Jazz: A Tool for Demand-Driven Structural Testing

Jonathan Misurda, James A. Clause, Juliya L. Reed, Bruce R. Childers, and Mary Lou Soffa

Department of Computer Science
University of Pittsburgh
Pittsburgh, Pennsylvania 15218 USA
{jmisurda,clausej,juliya,childers}@cs.pitt.edu

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22904 USA
soffa@cs.virginia.edu

Abstract

Software testing to produce reliable and robust software has become vitally important 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 tool, called Jazz, for testing Java programs with a novel demand-driven dynamic approach to structural testing. Jazz has a graphical user interface for specifying and running tests, a test planner to determine the most efficient way to test the program, and a dynamic instrumenter to carry out a test.

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.

To test software, a number of techniques can be applied. One class of techniques that is widely used is structural testing, which checks that a given coverage critierium 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 approaches are 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. This tool 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 the actual 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, and the test plan continues. The granularity of the instrumentation is flexible and includes statement and structure level (e.g., loops, functions). Because this 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.

Jazz has a specification language, which is used 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 has both a visual representation and textual form. The visual language is expressed through a graphical user interface (GUI). The GUI can be used to describe tests, automatically carry them out, and report results.

Jazz is a complete and new implementation of our SoftTest framework [REF] for structural testing. It implements a GUI, a test planner, and dynamic instrumenter for on-demand testing. Jazz is incorporated as a plug-in in the Eclipse integrated development environment (IDE) and the IBM Jikes Java Research Virtual

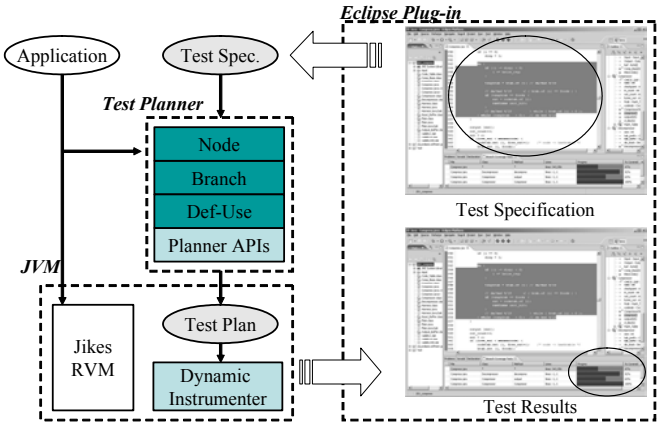

Figure 1: Jazz Tool Flow and Methodology

Machine. It supports branch, node and def-use testing over code regions in a program. Experiments show that Jazz's run-time overhead is very low in comparison to traditional testing tools that use static instrumentation.

2. Jazz: Testing Java Programs

Figure 1 shows Jazz's components and how they interact. To carry out a test, a user constructs a test specification with the GUI. Next, the graphical representation of the test specification is converted into a textual form in a language called *testspec*. A *testspec* specification includes the relevant parts of the program to be tested and the actions needed in the testing process. If desired, testers can write specifications directly in *testspec*, rather than use the GUI. 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. Jazz has a GUI for specifying the tests to apply, where to apply them, and under what conditions. A coverage criterium can also be specified for each region.

As shown in Figure 2, the GUI lets an user visually create and apply a test specification. The GUI shows the complete Eclipse iIDE and how Jazz is incorporated. The callouts in the figure show some of Jazz's GUI widgets for creating, running, and viewing test results.

To illustrate how the user interacts 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 Eclipse 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.

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 producte 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 (in the

