My main research interests are in Software Engineering and Formal Methods, with a focus on improving software quality via software testing and software model checking. Poor software quality can lead to financial loses and loss of life as demonstrated by numerous recent incidents. To improve software quality, scalable and efficient testing and verification techniques need to be developed. We can develop such techniques only by considering the complexity of practical software systems. My research has been inspired by fundamental challenges that affect practical software systems, and motivated by the slow technology transfer of testing techniques from research to industry.

More specifically, my research results include a number of scalable and efficient techniques for regression testing [7, 10, 13, 21, 22, 24, 25], automated test generation [2, 6, 8, 9, 16], testing concurrent code [3, 4, 12], software model checking [5, 14, 15, 23], and assessing test quality [1, 11, 17, 18, 20, 26]. Several of my tools have been deployed outside of academia, including at Apache and Google. In addition, my tools found hundreds of bugs in large projects including OpenMRS (an open medical record system widely used in developing countries), Oracle Java compiler, and Eclipse IDE.

Regression Testing

Developers often build regression test suites that are automatically run to check that code changes did not break any functionality. Modern software evolves fairly quickly, with changes pushed to repositories as often as several times per minute. Meanwhile, the number of tests in regression test suites also grow and could take days to execute. For example, Google recently reported that they observed quadratic increase in test-suite execution time (linear increase in the number of commits $\times$ linear increase in the number of tests). During my internships at Microsoft and Intel, I also observed a significant negative impact of long test executions on developers’ productivity. In brief, regression testing is becoming more important but also more expensive. To address this issue, I developed techniques [7, 10, 13, 21, 22, 24, 25] to speed up regression testing.

File-based regression test selection. One approach to speed up regression testing is regression test selection (RTS), which runs only a subset of tests that may be affected by the latest changes. To detect affected tests, RTS techniques statically analyze the latest changes to a codebase. To obtain overall time savings, compared to rerunning all the tests, RTS techniques have to balance the time spent on analysis vs. the time saved from not running non-selected tests. If the time to run non-selected tests is shorter than the analysis time, RTS does not provide overall savings. I proposed a new, lightweight RTS technique, called Ekstazi, that provides a “sweet-spot” balancing of the analysis time and time for running non-selected tests. Ekstazi tracks dynamic dependencies of tests on files [23]. Ekstazi guarantees to select tests affected by changes in code, libraries, and external files. I developed Ekstazi (www.ekstazi.org) for Java, and evaluated it on several hundred revisions of 32 open-source projects (totaling 5M lines of code). Ekstazi reduced the overall time 54% compared to running all tests. Ekstazi also has lower overall time, although it selects more tests, than the existing RTS techniques that track dependencies of tests on finer granularities (e.g., methods). Although this result may seem surprising at first, it follows from the fact that code changes are often small relative to the size of codebase. Finally, only a few months after the initial release, Ekstazi was adopted by Apache (including Camel and CXF projects) and used daily by several developers.

Regression testing for distributed software histories. Prior RTS techniques considered two versions of code at a time, effectively assuming a development process where changes to the code occur in a linear sequence. However, modern development processes that use distributed version-control systems are more complex. Their software version histories are generally modeled as directed graphs; in addition to version changes occurring linearly, multiple versions can be related by other commands (e.g., branch, merge, rebase, cherry-pick, revert, etc.). I developed the first RTS technique for software that uses modern distributed version-control systems [21]. By modeling different branch and merge commands directly, my technique computes test sets that can be substantially smaller than applying prior techniques to a linearization of the software history. I evaluated my technique on software histories of several large open-source projects. The results are encouraging: the technique obtained an average of 10.89× reduction in the number of tests over a prior technique, which linearizes a software history, while still selecting all tests whose behavior may differ.
Input Generation

Prior to my work on regression testing, I worked on automated generation of tests, as good tests are necessary to ensure software quality. The goal of my work was to develop an automated technique that is superior to manually written tests; manually writing tests is time consuming and often misses corner cases. My work resulted in UDITA [8], an automated technique that generates tests from a user specification (more details below). Tests generated by UDITA discovered a number of bugs in large systems, and the UDITA paper won an ACM SIGSOFT Distinguished Paper Award at ICSE 2010. My follow-up work examined existing software repositories to improve test input generation for refactoring engines [22], and discovered hundreds of bugs in refactoring engines of popular IDEs, including Eclipse and NetBeans. I also worked on optimizing and comparing various test generation techniques [2, 6, 9].

UDITA can be used to describe tests using non-deterministic test generation programs that allow combining declarative (filters) and imperative (generators) styles. To write such programs, I introduced a Java-based language with non-deterministic choice operators and an interface for generating linked structures. UDITA includes new algorithms that generate concrete tests by efficiently exploring the space of all executions of non-deterministic programs. UDITA was evaluated by generating tests for data structures, refactoring engines, and a Java PathFinder model checker. The experiments showed that test generation using UDITA is faster and leads to test descriptions that are easier to write than in previous frameworks. Moreover, the novel execution mechanism of UDITA is essential for making test generation feasible.

Concurrent Code Analysis

Large systems are inherently concurrent and often afflicted by bugs such as dataraces, deadlocks, and atomicity violations. To help developers prevent these bugs, I have developed techniques for unit-testing concurrent [3, 12] and distributed [4] code, and model checking for web/database applications [15] and programs written in a language for parallel computing [14]. Also, I optimized model checking techniques, which led to a substantial reduction in exploration time [5, 7, 25].

