This document explains how we have addressed each of the comments made by JSPE reviewers of the paper “A Bug Reproduction Approach Based on Directed Model Checking and Crash Traces”.

We have numbered each comment. Original comments from reviewers are in bold; our responses are in non-bold text. Also in an attached PDF version of the revised document we have inserted text in red that show the changes made to the paper. For each comment of the reviewers, we indicated the exact location in the revised paper where the changes were made.

We would like to thank the reviewers for their comments. We found the comments very helpful and many of the suggestions and comments have been incorporated into this revised version of the paper.

Reviewer 1:

R1.1. Providing a formal definition of the syntax of traces;

We added a formal definition of a crash trace generated from uncaught exceptions in Java. More formally, we define a Java crash trace T of size N as a sequence of frames T = f_0 , f_1 , f_2 , ..., f_N where each frame represents either a method call (with the location of the called method in the source code), an exception, or wrapped exception. In Figure 5 of the revised version of the paper, the frame f_0 represents an exception, the frame f_7 represents a method call to the method jsep.Foo.buggy in file Foo.java, located at line 7, causing the exception. F_6 represents a wrapped exception. Please see the first paragraph of Section 4.1 for the changes.

R1.2. Providing a formal definition of a frame and a slice and a better discussion ("reasoning") of the algorithm;

We have made significant changes to Section 4.3 (from Page 11 to Page 15) to address the above comment. The first two paragraphs provide a formal definition of a slice and a frame. We also added two figures (Fig. 7 and 8) to illustrate the concept of backward slicing used in this paper. We hope that this addresses the comment.

R1.3 Extending or modifying the evaluation section by a complete characterization of the partially and not reproduced bugs in the 30-x cases (table)

We are not sure what the reviewer means by a “complete characterization of the partially and not reproduced bugs”. The tables show the results of JCHARMING when applied to all the bugs, used in this study. The definitions of partially and not reproduced bugs are given in the beginning of Section 6. If the reviewer wishes to see more examples of partially or not reproduced bugs, we invite her or him to visit the website that accompanies this paper, where the result of JCHARMING on all the bugs is shown. The URL is: https://research.mathieu-nayrolles.com/jcharming.

R1.4. identifying, and making explicit, the limitations of JCHARMING

We made changes to the Conclusion Section by adding the limitations of JCHARMING. Addressing these limitations is the subject of future work. There are three main limitations that we have identified. The first one is that JCHARMING cannot reproduce bugs due to multi-threading. The second limitation is that JCHARMING cannot be used if the bug is caused by external input. We can always build a monitoring system to retrieve this data, but this may lead to privacy concerns. Finally, the third limitation is that the performance of JCHARMING relies on the quality of the crash trace. Please see the Conclusion section for the changes.

R1.5. The review instructions also ask for possibilities for condensing (or amplifying) the text. In my opinion, the prose in section 6 could be shortened.

We tried to provide as many details as possible to explain the results of JCHARMING in all cases (fully reproduced, partially reproduced, and not reproduced). We had to extend this section to address the comments of Reviewer 2 who wishes to see the length of counter-examples. For the partially and not reproduced bugs, we added a couple of sentences to point out the exact reason why these bugs were not fully reproduced. Please see the paragraph right before Section 6.3 and the one right before Section 7.

As for the style, we used similar style as the one used in other papers in the field. The other papers discuss the bugs in detail for a better evaluation on their approach. To make things more complete, we added a complete website with all the bugs studied in this paper as well as the result of JCHARMING when applied to reproducing each bug. Please see https://research.mathieu-nayrolles.com/jcharming.

R1.6. Page 2: l.16 "In [7]": poor style (writing rules say that the sentence must be complete if you leave out the reference), also see l.40

We modified the paper to adopt the proper style. We thank the reviewer for pointing this out.

R1.7. Page2: l.21 "uses a list of function outputs": every output comes from a function -> be more specific. Also, make clear that JCHARMING additionally uses the source code.

We meant that JCHARMING takes the crash trace as input. The crash trace represents the list of function calls that are output when an uncaught Java exception occurs. We modified the text to avoid the confusion. Please see the second paragraph on Page 2.

R1.8 Page 2: l.28 "control flow graph" seems a bit misleading since there is no 1-1 correspondence between statements in the source and program states.

We agree with the reviewer. We meant that the 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. Ultimately this graph can be mapped to the control flow, but we agree that there is no correspondence between statements and program states. We modified the text by changing “control flow graph” to “graph”.

R1.9. Page 2: l.38 "Hadoop and, " -> Hadoop, and

Fixed. Thanks for this.

R1.10. Page 2: l.43 systems are "relatively complex": you contradict yourself a bit here, since, at least for complex bugs, you write in Section 7 that the quality of the trace is what matters.

Yes, the quality of trace is really what matters. This said, crash traces generated from bugs in complex and large systems are expected to be harder to reproduce because they may involve many components. Since software engineers usually copy crash traces in bug descriptions, they may be reluctant to copy large traces. We modified the sentence in the paper to avoid the term “complex” since complexity can be measured in different ways. The sentence is now: “This is explained by the fact that the new systems used in this study are relatively larger compared to those [...]”.

