How Helpful Are Automated Debugging Tools?

Jeremias Rößler
Department of Computer Science
Saarland University
Saarbrücken, Germany
roessler@cs.uni-saarland.de

***Abstract*—The field of automated debugging, which is concerned with the automation of identifying and correcting a failure's root cause, has made tremendous advancements in the past. However, some of the reported progress may be due to unrealistic assumptions that underlie the evaluation of automated debugging tools. These unrealistic assumptions concern the work process of developers and their ability to detect faulty code without explanatory context, as well as the size and arrangement of fixes. Instead of trying to locate the fault, we propose to help the developer *understand* it, thus enabling her to decide which fix she deems most appropriate. This would entail the need to employ a completely different evaluation scheme that bases on feedback from actual users of the tools in realistic usage scenarios. With this paper we propose the details for a first such user study.**

***Keywords*-Automated Debugging, Statistical Debugging, User Study**

I. Introduction

Automated debugging—identifying and correcting a failure's root cause automatically—has made tremendous advancements in recent years. Many publications in the field focus on *statistical* approaches to defect localization [1]–[6]. These approaches identify potentially faulty code by observing the characteristics of a large number of failing and passing program executions. Intuitively, code that is exercised mostly by failing executions is more likely to be faulty than code that is exercised mostly by passing executions. So these techniques *rank* individual lines of code of the given program according to their likelihood to contain the defect. The evaluation scheme [7] commonly applied to these rankings, bases on some assumptions that a recent user study by Parnin and Orso [8] showed several issues with.

1) The evaluation assumes that the developer will proceed linearly through long lists of unrelated suspicious locations. However, this contrasts the findings of the user study, in which successful developers *jump* through such lists, and only developers that were unable to fix the defect proceeded in a linear fashion.
2) Success is measured in the percentage of the code of the whole program, the developer would have to inspect before reaching the defect in the ranking. In contrast, the user study showed that, after a certain number of statements inspected, the user's confidence in the results ceased and the users stopped using the tool. That number was *absolute* and not related to the size of the program under investigation.
3) The evaluation assumes *perfect bug understanding*, which means showing the defective line of code in isolation suffices for the developer to recognize the defect. Again, the user study showed that developers need some explanation or context to understand the defect before being able to recognize it in the code.

Also, Parnin and Orso [8] found something else during their user study: developers fix the same bug in *different locations*. However, while the authors simply assumed that some developers chose the "wrong" place to fix the bug, we think reality is much more intricate: we assume that there is no "right" location of the bug in the first place. Instead, the bug can be fixed equally well in multiple locations. The whole notion of the *location of the bug* also stems from a flaw in the evaluation scheme: in all evaluations that we are aware of (both user studies and quantitative evaluations), either a given correct program is changed to *introduce* (seed) bugs, then the "correct" fix is the one that reverts the change. Or the "correct" fix is the solution to the bug as recorded in a bug database, which we deem merely an accidental selection of the infinitely many possible fixes. And even more important, we assume that, often enough, the fix of a bug consists of more than a single line of code and can even be distributed over several locations in the code. This takes the whole notion of the "location of a bug" ad absurdum.

Instead of tying to locate the bug, we should aim to *help the developer understand the bug*—which simultaneously addresses all problems mentioned above. However, as bug understanding is not a directly measurable quantity, this makes the evaluation problematic. To address this issue we will perform a user study as detailed in the following.

In the remainder of this paper, we are first going to introduce the approach we want to evaluate (Section II). Then we will state the research questions and hypotheses (Section III), followed by some context of the user study, the participants and the evaluation scheme (Section IV). We finish with conclusions (Section V).

II. BugEx—Explaining Bugs

To overcome some of the issues of statistical debugging (as given in Section I), we came up with an approach named

