Automatic Software Repair: A Survey

Luca Gazzola, Daniela Micucci, *Member, IEEE*, and Leonardo Mariani, *Senior Member, IEEE*

Abstract—Despite their growing complexity and increasing size, modern software applications must satisfy strict release requirements that impose short bug fixing and maintenance cycles, putting significant pressure on developers who are responsible for timely producing high-quality software. To reduce developers workload, repairing and healing techniques have been extensively investigated as solutions for efficiently repairing and maintaining software in the last few years. In particular, *repairing solutions* have been able to automatically produce useful fixes for several classes of bugs that might be present in software programs. A range of algorithms, techniques, and heuristics have been integrated, experimented, and studied, producing a heterogeneous and articulated research framework where automatic repair techniques are proliferating. This paper organizes the knowledge in the area by surveying a body of 108 papers about automatic software repair techniques, illustrating the algorithms and the approaches, comparing them on representative examples, and discussing the open challenges and the empirical evidence reported so far.

Index Terms—Automatic program repair, generate and validate, search-based, semantics-driven repair, correct by construction, program synthesis, self-repairing

1 Introduction

Debugging software failures is still a painful, time consuming, and expensive process. For instance, recent studies showed that debugging activities often account for about 50 percent of the overall development cost of software products [1], [2].

There are many factors contributing to the cost of debugging, but the most impacting one is the extensive manual effort that is still required to identify and remove faults. In particular, the debugging process requires analyzing and understanding failed executions, identifying the causes of the failures, implementing fixes, and validating that the fixed program works correctly, that is, the problem has been fixed without introducing any side effect [3], [4], [5]. Most of these activities are executed manually or with partial tool support.

So far, the automation of debugging activities essentially concerned with the identification of the statements that are

Techniques that identify specific inputs and specific states that may trigger failures are often based on the well-known Delta Debugging technique [9]. The intuition is that by iteratively refining and reducing the input [10] and the state space [11], it should be possible to identify the smallest input and the smallest portion of the application state that cannot be eliminated to observe the failure. This information is relevant to understand the conditions that may trigger failures.

Finally, anomaly detection techniques can detect the operations that are executed by an application during failures but not during correct executions. These operations may explain why and how an application failed. Anomaly detection techniques usually exploit specification mining approaches [16], [17], [18], [19] to automatically generate models that represent the legal behavior of an application, and then use these models to analyze the failed executions