R1.11. Page 3: l.29 "NP-complex" does not exist as technical term IMO. A problem can be "NP-complete" or in the "complexity class@ NP.

We replaced NP-complex by NP-complete throughout the whole paper. Thanks for this.

R1.12. Page 3: l.35-39: don't you mean that it is impossible to reproduce the bug?

It is true that without having the input used to trigger the bug, it is almost impossible to reproduce the bug. We wrote in the paper that it is challenging. We changed the sentence to state that “it is almost impossible”. Please see the paragraph right before Section 2.1. In our opinion, it is a strong statement to say that it is impossible. One can always run various scenarios by varying different inputs until the bug is exercised. This technique can be time-consuming (and perhaps impractical), but may be needed if the bug is severe and reproducing it is the only way to understand the causes.

R1.13. Page 3: What does "specific data" refer to---simply the input?

This is related to Comment R1.12. We changed the sentence by removing “specific data” and replacing the sentence with “It is almost impossible to reproduce the bug without this input.” As we mentioned in the paragraph right before, by input we mean whatever input (files, network packets) was used to cause the bug.

R1.14. Page 4 : .08 "minimal"... what?

The authors of the paper where this approach was presented used the term “minimal” to mean that their approach compresses the data (traces, files, etc.) that is generated on the client’s host. We changed the paragraph to make this clear. Please see the first paragraph on Page 4.

R1.15. Page 4: l.39 typo in "JRapture"
Fixed.

R1.16. Page 4 l.42 "and on the replay": grammar
Fixed.

R1.17. Page 5:
l.03 "code-level" delete hyphen
l.03 ", there exist": put in a new sentence
l.28 "stack" -> stacks
l.33 "prior to the work ... begins": grammar
l.34 add a period after "i.e."

Fixed. Thanks for the detailed revision.

R1.18. Page 6: l.42 "P is the set of properties that each state satisfies": incorrect or at least misleading scope of "the set." You mean "P associates with each state the set of properties that it satisfies."

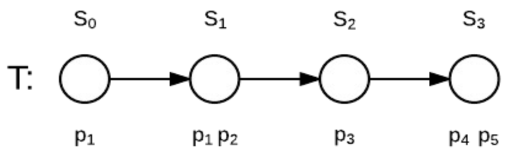
Yes. This is what we mean. In fact, we changed all the equations in Section 3, and Sections 4.2 and 4.3. This applies also to the next comments raised by the reviewer. Please see Comment R1.19.

R1.19. Page 6: l.47 what exactly is "x": a state or a _sequence_ of states (path)? I am familiar with the former, i.e., the situation where one asks the model checker whether property p holds in state x of model M, and then might get back the trace that constitutes a counter example. If "x" really is a path, then, for example, "x \in S" (Eq. 1) wouldn't hold, since S is a set of states. Also, in Eq. (3), you would quantify over paths rather than states.

All the equations of Section 3 (Preliminaries) have been reworked in order to adopt more standard definitions of model checking. The section now uses the following definitions (more details are provided in the paper, please see Section 3):

SU T =< S, S 0 , T, L >

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 exists a sequence of states x leading to a state where p holds. This can be written as:

(SU T, x) |= p

For the SUT of Figure 1, we can write (SU T, {S 0 , S 1 , S 2 }) |= 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, (SU T, {S 0 , S 1 , S 2 }) |= p 3 only ensures that ∃x such that p is reached at some point in the execution of the program and not that p 3 holds for ∀x. [...] 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:

∀x.(SU T, x) |= ¬c

If there exists a contradicting path (i.e., ∃x such that (SU T, x) |= 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.

We thank to reviewer for pointing out these imprecisions.

R1.20. Page 6: l.52 "specification" is not a good term here (and has a different meaning in formal methods), do you mean additional assumption?

We agree. We changed the sentence to: “In JCHARMING, we assume that SUTs must not crash…”

R1.21. Page 6: l.52 just as a remark, "fairness" has a special meaning in (LTL) model checking.

Good point. We agree with this. We changed the term “fair environment” to “ typical environment” that we define as: “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.” We added this to the paper. Please see Page 2, second paragraph right after Equation (2).

R1.22.
Page 7: l.17 "depicts" -> depict
Page 8: l.51 "are" -> is

Fixed. Thanks.

R1.23. Page 9: I don't understand Figure 5. Why is there a call to Foo.bar (even though in Figure 1 there is no edge from Bar.foo to Foo.bar)? What is "looptimes" - in Figure 1, the bound is called "loopCount"?

In Figure 1, the entry point of the program contains edges to Foo.Bar and Bar.Foo.

The Java language provides the InvalidActivityException class which is a specialization of the Exception class. The Exception class has a String constructor that allows developers to define a comment to be displayed when an exception is thrown. In Figure 5, a statement “throw new InvalidActivityException("loopTimes");” would have been inserted after “ if(loopCount > 2)”

R1.24. Page 9: Where is the call to Bar.foo the "crash trace contains" (l.23).

It is included in Figure 6 frame “8.. and 4 more". In Java, not all calls appear in the crash trace. Please refer to the following definition in the Java documentation (http://docs.oracle.com/javase/6/docs/api/java/lang/Throwable.html) : “This line [..and X more] 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.”