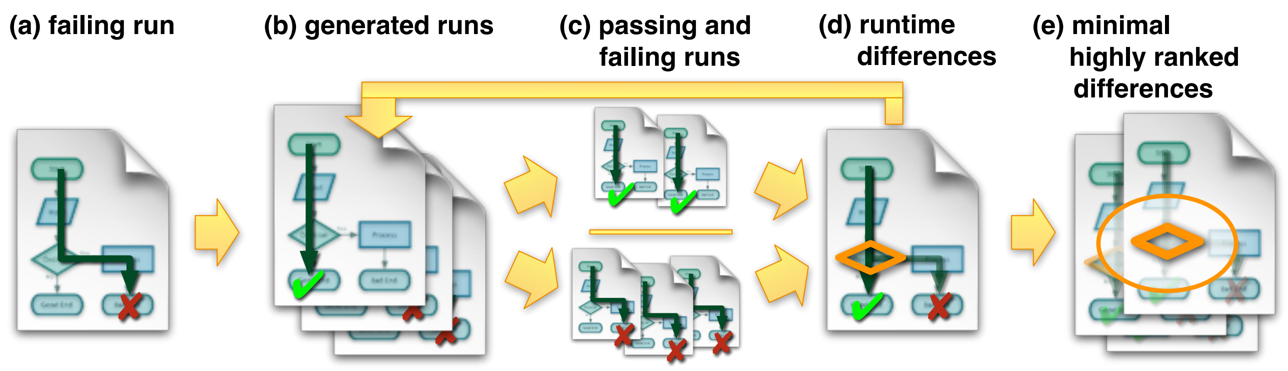
(a) failing run
(b) generated runs
(c) passing and failing runs
(d) runtime differences
(e) minimal highly ranked differences

Figure 1. BUGEX in a nutshell. Starting with a failing run (a), BUGEX generates additional runs (b). BUGEX establishes the differences between passing and failing runs (c) in terms of runtime facts (d)—right now, branches taken and state predicates. It statistically ranks these differences and generates more runs (b) to focus on and further minimize highly ranked differences. Eventually, BUGEX produces a minimal list of highly ranked fact differences (e).

BUGEX. In a nutshell, here is how BUGEX works:

a) Starting off with a failing test case,
b) we generate additional similar test cases that
c) either fail with the very same exception as the original test case or pass but still execute the critical code.
d) From all of these test cases, we extract runtime information that we call facts. Using a statistical debugging technique [6], we rank those facts according to their correlation to the failure. All highly correlated facts are target to test case generation (indicated by the loop), where we try to generate a passing or a failing test case that are as similar to the original test case as possible but differ in the targeted fact.
e) We end up with a minimal set of facts that are statistically highly correlated to the failure.

The feedback loop from statistical ranking to test case generation vastly increases performance and at the same time allows us to create a correlation with outstanding probability values. Usually the top ranked facts together account for more than 99,999% of the normalized probability to be correlated with the failure. It is a generic approach that can be implemented for all kinds of runtime information. We implemented the approach for branches and predicates. We think that the combination of different runtime information shows different aspects of the failure and gives context that helps the developer understand the underlying defect. In recent work we used a set of seven real world defects to qualitatively evaluate an implementation of the approach, and we think that the results show that the implementation meets its goal. However, it may well be that our evaluation is subjective. So in order to objectively evaluate our approach, we plan to perform the user study as proposed below.

III. RESEARCH QUESTIONS AND HYPOTHESES

We deem BUGEX helpful for developers to understand a defect. However, this has to be validated by a user study. Thus our first research hypothesis (RH) is:

RH1: BUGEX *is valuable for the debugging task.*

This is a purely subjective evaluation criterion that indicates whether users like to use the tool. We consider our tool to be valuable for users, if they continuously prefer to use the tool over not using it. Additionally, we will ask the users after each bug they fixed, whether they liked the tool, what they liked about it and what could be improved.

Our approach belongs to the field of automated debugging and has to compete with other approaches from that field:

RH2: BUGEX *is more helpful than state-of-the-art statistical debugging approaches.*

We will consider our tool to be more helpful than an alternative approach if users continuously prefer it over the alternative. Additionally, we request the estimated amount of time needed for fixing a bug, how helpful users considered the tool and how confident they are in the fix they implemented.

As indicated in the Introduction (Section I), we assume that if a defect is expressed only in terms of a failing test case, then there are many equally valid possibilities to fix it. The question is, whether developers are aware of this:

RH3: *Developers are aware that there are multiple equally valid ways to fix a defect.*

To test this hypothesis, we will ask users after each bug they fixed, whether they had alternatives and how many.

We further assume that different fixes have different probabilities to be chosen by the developer. The decision for a specific fix (including a rewrite of the complete program) is influenced or even mainly driven by the expected overall effort, which includes foreseeable future changes and fixes to probable future bugs. To make a concrete example: if the developer thinks, that implementing the program with a different design will save some effort in the long run, and if there are no additional factors such as upcoming program release dates, then this will be the fix of choice.

