model Checking (MC). The experiments have been conducted on a Linux machine (8 GB of RAM and using Java 1.7.0_51).

- The result is noted as "Yes" if the bug has been fully reproduced, meaning that the crash trace generated by the model checker is identical to the crash trace collected during the failure of the system.
- The result is "Partial" if the similarity between the crash trace generated by the model checker and the original crash trace is above t=80% and below t=100%. Given an 80% similarity threshold, we consider partial reproduction as successful. A different threshold could be used.
- Finally, the result of the approach is reported as "No" if either the similarity is below $t < 80\%$ or the model checker failed to crash the system given the input we provided.

As we can see in Table II, we were able to reproduce 24 bugs out of 30 bugs either completely or partially (80% success ratio). The average time to reproduce a bug was 19 minutes. The average time in cases where JCHARMING failed to reproduce the bug was 11 minutes. The maximum

DOI: 10.1002/smr

Table II. Effectiveness of JCHARMING using directed model checking (DMC) and model checking (MC) in minutes

SUT	Bug #ID	Reprod.	Time DMC	CE length	Time MC
Ant	38622	Yes	25.4	3	-
	41422	No	42.3	-	-
ArgoUML	2558	Partial	10.6	3	-
	2603	Partial	9.4	3	-
	311	Yes	11.3	10	-
	1786	Partial	9.9	6	-
DnsJava	38	Yes	4	2	23
jFreeChart	434	Yes	27.3	2	-
	664	Partial	31.2	3	-
	916	Yes	26.4	4	-
Log4j	11570	Yes	12.1	2	-
	40212	Yes	15.8	3	-
	41186	Partial	16.7	9	-
	45335	No	3.2	-	-
	46271	Yes	13.9	4	-
	47912	Yes	12.3	3	-
	47957	No	2	-	-
MCT	440ed48	Yes	18.6	3	-
PDFBox	1412	Partial	19.7	4	-
	1359	No	7.5	-	-
Mahout	486	Partial	34.5	5	-
	1367	Partial	21.1	7	-
	1594	No	14.8	-	-
	1635	Yes	31.0	14	-
Hadoop	2893	Partial	7.4	3	32
	3093	Yes	13.1	-	2
	11878	Yes	17.4	6	-
ActiveMQ	1054	Yes	38.3	11	-
	2880	Partial	27.4	6	-
	5035	No	1	-	-

fail time was 42.3 minutes, which was the time required for JCHARMING to fill all the available memory and stop and a "-" denotes that JCHARMING reached a sixty-minute timeout. Finally, we report the number of statements in the produced JUnit test, i.e. the counter example. While reproducing a bug is the first step in understanding the cause of a field crash, the step to reproduced should be as few as possible. Indeed, it is crucial for counter-examples to be short to effectively help the developer craft a fix. In average, JCHARMING counter-example were composed of 5.04 Java statements. While it is difficult to estimate if the generated counter-example are the shortest possible without a deep understanding of the system at hand, we are confident that five steps is short enough for the developer to understand the problem. This result demonstrates the effectiveness of our approach, more particularly, the use of backward slicing to create a manageable search space that guides adequately the model checking engine. We also demonstrated that our approach is usable in practice since it is also time efficient. Among the 30 different bugs we have tested, we will describe two bugs (chosen randomly) for each category (successfully reproduced, partially reproduced, and not reproduced) for further analysis. The bug report presented in the following sections are the

Prepared using smrauth.cls
DOI: 10.1002/smr

original reports as submitted by the reporter. As such, they contain typos and spelling mistakes that we did not correct.