R1.25. Page 9: Generally, I strongly recommend to include a grammar of a trace; this would also help answering some of my later questions (see below).

We used the ANTLR grammar (http://www.antlr.org/) for Java stack traces. The grammar was developed by Luca Dall'Olio, Christian Cassano and Gabriele Contini. A reference was added in the paper. This grammar is used extensively in software engineering. We did not include it in the paper because it is quite verbose. If the reviewer feels strong about adding the grammar, we can always put it as an appendix.

R1.26. Page 9: l.38 typo in the caption; I would also reformat the caption to avoid the lonely "2."

Fixed. Thanks.

R1.27. Page 9: l.42 what exactly is a frame? Put differently, what is the difference between a line number in a trace and a frame? Does f_0 have to contain a property (in terms of a programming language: an expression), as Figure 5 and p. 12 suggest, or does it have to contain an exception?

A Java crash trace T of size N is an ordered sequence of frames $T = f_0 , f_1 , f_2, ..., f_N$. A frame represents either a method call (with the location of the method in the source code), an exception, or a wrapped exception. For example, in Figure 6, the frame f_0 represents an exception, the frame f_7 a method call to the method jsep.Foo.buggy, declared in Foo.java line 17, and f_6 represents a wrapped exception.

We modified the paper to clarify what a frame is. We also added a formal definition of a crash trace. We also provided some background info on wrapped exceptions, which are less known that regular exceptions. Please see the two first paragraphs of Section 4.1.

R1.28. Page 9: l.46 "buggy ": delete the extra blank.

Fixed. We also proof read the paper over and over again to avoid these typos.

R1.30. Page 10

l.17 delete the extra "the."
l.21 "out of control" -> beyond control

Fixed.

R1.31. Page 10: l.48 I am not sure whether I completely understand what a slice is in your context: does a slice consist of statements and declarations from the program source? And do frames constitute slicing criteria, or (since this wouldn't work) phrased more precisely: the method invocation they contain? Please provide a formal definition of a slice.

This comment has been addressed at the same time as Comment R1.2. We have made significant changes to Section 4.3 (from Page 11 to Page 15) to explain the concept of slicing. We added examples and figures to illustrate in detail how the whole mechanism of slicing works (i.e., Algorithm 1). Please see Section 4.3 (from Page 11 to Page 15).

R1.32. Page 10: l.54 typo in f_{n 1}

Fixed.

R1.33. Page 10: l.55 in my understanding of a slice, the order of statements is essential, especially if the slice has to remain executable. Rather than forming the union of slides, you should therefore, e.g., concatenate them. (Perhaps the implementation is based on a union that happens to preserve the order, so that you never faced the difference between a set and a sequence). Algorithm 1 needs to be changed accordingly.

We prefer the term union that preserves the order. We added a sentence that this union can be seen as a concatenation of slice elements. Please see the paragraph before the last one on Page 12. To make things clearer, we added an example with a new figure to illustrate this concept. Please see Figure 10 with the surrounding description.

R1.34. Page 11: l.09 "allow to reach": grammar

Fixed.

R1.35. Page 11: Figure 6: what is "entry"? What is the purpose of the two self-loops in z0 and z1?

By entry, we mean the entry point of the program, i.e., the main function in Java. We just put the self-loops there because they can occur. It was just part of the example. We could have picked another example.

R1.36. Page 11: l.31 as a comment, the "worst case scenario" is the best-case scenario from the perspective of state space explosion...

We agree that we can at look at it this way, but what we are really interested in is the worst case scenario of our approach.

R1.37. Page 11: l.44 how can a slice be empty? Since there is an underlying execution between frame f_i and f_{i+1}, there have to be program statements, no? Again, a definition of a slice would have helped me (perhaps).

As we explained in the paper, there may be situations where f_{i+1} is not valid (due to copy-paste errors, etc.). In this case, the algorithm won't find the corresponding statement in the graph representation of the SUT. In other words, we can't generate the slice. The algorithm continues by checking f_{i+2}. If this frame is valid then the slice is generated between f_{i} and f{i+2}. We think that this is now much clearer with all the changes we made to Section 4.3 (from Page 11 to

Page 15). We added the definition of a slice. We also showed how the algorithm works (please see Figure 10).

R1.38. Page 11: l.48 "would": is colloquial

Fixed. We changed the sentence to: "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."

R1.39. Page 11: l.55 comma after WALA[31]
Fixed.

R1.40. Page 12: l.6-l.14 what is the point in this paragraph? "At first sight, it may appear that static slicing alone be used..."... OK, and what at second sight?

We removed the entire paragraph. We agree that it is out of context. The text flows better now. Thanks for this.

R1.41. Page 12: l.20 "frame" -> frames
We could not find this typo.

R1.42. Page 12: l.24 shouldn't "i+offset" be frames[i+offset]?Algorithm 1: shouldn't the slicing criterion change in each iteration of the loop?

Yes, it should be frames[i+offset]. Thanks for pointing this. Line 6 of Algorithm 1 is now: BSlice currentBSlice ← backward slice from frames[i] to frames[i + offset];

The slicing criterion does change after each iteration. At each iteration, we ask WALA to perform a new backward static slicing between two different points of the SUT. The slice is static as we don't assume any knowledge about the SUT nor the root cause of the crash. Consequently, WALA only needs the SUT locations to perform the slicing.