RH4: *The main reason to chose a specific fix is that it bears the least amount of estimated overall effort.*

To test this hypothesis, we will ask users after each fix for which they were aware of alternatives, which factors influenced their decision for the fix they eventually chose.

Many studies (including the one by Parnin and Orso [8]) show that users will only consider a few results before giving up on a tool. So we assume that less results are more valuable, even if they do not contain the optimal solution.

RH5: *Users prefer a smaller result set over a larger one—even if it does not contain the optimal solution.*

We consider this hypothesis confirmed if users continuously prefer a smaller result set over a larger one.

IV. Proposed User Study

Some of the typical problems of user studies with developers include the following:

1) The study is performed with only a few developers or does not exceed a certain time boundary (or both).
2) Either developers work on different tasks and thus the results are not comparable, or developers are all given the same program, which encompasses that
   a) all programmers are alien to the code and
   b) the code cannot exceed a certain size and complexity (see also 1).

All of this limits the generalizability of a study.

To address these problems and the ones given in the Introduction (Section I), we will perform a different user study. We utilize the participants of a course in programming that is due in the second semester of studying computer science. According to past experience, about 200 students of vastly differing programming experience are expected to register for the course, whereas only about 100 are expected to complete it. During the course, all of these students are given programming tasks with increasing levels of difficulty. These tasks start with a simple *Hello World*-implementation, and eventually end with a complete game such as *Tetris*, including Graphical User Interface and Artificial Intelligence. The students have a certain amount of time to complete the programming task that increases with the difficulty of the task. They work unmonitored on their chosen workstation, in their chosen environment and at their chosen time. They can suspend and continue to work any time.

All intermediate programming results are submitted to a central server and built and tested every night. The students receive feedback from automated unit tests. If all tests pass, the programming task is assumed to be completed. Some of the tests are given to the students together with the task and can be run on the workstations as often as needed and are also run on the server after every commit. Some other tests are unknown to the students—they are executed only on the server and only once every night (called *nightly* tests). Of these nightly tests, students only receive the failure message if a test fails. This creates two different usage scenarios: a normal usage scenario, where tests can be executed and manually debugged as often as needed, and a scenario, where students cannot rely on manual debugging at all.

For all failing tests (after every commit for the given tests and every night for the nightly tests), our BugEx prototype will generate explanations that the students can request. Using this feedback is optional. We will detect every request of the feedback, and thus will be able to see how beneficial the students deem the feedback to be (i.e., if they cease to request it). When requesting it, the students are required to complete a very short survey regarding the last feedback they received, before being given access to the new feedback. Some of the questions of this survey are

- How useful was the feedback?
- Was the bug fixed?
- How long they estimate it took them to fix the bug?
- How confident they are in the chosen fix?
- How many alternatives they noticed to fix the bug?
- Why they chose that alternative to fix the bug?

Completing the surveys will only be enforced by not giving access to further feedback. So if students cease to use the feedback, they will also cease to complete the surveys. However, as some of the tests are only executed on the server, based on past experience we expect students to be interested in getting as much additional information about these nightly tests as they can. Therefore, even if students will cease to use the feedback of the given tests in favor of conventional manual debugging, they will likely still want to receive the feedback for the nightly tests. And this bears another advantage: students might be reluctant to try and become acquainted with an alternative debugging method. But if they continue to use the feedback for the nightly tests, this might help to overcome this inhibition and familiarize them with the feedback from the system. As a downside, since complexity of the tasks increases also over time, this will mask whether automated debugging is more helpful for more complex debugging tasks.

The proposed approach addresses all of the typical problems a user study faces: The study is performed with a large group of developers and there is practically no boundary on the time the developers may need to fulfill the programming task. Also, the students will work with *their own* code. This means that they already know the code sufficiently well and that the code can be of arbitrary size and complexity (but, of course, directed towards solving the given problem). Still the results are comparable, as all participants work on the *same tasks*. The tasks are mandatory to pass the course, so the students have a strong motivation to complete them. And because they cannot use conventional manual debugging on the nightly tests, they have a strong *intrinsic* motivation to use the alternative debugging approach.

