UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO
DEPARTAMENTO DE CIÊNCIAS DE COMPUTAÇÃO E ESTATÍSTICA http://www.icmc.usp.br
Caixa Postal 668 – CEP 13560-970 – São Carlos, SP – Fone (16) 273-9655 – Fax (16) 273-9751

JaBUTi – Java Bytecode Understanding and Testing

User's Guide
Version 1.0 – Java

A. M. R. Vincenzi[†], M. E. Delamaro[‡] and J. C. Maldonado[†]

[†]Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo
São Carlos, São Paulo, Brazil
{auri, adenilso, jcmaldon}@icmc.usp.br

[‡]Faculdade de Informática
Centro Universitário Eurípides de Marília
Marília, São Paulo, Brazil
delamaro@fundanet.br

São Carlos, SP, Brazil
March, 2003

Contents

Abstract

This report describes the main functionalities of JaBUTi (Java Bytecode Understanding and Testing) toolsuite. JaBUTi is designed to work with Java bytecode such that no source code is required to perform its activities. It is composed by a coverage analysis tool, by a slicing tool, and by a complexity metric's measure tool. The coverage tool can be used to assess the quality of a given test set or to generate test set based on different control-flow and data-flow testing criteria. The slicing tool can be used to identify fault-prone regions in the code, being useful for debugging and also for program understanding. The complexity metric's measure tool can be used to identify the complexity and the size of each class under testing, based on static information. A simple example that simulates the behavior of a vending machine is used to illustrate such tools.

1 JaBUTi Functionality and Graphical Interface

JaBUTi implements a subset of the functionalities provided by *xSuds*, a complete tool suite for C and C++ programs [2]. This section describes the operations that can be performed by JaBUTi. The graphical interface allows the beginner to explore and learn the concepts of control-flow and data-flow testing. Moreover, it provides a better way to visualize which part of the classes under testing are already covered and which are not. A general view of the JaBUTi graphical interface, including its menus, is presented in Figure 1. A brief description of each option on each menu is presented below.

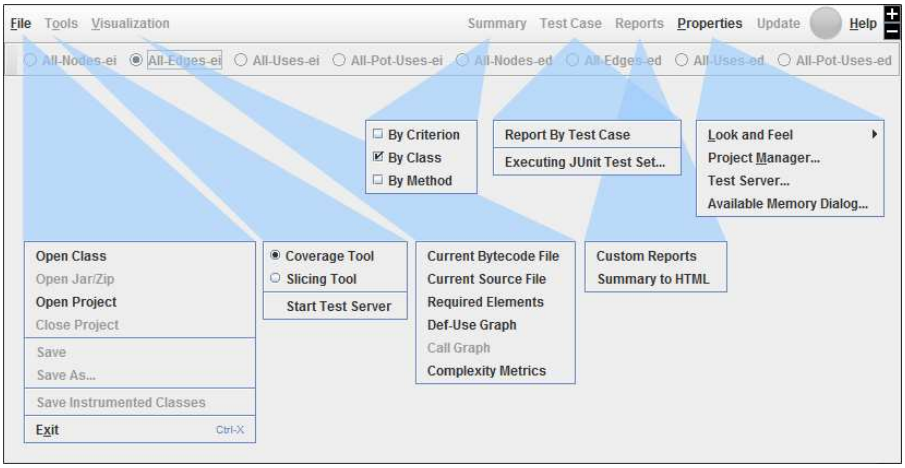


Figure 1: Operations available in the graphical interface.

- **File Menu** - provides options to create and manipulate a JaBUTi project.
  - **Open Class** - allows to select the base class file from where the classes to be tested are identified.
  - **Open Project** - opens a previously created project.
  - **Close Project** - closes the current project.
  - **Save** - saves the current project.
  - **Save As** - saves the current project with a different name.
  - **Save Instrumented Classes** - saves the classes from current project, already instrumented for external testing.
  - **Exit** - exits of the tool.
- **Tools** - provides the set of tools available in JaBUTi.
  - **Coverage Tool** - enables the JaBUTi coverage tool.
  - **Slicing Tool** - enables the JaBUTi slicing tool.
  - **Start Test Server** - starts the test server for mobile devices.
- **Visualization** - provides different forms of visualization of the classes and methods under testing.

  - **Current Bytecode File** - shows the highlighted bytecode of the current selected class file.
  - **Current Source File** - shows the highlighted source code of the current selected class file. Observe that this option requires that the source code is available to be performed.
  - **Required Elements** - shows the set of required elements for a given method of a given class, considering the current selected criterion (shown below the main menu). The presented screen also allows to mark a testing requirement as active/deactive or feasible/infeasible.
  - **Def-Use Graph** - shows the DUG of a given method of the current class.
  - **Complexity Metrics** - shows the resultant value of the set of complexity metrics implemented in JaBUTi for the complete set of user classes obtained from the base class. This set of classes includes the classes under testing.

- **Summary** - provides personalized coverage information in different levels of abstractions.
  - **By Criterion** - shows the cumulative coverage information for each testing criterion, considering all classes under testing.
  - **By Class** - shows the coverage information with respect to the current selected criterion, for each individual class under testing.
  - **By Method** - shows the coverage information with respect to the current selected criterion, for each individual method of each class under testing.

- **Test Case** - provides options for test set manipulation and report generation.
  - **Report By Test Case** - shows the coverage information with respect to the current selected criterion, for each individual test case, considering all class under testing. The presented screen also allows to enable/disable and delete/undelete test cases.
  - **Importing from JUnit** - allows to import a test set generated according to the JUnit framework.

- **Reports** - provides options to save JaBUTi's reports in HTML format.
  - **Custom Reports** - allows to generate a custom HTML report from the current testing project considering different levels of granularity.
  - **Summary to HTML** - allows to generate a HTML from any tabled style report provided by the JaBUTi graphical interface.

- **Properties** - provides general configuration options.
  - **Look and Feel** - allows to change the look and feel style considering three different options: Metal (default), Motif, and Windows.
  - **Project Manager...** - allows to verify and change the current set of classes under testing in the current project.
  - **Test Server...** - configuration of the test server.
  - **Available Memory Dialog...** - shows current available memory on system.

- **Update** - provides a visual information every time an event that affect the coverage occurs. For example, such a button becomes red in case additional test cases are imported or appended in the end of the trace file to indicate that a new event that affects the coverage information occurs. As soon as it is clicked, its background color changes to grey.

- **Help** - provides only one option to show information about the authors/developers of JaBUTi.

JaBUTi requires the creation of a testing project, such that the tester can specify only once the set of classes to be instrumented and tested. Section 2 describes how to create a JaBUTi's testing project. After having created the project, JaBUTi provides to the tester a coverage analysis tool, a slicing tool and a static metric measure tool. The coverage analysis tool is described in Section 3. Section 4 describes the slicing tool, and Section 5 describes the measure tool.

2 How to Create a Testing Project

In JaBUTi the testing activity requires the creation of a testing project. A testing project is characterized by a file storing the necessary information about (i) the base class file, (ii) the complete set of classes required by the base class, (iii) the set of classes to be instrumented (tested), and (iv) the set of classes that are not under testing. Additional information, such as the CLASSPATH environment variable necessary to run the base class is also stored in the project file, whose extension is .jbt. During the execution of any .class file that belongs to the set of classes under testing of a given project, dynamic control-flow information (execution trace) is collected and saved in a separate file that has the same name of the project file but with a different extension (.trc).

For example, considering the vending machine example, the vending package is composed by three .java files: VendingMachine.java, Dispenser.java and TestDriver.java. Suppose that these files are saved in a directory named `~\example`. The directory structure is like:

```
auri@AURIMRV ~
$ ls -l example/*
-rw-r--r--    1 auri         1313 Aug  6 09:07 Dispenser.java
-rw-r--r--    1 auri         1340 Aug  6 09:07 TestDriver.java
-rw-r--r--    1 auri          923 Aug  6 09:07 VendingMachine.java
```

To compile such an application one of the following command can be used:

```
auri@AURIMRV ~
$ javac -d example example/*.java
```

or

```
auri@AURIMRV ~
$ javac -g -d example example/*.java
```

Observe that in the later command, the debug option is activated, thus, the generated .class files contains more information, such as the real variable names. In this report we are using class files compiled with the debug option.

After the Java source files have been compiled, the `~\example` directory contains the following structure:

```
auri@AURIMRV ~
$ ls -l example/*
-rw-r--r--    1 auri         1313 Aug  6 09:07 Dispenser.java
-rw-r--r--    1 auri         1340 Aug  6 09:07 TestDriver.java
-rw-r--r--    1 auri          923 Aug  6 09:07 VendingMachine.java
example/vending:
vending:
total 6
-rw-r--r--    1 auri         1478 Aug  6 09:49 Dispenser.class
-rw-r--r--    1 auri         1340 Aug  6 09:49 TestDriver.class
-rw-r--r--    1 auri         1253 Aug  6 09:49 VendingMachine.class
```

Now, from the generated .class files, the user can create a project using JaBUTi. To do this, the first step is to invoke JaBUTi's graphical interface. Supposing that JaBUTi is installed on `~\Tools\jabuti`, the command below causes the invocation of its graphical interface.

```
auri@AURIMRV ~/example
$ java -cp ".;..\Tools\jabuti;\
>..\Tools\jabuti\lib\BCEL.jar;\
>..\Tools\jabuti\lib\jviewsall.jar;\
>..\Tools\jabuti\lib\crimson.jar;\
>..\Tools\jabuti\lib\junit.jar" gui.JabutiGUI
```

Observe that the tool requires that some third-party libraries (BCEL [5] to manipulate the Java bytecode files, ILOG JViews [10] to visualize the DUG, Crimson to manipulate XML files[1], and JUnit [3] to import test sets.) to be included in the CLASSPATH to allow its execution. The current directory, (“.” in our example), and the base directory of the tool (`..\Tools\jabuti`) are also included to allow the correct execution of our example and the tool itself. If desired, the user can set the CLASSPATH environment variable as a system variable that is initialized during the boot process. In this case, to call JaBUTi's GUI, the parameter -cp can be omitted. Figure 2 illustrates the JaBUTi's initial window.

To create a new project, the first step is to select a base .class file from File → Open Class menu. A dialog window, as illustrated in Figure 3(a) appears. From this dialog window, the tester selects the directory where the base class file is located and then select the base class file itself (Figure 3(b)). Once the base class file is selected the tool automatically identifies the package that it belongs (if any) and fills out the Package field with the package's name. The Classpath should contains only the path necessary to run the selected base class. In our example, since the tool is being called inside the `example` directory, to run vending.TestDriver it is necessary that the Classpath field contains only the current directory, as shown in Figure 3(b).