R1.43. Page 12: l.40, Eq.(6): I need some clarification on the premise of \models: assuming "x" is a path, what does "x." mean? Also, SUT and the union of slides are quite different types, why is it legal for both to write "x."?Further, what is the first subset relation good for (the one without the "x"-qualification)? Please introduce the notation and explain the different terms formally.

We thank the reviewer for pointing out this equation. We reworked the whole section in order to provide a better understanding of the equations. Equation 6 has been split into two equations (Equations 6 and 7) and follows the same notation as in Equations 1, 2 and 3. Please see the changes to Equation 6 (now Equations 6 and 7). We also explained the equations in the surrounding text. Please see the last paragraph on Page 14 and the first paragraph on Page 15.

R1.44. Page 12: l.45 What do you mean by "c_{i>1}" needs to be included"? Is this an extra step or does the slicer find that "c_{i>1}" automatically? And why can one include expressions anyway? Doesn't the slice contain statements?

What we meant is that the only frame that needs to be untouched for the backward static slice to be meaningful is f0. In Figure 6, f0 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 slicing process. The result is a non-directed model checking, which is likely to fail. We improved the paper to reflect this. Please see text right after Equation 7 (Page 14 and 15).

R1.45. Page 12: l.46 "needs to be untouched": I can't follow

The slicing process in JCHARMING needs to start somewhere. If the first line of the crash trace, i.e., frame 0, is corrupt then JCHARMING cannot perform the slicing because it does not know where to go. The result is a non-directed model checking. We added a sentence to make this clear (please see the last paragraph before Section 4.4).

R1.46. Page 13
l.05 delete comma after "each"
l.10 delete "entry"
l.11 typo in s_i+1
l.16 "As shown before": where?

The typos have been fixed. For comment l.16, we added a reference to Figure 6.

R1.47. Page 14: l.48 "an hypothetical" -> a hypothetical
Fixed.

R1.48. Page 15: What are the limitations of the unlowering steps? Which parts can you not reconstruct? Please make explicit.

We are not sure we understand what the reviewer means by "unlowering steps". In general, JCHARMING reconstructs all parts of the program that need to be executed depending on the validity of the crash trace (used to build the backward slice that guides the model checker). If some frames (except Frame 0) of JCHARMING are not valid then the resulting slice contains more information than it should. This can slow down the model checker, but won't affect much the accuracy of the approach. We hope that this addresses the comment. We also hope that the changes we made to Section 4.3 where we discussed explicitly the concept of slicing and showed the steps of the algorithm will contribute to addressing this comment.

R1.49. Page 15: Figure 8: faillure -> failure (3x)
Fixed

R1.50. Page 15: l.49 "Kilo Line": I've never seen KLoC spelled out like this.

It is spelled KLoC in many of our references related to this work. We prefer to keep the same spelling to be consistent with the literature in this area.

R1.51. Page 15: l.52 "make files": shouldn't it be Makefiles?

It is now Makefiles.

R1.52. Page 16
l.22 "player in" -> player among
l.55 "built on the top" -> built on top
l.56 "that" -> , which
l.08, l.09: delete blanks before the setTarget... method (line 3 and 4 in trace, also in the trace on page 22)
l.38: add comma after "class"

Fixed. Thanks.

R1.53. Page 16: Removal of lines in the preprocessing step could also mean that variables relevant for control- or data dependence are not discovered, correct? This could then cause problem in the construction of test cases, correct?

In fact, the removed lines belong to software applications that are not part of the SUT such as the Java primitives or embedded libraries. If the bug lies there (outside the SUT) we can't reproduce it because JCHARMING will not be able to find the point of entry to compute the static slice. The preprocessing step does not affect the SUT. In other words, we do not remove anything that is part of the SUT. Consequently, the removed variables and lines won't affect the construction of test cases since the test cases are designed to exercise the SUT (which calls Java libraries). The test cases are not designed to exercise Java libraries directly.

R1.53. Page 21: Both cases in Section 6.2 I don't find particularly enlightening. Surely, not match-able segments of traces present problems, but that is hardly surprising. Besides, in the second case, JCHARMING could employ a different definition of equality and in the first case include a heuristic. Instead of discussing what is more or less obvious, it would be much more interesting to count the frequency of those cases. Or to cluster resp. classify them, or to discuss how JCHARMING could improve its precision. Asked differently, what characterizes the "quality of ... crash trace" you refer to on page 23?

A good crash trace is a trace where all the frames are intact. In other words, there is no corrupt frame. This may happen but not always. This is why we introduced the partially reproduced bugs. We agree with the reviewer that a study that focuses on understanding what causes the poor quality of traces will be worth conducting. This is however beyond the scope of our paper. The results of such a study can help develop tools that would allow the generation and the reporting of good quality traces. We intend in the future to undertake this study.

