August 15, 2015

Dear Bram and Alex,

Please find attached an extended version of our SANER'15 paper: "JCHARMING: A Bug Reproduction Approach Using Crash Traces and Directed Model Checking," by Mathieu Nayrolles, Abdelwahab Hamou-Lhadj, Sofiène Tahar, and Alf Larsson.

The extended version is titled: "A Bug Reproduction Approach Based on Directed Model Checking and Crash Traces" by the same authors.

The new contributions of the paper consist of what follows:

- We addressed all the comments raised by SANER's reviewers. Some were addressed during the submission of the camera ready of the paper. Others have been addressed in this journal version (please see the following document that contains our responses to the reviewers' comments).
- We run additional experiments with JCHARMING on 10 new bugs of three additional systems, Hadoop, Mahoot, and ActiveMQ, bringing the total number of bugs examined in this paper to 30 and the total number of subject systems to 10. We would like to note that the new systems are larger than those used in the conference version of the paper.
- We added the analysis of three additional bugs.
- We added a section on JUnit Test Case Generation.
- We updated the related work section
- We updated the threats to validity section
- We updated the conclusion and future work section

We would like to thank you and SANER's anonymous reviewers for the constructive comments. We hope that you will find everything to your satisfaction.

We look forward to hearing from you.

Best regards,

Abdelwahab Hamou-Lhadj, PhD, PEng,
OCUP, OCEB
Associate Professor, Undergraduate Program Director
Software Behaviour Analysis (SBA) Research Lab
Electrical and Computer Engineering
Concordia University
1455 de Maisonneuve West
Montreal, Quebec H3G 1M8 Canada

Office: S-EV5.213
Tel: +1.514.848.2424 ext. 7949
Email: abdelw@ece.concordia.ca

This document explains how we have addressed each of the comments made by SANER's reviewers of the paper "JCHARMING: A Bug Reproduction Approach Using Crash Traces and Directed Model Checking". We have numbered each comment. Original comments from reviewers are in bold; our responses are in non-bold text.

Please note that many of these comments were also addressed during the submission of the camera ready of the paper. In some cases, we included additional clarifications to strengthen this journal version of the paper.

Reviewer 1:

R1.1 Section IV, B - just before C - I understand why you do not look at the Java GUI and JDK, but I wonder what if the crash is an interaction issue? These types of issues are not detectable by your method?

We did not focus on Java GUI and JDK components. We simply assumed that Java GUI and JDK components are robust. We agree that if the bug involves some of these components then we will not be able to reproduce it. A simple solution would be to include these components in our analysis.

R1.2 Section IV, C, just under figure 4 - If all the frames are corrupt, you perform full up model checking? How often does this happen?

If all the frames are corrupt then we are left with the full model checking option. In other words, we do not have anything to use to guide the model checker. In all the bugs we studied, we never had the case of a complete corrupted trace. So we suspect that this happens rarely. To answer the question in a precise way, we will need to conduct an empirical study on the quality of crash traces, which we think it is outside the scope of the paper. We defer this to future work. We modified the future work section to discuss this point.

R1.3. Section IV, E - This reader wonders if the user is not able to provide some information (which is why we want to do this in an automated fashion), can we trust a threshold that they offer? Will they be informed enough to know what 80% of the frames means?

This is a good point. In this study, we varied the threshold t from 10% with an increment of 5% until we were able to reproduce the bugs and also generate

the relevant test cases. Like any other similarity threshold, it is challenging to determine the threshold that would work for multiple systems. In practice, we expect that the tool that supports JCHARMING should provide some flexibility to software engineers to vary *t*. This said, based on our observations, 80% seems to be reasonable. We modified the paper (right before Section 4.6) to reflect this discussion.

R1.4. Section V, A, just below Table I - I understand that you used a random technique and it selected 7 bugs from one system, and one from others. It would have been nice to have at least 2 from each system though, for comparison.

We tried in earlier versions of the paper to describe two bugs for each system but the description turned out to be cumbersome, taking a huge amount of space. In this extended version, we preferred to describe three new bugs from the newly added systems, namely, Hadoop, Mahout, and ActiveMQ. We believe that this gives a good coverage of the various bugs. I hope this addresses the reviewer's concern.

R1.5. Table II - I wonder how long it took for the no scenarios to run. Did it take a very long time to realize that it did not work?

The average time to successfully reproduce a bug was 19 minutes. The average time in cases where JCHARMING failed to reproduce the bug was 11 minutes. The maximum fail time was 42.3 minutes, which was the time required for JCHARMING to fill out all the available memory and stop. We added this description to the paper as well. We also modified Table II.

R1.6. Just before Section VI, discussion of Struts - could this type of thing not happen often? How can you address it?

It is hard to assess whether this happens often or not. We haven`t conducted studies on the completeness of crash traces. This would be a very interesting thing to do. We added this to the future work section.

R1.7. Just before Section VI - Why not apply your approach to the "proper crash trace" and see if it did reproduce it?