By clicking the Open button (Figure 3(b)), the Project Manager window, as illustrated in Figure 4, will be displayed. From the selected base class file (TestDriver in our example) the tool identifies the complete set of system and non-system class files necessary to execute TestDriver. Currently, JaBUTi does not allows the instrumentation of system class files. Therefore, the complete set of non-system class files (user's classes) relate to TestDriver can be instrumented. From the Project Manager window (left side) the user can select the class files that will be tested. At least one class file must be selected. In our example, we select VendingMachine and Dispenser to be instrumented. TestDriver was not selected

[1]Such a package is required when a java compiler under version 1.4.1 is used (considering the Sun compiler)).

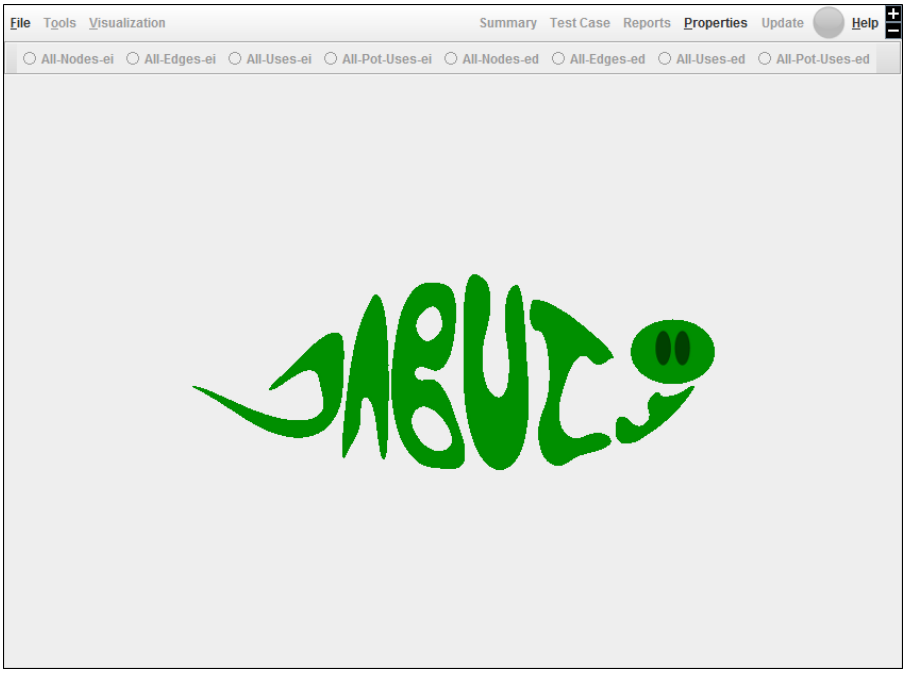
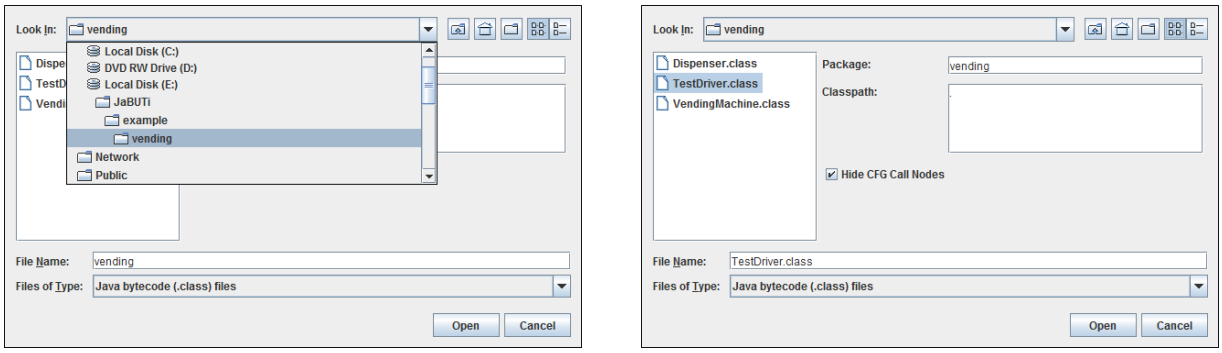


Figure 2: JaBUTi main window.



(a) (b)

Figure 3: Open Class: (a) Selecting the directory and (b) Selecting the main class file.

since it is only the driver that allows the tester to observe whether the behavior of VendingMachine and Dispenser are correct with respect to their specification.

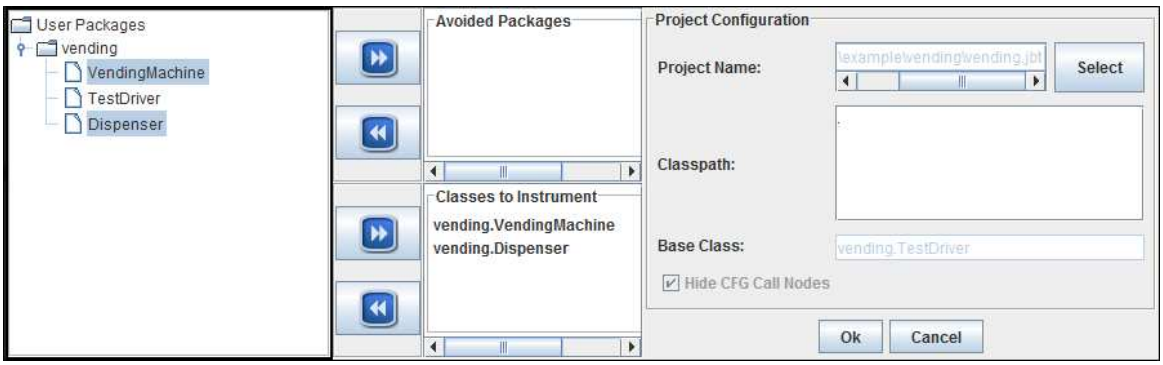


Figure 4: Selecting the class files to be tested.

Moreover, the tester must give a name to the project being created (vending.jbt in our example) by clicking on the Select button. By clicking on the Ok button, JaBUTi creates a new project (vending.jbt in our example), constructs the DUG for each method of each class under testing, derives the complete set of testing requirements for each criterion, calculates the weight of each testing requirement, and presents the bytecode of a given class under testing. By creating a project, the tester does not need to go through this entire process again if he/she intends to test the same set of class files. Next time, he/she can just go to File → Open Project menu option and reload a previously saved project.

Once the project is created, the tester just need to open it and the tool automatically detects the classes that compose it. JaBUTi can be used for analyzing the coverage of a given class file, for debugging the class file using the slice tool and for collecting static metrics information. JaBUTi also allows the visualization of the Def-Use Graph (DUG) of each method in a given class file as well as the visualization of the source code, when available. Different kinds of testing reports can also be generated considering different levels of abstraction. The following sections describe these features in detail.

3 How to use JaBUTi as a Coverage Analysis Tool

By default, the tool displays the bytecode instead of the Java source code, since the source code may not be available. The user can use the Visualization menu to change the display to show the bytecode or source code, as well as the DUG of each method in the current class. As can be observed in Figure 5, JaBUTi uses different colors to provide hints to the tester to ease the test generation. The different colors represent requirements with different weights, and the weights are calculated based on dominator and super-block analysis [1]. Informally, the weight of a testing requirement corresponds to the number of testing requirements that will be covered if this particular requirement is covered. Therefore, covering the requirement with the highest weight will increase the coverage faster[2]. The weight schema is used in all views of a given class/method: bytecode, source code and DUG. Figure 5 shows part of the colored bytecode of the Dispenser.dispense() method before the execution of any test case and Figure 6 part of the DUG of the same method. The different colors correspond to the weight of each node, considering the All-Nodes-ei criterion. Observe that the color bar (below the main menu) goes from white (weight 0) to red (weight 7).

In our example, observe that the node 105 in Figure 6, composed by bytecote instructions from 105 to 112 (Figure 5), is one of the highest weight. In this way, a test case that exercises node 105 increases the coverage in at least 7. A requirement with weight zero is a covered requirement and it is painted in white.

Although the source code is not required by JaBUTi, when it is available, the corresponding source code of the current class can also be displayed in colors, mapped back from the bytecode, facilitating the identification of which part of the code should be covered first. For example, Figure 7 shows that the statements on line 22 and 24 have the highest priority w.r.t. the All-Nodes-ei criterion. By covering one of these particular nodes, at least 7 other nodes will be covered.

There are two different types of nodes, represented by single and double line circles. Double circles represent call nodes, i.e., nodes where exists a method call to another method. This nodes are identified

[2]Observe that the weight is calculated considering only the coverage information and should be seen as hints. It does not take into account, for instance, the complexity or the criticality of a given part of the program. The tester, based on his/her experience may desire to cover first a requirement with a lower weight but that has a higher complexity or criticality and then, after recomputing the weights, uses the hints provided by JaBUTi to increase the coverage faster.

Figure 5: Dispenser.dispenser method: bytecode prioritized w.r.t. All-Pri-Nodes criterion.