R1.54. Page 21: Several bug reports contain spelling mistakes (the two reports on page 21 contain 2 resp. 1 typo, the one on p.22f. another one). Are those in the original reports? (If so I'm not sure if one is allowed to correct them. lease inform yourself and get back to me :-)

Yes, these are the original reports. We did not want to modify them. We added two sentences in the paper to say that the bug descriptions are reported in the paper as submitted by the developers. Please see the last two sentences right before Section 6.1.

R1.55. Page 22- l.42-46 I can't follow your argumentation: why did JCHARMING not find the crash location? What is different in the case of Struts?

This is because the error belongs to Struts, which is another software application used by Log4J, the SUT. We did not have access to Struts.

R1.56. Page 22
l.47 capitalize log4j
l.50 add comma after 46721
Fixed.

R1.57. Page 22: l.50/51 "We believe that JCHARMING could have successfully..": why don't you just try it? Is it so complicated to rerun JCHARMING?

The generated exception does not belong to Log4J. Indeed, the first frame is from the Struts program, the rest of the crash trace belongs to Log4j. Since we were using Log4J as the SUT, JCHARMING could not perform the backward static slicing and fell back to undirected model checking of Log4j as a whole. This undirected model checking failed for bugs 11570, 40212, 41186, 45335, 46271, 47912 and 47957 (Table II).

R1.58: Page 23: l.27 add an "a" before "program's behavior"
Fixed.

R1.59. Page 23: l.50-54: correct, certain failures are "hardly reproducible." What to conclude, though?

We can conclude that bug reproduction should not be the only way to uncover the causes of a failure. In our view, there will always be situations where it is almost impossible to reproduce a bug. We should therefore investigate techniques for fixing bugs that do not necessarily require bug reproduction. One possible direction is to invest in powerful (and “intelligent”) monitoring techniques that keep track of what goes on in the system. This way, if a bug appears, we can replay the system through the monitored data. The problem is that it is challenging to determine in advance what to monitor and to what extent. Heavy monitoring is simply not an option since it slows down the system, unless, perhaps, hardware solutions are used.

R1.60. References (plenty of problems):
Capitalization inconsistent in conference titles (e.g., [1,2,3,4]). Capitalization wrong in first names, tools, etc (e.g, [8,17,19]). Problems with non-ASCII letters (e.g., [15,25]). Incomplete reference (e.g., [14,17])

We have reviewed all the references and made sure that they are consistent and complete. Thanks.

<u>Reviewer: 2</u>

R2.1. The information obtained in the backward slicing operation serves as a way to guide the model checking activity. The approach is interesting, and reasonably well explained, but the formal aspects need to be improved. Whenever formal notations are used, I got quite confused by the inconsistencies that are currently in the text (see my detailed comments). I also would have liked to learn more about the way properties are expressed. This is not explained at all in the current text.

We have made significant changes to the paper by modifying the equations and adding descriptions where necessary. Please see red text in Section 3 and Sections 4.3 and 4.4.

For the properties, we added two paragraphs explaining how JPF handles properties. Please see the red text in the beginning of Section 4.4. In summary, 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. In opposition to symbolic model checker based on binary decision diagrams. 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.

R2.2. Regarding the directly related work, I think the authors give a nice overview, but with respect to "directed model checking", the authors do not explain its origins at all (again, see detailed comments).

In the related work we only focused on bug reproduction techniques. To our knowledge, directed model checking has never been used for bug reproduction.

This said, the reviewer suggested looking at some papers that discuss directed model checking as part of Comment R2.27. We reviewed those papers and cited them instead of the ones we had in the previous section. This is because the papers suggested by the reviewer seem to cover original work in the field. Please see paragraph that starts with “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.”

R2.3. The main benefit of model checking over testing is its ability to determine that a system is absolutely correct. By starting with a bug and guiding the checking to it, you are essentially not model checking anymore, but bug hunting. Of course, directed model checking in a way depends on there being a bug, otherwise is devolves into traditional model checking. A discussion along those lines should be added, I think, to make this clear to the reader, since "model checking" usually has this completeness characteristic of being able to show that a system is correct.

We agree with the reviewer. In fact, this is exactly what we are doing: bug hunting (and reproduction) using directed model checking. We added a paragraph on Page 8 (right before the beginning of Section 4) where we discuss the fact that unlike model checking, directed model checking is not complete. But the purpose of our work is not to “check” specific properties of the code but to reproduce the bug.

R2.4. Can you say anything regarding the length of your counter-example traces? For understandability, it is crucial that counter-examples are short. It would be interesting to know whether JCHARMING tends to produce the shortest counter-example or not, and if not, whether you could alter the technique to improve this.

Thanks for suggesting this. We added a new column to Table 2 (Length of counter-examples). We added a paragraph right after the table to explain the results in terms of length of counter-examples. In short, JCHARMING counter-examples were composed of 5.04 Java statements in average. 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 believe that having five statements in average is a reasonable number for the developers to understand the problem.

R2.5. Can you produce results that show to what extent your approach is optimal? How many states were visited vs. how many absolutely need to be visited to find the counter-example? Experiments like that would give an indication how well JCHARMING is able to find the counter-example based on the backward slice information.

