SoftTest: A Framework for Software Testing of Java Programs

B. Childers, M. L. Soffa, J. Beaver, L. Ber, K. Cammarata, T. Kane, J. Litman, J. Misurda

Department of Computer Science
University of Pittsburgh
Pittsburgh, Pennsylvania 15260
{childers, soffa, beaver, libst18, juliya, jmisurda}@cs.pitt.edu

Abstract

Producing reliable and robust software has become one of the most important software development concerns in recent years. Testing is a process by which software quality can be assured through the collection of information about software. While testing can improve software reliability, current tools typically are inflexible and have high overheads, making it challenging to test large software projects. In this paper, we describe a new scalable and flexible framework, called SoftTest, for testing Java programs with a novel path-based approach to coverage testing. We describe an initial implementation of the framework for branch coverage testing and demonstrate the feasibility of our approach.

1 Introduction

In the last several years, the importance of producing high quality and robust software has become paramount [2]. Testing is an important process to support quality assurance by gathering information about the software being developed or modified. It is, in general, extremely labor and resource intensive, accounting for 50-60% of the total cost of software development [3]. The increased emphasis on software quality and robustness mandates improved testing methodologies.

Testing approaches are hindered by the lack of quality tools. Current tools are not scalable in terms of both time and memory, limiting the number and scope of the tests that can be applied to large programs. These tools often modify the software binary to insert instrumentation code for testing. However, the tested version of the application is not the same version that is shipped to customers and errors may remain. Testing tools are usually not flexible and only implement certain types of testing, such as various kinds of structural testing.

In this paper, we describe a testing framework, called SoftTest that addresses these problems. Our approach uses techniques that apply different testing strategies in an efficient and automatic way. Our method relies on a novel scheme to employ test plans that describe what should be automatically inserted and removed in executing code to carry out testing strategies. A test plan is a "recipe" that describes how and where a test should be performed. The approach is path specific and uses the actual execution paths of an application to drive the instrumentation and testing. Once a test site is complete, the instrumentation is dynamically removed to avoid run time performance overhead, and the test plan continues. The granularity of the instrumentation is flexible and includes statement level and structure level (e.g., loops, functions). Because our approach is dynamic and can insert and remove tests as a program executes, the same program that is tested can be shipped to a customer.

To ensure that our framework is general, we are developing a specification language from which a dynamic test plan can be automatically generated by a plan generator. The test specification describes what tests to apply and under what conditions to apply them. For example, specifications could be written for structural testing, data flow testing, random testing, hot path testing, and user defined testing. The specification language has both a visual representation and textual form. The visual language is expressed through a graphical user interface (GUI). The graphical tool also includes the ability to collect test results and present them to the user with a test analyzer, highlighting relevant parts of the application with the test results. The test framework—the GUI, test planner, and test analyzer—are an Eclipse plug-in for building new flexible and scalable testing tools.

We have implemented a prototype tool built with SoftTest that can perform branch coverage testing

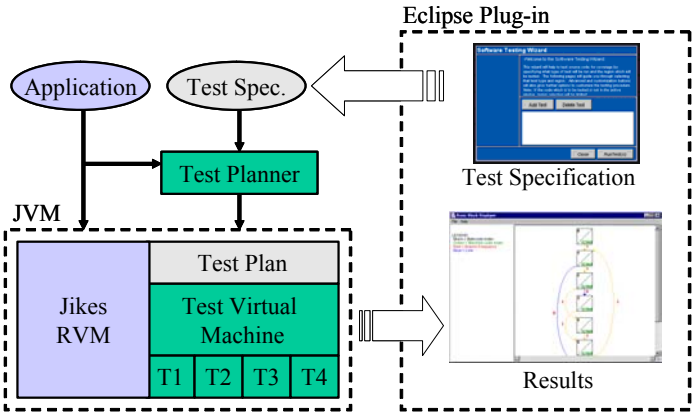

Figure 1: SoftTest testing framework for Java programs

over multiple regions of code in a Java program to demonstrate the feasibility and practicality of our approach. Our preliminary results show low runtime overhead for several small Java programs.