6.1. Successfully Reproduced

The first bug we describe in this discussion is the bug #311 belonging to ArgoUML. This bug was submitted in an earlier version of ArgoUML. This bug is very simple to manually reproduce thanks to the extensive description provided by the reporter, which reads: *"I open my first project (Untitled Model by default). I choose to draw a Class Diagram. I add a class to the diagram. The class name appears in the left browser panel. I can select the class by clicking on its name. I add an instance variable to the class. The attribute name appears in the left browser panel. I can't select the attribute by clicking on its name. Exception occurred during event dispatching:"*

The reporter also attached the following crash trace that we used as input for JCHARMING:

```
1. java.lang.NullPointerException:
2. at
3. uci.uml.ui.props.PropPanelAttribute .setTargetInternal (PropPanelAttribute.java)
4. at uci.uml.ui.props.PropPanel. setTarget(PropPanel.java)
5. at uci.uml.ui.TabProps.setTarget(TabProps.java)
6. at uci.uml.ui.DetailsPane.setTarget (DetailsPane.java)
7. at uci.uml.ui.ProjectBrowser.select (ProjectBrowser.java)
8. at uci.uml.ui.NavigatorPane.mySingleClick (NavigatorPane.java)
9. at uci.uml.ui.NavigatorPane$Navigator MouseListener.mouse Clicked(NavigatorPane.java)
10.at java.awt.AWTEventMulticaster.mouseClicked (AWTEventMulticaster.java:211)
11. at java.awt.AWTEventMulticaster.mouseClicked (AWTEvent Multicast er.java:210)
12.at java.awt.Component.processMouseEvent (Component.java:3168)
...
19. java.awt.LightweightDispatcher .retargetMouseEvent (Container.java:2068)
22. at java.awt.Container .dispatchEventImp l(Container.java:1046)
23. at java.awt.Window .dispatchEventImpl (Window.java:749)
24. at java.awt.Component .dispatchEvent (Component.java:2312)
25. at java.awt.EventQueue .dispatchEvent (EventQueue.java:301)
28. at java.awt.EventDispatchThread.pumpEvents
(EventDispatch Thread.java:90)
29. at java.awt.EventDispatchThread.run(EventDispatch Thread.java:82)
```

The cause of this bug is that the reference to the attribute of the class was lost after being displayed on the left panel of ArgoUML and therefore, selecting it through a mouse click throws a null pointer exception. In the subsequent version, ArgoUML developers added a TargetManager to keep the reference of such object in the program. Using the crash trace, JCHARMING's preprocessing step removed the lines between lines 11 and 29 because they belong to the Java standard library and we do not want either the static slice or the model checking engine to verify the Java standard library but only the SUT. Then, the third step performs the static analysis following the process described in Section IV.C. The fourth step performs the model checking on the static slice to produce the same crash trace. More specifically, the model checker identifies that the

Prepared using smrauth.cls
DOI: 10.1002/smr

method `setTargetInternal(Object o)` could receive a null object that will result in a `Null` pointer exception.

The second reproduced bug we describe in this section is Bug #486 belonging to MAHOUT. The submitter (Robin Anil) named the bug entry as *Null Pointer Exception running DictionaryVectorizer with ngram=2 on Reuters dataset*. He simply copied the following crash stack without further explanation:

```
1 java.io.IOException: Spill failed
2 at org.apache.hadoop.mapred.MapTask$MapOutputBuffer.collect(MapTask.java:860)
...
14 Caused by: java.lang.NullPointerException
15 at java.io.ByteArrayOutputStream.write(ByteArrayOutputStream.java:86)
16 at java.io.DataOutputStream.write(DataOutputStream.java:90)
17 at org.apache.mahout.utils.nlp.collocations.llr.Gram.write(Gram.java:181)
18 at org.apache.hadoop.io.serializer.WritableSerialization$WritableSerializer.serialize ...
19 at org.apache.hadoop.io.serializer.WritableSerialization$WritableSerializer.serialize ...
20 at org.apache.hadoop.mapred.IFile$Writer.append(IFile.java:179)
21 at org.apache.hadoop.mapred.Task$CombineOutputCollector.collect(Task.java:880)
22 at org.apache.hadoop.mapred.Task$NewCombinerRunner$OutputConverter.write ...
23 at org.apache.hadoop.mapreduce.TaskInputOutputContext.write ...
24 at org.apache.mahout.utils.nlp.collocations.llr.CollocCombiner.reduce(CollocCombiner.java:40)
25 at org.apache.mahout.utils.nlp.collocations.llr.CollocCombiner.reduce(CollocCombiner.java:25)
26 at org.apache.hadoop.mapreduce.Reducer.run(Reducer.java:176)
27 at org.apache.hadoop.mapred.Task$NewCombinerRunner.combine(Task.java:1222)
28 at org.apache.hadoop.mapred.MapTask$MapOutputBuffer.sortAndSpill(MapTask.java:1265)
29 at org.apache.hadoop.mapred.MapTask$MapOutputBuffer.access$1800(MapTask.java:686)
30 at org.apache.hadoop.mapred.MapTask$MapOutputBuffer$SpillThread.run(MapTask.java:1173)
```

