Research Papers Assignment

Andrea Polini

Software Engineering II – Software Testing
MSc in Computer Science
University of Camerino

Papers List

1. Testing environment for CPS by cooperating model checking with execution testing
2. Conformance Testing for Cyber-Physical Systems
3. Automatically Discovering, Reporting and Reproducing Android Application Crashes
4. Systematic Execution of Android Test Suites in Adverse Conditions
5. Sapienz: Multi-objective Automated Testing for Android Applications
6. Predicting Testability of Concurrent Programs
7. How well are your requirements tested?
8. Automatic Generation of Oracles Exceptional Behaviors

1. Testing environment for CPS by cooperating model checking with execution testing

Abstract

In this study, we propose a testing environment for cyber-physical systems (CPS). In system testing for CPS, many tests are difficult to design or implement because of these systems' many product variations. The proposed environment executes the tests and guarantees that these systems operate reliably using two methods. The first method provides easy management of test cases by managing functions to be tested and configurations to be tested separately. The second method involves automatic testing of real devices based on model checking technologies. The authors have developed a horizontal prototype of the proposed environment and confirmed its feasibility and applicability.

2. Conformance Testing for Cyber-Physical Systems

Abstract

Cyber-Physical Systems (CPS) require a high degree of reliability and robustness. Hence it is important to assert their correctness with respect to extra-functional properties, like power consumption, temperature, etc. In turn the physical quantities may be exploited for assessing system implementations. This article develops a methodology for utilizing measurements of physical quantities for testing the conformance of a running CPS with respect to a formal description of its required behavior allowing to uncover defects. We present foundations and implementations of this approach and demonstrate its usefulness by conformance testing power measurements of a wireless sensor node with a formal model of its power consumption.

3. Automatically Discovering, Reporting and Reproducing Android Application Crashes

Abstract

Mobile developers face unique challenges when detecting and reporting crashes in apps due to their prevailing GUI event-driven nature and additional sources of inputs (e.g., sensor readings). To support developers in these tasks, we introduce a novel, automated approach called CRASHSCOPE . This tool explores a given Android app using systematic input generation, according to several strategies informed by static and dynamic analyses, with the intrinsic goal of triggering crashes. When a crash is detected, CRASHSCOPE generates an augmented crash report containing screenshots, detailed crash reproduction steps, the captured exception stack trace, and a fully replayable script that automatically reproduces the crash on a target device(s). We evaluated CRASHSCOPE's effectiveness in discovering crashes as compared to five state-of-the-art Android input generation tools on 61 applications. The results demonstrate that CRASHSCOPE performs about as well as current tools for detecting crashes and provides more detailed fault information. Additionally, in a study analyzing eight real-world Android app crashes, we found that CRASHSCOPE's reports are easily readable and allow for reliable reproduction of crashes by presenting more explicit information than human written reports.

4. Systematic Execution of Android Test Suites in Adverse Conditions

Abstract

Event-driven applications, such as, mobile apps, are difficult to test thoroughly. The application programmers often put significant effort into writing end-to-end test suites. Even though such tests often have high coverage of the source code, we find that they often focus on the expected behavior, not on occurrences of unusual events. On the other hand, automated testing tools may be capable of exploring the state space more systematically, but this is mostly without knowledge of the intended behavior of the individual applications. As a consequence, many programming errors remain unnoticed until they are encountered by the users. We propose a new methodology for testing by leveraging existing test suites such that each test case is systematically exposed to adverse conditions where certain unexpected events may interfere with the execution. In this way, we explore the interesting execution paths and take advantage of the assertions in the manually written test suite, while ensuring that the injected events do not affect the expected outcome. The main challenge that we address is how to accomplish this systematically and efficiently. We have evaluated the approach by implementing a tool, Thor, working on Android. The results on four real-world apps with existing test suites demonstrate that apps are often fragile with respect to certain unexpected events and that our methodology effectively increases the testing quality:

5. Sapienz: Multi-objective Automated Testing for Android Applications

Abstract

We introduce Sapienz, an approach to Android testing that uses multi-objective search-based testing to automatically explore and optimise test sequences, minimising length, while simultaneously maximising coverage and fault revelation. Sapienz combines random fuzzing, systematic and search-based exploration, exploiting seeding and multi-level instrumentation. Sapienz significantly outperforms (with large effect size) both the state-of-the-art technique Dynodroid and the widely-used tool, Android Monkey, in 7/10 experiments for coverage, 7/10 for fault detection and 10/10 for fault-revealing sequence length. When applied to the top 1,000 Google Play apps, Sapienz found 558 unique, previously unknown crashes. So far we have managed to make contact with the developers of 27 crashing apps. Of these, 14 have confirmed that the crashes are caused by real faults. Of those 14, six already have developer-confirmed fixes.

6. Predicting Testability of Concurrent Programs

Abstract

Concurrent programs are difficult to test due to their inherent non-determinism. To address the nondetermin- ism problem, testing often requires the exploration of thread schedules of a program; this can be time-consuming for testing real-world programs. We believe that testing resources can be distributed more effectively if testability of concurrent programs can be estimated, so that developers can focus on exploring the low testable code. Voas introduces a notion of testability as the probability that a test case will fail if the program has a fault, in which testability can be measured based on fault-based testing and mutation analysis. Much research has been proposed to analyze testability and predict defects for sequential programs, but to date, no work has considered testability prediction for concurrent programs, with program characteristics distinguished from sequential programs. In this paper, we present an approach to predict testability of concurrent programs at the function level. We propose a set of novel static code metrics based on the unique properties of concurrent programs. To evaluate the performance of our approach, we build a family of testability prediction models combining both static metrics and a test suite metric and apply it to real projects. Our empirical study reveals that our approach is more accurate than existing sequential program metrics.

7. How well are your requirements tested?

Abstract

We address the question: to what extent does covering requirements ensure that a test suite is effective at revealing faults? To answer it, we generate minimal test suites that cover all requirements, and assess the tests they contain. They turn out to be very poor-ultimately because the notion of covering a requirement is more subtle than it appears to be at first. We propose several improvements to requirements tracking during testing, which enable us to generate minimal test suites close to what a human developer would write. However, there remains a class of plausible bugs which such suites are very poor at finding, but which random testing finds rather easily.

8. Automatic Generation of Oracles Exceptional Behaviors

Abstract

Test suites should test exceptional behavior to detect faults in error-handling code. However, manually-written test suites tend to neglect exceptional behavior. Automatically-generated test suites, on the other hand, lack test oracles that verify whether runtime exceptions are the expected behavior of the code under test. This paper proposes a technique that automatically creates test oracles for exceptional behaviors from Javadoc comments. The technique uses a combination of natural language processing and run-time instrumentation. Our implementation, Toradocu, can be combined with a test input generation tool. Our experimental evaluation shows that Toradocu improves the fault-finding effectiveness of EvoSuite and Randoop test suites by 8% and 16% respectively, and reduces EvoSuite's false positives by 33%.

Organizing the presentation

- Which is the problem?
- Why the problem is indeed a problem?
- Which is the proposed solution?
- Why the solution is a solution?
- Which are other approaches to solve the problem, how they relate to the presented one?

- The presentation should last 30 minutes

Organizing the presentation

- Which is the problem?
- Why the problem is indeed a problem?
- Which is the proposed solution?
- Why the solution is a solution?
- Which are other approaches to solve the problem, how they relate to the presented one?

- The presentation should last 30 minutes