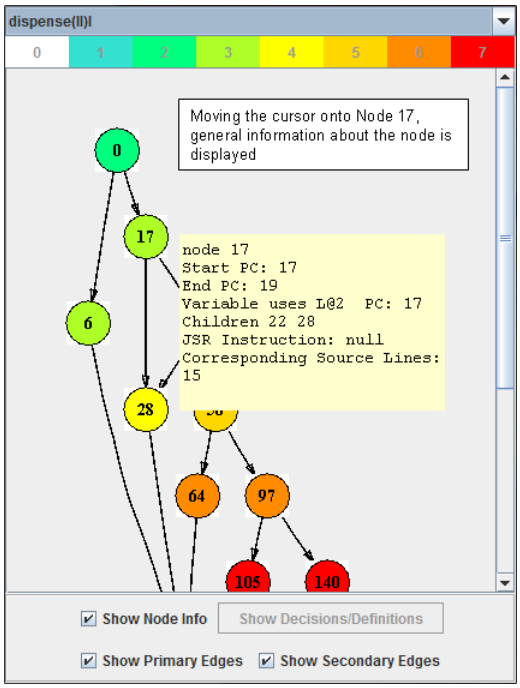


Figure 6: Dispenser.dispenser method: DUG prioritized w.r.t. All-Pri-Nodes criterion.

since we intend to extend our tool to deal, not only with intra-method testing criteria, but also with inter-method testing criteria that requires the identification of the relationship among methods. Bold circles, not visible in Figure 6, represent exit nodes. Observe that methods in Java can have more that one exit node due to the exception-handling mechanism. All the other nodes are represented

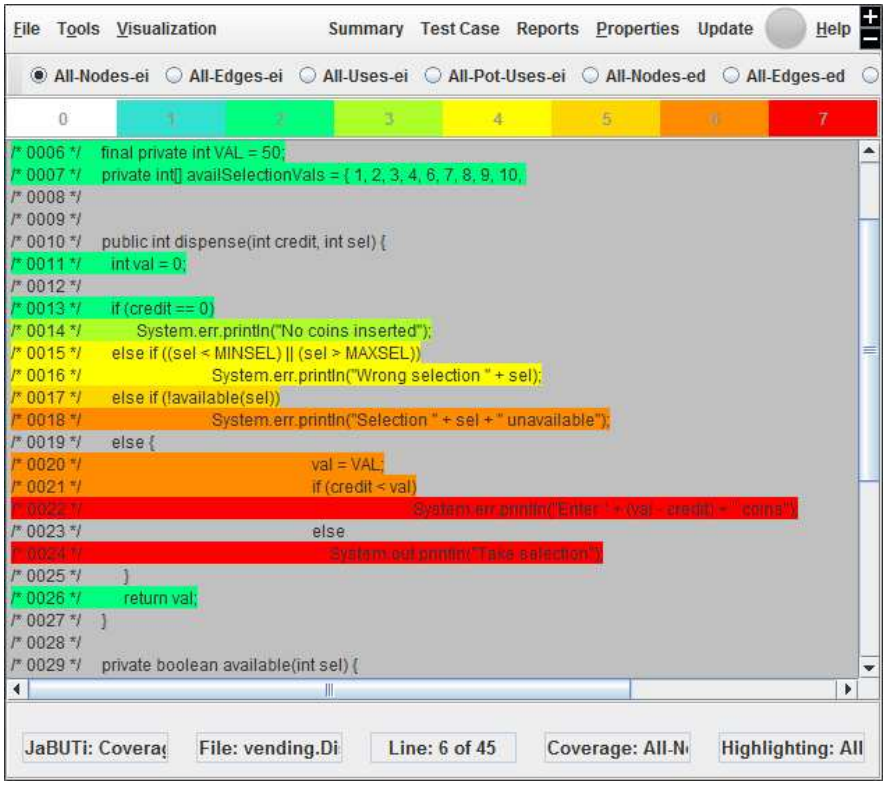


Figure 7: Dispenser.dispenser method: source code prioritized w.r.t. All-Pri-Nodes criterion.

as single line circles. We also have two different types of edges to represent the "normal" control-flow (continuous line – Primary Edges) and exception control-flow (dashed lines – Secondary Edges). Figure 6 does not contain exception edges. Primary and secondary edges can be hidden by deselecting the Show Primary and Show Secondary Edges check box, respectively. The node information shown when the cursor is moved onto the node, as illustrated in Figure 6, can also be disabled by deselecting the Show Node Info check box.

3.1 How the testing requirements are highlighted

As can be observed above the main menu in Figure 5, JaBUTi supports the application of six structural testing criteria: All-Nodes-ei, All-Nodes-ed, All-Edges-ei, All-Edges-ed, All-Uses-ei, and All-Uses-ed. Depending on which criterion is active, the bytecode, source code or DUG is colored in a different way. Figures 5, 6, and 7 show the color schema considering the All-Nodes-ei criterion, which is the same as the All-Nodes-ed, i.e., for these criteria, each requirement is a node, therefore, since each node has its own weight, the complete bytecode, source code or DUG appears highlighted according to the weight of each node.

Considering the All-Edges-ei and All-Edges-ed criteria, their requirements (DUG edges) are colored using a 2-layer approach. For All-Edges-ei criterion, only the nodes with more than one out-going edge (decision nodes) are painted in the first layer. For example, Figure 8 shows part of the decision nodes of method Dispenser.dispense() and how they are painted in the first layer. For each decision node, its weight is the maximum weight of its branches. Suppose a decision node has two branches: one has a weight of 2 and the other has a weight of 7. The weight of this decision node is 7. A decision node has a zero weight if and only if all its branches are covered. Such decisions are highlighted in white. Observe that, in the bytecode view, only the last bytecode instruction of a decision node is

highlighted, not the entire node as the All-Nodes-ei and the All-Nodes-ed criteria. For example, DUG decision node 22 goes from offset 22 to 25, although, in the bytecode view, only bytecode instruction at offset 25 is highlighted.

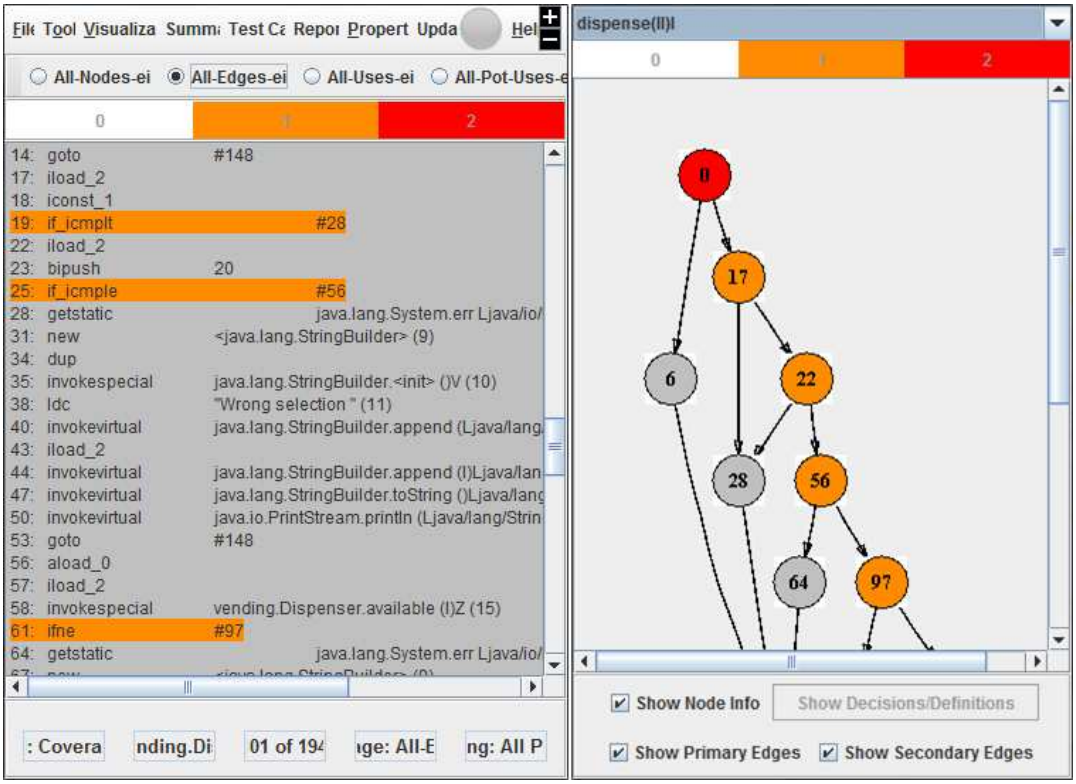


Figure 8: Color schema for All-Pri-Edges criterion: first layer.

By clicking either in a colored bytecode instruction or in a DUG node, all destination nodes of the branches associated with the selected decision node are highlighted and the decision node itself changes to a different color. For example, by clicking on the decision node with the highest weight (node 0 in the example), its children (nodes 6 and 17) are highlighted in different colors, considering their individual weights: node 6 has a weight of 2 (red) and node 17 has a weight of 1 (orange), as illustrated in Figure 9. Therefore, to improve the coverage with respect to the All-Edges-ei criterion, the tester should exercises first the edge $(0, 6)$, since it has the highest weight.

Observe that, in case a given method has no decision node, only the entry node is painted in the first layer and its child in the second layer. In this case, edges criterion is equivalent to nodes criterion, since, in a normal execution, once the entry node is exercised, all the other nodes and edges are. Figure 10 illustrates how the DUG of the default constructor of the class VendingMachine (VendingMachine.init()) is highlighted before (Figure 10(a)) and after (Figure 10(b)) the node selection.

For the All-Edges-ed criterion, the same approach is used. In the first layer, all nodes with at least one secondary out-going edge, instead of decision nodes, are highlighted. For each such a node, its weight is the maximum weight of its branches and a node with more that one secondary out-going edge is considered covered (zero weight) if and only if all exception-handler associated with such a node is covered.

Figure 11 and 12 shows the first and the second layer of Dispenser.available() method, considering the All-Edges-ed criterion, respectively. In this example, in the first layer, all highlighted nodes has the same weigh (one) since it has only one valid exception-handler for the entire method. In the

