New Generation Debuggers
Défis GDRGPL 2020

Steven Costiou, Thomas Dupriez, Stéphane Ducasse
RMoD, Inria Lille - Nord Europe

December 13, 2019

Abstract

Debugging is a painful and costly practice, due to the nature of bugs, of the debugged programs, or to tools limitations. We describe several difficulties of debugging that present scientific challenges (*i.e.*, we don't know how to do it) or technological challenges (*i.e.*, we can't do it). We believe that addressing these challenges will lead to new generation debuggers that will significantly ease and lower the cost of debugging.

1 Bugs are ineluctable

Software systems are getting more and more complex. Changes happen more and more dynamically. Many domains are complex by essence: *e.g.*, a Siemens robot machine to sculpt metal has 30 degrees of freedom and the program to drive it has to handle such intrinsic complexity. Today, we see new programs based on AI: in such systems the behavior may change based on provided sample and learning database. Bugs will always appear because building reliable software systems is an extremely complex task. While formal proofs of programs help to catch bugs ahead of time, the cost of writing specifications, and the complexity of modeling the domain of execution and its environment means that it is no silver bullet and that bugs will continue to occur. In this context, we need better and new debugging approaches.

2 Definition of *bug* and *debugging*

The literature describes very accurately what practitioners observe when debugging programs [17]. The source of a bug is a *defect*, *i.e.*, a piece of code that produces a run-time infection. An infection is an inconsistent state or behavior of the program that differs from what is expected, we may call that *errors*. This infection may spread and produce other errors, in a cascade effect which ultimately ends in an user-observable error: this is a *failure*. What we commonly call *bug* is actually the whole chain from the defect to the failure.

Debugging is the activity of tracking and fixing bugs. It follows a set of rigorous and systematic methods [15, 17, 14].

Reproducing the Failure. The first step is to make sure the failure can be reproduced. This is critical: without this step, gathering information on the bug is very difficult, and checking if a fix actually solves it is impossible.

Simplifying the Reproduction. After making the failure reproducible, the goal is to simplify and shrink the conditions needed to reproduce it, to have a simpler execution to inspect, and reduce the size of the suspect codebase.
Finding the Defect by using the Scientific Method. In order to reason backward from the failure to the defect, the developer observes the flawed execution, formulates a hypothesis as to what the defect is, tests this hypothesis through experiments, and refines her hypothesis based on the result. This loop continues until the developer finds the defect and why it caused the failure.
Fixing the Defect. The final step is fixing the defect, and checking that the failure no longer happens.

3 Challenges of debugging

In this section, we list problems that slow down, limit or prevent debugging, and formulate research questions to capture the inherent challenges they pose.

The symptom−source distance. This is one of the most common difficulty reported by practitioners when debugging. It refers to the fact that an observed failure (the symptom) does not occur at the same point in the code as the defect that provokes it (the source). For example, a defect in a program may produce an erratic value at some point, but the related failure is only observed when this value is used in a distant part of the code, for example when displayed in a GUI. This problem was first reported as a major difficulty in 1997 [7], but recent studies report that it is still a problem 20 years after [11].

- *How to design tools that help the developers overcome large symptom−source distances, and what are the requirements to build such tools?*

Concurrency and parallelism. They represent the second most frequent cause of today's hard bugs [11]. Reproducing bugs in concurrent/parallel programs requires more than running the same code and inputs. In such systems, processes and threads perform concurrent access to shared memory, and interact with each others in ways that may depend on external factors.

Ways by which single-threaded programs can be stepped and analyzed with existing debuggers cannot directly be applied to concurrent programs [10, 9]. Building adequate tools is hard, because we lack abstractions to express new debugging operations targeting concurrency bugs (*e.g.*, race conditions).

- *Can we find ways to systematically reproduce bugs in concurrent programs? If such ways do exist, what tools do we need to implement them and what abstractions do we need to support these implementations?*
- *We believe the best moment to observe a bug is the moment it actually appears and not in a post-mortem analysis. Could we debug a concurrent program on-the-fly [4], to catch and observe bugs at the moment they appear? How would this capability improve debugging practices?*

Bug Reproduction. Reproducing bugs is a vital step in understanding faults and finding the defect from which they originate [17, 14]. Developers can rerun the buggy program multiple times to better understand the bug, and test that the bug is actually gone after they make changes to the source code.

However, some bugs are hard (or even impossible) to reproduce in a controlled manner [12, 1, 17]. Bugs with non-deterministic aspects (*e.g.* concurrency) are notably hard to reproduce. Some bugs only happen after long executions and in specific conditions: those are materially inconvenient to reproduce. A company reported a bug occurring on its servers only after 15 days of heavy loads. Other bugs produce infections but then mask the faulty behavior or the inconsistent state of the program. Symptoms are no longer observable, which hinders bug reproduction and understanding.

