ABSTRACT

Unit testing has been widely recognized as an important and valuable means of improving software reliability, as it exposes bugs early in the software development life cycle. However, manual unit testing is often tedious and insufficient. Testing tools can be used to enable economical use of resources by reducing manual effort. Recently parameterized unit testing has emerged as a very promising and effective methodology to allow the separation of two testing concerns or tasks: the specification of external, black-box behavior (i.e., assertions or specifications) by developers and the generation and selection of internal, white-box test inputs (i.e., high-code-covering test inputs) by tools. A parameterized unit test (PUT) is simply a test method that takes parameters, calls the code under test, and states assertions. PUTs have been supported by various testing frameworks. Various open source and industrial testing tools also exist to generate test inputs for PUTs.

This tutorial presents latest research on principles and techniques, as well as practical considerations to apply parameterized unit testing on real-world programs, highlighting success stories, research and education achievements, and future research directions in developer testing. The tutorial will help improve developer skills and knowledge for writing PUTs and give overview of tool automation in supporting PUTs. Attendees will acquire the skills and knowledge needed to perform research or conduct practice in the field of developer testing and to integrate developer testing techniques in their own research, practice, and education.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—Validation; D.2.5 [Software Engineering]: Testing and Debugging—Symbolic execution, Testing tools

General Terms

Reliability, Verification

Keywords

Testing, unit testing, parameterized unit testing, theories, symbolic execution, mock objects, Pex

1. INTRODUCTION

A unit test is simply a method without parameters that performs a sequence of method calls that exercise the code under test and asserts properties of the code’s expected behavior. Unit tests are a key component of software engineering. The Extreme Programming discipline [1], for instance, leverages them to permit easy code changes. Being of such importance, many companies now provide tools, frameworks, and services around unit tests. Tools range from specialized testing frameworks to automatic unit-test generation tools. However, these tools do not provide any guidance for

• how to write code that can be tested in isolation,
• which tests should be written (for internal and for external behavior),
• how to come up with a minimal number of test cases, and
• what guarantees that the test cases provide.

In order to address these issues and give guidance, this tutorial teaches the notion of parameterized unit testing, test input generation by automated program analysis, and environment isolation by manual refactoring or automated code instrumentation.

More information about this tutorial can be found on the tutorial web:
https://sites.google.com/site/asergrp/put

2. PARAMETERIZED UNIT TESTING

Parameterized unit testing [10, 7] is a new methodology extending the current industry practice based on closed, traditional unit tests (i.e., test methods without input parameters). Test methods are generalized by allowing parameters. This generalization serves two main purposes. First, parameterized test methods are specifications of the behavior of the methods under test: they not only provide exemplary arguments to the methods under test, but ranges of such arguments. Second, parameterized unit tests describe a set of traditional unit tests that can be obtained by instantiating the parameterized test methods with given argument sets.
3. TEST GENERATION AND SELECTION

Instantiations should be chosen so that they exercise different code paths of the methods under test. Dynamic symbolic execution (DSE) \([4, 3]\) is a technique that combines static and dynamic analysis to automatically generate test inputs. Given a program that takes inputs, the goal of DSE is to generate test inputs that, upon execution of the program, will exercise as many reachable statements as possible. DSE combines concepts of symbolic execution, model checking, testing, and constraint solving. DSE is based on observing actual executions of the program under test. By leveraging observed concrete input/output values, DSE can simply concretize those operations that interact with the environment, or that are difficult to reason about (e.g., floating point arithmetic), while previous approaches based on symbolic execution and model checking would lose precision. Many implementations of DSE exist, ranging from academic open-source projects to industrial tools. They differ in their target environments (C programs, Java programs, .NET programs, x86 instructions, etc.) and their constraint solving abilities (programs with or without pointers, bitvector arithmetic or linear arithmetic, etc.). The tutorial uses the Pex tool \([6, 9]\).

4. ENVIRONMENT ISOLATION

It is desirable to test individual software components in isolation. It makes testing more robust and scalable. Especially in the context of unit testing, where the intention is to test a single unit of functionality, all irrelevant environment dependencies should be mocked \([11, 5]\), or simulated, so that the unit tests run quickly and give deterministic results. (In contrast, integration testing’s goal is to test an integrated (sub)system, including all environment dependencies, at the same time. Integration tests are usually neither quick, nor entirely deterministic.)

Ideally, the code should be written in a way that allows to substitute its constituent components at test time, in order to isolate a feature under test. In other words, it should be possible to treat all components as test parameters, so that mocked implementations or simulations can be used to instantiate parameterized tests.

The best solution to the problem is to refactor the code \([2]\), introducing explicit interface boundaries and allowing different interface implementations. When refactoring is not an option, e.g., when dealing with legacy code, other approaches can be used to detour environment-facing calls at test time. A common approach for detouring is code instrumentation.

5. LEARNING OBJECTIVES

Admittedly, writing open, parameterized unit tests is more challenging than writing closed, traditional unit tests. This tutorial serves as a starting point for familiarizing the attendees with principles, techniques, and applications of parameterized unit testing. This tutorial will greatly benefit from the increasing importance and popularity of the developer testing field and will help increase awareness of this important and promising field. The attendees will gain the skills and knowledge needed to conduct research and/or practice on developer testing, especially in the form of parameterized unit testing. In particular, the following skills and knowledge of conducting developer testing will be gained by the attendees and teaching these skills and knowledge will be gained by attendees with educational interests.

- Write parameterized unit tests
- Encapsulate or redirect relevant environment interactions
- Leverage dynamic symbolic execution tools to automatically generate test inputs \([12, 8]\)
- Analyze testing results
- Interact with tools to give guidance to tools
- Understand the limitations of the approach
- Gain skills and knowledge of developing new research techniques and tools in the area of developer testing

Acknowledgment

Tao Xie’s work is supported in part by NSF grants CCF-0725190, CCF-0845272, and CCF-0915400, and ARO grant W911NF-08-1-0443, and ARO grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI).

6. REFERENCES