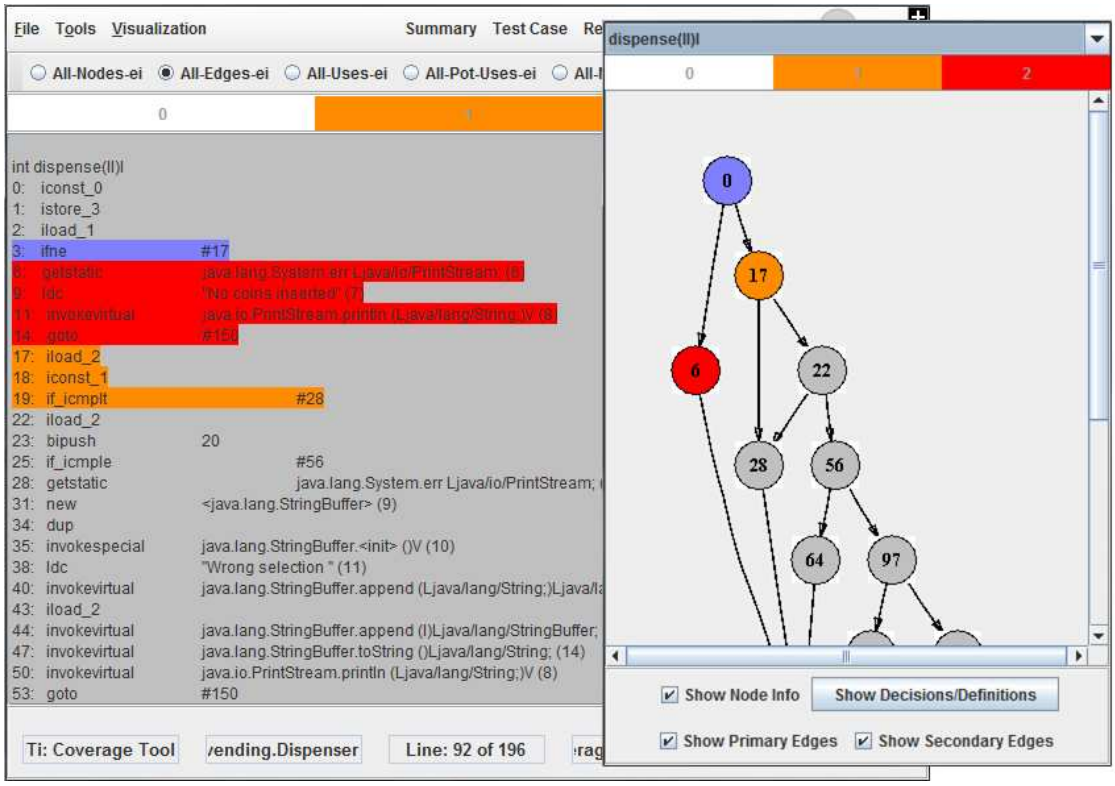
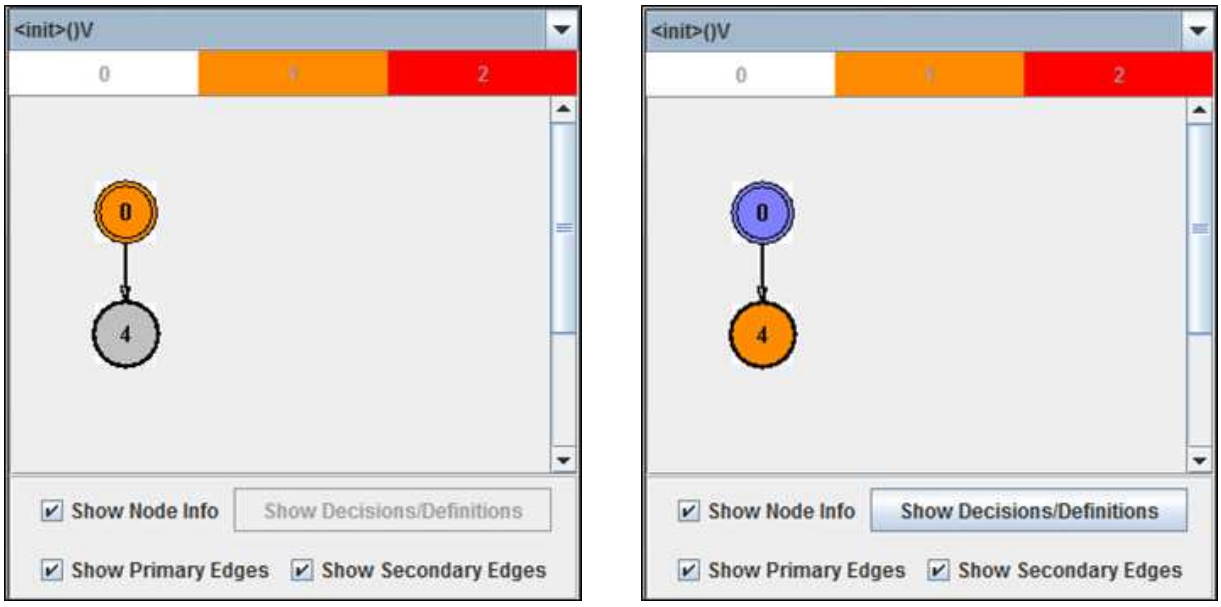


Figure 9: Color schema for All-Pri-Edges criterion: second layer.



(a) (b)

Figure 10: Special case when no decision node is found.

bytecode view, only the last bytecode instruction of each node is highlighted. For example, node 2 goes from offset 2 to 8, therefore, bytecode instruction at offset 8 is highlighted. By clicking on such an instruction or on the DUG node 2, Figure 12 shows the resultant color schema of the second layer.

Similar to a decision and its branches are displayed, a 2-layer representation is also used to display def-use associations for the All-Uses-ei and All-Uses-ed criteria. The first layer shows all the definitions

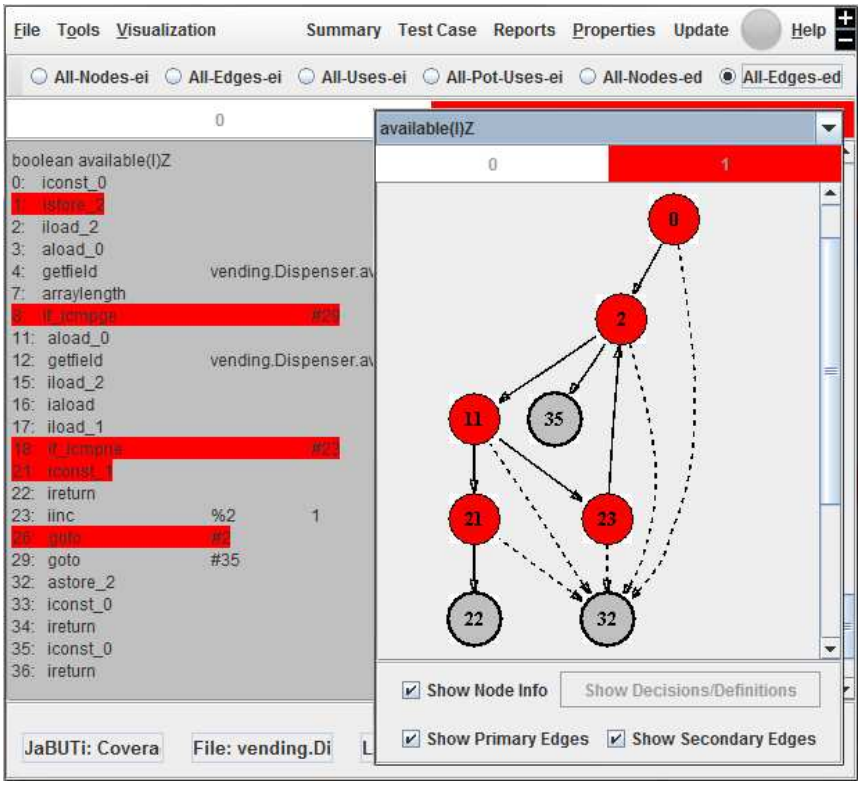


Figure 11: Color schema for All-Sec-Edges criterion: first layer.

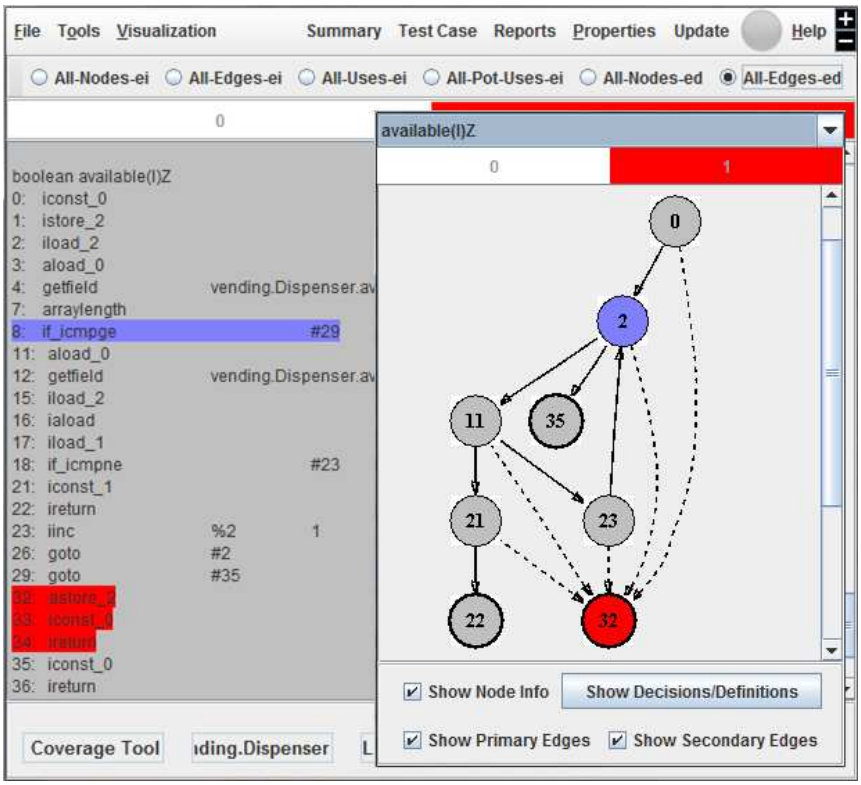


Figure 12: Color schema for All-Sec-Edges criterion: second layer.

which have at least one c-use or p-use association. Clicking on a definition goes to the second layer displaying all c-use and p-use associated with the selected definition. For each definition, its weight

is the maximum weight of its c-uses or p-uses. For example, suppose a definition has three def-use associations, one c-use and two p-uses, and the weights of these associations are 5, 3 and 7, respectively. The weight of this definition is 7. A definition has a zero weight if and only if all its c-uses and p-uses are covered. Such definitions are displayed with a white background.

Since even in the bytecode, more than one variable may be defined in the same offset, if desired, by clicking with the right mouse button over a definition point (bytecode offset, source code line or DUG node), a pop-up menu is opened showing all the variables defined in that point such that it is possible to choose which variable definition to select. For example, Figure 13 shows the set of defined variables at bytecode offset 0 that is part of the DUG node 0. Observe that in the DUG node 0 there is one additional variable definition because L@3 (val) is defined at bytecode offset 1 that also belongs to DUG node 0. When a definition point is clicked with the left mouse button the definition with the highest weight is considered selected. If all of then have the same weight, the first is selected. In our example, supposing that L@1 (credit) is selected (the definition with the highest weight), Figure 14 shows some of its uses (p-uses in this case) with the corresponding weight.