2 Test Framework

We are developing SoftTest as a complete framework for testing of Java software. Figure 1 shows the components in the framework, including a *test specifier*, a *test planner*, a *test virtual machine (TVM)*, and a *test analyzer*. One component is a language, *testspec*, for specifying a software test process. The specification includes the relevant parts of the program to be tested and the actions needed in the testing process. Testers can either write a specification in testspec or, better, use the GUI, which automatically generates a specification in testspec. A test planner consumes the testspec specification and determines a plan for testing the program given the specification. Using the plan, the TVM dynamically instruments an executing program to conduct the specified tests. Hence, the test plan is essentially a "program" that runs on the TVM to apply different software tests. The TVM is incorporated in the IBM Jikes Java RVM. Finally, the framework has an analyzer for reporting test results to the user.

In the following sections, we discuss SoftTest and our branch coverage tool, including test specification, test planning, the TVM, and the test analyzer.

2.1 Test Specification

In testing a software application, a developer may wish to apply different tests to various code regions. The tests are also often applied with different coverage criteria. SoftTest includes a graphical user interface for specifying the tests to apply, where to apply them, and under what conditions. Our tool for branch coverage testing includes the capability to select code regions using the GUI interface. A coverage criteria can also be specified for each region.

As shown in Figure 2, the GUI lets an user visually create a test specification. The main GUI features are identified with numbers in Figure 2. Feature 1 shows the button within the Eclipse platform that allows Eclipse to start the GUI. Feature 2 is the screen where the user creates and runs tests.

Feature 3 displays where the user selects the tests to run and defines regions to test. Feature 4 shows the text selection component of the GUI for highlighting lines of code to test in a Java program. In this way, the user is able to see the code and exactly what needs to be tested.

Features 5 and 6 aid the user in setting parameters for a specific test. One of these parameters is the number of times a code section should be hit to be considered covered. Finally, feature 5 sets parameters for whole test specification. From the visual specification, the GUI generates a specification in the testspec language. An example of the test specification for a method is shown in Figure 3.

As shown, the test specification language represents a user's desired test process. The language consists of two parts: DEFINITIONS and a BODY. In DEFINITIONS, regions of the code to test are defined. A region consists of a.java file name and class name, followed by any combination of line numbers, procedure names, loop identifiers or other regions. The purpose of DEFINITIONS is to

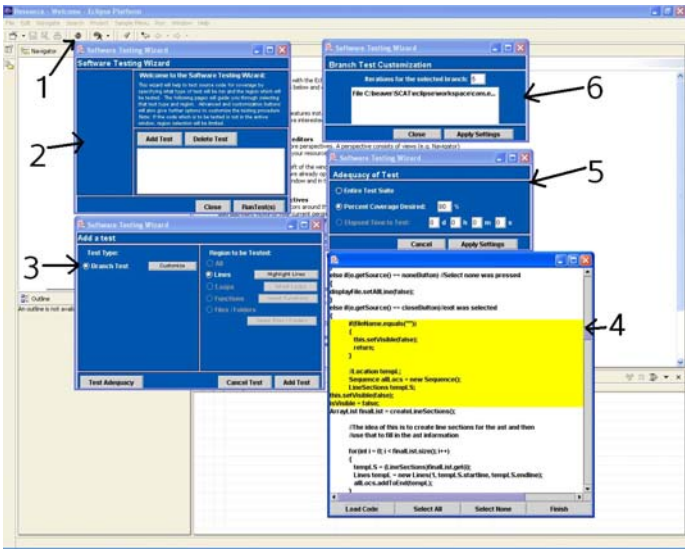

Figure 2: Test specification in Eclipse GUI

declare code regions for testing that can be referenced in the BODY section.

The BODY section has the actual test specifications. It has the test type (BRANCH_TEST means branch coverage testing) and what regions from DEFINTIONS to apply that test. In addition, conditions can be specified that must be met for the test to be considered complete. In Figure 3, the test coverage specified is 90%, meaning the test is complete when branch coverage reaches 90%.

```
public class simple6 {
 public int foo(int x) {
  if (x==200) x = x + 100;
  else x = x - 100;
  return x;
 } }
```

```
DEFINITIONS {
 NAME: X, REGION_D,
 LOCATION: FILE example.java {
  CLASS simple6, METHOD foo
 }
} BODY {
  DO BRANCH_TEST
   ON REGION X UNTIL: 90% }
```

Figure 3: Example test specification

2.2 Test Planner

From the test specification, an intermediate representation is generated for the test planner to decide how to instrument a Java program. For branch coverage, the test planner is invoked every time a method is loaded by the Jikes Just-in-Time compiler. The planner checks whether the loaded method is in the test specification to see whether it should be instrumented to apply branch coverage testing. Thus, only methods that are actually loaded and executed are instrumented by the planner.

