Contract-based Testing Tools

A primary purpose of testing is to detect software failures so that defects may be discovered and corrected. Testing cannot establish that a product functions properly under all conditions but can establish that it does function properly under specific conditions.

The scope of software testing often includes examination of code as well as execution of that code in various environments and conditions as well as examining the aspects of code: does it do what it is supposed to do and do what it needs to do. In the current culture of software development, a testing organization may be separate from the development team. There are various roles for testing team members. Information derived from software testing may be used to correct the process by which software is developed.

Assertions, as described in the Design by Contract(TM) software development methodology, contain the specification of the system. Thus, they provide an automated oracle for the testing process. Based on this insight, we are developing a tool for fully automatic testing of the "C" language code.

The goal of our work is to fully automate the entire testing process. We are working in the development of a testing tool that integrates several efficient testing techniques, such as: Random Testing, Black and White Box Testing, Delta Debugging, Mutation Testing, Refactoring.
Automatic Detection of Bugs using Static Analysis

Static analysis tools automate the inspection of source code and can identify common coding problems, inconsistent, and deviant code early in the development process. Alerts generated by automated static analysis tools require inspection by a developer to determine if the alert is an indication of an anomaly important enough for the developer to fix, called an actionable alert. When an alert is not an indication of an actual code anomaly or is deemed unimportant to the developer (e.g. the alert indicates a programmer anomaly inconsequential to the program's functionality), the alert is called unactionable. Automated static analysis tools may generate an overwhelming number of alerts, the majority of which are likely to be unactionable. To mitigate the costs of false positives when using static analysis, we want to build project specific models to predict or prioritize which alerts are actionable.

The goal of this research is to decrease the inspection latency and increase the rate of anomaly removal when using automated static analysis tools by creating and validating an adaptive false positive mitigation model to prioritize automated static analysis alerts by the likelihood the alert is actionable by a developer. We hypothesize that false positive mitigation models can be built that predict which alerts are actionable by developers, and these models can be used to prioritize alerts for developer inspection. False positive mitigation models are built by observing patterns in the characteristics about alerts that have been fixed or suppressed by a team or developer in the past, and using these patterns to predict which alerts are likely to be actionable and unactionable in the future.
Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards. Kent Beck, who is credited with having developed or ‘rediscovered’ the technique, stated in 2003 that TDD encourages simple designs and inspires confidence.

Test-driven development is related to the test-first programming concepts of extreme programming, begun in 1999, but more recently has created more general interest in its own right.

This research project describes a controlled experiment that examines the effects of TDD on internal software design quality. The experiment will be conducted with graduate students in a software engineering course. Students in three groups completed semester-long real programming projects using either an iterative test-first (TDD), iterative test-last, or linear test-last approach. We are interested in obtaining the benefits and advantages of using TDD as effective software design approach to improve both aspects such as test coverage and software quality and productivity enhancement. In addition, we are interested in showing that iterative development approaches that include automated testing provide additional benefits over a more traditional linear approach with manual tests.