Software Testing

Gordon Fraser and José Miguel Rojas

Abstract Any nontrivial program contains some errors in the source code. These "bugs" are annoying for users if they lead to application crashes and data loss, and they are worrisome if they lead to privacy leaks and security exploits. The economic damage caused by software bugs can be huge, and when software controls safety critical systems such as automotive software, then bugs can kill people. The primary tool to reveal and eliminate bugs is software testing: Testing a program means executing it with a selected set of inputs and checking whether the program behaves in the expected way; if it does not, then a bug has been detected. The aim of testing is to find as many bugs as possible, but it is a difficult task as it is impossible to run *all* possible tests on a program. The challenge of being a good tester is thus to identify which are the best tests that help us find bugs, and to execute them as efficiently as possible. In this chapter, we explore different ways to measure how "good" a set of tests is, as well as techniques to generate good sets of tests.

All authors have contributed equally to this chapter.

G. Fraser
University of Passau, Passau, Germany
e-mail: gordon.fraser@uni-passau.de

J. M. Rojas
University of Leicester, Leicester, UK
e-mail: j.rojas@leicester.ac.uk

S. Cha et al. (eds.), *Handbook of Software Engineering*,
https://doi.org/10.1007/978-3-030-00262-6_4

1 Introduction

Software has bugs. This is unavoidable: Code is written by humans, and humans make mistakes. Requirements can be ambiguous or wrong, requirements can be misunderstood, software components can be misused, developers can make mistakes when writing code, and even code that was once working may no longer be correct when once previously valid assumptions no longer hold after changes. Software testing is an intuitive response to this problem: We build a system, then we run the system, and check if it is working as we expected.

While the principle is easy, testing *well* is not so easy. One main reason for this is the sheer number of possible tests that exists for any nontrivial system. Even a simple system that takes a single 32 bit integer number as input already has 2^{32} possible tests. Which of these do we need to execute in order to ensure that we find all bugs? Unfortunately, the answer to this is that we would need to execute *all* of them—only if we execute all of the inputs and observe that the system produced the expected outputs do we know for sure that there are no bugs. It is easy to see that executing all tests simply does not scale. This fact is well captured by Dijkstra's famous observation that testing can never prove the absence of bugs, it can only show the presence of bugs. The challenge thus lies in finding a subset of the possible inputs to a system that will give us reasonable certainty that we have found the main bugs.

If we do not select good tests, then the consequences can range from mild user annoyance to catastrophic effects for users. History offers many famous examples of the consequences of software bugs, and with increased dependence on software systems in our daily lives, we can expect that the influence of software bugs will only increase. Some of the most infamous examples where software bugs caused problems are the Therac-25 radiation therapy device (Leveson and Turner 1993), which gave six patients a radiation overdose, resulting in serious injury and death; the Ariane 5 maiden flight (Dowson 1997), which exploded 40 s after lift-off because reused software from the Ariane 4 system had not been tested sufficiently on the new hardware, or the unintended acceleration problem in Toyota cars (Kane et al. 2010), leading to fatal accidents and huge economic impact.

In this chapter, we will explore different approaches that address the problem of how to select good tests. Which one of them is best suited will depend on many different factors: What sources of information are available for test generation? Generally, the more information about the system we have, the better we can guide the selection of tests. In the best case, we have the source code at hand while selecting tests—this is known as *white box* testing. However, we don't always have access to the full source code, for example, when the program under test accesses web services. Indeed, as we will see there can be scenarios where we will want to guide testing not (only) by the source code, but by its intended functionality as captured by a specification—this is known as *black box* testing. We may even face a scenario where we have neither source code, nor specification of the system, and even for this case we will see techniques that help us selecting tests.