An Artificial Intelligence Paradigm for Troubleshooting Software Bugs

December 24, 2017

Abstract

Troubleshooting is an important part of software development. It starts when a bug is detected, e.g., when testing the system, and ends when the relevant source code is fixed. Key actors in this process are the *tester*, who runs the tests, and the *developer*, who writes the program and is (hopefully) able to fix bugs (i.e., to debug them). In modern software engineering, the interaction between tester and developer during troubleshooting is usually as follows. First, the tester executes a suite of tests and finds a bug. The tester then files a *bug report*, usually in some *issue tracking system* such as Bugzilla. Later, the bug report is assigned to a developer, who is tasked to fix it. This usually means first isolating the bug to find its root cause – the faulty software module that caused the bug – and then fixing it. The fixed software components are then *committed* to a version control system such as Git so that the fix will be made available to the rest of the development team and eventually in the next software version deployed. Figure 1 provides a visual illustration of this process.

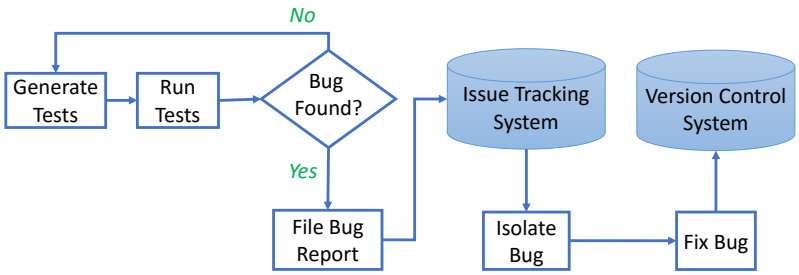

Figure 1: An illustration of the standard workflow for troubleshooting software in current software companies.

This process is known to be very costly. One reason for this is that it is often difficult for the developer to reproduce the bug observed by the tester and fix it, because the developer runs on a different machine

than the tester and in a different context. Another reason is that programs are often large and complex.

To reduce the costs of software troubleshooting we propose a novel paradigm for software troubleshooting that incorporates a helpful intelligent software agent. This agent employs a range of techniques from the Artificial Intelligence (AI) literature to improve detection and isolation of software bugs. (1) First, it learns which source files are likely to fail by analyzing the source-code structure, revisions history and past failures. (2) Then, it plans a test suit such that the software components have a higher likelihood to have a bug will be extensively tested. (3) When a test fails, a diagnosis (DX) algorithm considers the observed tests (failed and passed), as well as knowledge learned from past data, to suggest possible diagnoses and estimate their likelihoods. (4) If further tests are needed to isolate the faulty software components, then the AI agent automatically plans a minimal set of additional tests. After executing these tests, additional diagnostic information is added and fed to the DX algorithm. This iterative process continues until a sufficiently accurate diagnosis is found. (5) Lastly, an automatic genetic algorithm will try fixing the bug. We call this AI-integrated paradigm for software troubleshooting the *Learn, Test, Diagnose, Plan and Fix (LTDPF)* paradigm. This process is illustrated in Figure 2.

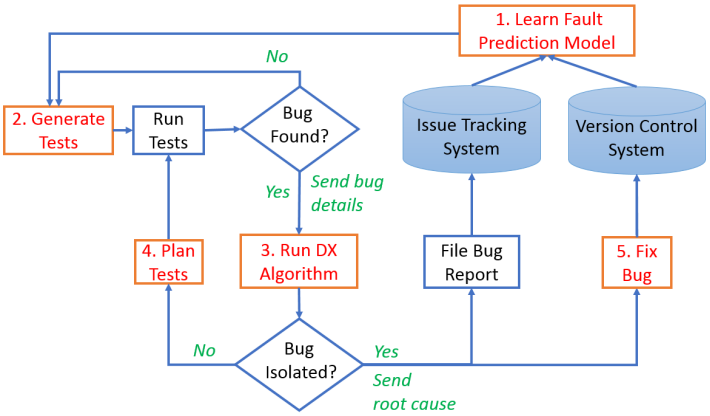

Figure 2: An illustration of the workflow with LTDPS.

LTDPF utilizes four AI techniques: **machine learning**, **diagnosis**, **planning** and **genetic programming**. Using these techniques we propose to develop the five components of LTDPF paradigm. Some of these components have been studied individually [1, 2, 3, 4, 5], but in this proposal we aim to research and develop each one of the components and integrate them into the modern standard troubleshooting process, exploiting data generated by industry-standard software engineering tools. Moreover, we aim to modify these AI components so that they can affect and take advantage of the other AI components, resulting in an effective synergy between them.

References

[1] W. E. Wong, R. Gao, Y. Li, R. Abreu, F. Wotawa, A survey on software fault localization, IEEE Transactions on Software Engineering 42 (8) (2016) 707–740.

[2] N. Cardoso, R. Abreu, Enhancing reasoning approaches to diagnose functional and non-functional errors, in: the International Workshop on Principles of Diagnosis (DX), 2014.

[3] T. Zamir, R. Stern, M. Kalech, Using model-based diagnosis to improve software testing, in: AAAI Conference on Artificial Intelligence, 2014.

[4] A. Elmishali, R. Stern, M. Kalech, Data-augmented software diagnosis., in: AAAI, 2016, pp. 4003–4009.

[5] D. Radjenovic, M. Hericko, R. Torkar, A. Zivkovic, Software fault prediction metrics: A systematic literature review, Information & Software Technology 55 (8) (2013) 1397–1418.