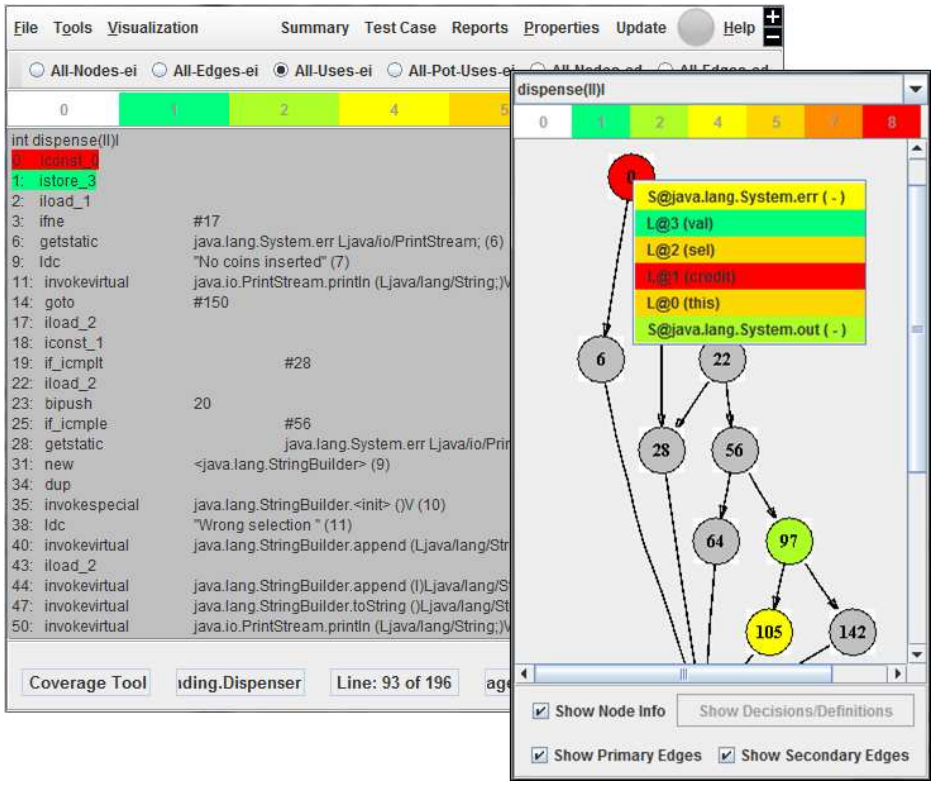


Figure 13: Color schema for All-Pri-Uses criterion: first layer.

It is important to observe that the weights provided by the tool may be seen as hints for the tester to facilitate the generation of test cases, such that a higher coverage can be obtained with fewer test cases. Since the prioritization does not consider the criticality of a given part of the code, if desired, the tester may choose a different part of the code to be tested first based on his/her knowledge about the application, even if such a part has a lower weight than other parts of the code. After the critical parts have to be tested, additional test cases can be developed considering the hints.

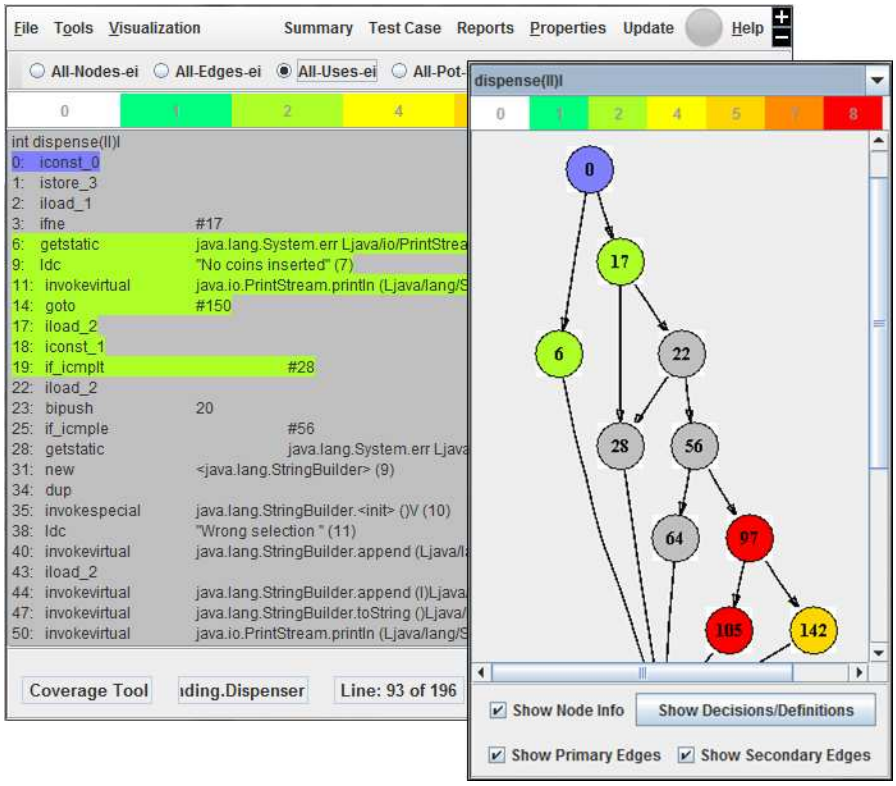


Figure 14: Color schema for All-Pri-Uses criterion: second layer.

3.2 How to generate testing reports

To evaluate the coverage obtained, the tool provides personalized tabled style testing reports that can be accessed from the Summary and Test Case menus. Any tabled style report can be saved as a HTML file by accessing Reports → Summary to HTML menu option. The tool provides reports w.r.t. each testing criterion (Summary → By Criterion), w.r.t. each class file (Summary → By Class) and w.r.t. each method (Summary → By Method). Figures 15, 16, and 17 show each one of these reports, respectively, considering that no test case have been executed.

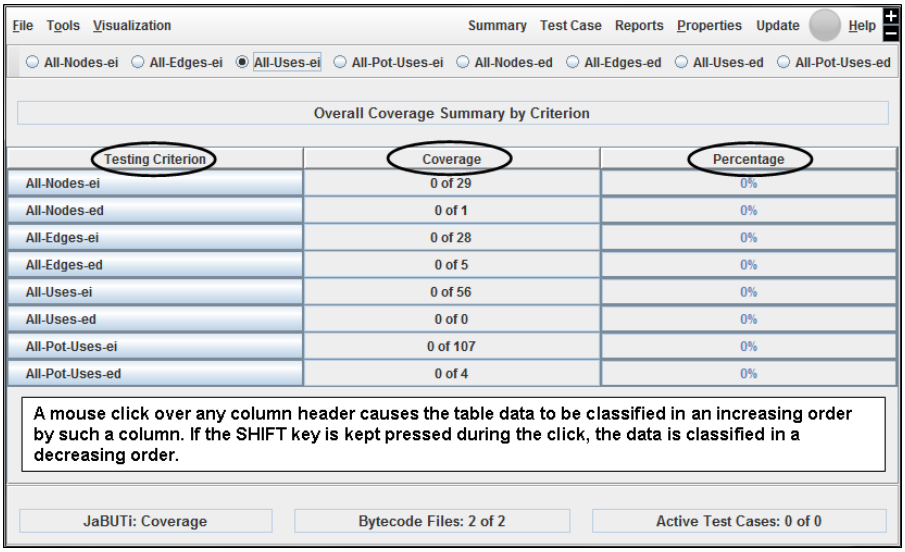


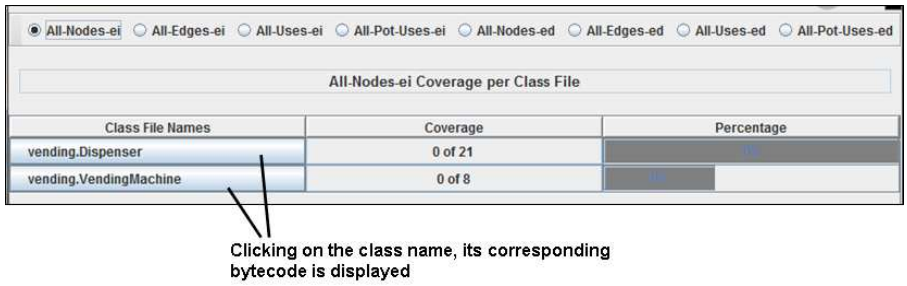
Figure 15: Initial summary information per testing criterion.

For example, in Figures 15 it is possible to evaluate the number of required elements by criterion that were generated for the entire project, i.e., classes VendingMachine and Dispenser. The individual information per class file can be obtained by generating the summary report by class (Figures 16).

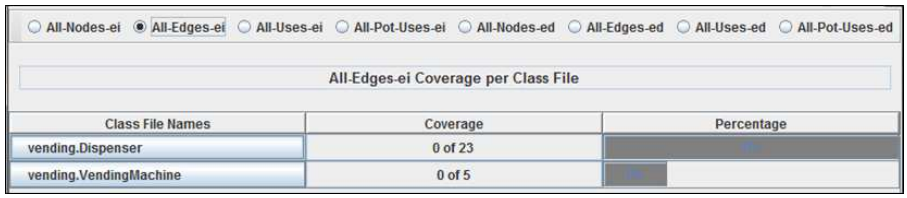
When showing the summary by class or by method, the tester can choose, among the available testing criteria, which one he/she wants to evaluate. Figures 16(a), 16(b), and 16(c) illustrate the summary per individual class w.r.t. the All-Nodes-ei, All-Edges-ei, and All-Uses-ei criteria, respectively. Considering the summary by class and by method, by clicking on the class file name or in the method name, the corresponding bytecode is highlighted and displayed considering the weight w.r.t. the selected criterion.

As commented in Figures 15, double-clicking on the desired column header, any tabled report generated by JaBUTi is classified in an increasing order, considering the clicked column header. A double-click with the SHIFT key pressed causes the table data to be sorted in a decreasing order.

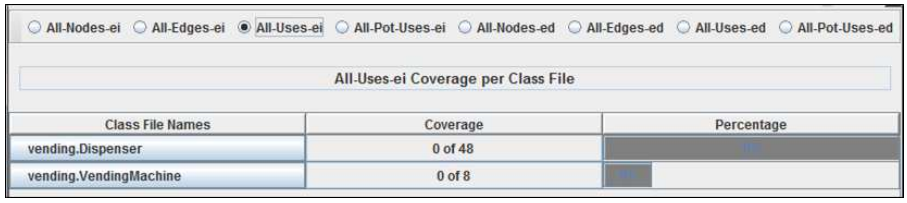
Using these summary reports the tester can evaluate the current coverage of each class under testing with different granularity and decides whether a given class requires additional testing w.r.t. six different intra-method testing criteria. Section 3.4 illustrates how to include test cases and how to generate testing reports by test case and by test case paths.



(a)



(b)



(c)

Figure 16: Initial summary per class file: (a) All-Pri-Nodes criterion, (b) All-Pri-Edges criterion, and (c) All-Pri-Uses criterion.