We thought of providing such data, but unfortunately, JPF, the model checking engine we are using, does not provide this information by default. We could have implemented another listener, very much like what we did for generating test cases (see Section 4.6). However, implementing such a listener would require a large development effort. Furthermore, we will have to run all our experiments again in order to extract the amount of computed states for the sliced programs. Finally, the amount of states that would need to be visited on the complete program (i.e., without the JCHARMING slicing approach) will be, at best, an educated guess. Indeed, as shown by our experiments (Table II), only two reproduced bugs out of 30 were computed using undirected model checking. For the 28 remaining bugs, the states filled up the 8GB of RAM of our test machine before completion. We could only report the amount of states generated before running out of memory and the estimate of the remaining states will be hardly accurate.

In short, it would have taken a considerable amount of time and effort to implement a mechanism that outputs the number of visited states and even if we did this, we could not guarantee the accuracy of the results because of memory limitations.

R2.6.
Page 1 - piece -> pieces
Page 2 - functions output -> function outputs
Page 3 - oversee -> oversees
Page 3 - a NP-Complex -> an NP-complete

We fixed all these typos. We changed NP-complex to NP-complete everywhere in the paper.

R2.7. different SMT (satisfiability modulo theories) solvers: different from what?

We meant various SMTs and not ‘different’. We replaced the word “different” by “various”.

R2.8. model checking techniques: SMT is being used for model checking itself, so the "even" remark seems to be odd.

We rephrased the sentence to read: “In order to overcome these limitations, some researchers have proposed to use various SMT (satisfiability modulo theories) solvers \cite{Dutertre2006} and model checking techniques \cite{Visser2003}.

R2.9. For example, the reading of a file that is only present on the hard drive of the customer or the reception of a faulty network packet: This is not a complete sentence. Please complete it or attach it to the previous sentence.

We modified the sentence as follows: “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 such as the reading of a file that is only present on the hard drive of the customer or the reception of a faulty network packet \cite{Chen2013a, Nayrolles2015}.”

R2.10. reproduce a crash to be purely external.: so, is it possible to reproduce the crash / bug or not? You seem to indicate both.

We believe that it is almost impossible to reproduce a crash that is purely external. We changed the sentence to state that “it is almost impossible” instead of “that it is challenging”. Please see the paragraph right before Section 2.1. In our opinion, it is a strong statement to say that “it is impossible”. One can always run various scenarios by varying different inputs until the bug is exercised. This technique can be time-consuming (and perhaps impractical), but may be needed if the bug is severe and reproducing it is the only way to understand the causes.

R2.11. Page 4 - file whose size averages 70KB -> files that are 70Kb on average
Fixed.

R2.12. Similarly, private: similar to the approach of Clause et al.? Please mention this

We agree with this. We modified the sentence to make it explicit that the approach presented by Burg et al. [20] does not address the problem of data privacy, similar to the approach of Clause et al. Please see the changes in red (Page 4, the second paragraph from the top) .

R2.13
clones: remove s
JRaptrue -> JRapture
creator -> creators
it saw -> as
phase: a proper ending of this sentence is missing, e.g. "are presented"

Fixed. We thank the reviewer for the thorough review. It is very much appreciated.

R2.14. that can also monitor other software system than the intended ones: why is that a problem?

It is simply not practical to monitor every application that runs on the host. This will be computationally demanding, causing a significant overhead. In addition, this type of monitoring may cause privacy concerns since some applications may be using confidential data.

R2.15.
system -> systems
Page 5 - level, there -> level. There
Page 5 - lies: relies?
Page 5 - really begins -> really beginning
Page 6 - used based -> based

Fixed. Thanks.

R2.16. Except for STAR: so how does JCHARMING compare to STAR?

STAR uses symbolic analysis, whereas JCHAMRING uses directed model checking. Our approach achieves 80% accuracy (when including partially reproduced bugs) compared to 55% for STAR. We used the same systems as the ones used in STAR, but we did not use the same bugs. Some of the bugs used by STAR authors were no longer available.

We wanted to compare experimentally the two tools on the same dataset, but unfortunately, we were not able to compile or run STAR. We contacted the authors but were not able to get the assistance needed. It would indeed be interesting to conduct such a comparative study with the same set of systems and bugs. We hope we can do this in the future. We will most likely need to re-implement STAR from scratch.

We want also to note that we improved the related work section by indicating the accuracy of the surveyed approaches. However, comparing JCHARMING with these approaches just by looking at the accuracy won't be fair. As we discussed in the paper, most of these approaches use heavy instrumentation of the code or violate privacy of the data.

R2.17. The system: for model checking, it is crucial to mention that the system is formally defined (has a clear semantics)

We agree with the reviewer. We modified the text to: "Model checking (also known as property checking) will, given a formally defined system (that could be software \cite{Visser2003} or hardware based \cite{kropf1999introduction}), 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 \cite{Kripke1963} structure:"

R2.18. P is the set of properties that each state satisfies → P is a set of state predicates, and there should be a function assigning subsets of P to states. Typically, however, P does not hold in its entirety in all reachable states
R2.19. The SUT is said to satisfy a set of properties: you seem to give a definition here of satisfying a property that corresponds with testing. For model checking, a property is said to hold if there exists no trace contradicting it.
R2.20. not that p holds nor that ∀x, p is satisfiable: if that is the case, then the system does not satisfy p.