Unit testing for concurrent code. A multithreaded test exercises the code under test with two or more threads. Each test execution follows some schedule/interleaving of the multiple threads, and different schedules can give different results. Developers often want to enforce a particular schedule for test execution, and to do so, they use time delays (Thread.sleep in Java). Unfortunately, this approach can produce false positives or negatives, and can result in unnecessarily long testing time. I worked on IMUnit [3, 12], a novel approach to specifying and executing schedules for multithreaded tests. IMUnit includes a tool that automatically instruments the code to control test execution to follow the specified schedule, and a tool that helps developers migrate their legacy tests, based on time delays, into IMUnit tests. The migration techniques have high precision and recall, and IMUnit reduces testing time compared to legacy tests on average 3.4×.

Model checking. Web applications maintain shared state in databases and typically there are relatively few database accesses for each request. This implies concurrent interactions are limited to relatively few and well-defined points, enabling scalable model checking. I developed DPF [15], an explicit-state model checker for database-backed web applications. DPF interposes between the program and the database layer, and precisely tracks the effects of queries made to the database. I defined independence relations at different granularity levels (at the database, relation, record, attribute, or cell level), and show the effectiveness of dynamic partial-order reduction based on these relations. DPF was able to find new concurrency bugs in two open-source web applications, including in a standard example distributed with the Spring framework.

Test Quality Assessment

While working on test input generation and techniques for testing concurrent code, there was always a question about the quality of the generated tests and comparison of the testing techniques. I have worked on assessing the quality of test suites and testing techniques for both sequential [11, 18, 20, 26] and concurrent code [1, 17]. My recent work [18, 26] helps researchers to compare test suites and testing techniques.

Researchers frequently compare test suites by measuring their coverage. A coverage criterion C provides a set of test requirements and measures how many requirements a given suite satisfies. A suite that satisfies 100% of the (feasible) requirements is C-adequate. Previous evaluations of coverage criteria mostly focused on such adequate test suites: given criteria C and C’, are C-adequate suites (on average) more effective
Future Directions

The increasing complexity of software impacts software quality and poses challenges for many existing testing and verification techniques. These challenges can be addressed by considering fundamental problems that affect practical systems. While my prior work has provided successes in several different domains of testing and verification, many challenges still lie ahead. In the short term, I would like to improve regression testing. In the long term, I would like to develop new testing and verification techniques for emerging platforms and utilize these platforms to improve testing and verification.

**Improved Regression Techniques.** I envision multiple novel techniques to push forward the state of the art in regression testing. To achieve this goal, I will extend, combine, and build on the techniques that I have developed. I want to optimize regression test selection by dynamically adapting granularities on which dependencies are tracked. Although Ekstazi, which uses coarse dependency granularity, provides overall time savings compared to rerunning all the tests or using a finer dependency granularity, Ekstazi can be further improved by adapting dependency granularity based on the code changes. I also want to improve regression testing of concurrent code. Specifically, I plan to combine IMUnit with regression techniques to detect code changes that do not affect the specified schedule. Finally, I would like to improve test input generation and maintenance of generated tests, for example by combining UDITA and regression techniques — by detecting invariants that are affected by the changes, I plan to regenerate new inputs by starting from previously generated inputs. This approach will simplify maintenance of large, automatically generated test inputs.

**Testing and Verification for Cloud and Mobile Applications.** I am interested in developing testing and verification techniques for cloud and mobile applications. Today, many of the world’s popular applications are running in the cloud. This scale triggers code execution that is not encountered in common testing scenarios (e.g., data transfer between servers under high load). Existing techniques, such as model checking, are likely not to scale to these complex systems as state-space explosion is an issue even for smaller applications. I plan to extend my work on model checking programs written in concurrent languages [14] to support model checking for cloud applications. In order to scale model checking, new approaches have to consider not only the semantics of the language constructs, but also programming model, runtime environment, implementation details of a compiler, and data distribution. Further, model checking cloud applications will require dynamic adjustments of granularity on which independence relation is defined [15]. Additionally, inspired by manual testing of my mobile game (www.snapshot.info) that interacts with the cloud, I am interested in automating performance/resilience testing for mobile and cloud platforms, and testing communication protocols, which will require automated generation and maintenance of mocks to make the testing feasible.

**Leveraging the Cloud to Improve Testing and Verification.** I would like to leverage the power of the cloud to speed up software testing and verification. One advantage of running testing and verification in the cloud is the reuse of the historical results among the developers. My work at Microsoft [23], on a cloud-based build system, was the initial step in this direction. Testing and verification in the cloud, similar to build process, would consist of two phases: analysis (finding dependencies between the tests) and execution (running the tests). Although execution is usually embarrassingly parallel (but not always, due to test dependencies), analysis is commonly sequential. I am interested in distributing the analysis phase. The problem is even more challenging in the context of regression techniques to improve analysis for runs for evolving software. I plan to develop prediction models for regression selection and regression verification to achieve the best balance in subsequent runs; as an example, selected tests/properties should not be on the same machine, data transfer should be minimal, tests/properties with similar dependencies should be grouped, long running tests should be separated and so on. To address these challenges, I plan to design techniques that combine static and dynamic analyses to split the exploration space and rebalance the execution/exploration based on the historical runs. While this research direction focuses on the cloud platform, in the future I want to leverage other new platforms that will emerge to improve software quality and developers’ productivity.
Refereed Journal Articles

[26] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. “Guidelines for Coverage-Based Comparisons of Non-Adequate Test Suites”. In: Transactions on Software Engineering and Methodology. Accepted in September 2014.


Refereed Conference Papers


Refereed Workshop Papers


