

Jazz: A Tool for Demand-Driven Structural Testing

Jonathan Misurda¹, Jim Clause¹, Juliya Reed¹, Bruce R. Childers¹, and Mary Lou Soffa²

¹ University of Pittsburgh, Pittsburgh PA 15260, USA,
{jmisurda, clausej, juliya, childers}@cs.pitt.edu

² University of Virginia, Charlottesville VA 22904, USA,
soffa@cs.virginia.edu

Abstract. Software testing to produce reliable and robust software has become vitally important. Testing is a process by which quality can be assured through the collection of information about software. While testing can improve software quality, current tools typically are inflexible and have high overheads, making it a challenge to test large projects. We describe a new scalable and flexible tool, called Jazz, that uses a demand-driven structural testing approach. Jazz has a low overhead of only 17.6% for branch testing.

1 Introduction

In the last several years, the importance of producing high quality and robust software has become paramount. 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 [1]. The increased emphasis on software quality and robustness mandates improved testing methodologies.

To test software, a number of techniques can be applied. One class of techniques is structural testing, which checks that a given coverage criterion is satisfied. For example, branch testing checks that a certain percentage of branches are executed. Other structural tests include def-use testing in which pairs of variable definitions and uses are checked for coverage and node testing in which nodes in a program's control flow graph are checked.

Unfortunately, structural testing is often hindered by the lack of scalable and flexible 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 for testing. In this case, the tested version of the application is not the same version that is shipped to customers and errors may remain. Testing tools are usually inflexible and only implement certain types of testing. For example, many tools implement branch testing, but do not implement node or def-use testing.

In this paper, we describe a new tool for structural testing, called Jazz, that addresses these problems. Jazz uses a novel demand-driven technique to apply

different testing strategies in an efficient and automatic way. Our method relies on test plans that describe what test instrumentation should be inserted and removed on-demand 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 execution paths of an application to drive the instrumentation and testing. Once a test site is covered, the instrumentation is dynamically removed to avoid performance overhead.

Jazz uses a specification language to describe what to test. From the specification, a test plan can be automatically generated by a test planner. The test specification describes what tests to apply and under what conditions to apply them. The specification language can be described with a GUI or through a textual representation. Jazz implements a GUI, a test planner, and a dynamic instrumenter for demand-driven testing. Jazz is incorporated as a plug-in in Eclipse and the IBM Jikes Java Research Virtual Machine. It supports branch, node and def-use testing over code regions in a program.

2 Testing Java Programs

To carry out a test with Jazz, a user constructs a test specification with a GUI. Next, the graphical specification is converted into a textual form in a language called *testspec*. A testspec specification includes the relevant code segments to be tested and the actions needed in the testing process. Once the user is ready to test the program, the specification is passed to a test planner. This step translates the specification into a test plan. In the next step, the test plan is used by the dynamic instrumenter to instrument the program and determine coverage. Finally, the test results are displayed by the GUI.

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. The Jazz GUI can specify the tests to apply, where to apply them, and under what conditions. A coverage criterion can also be specified for each region. As shown in Figure 1, the GUI lets an user create and apply a test specification. To illustrate user interaction with the tool, the figure shows several steps. The figure shows that the user has selected several source lines in the Eclipse source editor (step 1). The selected lines are used to build a test specification. In this case, lines 343-356 in the file `Compress.java` have been selected as a test region for branch testing. When a region is selected, a test specification is created and displayed by the GUI. Test specifications are shown in a “specification viewer” window (step 2). A specification may be changed or deleted from this window.

To run the current tests, the user clicks a button on the toolbar (step 3). Jazz automatically invokes the test planner, Jikes and the dynamic instrumenter. When the program completes, the test results are displayed as a bar graph in the specification viewer (step 4). The GUI also highlights covered and uncovered source lines in the Eclipse editor window.