- *We believe that to ease bug reproduction, we need to capture and leverage contextual run-time information. But how can we identify information relevant with the investigated bug? In addition, the evolution of the program state and execution path throughout an entire execution represents a lot of data. How can we overcome this limitation?*
- *However, if there are bugs that truly cannot be reproduced in a controlled manner, what alternative methodology can we use, and if it does not exist can we define new ways of debugging these bugs?*

Debugging the debugger. Debuggers are programs used to debug other programs. So debuggers themselves can have bugs. But debuggers make use of special techniques to monitor and control the execution of the programs they debug. These techniques are geared towards normal executions, and not executions using these techniques like debuggers. For example, Aspect-oriented programming is unable to debug itself, as aspects cannot be added to other aspects [16]. Kansas is a system able to debug itself [13]. It is a reflective system where developers interact with objects in a world. When a Kansas world is broken, another one is created from which the first world can be repaired. But Kansas cannot be debugged from within itself when that ability to create new worlds is broken. This illustrates a limit that seems impossible to overcome: a system cannot debug itself when one of its core features is broken.

- *Is it possible to design a debugger that cannot break, in the sense that it can always be used to fix its own bugs? If not, what is the minimal set of working debugging features required for the debugger to be able to debug itself?*

Designing usable debugging tools. In 1997, the debugging scandal[8] was the observation that debugging tools had progressed very little over 30 years. Recent studies [11, 2] show that new technologies have not significantly improved the debugging experience. Tools implementing these technologies are hard to understand and require considerable learning time, sometimes developers do not know about their existence [11].

Another pitfall for tools is to be either too specific, in such a way that they are almost never applicable in practice, or too generic, in such a way that they are always usable but not helpful enough for the specific bug being encountered. The challenge is to design tools that actually help the developers in the field, by striking the right balance between genericity and specialization, and the right balance between ease-of-use and feature-wealth [3, 5].

- *Can we pinpoint what characteristics a tool needs to have to be easy to pick-up by developers and powerful enough to help them?*
- *What is a debugger API that is powerful and versatile enough to allow developers to perform many debugging tasks that would normally require specific debugging tools or tedious manual operations?*

4 New Generation Debuggers

We believe that tackling the challenges laid out in this paper is the key to unlock new debugging capabilities, allowing the design and implementation of better debugging strategies and tools: **New Generation Debuggers!**

We believe a good research direction would be to identify and study the properties that programming languages and their infrastructure (*i.e.*, virtual machines) must exhibit to support debugging features that effectively help debugging. We started this work in the last few years, and implemented debugger prototypes [4, 5, 6], but there is still much to do.

References

[1] Agans, D. J. *Debugging: The 9 indispensable rules for finding even the most elusive software and hardware problems.* Amacom, 2002.

[2] Beller, M., Spruit, N., Spinellis, D., and Zaidman, A. On the dichotomy of debugging behavior among programmers. In *Proceedings of ICSE 18: 40th International Conference on Software Engineering* (2018).

[3] Chiş, A., Gîrba, T., and Nierstrasz, O. The Moldable Debugger: A framework for developing domain-specific debuggers. In *Software Language Engineering* (2014), Springer, pp. 102–121.

[4] Costiou, S. *Unanticipated behavior adaptation : application to the debugging of running programs.* Theses, Université de Bretagne occidentale - Brest, Nov. 2018.

[5] Dupriez, T., Polito, G., Costiou, S., Aranega, V., and Ducasse, S. Sindarin: A versatile scripting api for the pharo debugger. In *DLS'19, Dynamic Language Symposium* (2019).

[6] Dupriez, T., Polito, G., and Ducasse, S. Analysis and exploration for new generation debuggers. In *Proceedings of the 12th Edition of the International Workshop on Smalltalk Technologies* (New York, NY, USA, 2017), IWST '17, ACM, pp. 5:1–5:6.

[7] Eisenstadt, M. My hairiest bug war stories. *Commun. ACM 40*, 4 (1997), 30–37.

[8] Lieberman, H. Introduction. *Commun. ACM 40*, 4 (Apr. 1997), 26–29.

[9] Lopez, C. T., Singh, R. G., Marr, S., Boix, E. G., and Scholliers, C. Multiverse debugging: Non-deterministic debugging for non-deterministic programs. In *33rd European Conference on Object-Oriented Programming* (2019).

[10] Marr, S., Lopez, C., Aumayr, D., Gonzalez Boix, E., and Mossenbock, H. Kompos: A platform for debugging complex concurrent applications. In *Programming'17* (apr 2017), pp. 1–2.

[11] Perscheid, M., Siegmund, B., Taeumel, M., and Hirschfeld, R. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal 25*, 1 (2017), 83–110.

[12] Raymond, E. S., and Steele, G. L. *The new hacker's dictionary.* Mit Press, 1996.

[13] Smith, R. B., Wolczko, M., and Ungar, D. From kansas to oz: collaborative debugging when a shared world breaks. *Commun. ACM 40*, 4 (Apr. 1997), 72–78.

[14] Spinellis, D. Modern debugging: The art of finding a needle in a haystack. *Commun. ACM 61*, 11 (Oct. 2018), 124–134.

[15] Telles, M., and Hsieh, Y. *The science of debugging.* Coriolis Group Books, 2001.

[16] Yin, H. *Defusing the Debugging Scandal - Dedicated Debugging Technologies for Advanced Dispatching Languages.* PhD thesis, University of Twente, Dec. 2013.

[17] Zeller, A. *Why programs fail: a guide to systematic debugging.* Elsevier, 2009.