Drew Farris, who was assigned to fix this bug, commented *"Looks like this was due to an improper use of the Gram default constructor that happened as a part of the 0.20.2[§] refactoring work."* While this quick comment, made only two and a half hours after the bug submission, was insightful as shown in our generated test case[¶], the fix happened in the `CollocCombiner` class that is one of the `Reducer`[‖] available in Mahout. The fix (commit #f13833) involved creating an iterator to combine the frequencies of the `Gram` and a null check of the final frequency.

JCHARMING's preprocessing step removed the lines between lines 1 to 14 because they belong to the second thrown exception since `java.lang.NullPointerException` occurred when writing in a `ByteArrayOutputStream`. JCHARMING aims to reproduce the root exceptions and not the exceptions that derive from other exceptions. This said, the

[§] Farris certainly meant 0.10.2 which was the last refactor of the incriminated class, and the current version of Mahout is 0.11

[¶] As a reminder, the generated test cases are made available at research.mathieu-nayrolles.com/jcharming

[‖] As in Map/Reduce

Prepared using smrauth.cls
DOI: 10.1002/smr

`java.io.IOException: Spill failed` was ignored and our directed model checking engine focused, with success, on reproducing the `java.lang.NullPointerException`.

6.2. Partially Reproduced

As an example of a partially reproduced bug, we explore Bug #664 of the Jfreechart program. The description provided by the reporter is: "*In ChartPanel.mouseMoved there's a line of code which creates a new ChartMouseEvent using as first parameter the object returned by getChart(). For getChart() is legal to return null if the chart is null, but ChartMouseEvent's constructor calls the parent constructor which throws an IllegalArgumentException if the object passed in is null.*"

The reporter provided the crash trace containing 42 lines and then replaced an unknown number of lines by the following statement "<deleted entry>". While JCHARMING successfully reproduced a crash yielding almost the same trace as the original trace, the "<deleted entry>" statement – which was surrounded by calls to the standard Java library – was not suppressed and stayed in the crash trace. That is, JCHARMING produced only the 6 (out of 7) first lines and reached 83% similarity, and thus a partial reproduction.

```
1. java.lang.IllegalArgumentException: null source
2. at java.util.EventObject.¡init¿( EventObject.java:38)
3. at
4 org.jfree.chart.ChartMouseEvent.¡init¿ (ChartMouseEvent.java:83)
5. at org.jfree.chart.ChartPanel .mouseMoved(ChartPanel.java:1692)
6. <deleted entry>
```

