Engineering Applications of Artificial Intelligence 69 (2018) 147–156

Contents lists available at ScienceDirect

Engineering Applications of Artificial Intelligence

journal homepage: www.elsevier.com/locate/engappai

An Artificial Intelligence paradigm for troubleshooting software bugs

Amir Elmishali, Roni Stern, Meir Kalech *

Ben Gurion University of the Negev, Department of Software and Information Systems Engineering, Israel

ARTICLE INFO

Keywords:
Artificial Intelligence
Automated diagnosis
Automated troubleshooting
Software engineering
Software fault prediction

ABSTRACT

Software bugs are prevalent and fixing them is time consuming, and therefore troubleshooting is an important part of software engineering. This paper presents a novel paradigm for incorporating Artificial Intelligence (AI) in the modern software troubleshooting process that can drastically reduce troubleshooting costs. In this paradigm, which we call Learn, Diagnose, and Plan (LDP), we integrate three AI technologies: (1) **machine learning:** learning from source-code structure, revisions history and past failures, which software components are more likely to contain bugs, (2) **automated diagnosis:** identifying the software components that need to be modified in order to fix an observed bug, and (3) **automated planning:** planning additional tests when such are needed to improve diagnostic accuracy. Importantly, these AI technologies are integrated in LDP in a synergistic manner: the diagnosis algorithm is modified to consider the learned fault predictions and the planner is modified to consider the possible diagnoses outputted by the diagnosis algorithm. The overall solution is demonstrated on real faults observed in four open source software projects.

1. Introduction

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. Fig. 1 provides a visual illustration of this process.

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, 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 present 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. First, it learns which source files are likely to fail by analyzing the source-code structure, revisions history and past failures. Then, 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. Lastly, 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. We call this AI-integrated paradigm for software troubleshooting the *Learn, Diagnose, and Plan (LDP)* paradigm. This process is illustrated in Fig. 2.

LDP consists of three AI components: a **learning algorithm**, a **diagnosis algorithm**, and a **planning algorithm**. Each of these components have been studied individually (Cardoso and Abreu, 2014; Zamir et al., 2014; Elmishali et al., 2016; Radjenovic et al., 2013), but in this work we show how they can be integrated into the modern standard troubleshooting process, exploiting data generated by industry-standard software engineering tools. Moreover, we show how 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.

* Corresponding author.
E-mail addresses: amirelm@bgu.ac.il (A. Elmishali), sternron@bgu.ac.il (R. Stern), kalech@bgu.ac.il (M. Kalech).

https://doi.org/10.1016/j.engappai.2017.12.011
Received 25 February 2017; Received in revised form 5 November 2017; Accepted 18 December 2017
Availableonline 12 January 2018
0952-1976/

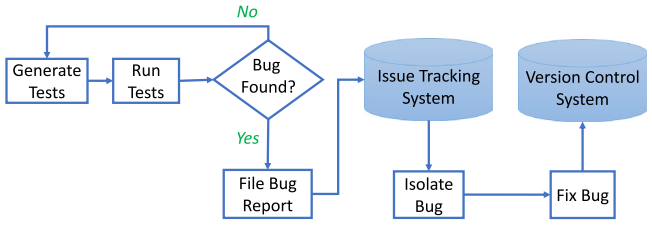

Fig. 1. An illustration of the standard workflow for troubleshooting software in current software companies.

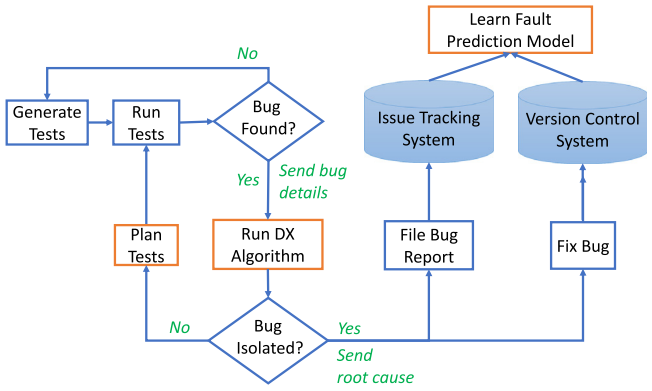

Fig. 2. An illustration of the workflow with LDP.

We implemented all LDP components and evaluated them on several open source software projects: Eclipse CDT (Eclipse CDT), Apache Ant (Apache Ant), Apache POI (Apache POI), and OrientDB (OrientDB). The results demonstrate that LDP is feasible and effective. Moreover, the results show that the performance of the individual AI components of LDP (fault prediction, DX algorithm, and planner) is improved when they are integrated together as we propose.