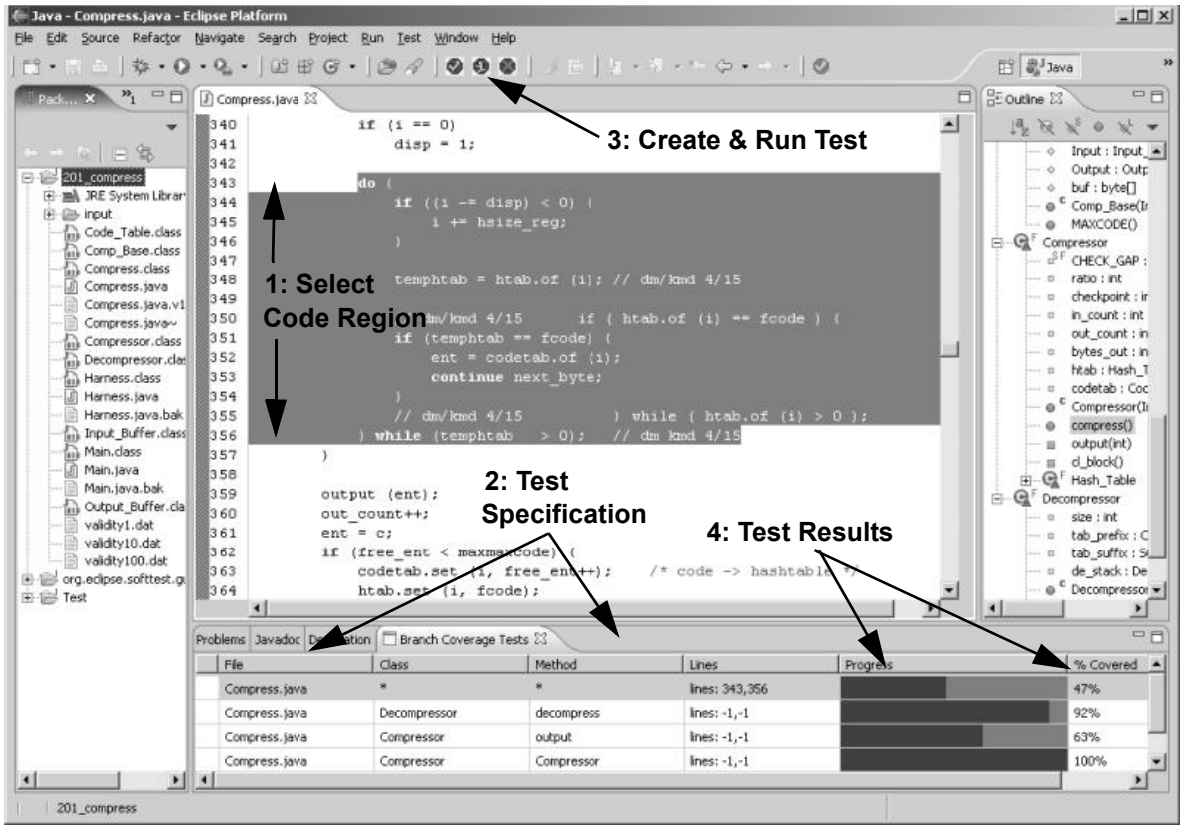

Figure 2: Branch Coverage GUI for Jazz

control flow graph of a test region) where coverage information is collected. For example, in def-use testing, there are instrumentation locations for 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 out of a block 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 "test payload" that is executed at an instrumentation 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.

When multiple tests are applicable at a location, the test planner generates a "customized payload" for that location. For example, in def-use testing, there may be several definitions and uses within a single basic block. The custom payload for such a block inserts instrumentation at uses reachable from the definitions. It also updates the coverage for the uses in the current block.

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. The dynamic instrumenter operates on target machine code generated by Jikes. Dynamic instrumentation (that can be removed/inserted at run-time) is implemented 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.

The dynamic instrumenter's API provides primitives, such as the placement of successor breakpoints, storing test-specific data, and removal of breakpoints, for con-

structing fast breakpoints with varying payloads. The instrumentation constructed with the API is highly scalable since only relevant portions of the program are instrumented for only as long as needed.

3. Experimental Results

We used Jazz to measure the node, branch, and def-use coverage of the SPECjvm98 benchmarks. In this experiment, the test specification covers all loaded methods and the test inputs are the data sets from SPECjvm98. The benchmarks were run on a 2.4 GHz Pentium IV machine with 1 GB of RAM and RedHat Linux.

Benchmark	Branch	Def-Use	Node
compress	58%	89.8%	90.6%
jess	46.8%	71.8%	80.3%
db	44.4%	75%	76.9%
javac	38.9%	66.9%	75%
mpegaudio	60.9%	90.5%	88.7%
mtrt	50.6%	87.3%	90.3%
jack	55.6%	73.4%	82.2%

Table 1: Percentage coverage

Table 1 shows the coverages for the three tests supported by Jazz. As the table shows, for branch testing, Jazz reported a coverage of 38.9–58% and for node testing 75–90.6%. The tool reported coverage of 66.9–90.5% for def-use testing.

We also investigate Jazz's performance and compared it to a traditional tool based on static instrumentation. For fairness, we implemented a tool that uses static instrumentation in our testing environment and framework. This tool instruments a method's binary code before run time and does not remove instrumentation. It is similar to Rational PurifyPlus [REF] and JCover [REF]. Jazz and the static tool perform the same actions when conducting a test. They differ only in on-demand versus static instrumentation.

We measured run-time when the benchmarks were run directly in Jikes without testing, with Jazz and with the static tool. For brevity, we report 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. Indeed, in tight loops, instrumentation is typically removed within the first few loop iterations, which keeps overhead low (e.g., *compress* has a tight loop). As these results show, Jazz is an effective tool, with minimal performance overhead.

4. Related Work

There are a number of commercial tools that perform coverage testing on Java programs, including JCover and IBM's Rational PurifyPlus.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 pass removes the instrumentation. Our technique instruments only executed paths and removes instrumentation as soon as possible.

5. 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 17.6%, while a tool that used static instrumentation had an overhead of 89.9%. Jazz supports branch, node, and def-use testing. It is available for Eclipse 3.0 and the IBM Jikes Java RVM.

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.