The second partially reproduced bug we present here is Bug #2893 belonging to Hadoop. This bug, reported in February 2008 by Lohit Vijayarenu, was titled *checksum exceptions on trunk* and contained the following description: *While running jobs like Sort/WordCount on trunk I see few task failures with ChecksumException. Re-running the tasks on different nodes succeeds. Here is the stack*

```
1 Map output lost, rescheduling: getMapOutput(task_200802251721_0004_m_000237_0,29) failed :
2 org.apache.hadoop.fs.ChecksumException: Checksum error: tmps4mapredttmapred-
local/task_200802251721_0004_m_000237_0file.out at 2085376
3 at org.apache.hadoop.fs.FSInputChecker.verifySum(FSInputChecker.java:276)
4 at org.apache.hadoop.fs.FSInputChecker.readChecksumChunk(FSInputChecker.java:238)
5 at org.apache.hadoop.fs.FSInputChecker.read1(FSInputChecker.java:189)
6 at org.apache.hadoop.fs.FSInputChecker.read(FSInputChecker.java:157)
7 at java.io.DataInputStream.read(DataInputStream.java:132)
8 at org.apache.hadoop.mapred.TaskTracker$MapOutputServlet.doGet(TaskTracker.java:2299)
...
23 at org.mortbay.util.ThreadPool$PoolThread.run(ThreadPool.java:534)
```

Similarly to the first partially reproduced bug, the crash traces produced by our directed model checking engine and the related test case did not match 100% of the attached crash stack. While JCHARMING successfully reproduced the bug, the crash stack contains timestamps information

DOI: 10.1002/smr

(e.g., 200802251721), that was logically different in our produced stack trace as we ran the experiment years later.

In all bugs that were partially reproduced, we found that the differences between the crash trace generated from the model checker and the original crash trace (after preprocessing) consists of a few lines only.

6.3. Not Reproduced

To conclude the discussion on the case study, we present a case where JCHARMING was unable to reproduce the failure. For the bug #47957, belonging to LOG4J and reported in late 2009 the reporter wrote: "*Configure SyslogAppender with a Layout class that does not exist; it throws a NullPointerException. Following is the exception trace:*" and attached the following crash trace:

```
1. 10052009 01:36:46 ERROR [Default: 1] struts.CPExceptionHandler.execute
RID[(null;25KbxlK0voima4h00ZLBQFC;236Al8E60000045C3A 7D74272C4B4A61)]
2. Wrapping Exception in ModuleException
3. java.lang.NullPointerException
4. at org.apache.log4j.net.SyslogAppender .append(SyslogAppender.java:250)
5. at org.apache.log4j.AppenderSkeleton .doAppend(AppenderSkeleton.java:230)
6. at org.apache.log4j.helper.AppenderAttachableImpl .appendLoopOnAppen-
ders(AppenderAttachableImpl .java:65)
7. at org.apache.log4j.Category.callAppenders (Category.java:203)
8. at org.apache.log4j.Category .forcedLog(Category.java:388)
9. at org.apache.log4j.Category.info (Category.java:663)
```

The first three lines are not produced by the standard execution of the SUT but by an ExceptionHandler belonging to Struts [46]. Struts is an open source MVC (Model View Controller) framework for building Java web applications. JCHARMING examined the source code of Log4J for the crash location `struts.CPExceptionHandler.execute` and did not find it since this method belongs to the source base of Struts – which uses log4j as a logging mechanism. As a result, the backward slice was not produced, and we failed to perform the next steps. It is noteworthy that the bug is marked as duplicate of the bug #46271 which contains a proper crash trace. We believe that JCHARMING could have successfully reproduced the crash, if it was applied to the original bug.

The second bug that we did not reproduce and that we present in this section belongs to Mahout. It was reported by Jaehoon Ko on July 2014. Bug #1594 is titled *Example factorize-movielens-1M.sh does not use HDFS* and reads *It seems that factorize-movielens-1M.sh does not use HDFS at all. All*

Prepared using smrauth.cls
DOI: 10.1002/smr

paths look local paths, not HDFS. So the example crashes immeidately because it cannot find input data from HDFS:

```
1 Exception in thread "main" org.apache.hadoop.mapreduce.lib.input.InvalidInputException: Input
path does not exist: /tmp/mahout-work-hoseog.lee/movielens/ratings.csv
2 at org.apache.hadoop.mapreduce.lib.input.FileInputFormat.singleThreadedListStatus ...
3 at org.apache.hadoop.mapreduce.lib.input.FileInputFormat.listStatus ...
...
31 at org.apache.hadoop.util.RunJarm.theain(RunJar.java:212)
```

This entry was marked as "Not A Problem" / "WONT_FIX" meaning that the reported bug was not a bug in the first place. The resolution of this bug** involved the modification of a bash script that Ko (the submitter) was using to *query* Mahout. In other words, the cause of the failure was external to Mahout itself and this is why JCHARMING could not reproduce it.