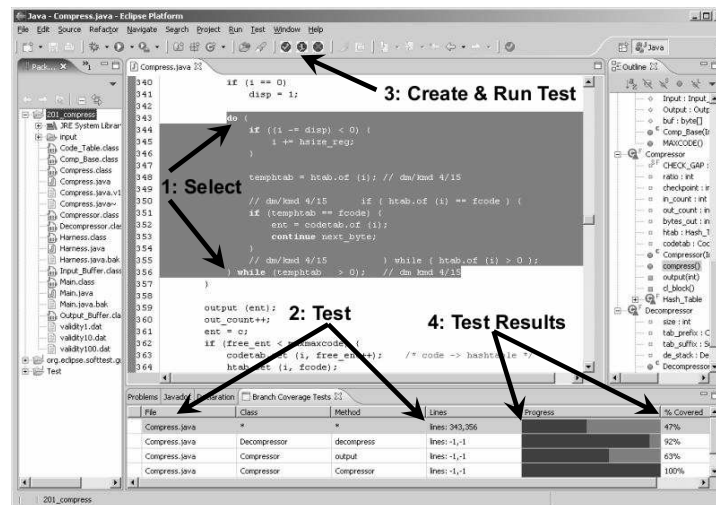


Fig. 1. Branch coverage GUI for Jazz

2.2 Test Planner

Using the test specification, the test planner decides how to test Java methods. The test planner is invoked every time a method is loaded by Jikes' Just-in-Time compiler. The planner checks whether there is a test specification for any portion of the method. If a specification exists, then the planner generates a test plan for the relevant code in the method. Thus, only methods that are actually loaded and executed are tested.

The main function of the test planner is to produce a test plan that determines where and how to instrument a method to do the test actions. The test plan describes how best to dynamically instrument a method to determine coverage. To generate a test plan, the planner identifies the locations where to instrument a test region, when to insert and remove instrumentation at each location, and what to do at each location. Typically, instrumentation locations correspond to basic blocks where coverage information is collected. For example, in def-use testing, there is instrumentation at each variable definition and all uses reachable from a definition.

Instrumentation is inserted and removed on-demand as the program executes. For example, in node testing, when a particular basic block is executed, instrumentation is inserted in successor blocks. Once a block is hit, its instrumentation can be removed because the block is covered. In branch and def-use testing, the planner ensures that instrumentation remains until all edges or all uses of a definition are covered.

Finally, the planner determines what actions to perform at each location. The actions are encoded in a "payload" that is executed at each location. In node testing, the payload updates coverage information, inserts instrumentation

at successor blocks, and removes the instrumentation in the current block. The payloads for branch and def-use testing are similar, except they check whether all edges or def-use pairs are covered.

2.3 Dynamic Instrumenter

With the test plan from the planner, the dynamic instrumenter provides the functionality to insert and remove instrumentation at run-time. This interface is targeted by the test planner. Dynamic instrumentation (that can be removed/inserted at run-time) is implemented with fast breakpoints[2]. A fast breakpoint replaces an instruction in the target machine code generated with a jump to a breakpoint handler that invokes the test instrumentation payload from the test planner.

3 Experimental Results

We investigated Jazz's performance and compared it to a traditional approach based on static instrumentation. To ensure a fair comparison, we implemented a separate tool that uses static instrumentation in our framework. This tool instruments a method's binary code before run time and does not remove instrumentation. It is similar to IBM Rational PurifyPlus and JCover. Jazz and the static tool differ only in on-demand versus static instrumentation. In the experiments, all loaded methods were covered and the benchmarks were run on a Linux 2.4 GHz Pentium IV with 1 GB RAM.

We measured run-time when the benchmarks were run directly in Jikes without testing, with Jazz and with the static tool. For brevity, we summarize the run-times only for branch testing. When run without testing, the benchmarks take 13.8-44.7 seconds. With the static branch testing tool, run-time is increased dramatically. It varies from 20.7-96.1 seconds and incurs an overhead of 11.7-241% (average 89.9%) over native execution. Jazz has much lower run-times than the static tool. Its run-time is 20.6-43.9 seconds and the performance overhead is only 0.3% to 7.8% (average 17.6%). Jazz has less overhead than the static tool because instrumentation is inserted and removed on-demand.

4 Summary

This paper described a new tool, called Jazz, for software testing of Java programs that relies on a novel scheme for dynamically inserting and removing instrumentation on-demand. The performance results with Jazz are very encouraging: The average overhead for branch testing with Jazz was only 17.6%.

References

- [1] W. Perry: *Effective Methods for Software Testing*. John Wiley and Sons, Inc., New York, 1996.
- [2] P. Kessler: *Fast breakpoints: Design and implementation*. ACM SIGPLAN Conference on Programming Languages, Design and Implementation, 1990.