Chapter 33
Cloud-Enabled Software Testing Based on Program Understanding

Chia-Chu Chiang
University of Arkansas at Little Rock, USA

Shucheng Yu
University of Arkansas at Little Rock, USA

ABSTRACT

Cloud computing provides an innovative technology that enables Software as a Service (SaaS) to its customers. With cloud computing technologies, a suite of program understanding tools is suggested to be deployed in a cloud to aid the generation of test cases for software testing. This cloud-enabled service allows customers to use these tools through an on-demand, flexible, and pay-per-use model. Lastly, the issues and challenges of cloud computing are presented.

INTRODUCTION

The quality of software testing lies in the use of effective test cases. Creating effective test cases requires the in-depth study of the application for which test cases are being generated. In this chapter, a suite of tools developed to aid the understanding of programs is suggested to be hosted on a cloud with the innovative technology of cloud computing. This suite of tools on the cloud provides a cloud-enabled service to customers without the need to purchase the tools. Most importantly, customers can use the service through the on-demand, flexible, and pay-per-use model. The purpose of this chapter is to propose a technical solution for designing a cloud-based environment that enables detailed understanding of the programs in the portfolio and further aids the generation of test cases for software testing.

This chapter is organized as follows: First, the needs of program understanding for software testing are explained. Following that, reverse engineering techniques for program comprehension are briefly overviewed. A suite of automated program understanding tools is presented. Then, a cloud-based platform with its implementation

DOI: 10.4018/978-1-4666-6539-2.ch033

alternatives for hosting the tools is given. Later, the issues and challenges of cloud computing are presented. Customers should be aware that cloud computing has limitations. They should carefully evaluate these trade-offs before making decisions to migrate their proprietary programs into cloud computing.

BACKGROUND ON PROGRAM COMPREHENSION

Software testing serves to discover defects in the software where the outputs do not meet the expected results or fail to meet the user requirements. One essential activity of software testing is creating test cases. Two major software testing techniques that include white-box and black-box testing are used to generate test cases. The white-box testing generates the test cases from the programs. The black-box testing generates the test cases from the specifications and designs. Test cases are not created from the source code. For both software testing techniques, program (specification) understanding is a key activity in the testing process. Programmers can understand a program by reading its documentation (such as specification and design documents), reading the source code, or just executing the program. Reading the documents can either be effective or misleading. Unfortunately, one of the major problems of documentation is keeping the documents current to reflect the changes of the code. If the documents are outdated, reading the documentation for program understanding is not a good idea. Executing a program to understand the program's dynamic behavior can dramatically improve understanding which cannot be assimilated from reading the source code alone. However, the knowledge of the program mainly lies in the source code. Currently, programmers still count on reading the source code to gain knowledge about a program. Several reverse engineering techniques and tools have been developed to help automate knowledge extraction from the programs used for program understanding (Bellay & Gall, 1997; Chikofsky & Cross, 1990; Gannod & Cheng, 1999; Zvegintzov, 1997). The techniques of reverse engineering are applied to automate program understanding by analyzing a system (1) to identify the system's components, their interrelationships, and (2) to create representations of the system in another form or at a higher level of abstraction (Chikofsky & Cross, 1990). The outputs of reverse engineering include structure chart, module calling hierarchy, data dictionary, data flow dependence, and control flow dependence. The knowledge about the programs is embedded in these artifacts. Programmers gain the understanding of programs by reading these artifacts.

Existing reverse engineering techniques mainly work on code for white-box testing. These reverse engineering techniques can be classified into two categories: static and dynamic analyses. Static analysis relies on code and its associated documentation that can be broken down into formal and informal methods. A formal method generates a formal specification in mathematical notations such as logic, set, and axioms from source code (Gannod & Cheng, 1996). An informal method for reverse engineering can be classified into two categories: plan-based and parsing-based approaches. The plan-based approaches automate the recognition of abstract concepts in source code using a library of programming plan templates and concepts with top-down and bottom-up search strategies (Abd-El-Hafiz & Basili, 1996; Woods & Yang, 1996). To date, industrial adoption of formal methods is still limited. In addition, many legacy systems have the problems of delocalized plans caused by enhancements and patches as the systems evolve (Bennett, 1995; Rugaber, Stirewalt, & Wills, 1995). Therefore, most plan-based approaches are limited as well. The parsing-based approach is usually applied to most programming languages. A parsing-based approach analyzes the