7. THREATS TO VALIDITY

The selection of SUTs is one of the common threats to validity for approaches aiming to improve the understanding of a program's behavior. It is possible that the selected programs share common properties that we are not aware of and therefore, invalidate our results. However, the SUTs analyzed by JCHARMING are the same as the ones used in similar studies. Moreover, the SUTs vary in terms of purpose, size and history.

Another threat to validity lies in the way we have selected the bugs used in this study. We selected the bugs randomly to avoid any bias. One may argue that a better approach would be to select bugs based on complexity or other criteria (severity, etc.). We believe that a complex bug (if complexity can at all be measured) may perhaps have an impact on the running time of the approach, but we are not convinced that the accuracy of our approach depends on the complexity or the type of bugs we use. Instead, it depends on the quality of the produced crash trace. This said, in theory, we may face situations where the crash trace is completely corrupted. In such cases, there is nothing that guides the model checker. In other words, we will end up running a full model checker. It is difficult to evaluate the number of times we may face this situation without conducting an empirical study on the quality of crash traces. We defer this to future work.

In addition, we see a threat to validity that stems from the fact that we only used open source systems. The results may not be generalizable to industrial systems. The last author of the paper expressed an interest to apply these techniques to Ericsson systems. We intend to undertake these studies in future work.

Field failures can also occur due to the running environment on which the program is executed. For instance, the failure may have been caused by the reception of a network packet or the opening of a given file located on the hard drive of the users. The resulting failures will hardly be reproducible by JCHARMING.

**https://github.com/apache/mahout/pull/38#issuecomment-51436303

DOI: 10.1002/smr

Finally, the programs we used in this study are all written in the Java programming language and JCHARMING leverages the crash traces produced by the JVM to reproduce bugs. This can limit the generalization of the results. However, similar to Java, .Net, Python and Ruby languages also produce crash traces. Therefore, JCHARMING could be applied to other object-oriented languages.

In conclusion, internal and external validity have both been minimized by choosing a relatively large set of different systems and using input data that can be found in other programming languages.

8. CONCLUSION AND FUTURE WORK

In this paper, we presented JCHARMING (Java CrasH Automatic Reproduction by directed Model checking), an automatic bug reproduction technique that combines crash traces and directed model checking. JCHARMING relies on crash traces and backward program slices to direct a model checker. This way, we do not need to visit all the states of the subject program, which would be computationally taxing. When applied to thirty bugs from ten open source systems, JCHARMING was able to successfully reproduce 80% of the bugs. The average time to reproduce a bug was 19 minutes, which is quite reasonable, given the complexity of reproducing bug that cause field crashes.

To build on this work, we need to experiment with additional (and more complex) bugs with the dual aim to (a) improve and fine tune the approach, and (b) assess the scalability of our approach when applied to even larger (and proprietary) systems. We also need to conduct empirical studies that assess the quality of crash traces. This is because poorly reported crash traces impact our approach as we showed in cases where JCHARMING was not able to reproduce the bug. Finally, we should also extend JCHARMING to consider bugs that involve multi-threading.

