Research Papers Assignment

Andrea Polini

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

Papers List

1. Test Coverage of Data-Centric Dynamic Compositions in Service-Based Systems
2. A Web Service Test Generator
3. Using Rules for Web Service Client Side Testing
4. Test Case Prioritization for Audit Testing of Evolving Web Services using Information Retrieval Techniques
5. Server Interface Descriptions for Automated Testing of JavaScript Web Applications
6. Feedback-Directed Instrumentation for Deployed JavaScript Applications
7. Test Generation Using Symbolic Execution
8. CUTE: A Concolic Unit Testing Engine for C
9. SAGE: Whitebox Fuzzing for Security Testing
10. Systematic Execution of Android Test Suites in Adverse Conditions

1. Test Coverage of Data-Centric Dynamic Compositions in Service-Based Systems

Abstract

This paper addresses the problem of integration testing of data-centric dynamic compositions in service-based systems. These compositions define abstract services, which are replaced by invocations to concrete candidate services at runtime. Testing all possible runtime instances of a composition is often unfeasible. We regard data dependencies between services as potential points of failure, and introduce the k-node data flow test coverage metric. Limiting the level of desired coverage helps to significantly reduce the search space of service combinations. We formulate the problem of generating a minimum set of test cases as a combinatorial optimization problem. Based on the formalization we present a mapping of the problem to the data model of FoCuS, a coverage analysis tool developed at IBM. FoCuS can efficiently compute near-optimal solutions, which we then use to automatically generate and execute test instances of the composition. We evaluate our prototype implementation using an illustrative scenario to show the end-to-end practicability of the approach.

2. A Web Service Test Generator

Abstract

An automated process for generating test inputs for web services from a WSDL is presented. A grammatical representation of the web service is extracted from the WSDL and used to produce test cases. A context-free grammar (CFG) is generated from the XSD that is stored in the WSDL. The CFG is provided as input into a constraint-satisfaction problem solver to automatically generate a diverse set of structurally correct XML documents. Testing data is then inserted into the XML templates in accordance with any constraints specified in the XSD. Web service-specific testing can be performed with the inclusion of external datasets and service-specific configurations.

3. Using Rules for Web Service Client Side Testing

Abstract

Web Services (WS) are software components accessible over the Internet through a well-defined set of standards. When consumers invoke a service, they expect to receive a valid response. However, the problem is to determine the structure of a valid request. WS specifications are used to solve this problem since they are considered the primary piece of information for building service requests. Unfortunately, existing specifications do not provide enough support for this type information (e.g., WSDL) or there is little support on the client side (e.g., OWL-S). In this paper we address this issue by implementing a technique to reduce the number of faulty requests. We specifically propose an approach for extending WSDL with service input parameters rules that help consumers and integrators to verify their calls on the client side.

4. Test Case Prioritization for Audit Testing of Evolving Web Services using Information Retrieval Techniques

Abstract

Web services evolve frequently to meet new business demands and opportunities. However, service changes may affect service compositions that are currently consuming the services. Hence, audit testing (a form of regression testing in charge of checking for compatibility issues) is needed. As service compositions are often in continuous operation and the external services have limited (expensive) access when invoked for testing, audit testing has severe time and resources constraints, which make test prioritization a crucial technique (only the highest priority test cases will be executed). This paper presents a novel approach to the prioritization of audit test cases using information retrieval. This approach matches a service change description with the code portions exercised by the relevant test cases. So, test cases are prioritized based on their relevance to the service change. We evaluate the proposed approach on a system that composes services from eBay and Google.

5. Server Interface Descriptions for Automated Testing of JavaScript Web Applications

Abstract

Automated testing of JavaScript web applications is complicated by the communication with servers. Specifically, it is difficult to test the JavaScript code in isolation from the server code and database contents. We present a practical solution to this problem. First, we demonstrate that formal server interface descriptions are useful in automated testing of JavaScript web applications for separating the concerns of the client and the server. Second, to support the con- struction of server interface descriptions for existing applications, we introduce an effective inference technique that learns communi- cation patterns from sample data. By incorporating interface descriptions into the testing tool Artemis, our experimental results show that we increase the level of automation for high-coverage testing on a collection of JavaScript web applications that exchange JSON data between the clients and servers. Moreover, we demonstrate that the inference technique can quickly and accurately learn useful server interface descriptions.

6. Feedback-Directed Instrumentation for Deployed JavaScript Applications

Abstract

Many bugs in JavaScript applications manifest themselves as objects that have incorrect property values when a failure occurs. For this type of error, stack traces and log files are often insufficient for diagnosing problems. In such cases, it is helpful for developers to know the control flow path from the creation of an object to a crashing statement. Such crash paths are useful for understanding where the object originated and whether any properties of the object were corrupted since its creation. We present a feedback-directed instrumentation technique for computing crash paths that allows the instrumentation overhead to be distributed over a crowd of users and to reduce it for users who do not encounter the crash. We implemented our technique in a tool, Crowdie, and evaluated it on 10 real-world issues for which error messages and stack traces are insufficient to isolate the problem. Our results show that feedback-directed instrumentation requires 5% to 25% of the program to be instrumented, that the same crash must be observed 3 to 10 times to discover the crash path, and that feedback-directed instrumentation typically slows down execution by a factor 2x-9x compared to 8x-90x for an approach where applications are fully instrumented.

7. Test Generation Using Symbolic Execution

Abstract

This paper presents a short introduction to automatic code-driven test generation using symbolic execution. It discusses some key technical challenges, solutions and milestones, but is not an exhaustive survey of this research area.

8. CUTE: A Concolic Unit Testing Engine for C

Abstract

In unit testing, a program is decomposed into units which are collections of functions. A part of unit can be tested by generating inputs for a single entry function. The entry function may contain pointer arguments, in which case the inputs to the unit are memory graphs. The paper addresses the problem of automating unit testing with memory graphs as inputs. The approach used builds on previous work combining symbolic and concrete execution, and more specifically, using such a combination to generate test inputs to explore all feasible execution paths. The current work develops a method to represent and track constraints that capture the behavior of a symbolic execution of a unit with memory graphs as inputs. Moreover, an efficient constraint solver is proposed to facilitate incremental generation of such test inputs. Finally, CUTE, a tool implementing the method is described together with the results of applying CUTE to real-world examples of C code.

9. SAGE: Whitebox Fuzzing for Security Testing

Abstract

“program verification research” as mostly theoretical with little impact on the world at large. Think again. If you are reading these lines on a PC running some form of Windows (like over 93% of PC users-that is, more than one billion people), then you have been affected by this line of work-without knowing it, which is precisely the way we want it to be. Every second Tuesday of every month, also known as “Patch Tuesday,” Microsoft releases a list of security bulletins and associated security patches to be deployed on hundreds of millions of machines worldwide. Each security bulletin costs Microsoft and its users millions of dollars. If a monthly security update costs you $0.001 (one tenth of one cent) in just electricity or loss of productivity, then this number multiplied by one billion people is $1 million. Of course, if malware were spreading on your machine, possibly leaking some of your private data, then that might cost you much more than $0.001. This is why we strongly encourage you to apply those pesky security updates.

10. 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: Of 507 individual tests, 429 fail when exposed to adverse conditions, which reveals 66 distinct problems that are not detected by ordinary execution of the tests.