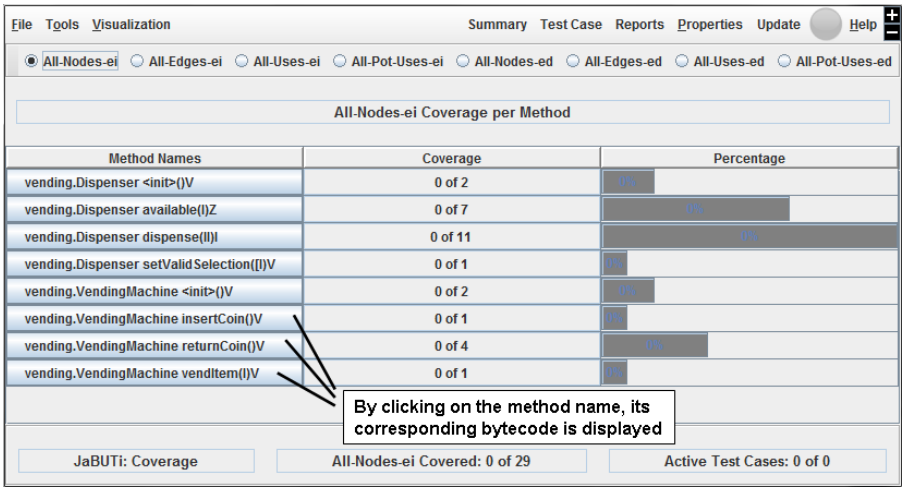
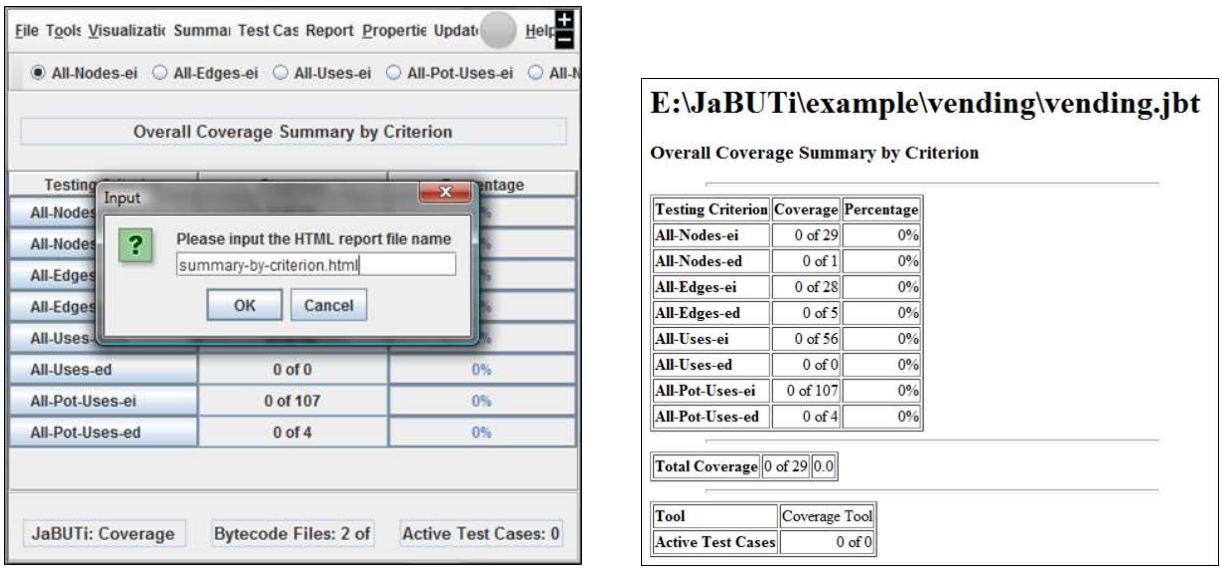


Figure 17: Initial summary information per method: All-Pri-Nodes criterion.

3.3 How to generate an HTML version of a JaBUTi report

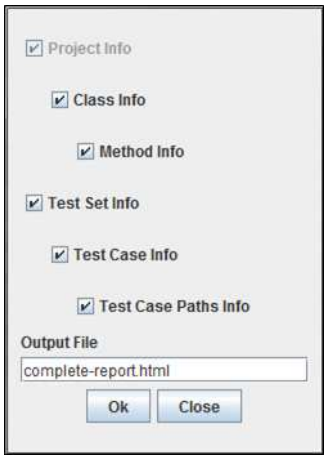
As mentioned above, any kind of tabled report presented in JaBUTi's graphical interface can be saved as an HTML file through the Reports → Summary to HTML menu option. For example, Figure 18(a) shows how to create a summary-by-criterion.html file, corresponding to the current JaBUTi screen. Figure 18(b) illustrates the generated HTML file in a browser window. In this way the tester can collect and save different testing report showing the evolution of the testing activity.



(a) (b)

Figure 18: Example of an HTML report generated by JaBUTi.

A different kind of report can be generated through the Reports → Custom Report menu option. By selecting this option, a dialog windows, as illustrated in Figure 19(a), appears and the tester can choose the level of information that will be saved in a HTML file. Observe that to save information at method level it is also required to save information about the class and project level, since the information is saved hierarchically. Part of a complete report for the entire project is presented in Figure 19(b).



(a)

Project Report: Complete

General Info

Name	E:\JaBUTi\example\vending\vending.jbt
Type	Research
Mobility	false
CFG Option	1
Base Class	vending.TestDriver
Avoided Packages	

Project Coverage

Criterion	Number Of Covered Requirements	Percentage
All-Nodes-ei	0 of 29	0.0
All-Nodes-ed	0 of 1	0.0
All-Edges-ei	0 of 28	0.0
All-Edges-ed	0 of 5	0.0
All-Uses-ei	0 of 56	0.0
All-Uses-ed	0 of 0	0.0
All-Pot-Uses-ei	0 of 107	0.0
All-Pot-Uses-ed	0 of 4	0.0

(b)

Figure 19: Example of custom HTML report generated by JaBUTi for the entire project.

3.4 How to include a test case

JaBUTi is designed to support test case set adequacy evaluation and to provide guidance in test case selection. If there is a previously developed test set, e.g. a functional test set, such a test set can be evaluated against the structural testing criteria implemented by JaBUTi. This allows the tester to check, for example, whether all instructions/statements are executed by a particular test set. If they are not yet executed, additional test cases can be developed to improve the quality of the previous test set.

On the other hand, if there is no previous test set available, the tester can use the hints provided by the tool to create test cases aiming at to cover the different testing requirements generated by JaBUTi based on its structural criteria. Once the test criteria can be applied incrementally, the tester can start by developing an adequate test set w.r.t. the All-Nodes-ei criterion, then, if necessary, to develop additional test cases to evolve the All-Nodes-ei-adequate test set to a All-Edges-ei-adequate test set, and later, if necessary, to develop additional test cases to evolve it to an All-Uses-ei-adequate test set. All these criteria, prefixed by All-Pri-, are related with the normal program execution. If desired, the tester can check the coverage of the exception-handling mechanism, by using the All-Nodes-ed, All-Edges-ed and All-Uses-ed criteria. The idea is to use these criteria incrementally to reduce their complexity and also to allow the tester to deal with the cost and time constraints.

JaBUTi allows to include test cases in two different ways: 1) using the JaBUTi's class loader; or 2) importing from a JUnit test set. The first is done by a command line application (probe.ProberLoader, the JaBUTi's class loader). This command line application extends the default Java class loader [11] such that, from a given testing project, it identifies which classes should be instrumented before to be loaded and executed. By detecting that a given class belongs to the set of classes under testing, JaBUTi's class loader inserts the probes to collect the execution trace and then loads the instrumented class file to be executed. The trace information w.r.t. the current execution is appended in a trace file with the same name of the testing project but with the extension .trc instead of .jbt.

For example, the test case file input1, presented in Figure 20, touches the testing requirement (considering the All-Nodes-ei criterion) with the highest weight illustrated in Figure 7 and also reveals a fault contained in the Dispenser component. Such a test case will be used in the next section to show how to use JaBUTi to ease the fault localization.

```
auri@AURIMRV ~/example
$ cat input1
insertCoin
vendItem 3
```

Figure 20: Example of test case file.

The command line in Figure 21(a) shows the output of the vending.TestDriver when executed by the default class loader. Figure 21(b) shows the resultant output when vending.TestDriver is executed using the JaBUTi's class loader. The execution of the command as shown in Figure 21(b) causes the vending.trc to be generated/updated with the execution path of test case file input1.

Considering Figure 21(b), observe that the CLASSPATH variable is set containing all the paths necessary to run the JaBUTi's class loader and also the vending machine example. The next parameter, probe.ProberLoader, is the name of the JaBUTi's class loader. It demands two parameters to execute: the name of the project (-P vending.jbt), and the name of the base class to be executed (vending.TestDriver). In our example, since vending.TestDriver requires one parameter to be executed,

```
auri@AURIMRV ~/example
$ java -cp "." vending.TestDriver input1
VendingMachine ON
Current value = 25
Enter 25 coins
Current value = -25
VendingMachine OFF
```

(a)

```
auri@AURIMRV ~/example
$ java -cp ".;..\Tools\jabuti;\
>..\Tools\jabuti\lib\BCEL.jar;\
>..\Tools\jabuti\lib\crimson.jar;\
> probe.ProberLoader -P vending.jbt \
> vending.TestDriver input1
Project Name: vending.jbt
Trace File Name: vending.trc
Processing File vending.jbt
VendingMachine ON
Current value = 25
Enter 25 coins
Current value = -25
VendingMachine OFF
```

(b)

Figure 21: vending.TestDriver output: (a) default class loader, and (b) JaBUTi class loader.

this parameter is also provided (input1). During the execution of vending.TestDriver, VendingMachine and Dispenser classes are required to be loaded. From the project file (vending.jbt) the JaBUTi's class loader detects that these class files have to be instrumented before to be executed. The instrumentation is performed on-the-fly every time the probe.ProberLoader is invoked.

The resultant execution trace information, corresponding to the execution of the test case input1, is appended in the end of the trace file vending.trc . Every time the size of the trace file increase, the Update button in the JaBUTi's graphical interface becomes red, indicating that the coverage information can be updated considering the new test case(s). Figure 22 illustrates this event.

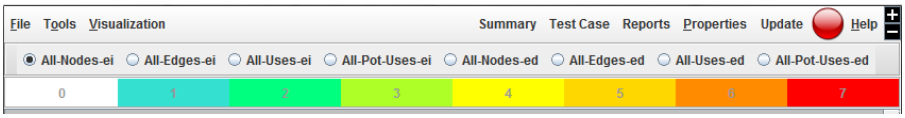


Figure 22: Update button with a different background color: new test case(s) available.

By clicking on the Update button, JaBUTi imports the new test case(s), empties the trace file vending.trc and updates the coverage information and the weights for each testing criterion. For example, after the importation of test case input1, the weight of the Java source code w.r.t. the All-Nodes-ei criterion changed as illustrated in Figure 23.