The planner also retrieves the source code to bytecode line number mapping from a Java class file. If a user specified certain test regions, the planner will identify the basic blocks that need to be tested and set the appropriate parameters in the test plan.

The main function of the test planner is to determine where and how to test the application by producing a test plan. A test plan has two parts: a *test table* and *instrumentation payload*. The test table has information about how to conduct a specific test. For branch coverage, it says when to insert and remove instrumentation for covering each edge of a method's control flow graph (CFG). The payload code is target machine code that is executed at each instrumentation point. The payload for branch coverage updates a table that records which edges have been covered. The payload also removes instrumentation once an edge has been covered and inserts new instrumentation to cover edges that are next to execute and have not yet been hit. In this way, the payload inserts and removes instrumentation dynamically along a path of execution.

To apply branch coverage testing with minimal overhead, the test planner must determine how best to instrument the program. The goal is to minimize the amount of instrumentation that is executed

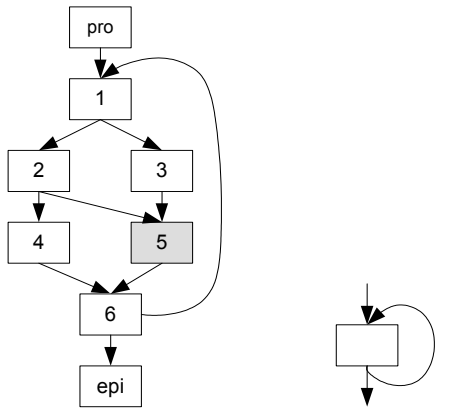

Figure 4: Example CFGs for test planner

when the method runs. Instrumentation is placed dynamically by having a block that is hit place instrumentation in its successors, according to the plan. To determine the edge that was executed for branch testing, predecessors and successors are needed. To get this information, the planner constructs a method's CFG.

The CFG is also used to identify special code constructs to identify the best way to instrument them. For instance, a reflexive block is a special case that occurs when a basic block has itself as a successor (see Figure 4b). The planner recognizes this condition and ensures that instrumentation is inserted to account for the reflexive loop by describing where its successor should place the instrumentation.

Other cases arise when there may not be enough information to determine which edge was taken by a branch. To produce correct branch coverage results, it is necessary to track of the current block's predecessor at run-time, so that the appropriate edge can be marked as covered. One special case occurs for a *stranded basic block*. Assume a basic block A has multiple predecessors and at least one of A's predecessors has multiple children. Then control comes from any predecessor but the first, we must ensure that there is instrumentation in the predecessor so the edge can be identified. In Figure 4a, the shaded block (5) is stranded if the paths 1->2->4->6 and 1>3>5>6 are taken and then the path 1->2->5->6 is taken. However, since instrumentation is inserted and deleted dynamically in our scheme, instrumentation would be removed from blocks 2 and 3 as soon as instrumentation is put in its successors (4 and 5). In the second time through block 5, we do not know whether the actual predecessor was block 2 or 3 because the instrumentation for covering block 2 and 3 was removed. To solve the stranded block case, instrumentation is placed in predecessors' basic blocks if the above conditions are satisfied.

Finally, the test planner can reduce the instrumentation overhead by inserting specialized payloads for single-entry blocks and their predecessors. Since single-entry blocks have a single predecessor, a simplified version of instrumentation can be used that updates the coverage table without knowledge about predecessor blocks.

2.3 Test Virtual Machine

Using the information gathered from the test planner, the TVM provides the functionality to insert and remove instrumentation at run-time. The TVM operates on target machine code generated by the Jikes JIT compiler. The TVM implements an interface for inserting and removing instrumentation with *fast breakpoints*. A fast breakpoint replaces an instruction in the target machine code with a jump to a breakpoint handler that invokes the test instrumentation payload from the test planner.