To address the above comments and other related comments, we have made significant changes to the paper by modifying the equations and adding descriptions where necessary. Please see Section 3 and Section 4.3 as well as the surrounding text in red. We believe that these changes have improved the formal aspect of the paper. We thank this reviewer and Reviewer 1 for pointing out the imprecisions in the equations.

R2.21. fair environment: is there any relation to fairness in formal verification, as in "Behaviour that is infinitely often enabled is also infinitely often executed"?

We used the term “fair environment” to mean a typical environment where everything runs normally. To avoid confusion, we changed the word “fair” to “typical” and added a sentence to describe what we mean by typical environment. Please see Page 7, second paragraph right after Equation (2).

R2.22. properties: How are the properties expressed, using which temporal logics?

As described when addressing Comment R2.1, 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. This is contrasted with a symbolic model checker based on binary decision diagrams. 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, JPF does support LTL definitions and we might leverage this in a near future in order to reproduce other complex bugs such as the ones related to multi-threading.

R2.23. Page 7: - (3): this is closer to satisfying a property in a model checking context. Usually, the "for all x" does not need to be added though. In fact, the shorter notation SUT |= p would be more in line with what is common

We agree. We modified the equation. In fact, we made significant improvements to the equations in order to improve the formal aspect of the paper and follow common practice. Please see red text in Sections 3, 4.2, and 4.3.

R2.24. such a path exists -> there exists a path contradicting this

We agree with this as well. It makes the paper sound better. We modified the sentence as follows: “If there exists a path contradicting …..”

R2.25. what to look for in order to detect the causes: do you mean intermediately, i.e. along the trace? Because the final goal is known here (i > 2)

We used a simple example where the cause is known, but we still do not how to reproduce this bug in order to fix it. For this simple example, a software developer will most likely go and investigate the statements surrounding the for-loop. But for real bugs, this might not be as straightforward. What we need is a way to reproduce the bug, which is really the objective of JCHARMING. The crash trace is used to identify the program slice, which is then fed to the directed model checking engine. The model checking engine exercises the system until it crashes (due to the same bug). In the meantime, the data used by the model checker during execution is used to build the JUnit test. This way, the developers can reproduce the crash and understand the causes.

R2.26. complete, but impractical: that depends on the search order, which, as you mention later, opens the door for directed model checking

Exactly. In JCHARMING, the use of crash traces and the resulting backward slice define the search order that narrows the number of states that need to be checked. Without these, we will have to resort to complete model checking, which, in our opinion, won’t be practical for large systems.

R2.27. directed (or guided) model checking has been introduced [28]: actually, the term "directed model checking" was coined in an earlier paper, namely "Directed Explicit-State Model Checking in the Validation of Communication Protocols", by Edelkamp, Leue, and Lluch-Lafuente. Also, you should refer to "Survey on Directed Model Checking" by Edelkamp, Schuppan, Bosnacki, Wijs, Fehnker and Aljazzar

We have added the references when introducing directed model checking. Please see Section 3.

R2.28
use insights -> uses insights
Page 9: Figure 5: Bar.Goo -> Bar.Foo. Also, it may improve readability if you add f0, f1 etc to the figure, as you refer to these in the text
Page 10: that led to the: remove the
are not limited to, -> but not necessarily, to
also reduce -> reduce
foo: previously, "foo" was capitalised. Please be consistent
foo: see previous statement

Fixed. Thanks.

R2.29.
Page 11:
- equation 5: I have a number of comments regarding (5):
*** i ranges from 0 to entry, but f_entry was previously not defined, only just entry**
*** i should range to entry-1, since otherwise there should also be an f_entry+1, and this appears not to be the case**
*** equation 4 states that the union equals the bslice from f_0 to entry, here it is a subset. Which one is correct?**

We have made significant changes to the paper to improve all the equations. Please see Section 3, 4.2, and 4.3. We have also added examples to illustrate how Algorithm 1 works. We added several figures to explain in detail how the process works. We believe that the new version of the paper clarifies many ambiguities raised by the reviewer.

R2.30. Indeed, in Figure 6, the set of states: based on figure 6, this set should be empty, as f_2 has no outgoing transitions. The other sets you mention (f2 to f1 and f1 to f0) are also empty.
assuming that z2 is a prerequisite: what is a prerequisite?
Which state in figure 6 corresponds with entry?
Why is bslice from f_0 to entry the given set of states?And the sentence ends prematurely. Which set corresponds with the union of the given bslices? This example is very confusing. Please fix this.
between each frame -> between each two frames

We acknowledge that Figure 6 of the original submission was confusing. We removed it and replaced it with Figure 9 (revised version). The figure and surrounding text explain the benefits of performing the union of slices over computing one backward slice between the last frame and the first frame. We added the new figure and described the reasoning behind this in the surrounding text. Please see Figure 9 and 10 (Pages 12, 13 and 14).

R2.31. From line 1 to line 5: in Alg. 1, please number the lines, as you use these numbers in the text. Also the lines in the text are wrong. Instead of "line 1 to 5", you should write "line 1 to 6", and instead of "line 6 and ends at line 15" you should write "line 7 and ends at line 15".

We numbered the lines of the algorithm and fixed the use of these lines in the text. Thanks.