REFERENCES

1. Bettenburg N, Just S, Schröter A, Weiss C, Premraj R, Zimmermann T. What makes a good bug report? *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering - SIGSOFT '08/FSE-16*, ACM Press: New York, New York, USA, 2008; 308, doi:10.1145/1453101.1453146. URL http://portal.acm.org/citation.cfm?doid=1453101.1453146.
2. Narayanasamy S, Pokam G, Calder B. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. *Proceedings of the 32nd annual international symposium on Computer Architecture*, vol. 33, ACM, 2005; 284–295, doi:10.1145/1080695.1069994. URL http://dl.acm.org/citation.cfm?id=1080695.1069994.
3. Artzi S, Kim S, Ernst MD. Recrash: Making software failures reproducible by preserving object states. *Proceedings of the 22nd European conference on Object-Oriented Programming*, 2008; 542–565.
4. Jaygarl H, Kim S, Xie T, Chang CK. OCAT: Object Capture based Automated Testing. *Proceedings of the 19th international symposium on Software testing and analysis*, 2010; 159–170.
5. Manevich R, Sridharan M, Adams S. PSE: explaining program failures via postmortem static analysis. *ACM SIGSOFT Software Engineering Notes*, vol. 29, ACM, 2004; 63, doi:10.1145/1041685.1029907. URL http://dl.acm.org/citation.cfm?id=1041685.1029907.
6. Chandra S, Fink SJ, Sridharan M. Snugglebug: a powerful approach to weakest preconditions. *ACM Sigplan Notices*, vol. 44, ACM, 2009; 363–374.
7. Nayrolles M, Hamou-Lhadj A, Sofiene T, Larsson A. JCHARMING : A Bug Reproduction Approach Using Crash Traces and Directed Model Checking. *SANER'15*, 2015; 101–110.
8. Baier C, Katoen Jp. *Principles of Model Checking*. MIT press Cambridge, 2008.
9. Steven J, Chandra P, Fleck B, Podgurski A. jRapture: A Capture/Replay Tool for Observation-Based Testing. *Proceedings of the International Symposium on Software Testing and Analysis.*, August, 2000; 158–167.