In the prototype for branch coverage, the TVM inserts a breakpoint in the first point specified by the test plan. This single breakpoint is then responsible for placing the next breakpoints needed, which, in the case of branch coverage, would be the nodes as identified by the planner as successor basic blocks. These successor breakpoints in turn, as they are hit, execute payload code that is responsible for placing breakpoints in successor blocks. In this manner, we can minimally affect the execution of the program since we are guiding the instrumentation by the currently executing path. This allows for performance to be minimally affected since hot paths and tight loops will be instrumented only for a few passes. Hence, most of the execution time is spent executing code without instrumentation.

The TVM's API provides primitives, such as the placement of successor breakpoints, storing test-specific data, and removal of breakpoints, for constructing fast breakpoints with varying payloads. This API allows for flexible instrumentation that can be specified in a variety of ways. The instrumentation constructed with the API is also highly scalable since only relevant portions of the program are instrumented for only as long as needed.

2.4 Test Analyzer

The test analyzer displays the results of tests conducted in the SoftTest framework. For branch coverage, the analyzer displays the CFG for a method and highlights the edges that were covered for a particular test input. The prototype displays the CFG for the target machine code; we are implementing support for source level coverage as well.

3 Preliminary Results

Our branch coverage tool implements the methodology outlined in this paper. A user can specify code regions on which to apply branch coverage for arbitrary methods in a Java program.

Using our tool, we have done preliminary experiments to determine performance and memory overhead. The benchmarks are small Java programs that uncompress a file, transpose a matrix, and create parallel threads. On these programs, the performance overhead for coverage varied from 1% to 5.5% of the total execution time. Of the total performance overhead, the test planner accounted for 27% to 64% and the instrumentation code 36% to 72%. In some cases, the planner overhead was more than the instrumentation overhead. This case can occur when a method has many basic blocks, but a short path is taken through the method. Similarly, the instrumentation overhead can be higher than the planning overhead when some instrumentation is repeatedly hit (e.g., in a stranded block in a loop).

For the benchmarks, the memory overhead was 178 to 822 bytes for the test table. The payload code has a common portion that is shared by all instrumentation points and a portion specific to each breakpoint inserted. The common code is 110 bytes and the breakpoint specific code is 31 bytes.

These initial experiments demonstrate that our approach has both low performance and memory overhead. We are currently evaluating the overhead for larger programs, including the SPECjvm98 benchmarks, and we expect that our preliminary results will scale to these programs.

4 Related Work

There are a number of commercial tools that perform coverage testing on Java programs, including JCover and IBM's Rational TestStudio. Most of these tools statically instrument the program to perform coverage testing. The work that is most closely related to ours is a tool developed with the ParaDyn parallel instrumentation framework [4]. This tool dynamically inserts and removes instrumentation on method invocations to do node coverage, where we take a similar approach for branch coverage. Unlike our approach, instrumentation is inserted in the whole method when it is invoked and a separate garbage collection process is done to remove instrumentation. Our technique instruments only along executed paths and removes instrumentation on-demand as soon as possible.

5 Summary

This paper described a framework, called SoftTest, for software testing of Java programs that relies on a novel scheme for dynamically inserting and removing instrumentation based on execution paths. We presented an initial prototype tool built with SoftTest for branch coverage testing. Our preliminary results are very encouraging: The test overhead on several small benchmarks ranged from 1% to 5.5%. We are currently extending our framework and tools to support other test types, including statement coverage and def-use coverage. Our future work will also include the ability to conduct software tests on optimized Java code.

Acknowledgements

This research was supported in part by an IBM Eclipse Innovation Grant.

References

[1] P. Kessler, "Fast breakpoints: Design and implementation", *ACM SIGPLAN Conf. on Programming Languages, Design and Implementation*, June 1990.

[2] L. Osterweil et al., "Strategic directions in software quality", *ACM Computing Surveys*, Vol. 4, December 1996.

[3] W. Perry, *Effective Methods for Software Testing*, John Wiley & Sons, Inc., New York, New York, 1995.

[4] M. Tikir and J. Hollingsworth, "Efficient instrumentation for code coverage testing", *Int'l. Symp. on Software Testing and Analysis*, Rome, Italy, 2002.