Comparing the updated source of Figure 23 (front) with the one presented in Figure 23 (back) it can be observed that the requirement with the highest changed from source lines 22 and 24 (now covered – painted in white) to source lines 16, 18, 20 and 27. Considering the entire project, Figure 24(b) shows the updated coverage for each method in the Dispenser and VendingMachine classes w.r.t. the All-Nodes-ei criterion. Observe that input1 covered 7 primary nodes (63%) of Dispenser.dispense method and 19 of 29 primary nodes (65%) considering the entire project. By accessing the Test Case → Report by Test Case menu option, the tester can visualize the coverage of the entire project by test case, as illustrated in Figure 24(a).

If desired, the tester can develop additional test cases to improve the coverage w.r.t. All-Nodes-ei criterion. For example, besides input1, Table 1 shows four additional test cases developed to improve the coverage of such a criterion.

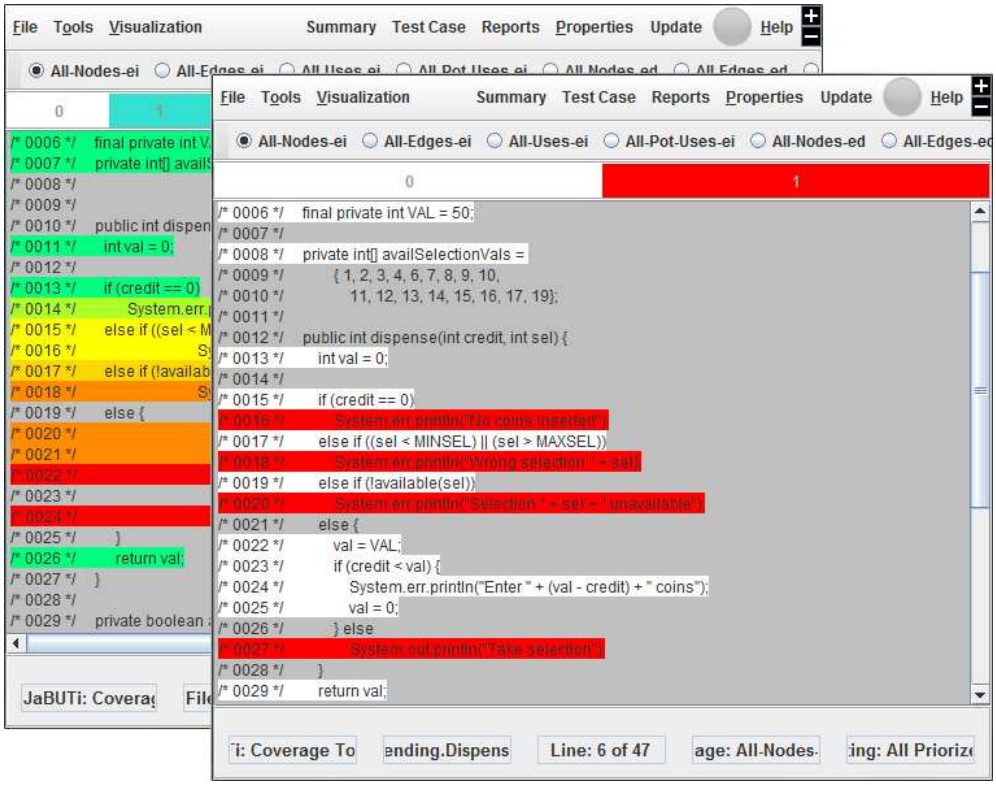
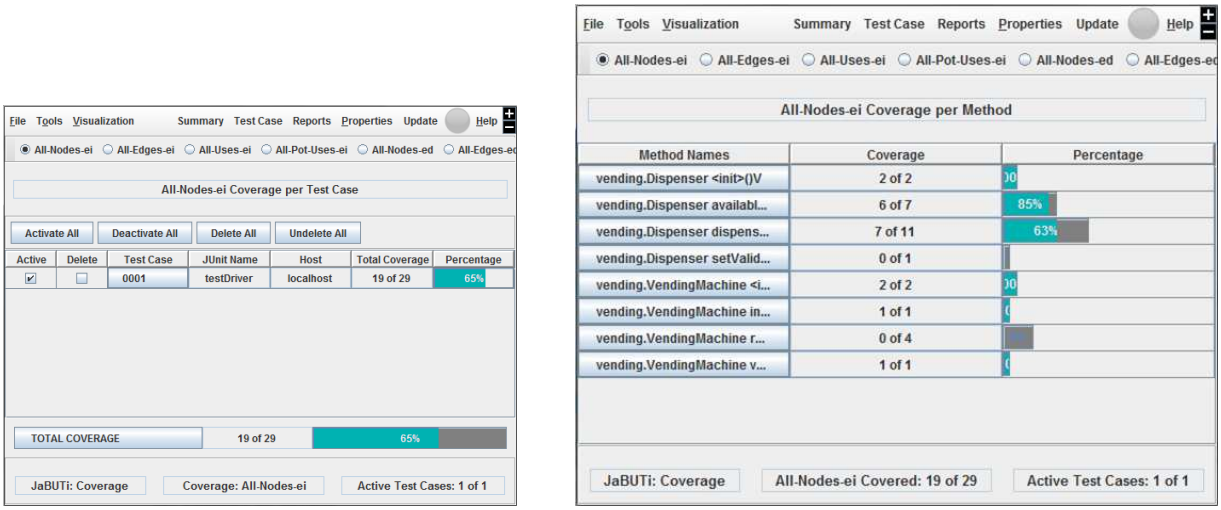


Figure 23: Dispenser.dispenser() method before and after the weight's updating w.r.t. All-Pri-Nodes criterion.



(a) Report by test case

(b) Summary by Method

Figure 24: Updated coverage after test case input1.

Table 1: Adequate test set to the Dispenser component.

Name	**Content**	**Correct output**
input1	insertCoin vendItem 3	no
input2	insertCoin vendItem 18	yes
input3	insertCoin insertCoin vendItem 25	yes
input4	vendItem 3	yes
input5	insertCoin insertCoin vendItem 3	yes

3.5 How to import test cases from JUnit framework

Observe that after five test cases, the coverage of almost all methods, w.r.t. the All-Nodes-ei criterion, is 100%. The only two methods uncovered are Dispenser.setValidSelection() and VendingMachine.returnCoin(). To obtain a adequate test set w.r.t. the All-Nodes-ei, additional test cases are required. Figure 25 shows one additional test case, specified according to the JUnit framework, to cover 100% of required elements of the All-Nodes-ei for the Dispenser class. More information about how to develop test set using the JUnit framework can be found elsewhere [3, 18].

```
import org.junit.Before;
import org.junit.Test;

import br.jabuti.probe.DefaultProber;

public class DispenserTestCase {
    protected Dispenser d;

    @Before
    protected void setUp() throws Exception {
        d = new Dispenser();
    }

    @Test
    public void testDispenserException() {
        int val;

        d.setValidSelection( null );
        val = d.dispense( 50, 10 );
        assertEquals( 0, val );
    }

    @After
    public void tearDown() {
        DefaultProber.dump();
    }

}
```

Figure 25: Example of a test set using the JUnit framework.

As can be observed in Figure 25, JaBUTi requires that a static method (DefaultProber.dump()) to be called at the end of each test case Such a method is responsible to write the execution trace in the trace file. Observe that to call such a static method it is also necessary to include the import statement for the class. Once compiled, such a JUnit test case can be imported by JaBUTi from the Test Case → Import from JUnit menu option. Figure 26 shows the screen to import a JUnit's test case.

The fields to be entered are:

- **Path to application:** automatically filled by JaBUTi.
- **Path to JUnit test suite source/binary code:** if you followed the manual it should be your current path; otherwise it's the path to your application's source and binary codes.
- **Test suite full qualified name:** the full name of the JUnit test which will be used.
- **JaBUTi's library:** the .jar used to run JaBUTi.
- **Other application specific libraries:** other libraries your application may need.
- **javac:** full path to your java compiler.
- **Trace file name:** automatically filled by JaBUTi.

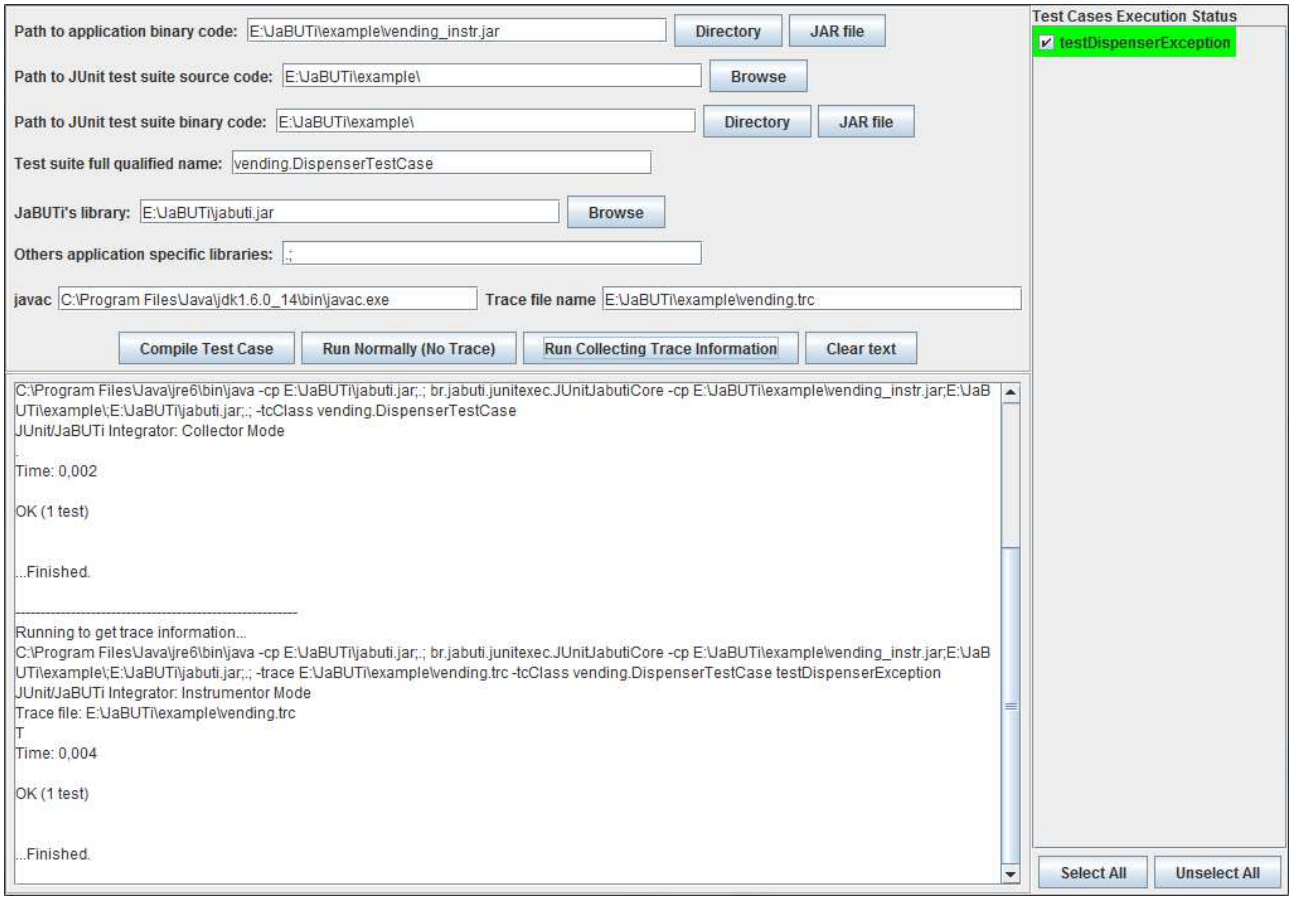


Figure 26: Importing a JUnit Test Case.