To summarize, the contributions of this paper are as follows. First, we propose the LDP paradigm. Second, we explain how each of the AI components of LDP can be implemented, and how they can be integrated together effectively. Third, we demonstrate the applicability of LDP in practice on real open-source projects with real bugs.

This paper is structured as follows. Section 2 describes the proposed LDP paradigm, where its subsections Sections 2.1–2.3) describe the AI components of LDP and how they integrate. Then, we describe how we implemented LDP and present the results of our empirical study (Section 3). Section 4 presents related work. Finally, Section 5 concludes with directions for future work.

2. Learn, diagnose, and plan (LDP)

LDP is a paradigm that involves both human and AI elements. The humans in the loop have one of the following roles.

- **Tester.** The tester is the one that observed and reported the abnormal system behavior. Under this definition, a tester can be a human quality assurance (QA) professional, a user of the system, or even an automated testing script.
- **Developer.** This is the person(s) in charge of developing the software. Its role in LDP is to fix given software components (file/method/line of code) that were identified as faulty.

The AI elements of LDP are:

- **Fault predictor.** This is an algorithm or a model that estimates the probability of each software component to be faulty. Importantly, the fault predictor does not consider the observed abnormal behavior of the system. This is important as the probabilities generated by the fault predictor will be used as priors for the diagnoser (this is explained later in greater detail). The fault predictor can consider all data that does not include the current observations, such as historical data – past software versions and bugs – and static code analysis, e.g., various code complexity measures.
- **Diagnoser.** This is an algorithm that accepts as input the observed abnormal behavior of the system as reported by the tester, and outputs one or more possible explanations for that behavior. Each of these explanations, referred to as *diagnoses*, is an assumption that a specific set of software components is faulty. The diagnoser is expected to output for every diagnosis an estimate of the probability that it is correct.
- **Test planner.** This is an algorithm that accepts the set of diagnoses outputted by the diagnoser, and, if needed, suggests additional tests that the tester should perform in order to find the correct diagnosis, i.e., the software components that caused the abnormal system behavior.

LDP is intended to help the developer to debug an observed bug by automatically diagnosing it and, if needed, intelligently guiding a tester to perform additional specially tailored tests to collect further diagnostic information. The first stage in LDP, which is done periodically before a bug is observed, is to learn a fault predictor based on information about the past failures. This is done using standard machine learning techniques. Then, when a bug is observed and reported by the tester, it is inputted to the diagnoser that outputs a set of possible diagnoses. The diagnoser does this by applying an automated diagnosis algorithm that considers both the observed system behavior and the probabilities generated by the fault predictor. If a single diagnosis is found whose probability of being correct is high enough (how much is "high enough" is a parameter), then this diagnosis is passed to the developer for fixing. If not, then the test planner proposes an additional test to be performed to the tester to narrow down the set of possible diagnoses. This initiates an iterative process in which the test planner plans an additional test, the tester performs it, and the diagnoser recomputes the set of possible diagnoses and their likelihoods. This process stops when a diagnosis is found whose probability of being correct is high enough. At this stage a bug report is added to the issue tracking system and the diagnosed bug is given to a developer to fix the (now isolated) bug. This entire process is illustrated in Fig. 2.

Next, we describe how the AI components of LDP – fault predictor, diagnoser, and test planner – are implemented in practice.

2.1. Software fault prediction

Fault prediction in software is a classification problem. Given a software component, the goal is to determine its class – healthy or faulty. Supervised machine learning algorithms are commonly used to solve classification problems. They work as follows. As input, they are given a set of *labeled instances*, which are pairs of instances and their correct labeling, i.e., the correct class for each instance. In our case, instances are software components and the labels are which software component is healthy and which is not. They output a *classification model*, which maps an (unlabeled) instance to a class. This set of labeled instances is called the *training set* and the act of generating a classification model from the training set if referred to as *learning*.

Learning algorithms extract *features* from a given instance, and try to learn from the training set the relation between the features of an instance and its class. A key to the success of machine learning algorithms is the choice of *features* used. Many possible features were proposed in the literature for software fault prediction.

Radjenovic et al. (2013) surveyed the features used by existing software prediction algorithms and categorized them into three families:

- **Traditional.** These features are traditional software complexity metrics, such as number of lines of code or more sophisticated