10. Roehm T, Nosovic S, Bruegge B. Automated extraction of failure reproduction steps from user interaction traces. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, 2015; 121–130, doi:10.1109/SANER.2015.7081822. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7081822.
11. Jin W, Orso A. BugRedux: Reproducing field failures for in-house debugging. *2012 34th International Conference on Software Engineering (ICSE)*, Ieee, 2012; 474–484, doi:10.1109/ICSE.2012.6227168. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6227168.
12. Jin W, Orso A. F3: fault localization for field failures. *Proceedings of the 2013 International Symposium on Software Testing and Analysis - ISSTA 2013*, ACM Press: New York, New York, USA, 2013; 213–223, doi:10.1145/2483760.2483763. URL http://dl.acm.org/citation.cfm?doid=2483760.2483763.
13. Zuddas D, Jin W, Pastore F, Mariani L, Orso A. MIMIC: locating and understanding bugs by analyzing mimicked executions. *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*, ACM Press: New York, New York, USA, 2014; 815–826, doi:10.1145/2642937.2643014. URL http://dl.acm.org/citation.cfm?id=2642937.2643014.
14. Chen N. Star: stack trace based automatic crash reproduction. PhD Thesis 2013.
15. Rößler J, Zeller A, Fraser G, Zamfir C, Candea G. Reconstructing Core Dumps. *Proceedings of the 6th International Conference on Software Testing, Verification and Validation, ser. ICST*, 2013.
16. Clause J, Orso A. A Technique for Enabling and Supporting Debugging of Field Failures. *ICSE '07 Proceedings of the 29th international conference on Software Engineering*, 2007; 261–270.
17. Dutertre B, Moura LD. The yices smt solver. *Technical Report* 2006.
18. Visser W, Havelund K, Brat G, Park S, Lerda F. Model Checking Programs. *Automated Software Engineering*, vol. 10, Springer, 2003; 203–232.
19. Visser W, Psreanu CS, Khurshid S. Test input generation with java PathFinder. *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis - ISSTA '04* 2004; :97doi:10.1145/1007512.1007526. URL http://portal.acm.org/citation.cfm?doid=1007512.1007526.
20. Burg B, Bailey R, Ko AJ, Ernst MD. Interactive record/replay for web application debugging. *Proceedings of the 26th annual ACM symposium on User interface software and technology - UIST '13*, ACM Press: New York, New York, USA, 2013; 473–484, doi:10.1145/2501988.2502050. URL http://dl.acm.org/citation.cfm?id=2501988.2502050.
21. Herbold S, Grabowski J, Waack S, Bünting U. Improved Bug Reporting and Reproduction through Non-intrusive GUI Usage Monitoring and Automated Replaying. *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, IEEE, 2011; 232–241, doi:10.1109/ICSTW.2011.66. URL http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=5954414.
22. Zeller A, Hildebrandt R. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 2002; **28**(2):183–200, doi:10.1109/32.988498. URL http://dl.acm.org/citation.cfm?id=506201.506206.
23. Bell J, Sarda N, Kaiser G. Chronicler: Lightweight recording to reproduce field failures. *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, 2013; 362–371, doi:10.1109/ICSE.2013.6606582. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6606582.
24. Zamfir C, Candea G. Execution Synthesis: A Technique for Automated Software Debugging. *Proceeding of EuroSys '10 Proceedings of the 5th European conference on Computer systems*, 2010; 321–334.
25. De Moura L, Bjørner N. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008; 337–340.
26. Kropf T. *Introduction to formal hardware verification*. Springer, 1999.
27. Kripke SA. Semantical Considerations on Modal Logic. *Acta Philosophica Fennica* 1963; **16**(1963):83–94.
28. Edelkamp S, Leue S, Lluch-Lafuente A. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer (STTT)* mar 2004; **5**(2-3):247–267, doi:10.1007/s10009-002-0104-3. URL http://link.springer.com/10.1007/s10009-002-0104-3.
29. Edelkamp S, Schuppan V, Bosnacki D, Wijs A, FehnkerAnsgar, Aljazzar H. *Survey on directed model checking*. Springer, 2009.
30. Oracle. Throwable (Java Plateform SE6) 2011. URL http://docs.oracle.com/javase/6/docs/api/java/lang/Throwable.html.
31. De Lucia A. Program slicing: Methods and applications. *International Working Conference on Source Code Analysis and Manipulation*, IEEE Computer Society, 2001; 144.
32. IBM. T. J. Watson Libraries for Analysis (WALA) 2006.
33. Holzmann GJ. The model checker SPIN. *IEEE Transactions on software engineering* 1997; **23**(5):279–295.
34. McMillan KL. *Symbolic model checking*. Springer, 1993.

35. Kim S, Zimmermann T, Nagappan N. Crash Graphs: An Aggregated View of Multiple Crashes to Improve Crash Triage. *International Conference on Dependable Systems and Networks (DSN)*, 2013; 486–493.
36. Apache Software Foundation. Apache Ant. URL http://ant.apache.org/.
37. CollabNet. Tigris.org: Open Source Software Engineering. URL http://www.tigris.org/.
38. Wellington B. dnsjava 2013. URL http://www.dnsjava.org/.
39. Object Refinery Limited. JFreeChart 2005. URL http://jfree.org/jfreechart/.
40. The Apache Software Foundation. Log4j 2 Guide - Apache Log4j 2 1999. URL http://logging.apache.org/log4j/2.x/.
41. NASA. Open Mission Control Technologies 2009. URL https://sites.google.com/site/openmct/.
42. Apache Software Foundation. Apache PDFBox — A Java PDF Library 2014. URL http://pdfbox.apache.org/.
43. Hadoop A. Hadoop 2011.
44. Mahout A. Scalable machine learning and data mining 2012.
45. Snyder B, Bosanac D, Davies R. *ActiveMQ in Action*. Manning Publications Co., 2011.
46. Apache Software Foundation. Apache Struts Project 2000. URL http://struts.apache.org/.

DOI: 10.1002/smr