R2.32. then JCHARMING: remove then
Fixed.

R2.33. the possibility to resort to non-directed model checking: and what if the final slice is not empty? How does it help the directed model checking? This is explained later, but it would improve the text if you already give some indication how this works here.

R2.34. provides -> provide
The typo has been fixed.

R2.35. Page 12: Frames frames ← extract frames from crash stack;: do not write text in math mode

We fixed this by no longer using the math mode when writing text.

R2.36. size of frame: which frame? Multiple were extracted in previous line. Or do you mean frames?
Yes, we meant frames. This typo has been corrected.

R2.37. equation 6: I have the same issues with this equation as with equation 5. In addition:
*** are the two cases between the brackets conjunctives? How are they combined?**
*** what is the difference between the two cases? How should I read the second one?**

A comma was missing in the equation. I believe it's easier to understand now.

R2.38. That is the -> That is, the
Fixed.

R2.39. only frame that needs to be untouched for the backward static slice to be meaningful is f0.: can you elaborate on this? Why is this the case?

F_0 is the crash point. If the frame containing the crash point is corrupt or missing, then we can't perform the slicing (as we don't have any direction to point to). If this is the case, the slicing will fail and we will fall back to undirected model checking. We added this to the paper (please see the paragraph right before Section 4.4. (Page 14)).

R2.40. Page 13:
choose -> chose
each, forward -> each forward
with entry the states: something is wrong here. Please fix
that falls -> that fall
a set a property -> either "a property" or "a set of properties"
transitions -> transition

Fixed.

R2.41. t is the percentage: do not start a sentence with a mathematical symbol

We modified the sentence (see the last sentence of the second paragraph of Section 4.5). We also proof read the paper to correct similar mistakes.

R2.42. the exercise of the bug -> execution of the bug, or "the bug to appear"

We agree with this. The sentence now reads: “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 appear.”

R2.43.
Page 14
industrial size -> industrial sized
listens the -> listens to the

Fixed.

R2.44. Page 15: a loop from 0 to 3: what do you mean by this?

We changed the sentence to explicitly indicate that the jsme.Bar.foo(int) method will execute the for-loop from i=0 until i=3 and throw an exception at i = 3.

R2.45. as shown in Figure 8.: please discuss figure 8

We added a discussion of Figure 8 (which is now Figure 12) , which shows the test case, generated based on JUnit templates. Please see the discussion on Pages 17 and 18, right before Figure 12.

R2.46. study -> studies
Fixed.

R2.47. reasonable amount of time: what is a reasonable amount of time?

This is good point. What is reasonable for one person may not be reasonable for another. For us, we did not want to have a bug reproduction approach that takes hours. JCHARMING takes in average 19 minutes to reproduce a bug, which seems to be acceptable.

R2.48. Page 16:
time JfreeChart -> time. JfreeChart
on the top -> on top

Fixed.

R2.49- Mahout: in the table, swap the entries for Hadoop and Mahout, to keep it in line with the order in the text

The reviewer is right. We have swapped them. Thanks for noticing this.

R2.50. Page 17 above t=80%: and below 100%, I assume

Yes. We changed the sentence to become: “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%.”

R2.51. fill out all -> fill all

Fixed

R2.52. Page 18: dispatching:”: end sentence with '.'

This is a quote from the original bug report where the sentence ends with a “:”. Furthermore, the “:” introduces the stack traces to come. We kept the reports as they were submitted. We added two sentences in the paper to explain that the bug descriptions were reported in the paper as submitted by the developers. Please see the last two sentences right before Section 6.1.

R2.53.
Page 19:
neither -> either
nor -> or
Page 20:
hour -> hours
Page 21:
ChecksumException Re-running -> ChecksumException. Re-running
the attached -> of the attached
While, -> While
were -> was
Page 22:
of few -> of a few
Application -> Applications

Fixed.

R2.54. We believe that JCHARMING could have successfully reproduced the crash: couldn't you test this?

We could not test this because the first frame belongs to Struts, an external system (not one of the SUTs). We did not think that it was necessarily to add Struts to the list of SUTs (we already have a good representative list of SUTs). The argument we were making in the paper is that if Struts was one of the SUTs, we would have been able to reproduce the bug that had that frame as the first frame, just like the bugs we reproduced for other systems.

R2.55 - Page 23: the SUTs analyzed by JCHARMING are the same as the ones used in similar studies: how do the results of JCHARMING actually compare to those obtained in similar studies?

We completed the related work section with the results of each approach, when they are clearly reported. Many papers focus on implementations details and/or only report successfully-reproduced bug without discussing the bugs that were not reproduced. Please see modifications made to the related work section. They are in red text.

R2.56 -
footnote "described here ... ": make this note a full sentence
Page 24:
all based on -> all written in
bug -> bugs
involves -> involve

Fixed.

R2.57 –
Katoen Jp -> Katoen, J-P.
Page 26:
41. Hadoop A. Hadoop 2011.
42. Mahout A. Scalable machine learning and data mining 2012.
43. Snyder B, Bosanac D, Davies R. ActiveMQ in Action 2011; :408.: these are not very helpful references. Please add more information

We reviewed all the references and made sure that they are complete and consistent.