For the bugs that were successfully reproduced, we applied the approach to the proper crash traces. Unfortunately, in practice, we do not always have proper crash traces. So, we preferred to show an example where we have corrupt frames and how partial traces can be dealt with.

<u>Reviewer 2:</u>

R2.1. Page 2, 1. col
- coverage of automated test -> coverage of automated tests
- 'pass' and 'fail' path which -> 'pass' and 'fail' path, which

Page 2, 2nd col
- on-field bug -> on-field bugs
- core dump and try -> core dump and tries
- an in-memory which -> an in-memory, which (read up on the use of 'which'
and the difference between w and w/o comma, and check the paper)
- own JVM allows to capture -> own JVM allows one to capture
Page 3, 1st col
- Except for STAR, the majority: rephrase (leave out either 'except for'
or 'majority')
- when there exists a sequence of state transition: whether or not it is
a "sequence" depends on the logic formula
Page 3, 2nd col
- superior -> larger

Fixed. We would like to thank the reviewer for proof reading the paper. It is much appreciated. We fixed all the typos raised by the reviewer. We also proof read the paper again and again.

R2.2. Page 4, 2nd col- address the problem of nested exceptions: namely?

We added an explanation on what we mean by nested exceptions and how our approach focuses on the root exception. There is no need to consider the inner exceptions since fixing the root exception (i.e., the one that triggered the inner exceptions) will eventually fix the other problems. Please see the first paragraph of Section 4.2.

R2.3. Page 5, 1st col.
- for our directed: missing noun
- does not necessary -> does not necessarily
- allows to reach -> allows one to reach
Fixed.

R2.4. Page 5, 2nd col
- Bytecode -> byte code
- in Fig. 4, what is the entry?
- did you take WALA's slicer as-is or did you have to extend it?

We reviewed Fig. 4 and fixed the figure by specifying the entry. We did not have to extend WALA. We used it as it is. We modified the text to reflect this.

R2.5. Page 6, 1st col
- Fig.5: do you call WALA's slicer?
- Fig.5: the two arrows in line 7 are fairly non-standard

- Fig.5: typo in 'slide'
- states transitions -> state transitions (everywhere)
Fixed.

R2.6 Page 6, 2nd col
- why does serialization help with optimization?

We are no longer using serialization. We have modified completely the way test cases are being generated. Please refer to the newly added section, Section 4.6.

R2.7. Page 7, 1st col
- Can we use crash traces ... to reproduce on-field bugs: why should the answer be anything other than 'yes'? Be more specific.

The emphasis here is really on using directed model checking with stack traces (from which we extract program slices) in a reasonable amount of time. The efficiency of the approach is important here since most model checking techniques (in different areas) suffer from efficiency problems. Our experiments (including the new ones we include in this journal version of the paper) show that we can reproduce a bug in 80% of the cases in average time of 19 minutes.

R2.8.Page 8, 1st col
- fonts in line 29
Page 9, 1st col
- lines and the replaced -> lines and then replaced
Page 9, 2nd col
- We believe that JCHARMING ..: what prevents you from checking it?

We fixed the typos. We checked that JCHARMING works by producing test cases automatically that can be used to reproduce the bugs. We added a section on JUnit test case generation to discuss this.

Reviewer 3:

R3.1 In section IV.D, it would be good to have a bit more description (maybe an example) of how the backwards slice information is used to guide the model checker, since this is the central idea. The 7 lines quite formal description does not seem adequate (and is IMO not so easy to follow).

We added a paragraph right after the formal description to clarify the process. 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.

R3.1 One thing that needs to be fixed is "Figure 5", the algorithm for computing the union of slices (btw., it's not really a figure). There are some inconsistencies (or bugs) in there:
- CurrentFrames = frames[i] when i has not been initialized
- the "for" loop is a bit strange. It's neither Pascal nor C/Java like, its semantics are not completely clear
- shouldn't the slice go from CurrentFrame to Frame[i+offset]?
- "Slide" should be "Slice"
- I wonder if the description cannot be simplified. I mean, the branch/offset is just for ignoring corrupted entries, right?

We fixed the algorithm. As for the last point, the branch/offset is used to ignore corrupted entries. The main purpose, however, is to find the smallest slice connecting all the frames.

R3.3 Minor issues:
- p. 2: "try (with evolutionary algorithms) to..." => tries
- p. 3: "there exists a sequence of states transition x leading..." => sequence of states or sequence of transitions, or both?
- p. 5: "for our directed, ..." => a noun missing?
- p. 5, introduction of bslice: my impression was that sometimes m and n are exchanged
- p. 6: "that need to be" => needs
- p. 6: "current sequence of states transitions x" => see above
- p. 6: "satisfies a set a property" => ?
- p. 7: Case studies: The description of subject systems could be shortened.
- p. 8: the example stack trace for bug #311 could be shortened to the first half, since the second half is thrown away anyway. It does not seem adequate to spend 1/3 page for that.

We addressed all these issues, except for the last one. We kept the description as it is since it is a journal version of the paper. Thank you to the reviewer for the comments and pinpointing the typos.