To appropriately address the research questions formulated in Section III, we are going to offer four different options for feedback each participant can choose to receive. After choosing an option and before being able to receive the feedback of the next option, the participant will have to complete a short survey. Additional to the survey results, for each option we will continuously capture the number of times users requested it. The four options are:

- **OPT1** This option will be random feedback. It substitutes a control group and allows us to check research question **RH1**.
- **OPT2** This option will be the feedback of a state-of-the-art statistical debugging approach and establish the ground truth that allows us to check research questions **RH2** and **RH5**.
- **OPT3** This option will be the feedback of a state-of-the-art statistical debugging approach, but the feedback will only contain a limited number of results. This option allows us to check research question **RH5**.
- **OPT4** This option will be the feedback from BugEx that will be compared to **OPT1** and **OPT2** to check **RH1** and **RH2**.

The options will be assigned neutral names and the assignment of names to options will differ for each student. After each project, the assignment of the names to the options will be randomized (and this will be announced), so students have to reevaluate all options at least once per project.

Additional to that, we collect the results from the surveys after fixing individual bugs. The research questions **RH3** and **RH4** will be addressed by these surveys about bug fixes the students implemented.

Threats to Validity

One of the drawbacks we conceive for this user study is that it will be conducted with students only, which introduces a sampling bias. However, a short mandatory survey at the beginning of the course about the programming skills and experience of the participating students will allow us to draw some conclusions about how the results may generalize.

One problem that may arise with the design of the study is the introduction of a self-selection bias: by selecting an option and respectively a system students want to receive feedback from, they select themselves into a group. We counter this thread by three measures:

1) The assignment of the system that generates the feedback to the neutral option name is kept secret.
2) The assignment is different for every student.
3) The assignment will be randomized after every project.
4) **OPT1** serves a similar purpose as a control group.

Another risk comes from the *Hawthorne-Effect*. This effect describes the phenomenon, that participants of an intervention study in worklife will behave differently simply because they know that they participate in a study. However, the very existence of this effect has been questioned recently [9], and the results of the corresponding studies have been attributed to other unmonitored factors. Generally, it is impossible to conduct studies with human beings where all possible factors are controlled or even monitored. Observing effects that originate from factors that have been (intentionally or unintentionally) ignored is a major thread to every study on human beings.

V. Conclusions

To the best of our knowledge, the user study as proposed in this paper is unprecedented in the field of automated debugging. This makes it very promising in its ability to counter issues we detected in earlier studies. But at the same time, this causes many uncertainties and risks due to the lack of experience with such a study.

Acknowledgments

We would like to thank Sebastian Hack, Klaas Boesche and Eva May for their support of our experiment during the course *Programmierung 2* at Saarland University as well as feedback on earlier versions of this paper. We would also like to thank Matthias Höschele for some inspiring ideas.

References

[1] J. Jones, M. J. Harrold, and J. Stasko, “Visualization of test information to assist fault localization,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2002, pp. 467–477.

[2] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, “Scalable statistical bug isolation,” in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2005, pp. 15–26.

[3] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, “SOBER: Statistical model-based bug localization,” in *Proceedings of 10th European Software Engineering Conference and 13th Foundations on Software Engineering (ESEC/FSE)*, 2005, pp. 286–295.

[4] D. Hao, Y. Pan, L. Zhang, W. Zhao, H. Mei, and J. Sun, “A similarity-aware approach to testing based fault localization,” in *Proceedings of the Conference on Automated Software Engineering (ASE)*, 2005, pp. 291–294.

[5] C. Liu and J. Han, “Failure proximity: A fault localization-based approach,” in *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, 2006, pp. 286–295.

[6] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, “An observation-based model for fault localization,” in *Sixth International Workshop on Dynamic Analysis (WODA)*, 2008, pp. 64–70.

[7] M. Renieris and S. P. Reiss, “Fault localization with nearest neighbor queries,” in *Proceedings of the Conference on Automated Software Engineering (ASE)*, 2003, pp. 30–39.

[8] C. Parnin and A. Orso, “Are Automated Debugging Techniques Actually Helping Programmers?” in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2011, pp. 199–209.

[9] G. Wickström and T. Bendix, “The ”hawthorne effect”—what did the original hawthorne studies actually show?” *Scandinavian Journal of Work, Environment & Health*, vol. 26, no. 4, pp. 363–367, 2000.