If everything is correct, you should be able to (1) compile the test case, (2) run it normally (to make sure it works) and (3) run it collecting trace information.

In our example (Figure 3.5), there is only one test case to be imported. Observe that the name of such a test case testDispenserException is highlighted in green. This is because JUnit framework provides several assertions statements, like the one at line 22 of Figure 25, that allows to verify if the test case's output is according to its specification or not. We use a green background color to represent test cases that behave accordantly to their specifications and a red background color to represent the ones that do not.

As soon as JaBUTi detects a new test case execution trace is appended in the end of the trace file, the Update button becomes red indicating that the coverage needs to be updated. Updated the coverage information, Figure 27 shows the summary report by methods and the corresponding coverage w.r.t. the All-Nodes-ei criterion. The summary report by criterion can be seen in Figure 28.

After the execution of these six test cases, all methods of the Dispenser component are 100% covered w.r.t. the All-Nodes-ei criterion, but this test set is adequate to the VendingMachine class. More specifically, the VendingMachine.returnCoin method is not covered yet. So, after six test cases, the coverage of the entire project w.r.t. All-Nodes-ei and All-Nodes-ed are 86% and 100%, respectively. The coverage for the All-Edges-ei and the All-Edges-ed are 82% and 20%, respectively, and the coverage for the All-Uses-ei and All-Uses-ed are 82% and 0%, respectively.

Independently of the number of test cases available, if desired, the tester can enable/disable any combination of test cases by accessing the Test Case → Report By Test Case menu option and choosing

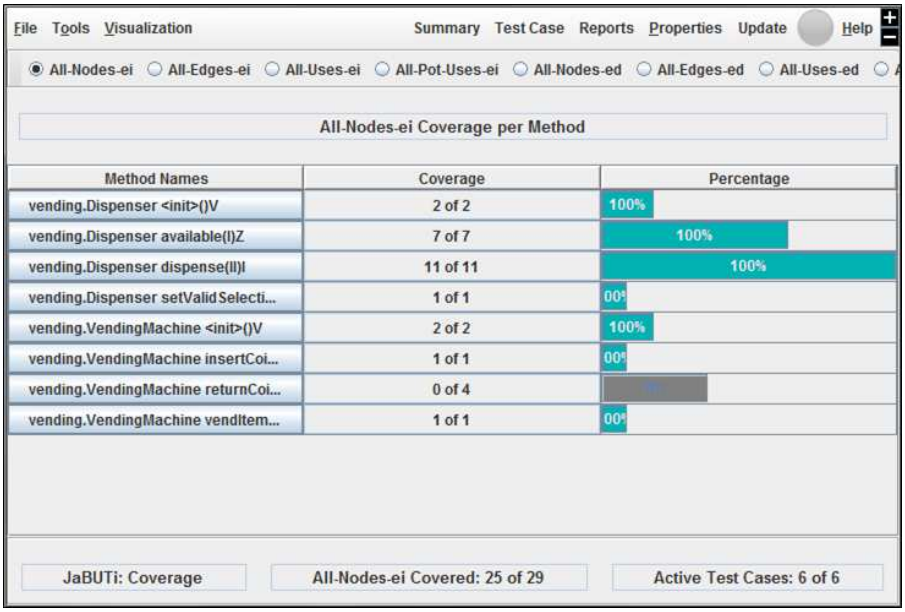
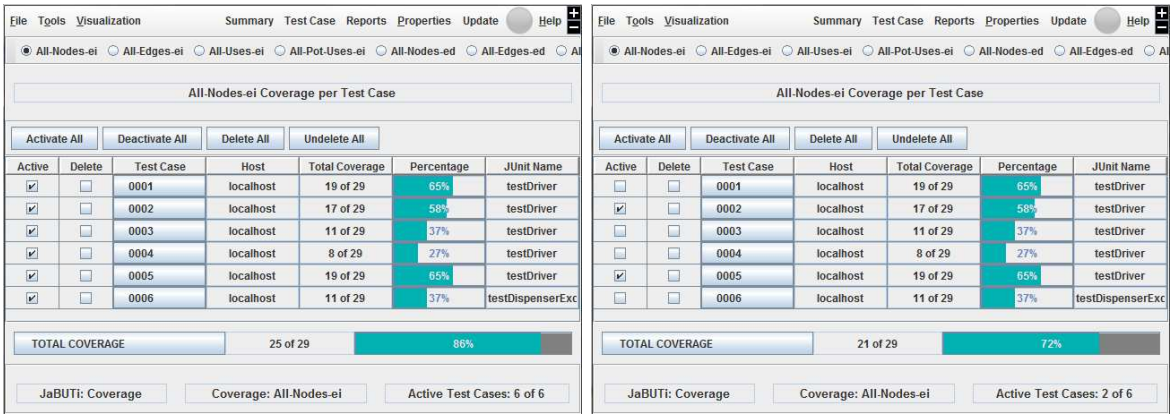


Figure 27: Updated coverage after six test cases w.r.t. the All-Pri-Nodes criterion.

Figure 28: Updated coverage after six test cases w.r.t. all criteria.

which test case should be enabled/disabled. Once a different subset of test cases are enabled/disabled all the coverage information should be updated by clicking on the Update button. Figure 29(a) shows the complete test set with all test cases enabled. Figure 29(b) shows the updated coverage considering only the test cases number 2 and 5.

Using the resource of activate/deactivate test cases, the tester can evaluate how the coverage changes, considering different combinations of test case. Moreover, besides activate/deactivate test cases, the tester can also mark a test case to be deleted. Once a test case is marked to be deleted it is only logically deleted. Such a test case is physically deleted when the project is saved and closed. While the project is not closed, a test case marked as deleted can be undeleted.



(a) (b)

Figure 29: Summary by test case: (a) all enabled, and (b) 2 and 5 enabled.

3.6 How to mark a testing requirement as infeasible

When applying structural testing criteria, one problem is to identify infeasible testing requirements since, in general, it is an undecidable problem. Vergilio [25] developed some heuristics to automatically determine infeasible testing requirements but such heuristics are not yet implemented in JaBUTi. Therefore, it is a responsibility of the tester to identify such infeasible testing requirements.

By accessing the Visualization → Required Elements menu option, JaBUTi allows the tester to visualize and mark testing requirements as infeasible, as well as activate/deactivate testing requirements. Once a testing requirement is marked as infeasible or deactivated, the Update button becomes red indicating that the coverage information should be updated to reflect such a change. For example, Figure 30 shows part of the testing requirements for the method Vending.returnCoin() method, considering the All-Nodes-ei criterion.

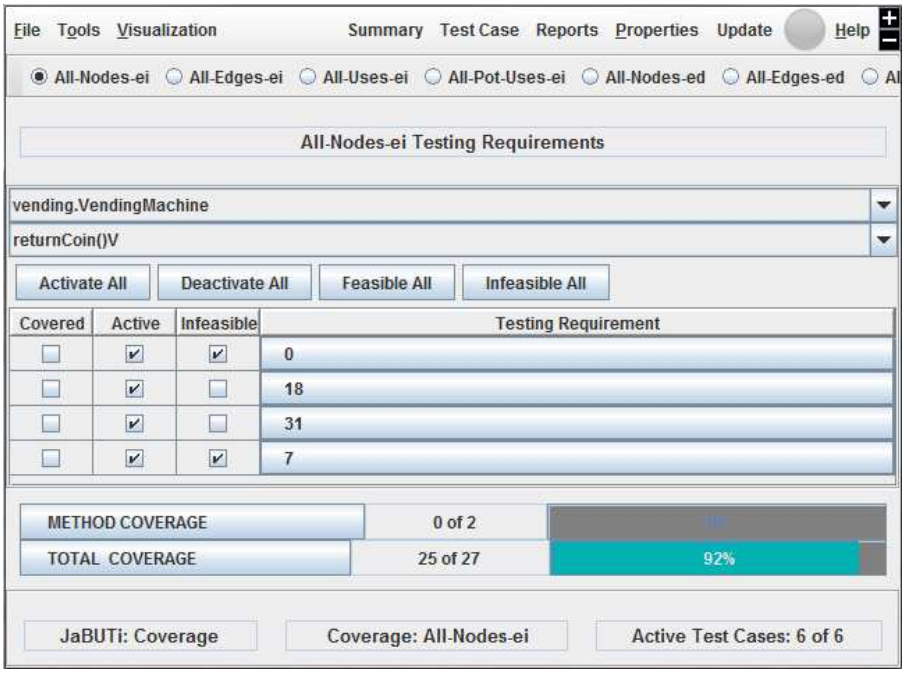


Figure 30: Dispenser.dispense() required elements for the All-Pri-Nodes criterion.

Considering Figure 30, two uncovered requirements were marked as infeasible, primary nodes 0 and 7. Since the tester can mark erroneously a testing requirement as infeasible, in case such a testing requirement is covered in the future by an additional test case, the tool indicates such a inconsistency, highlighting the infeasible check box of a covered testing requirement in red, as illustrated in Figure 31. In this case, primary node 0 is covered by the new test case added to exercise the Vending.returnCoin() method, so node 0 is feasible.

During the importation of a given test case, the tester has to check whether the obtained output is correct w.r.t. the specification. As can be observed in Table 1, only test case number 1 does not behave according to the specification. Once any discrepancy is detected, the fault should be localized and corrected. One alternative for fault localization is slicing. In the next section we describe how to use the slicing tool implemented in JaBUTi to help the fault localization and smart debugging.

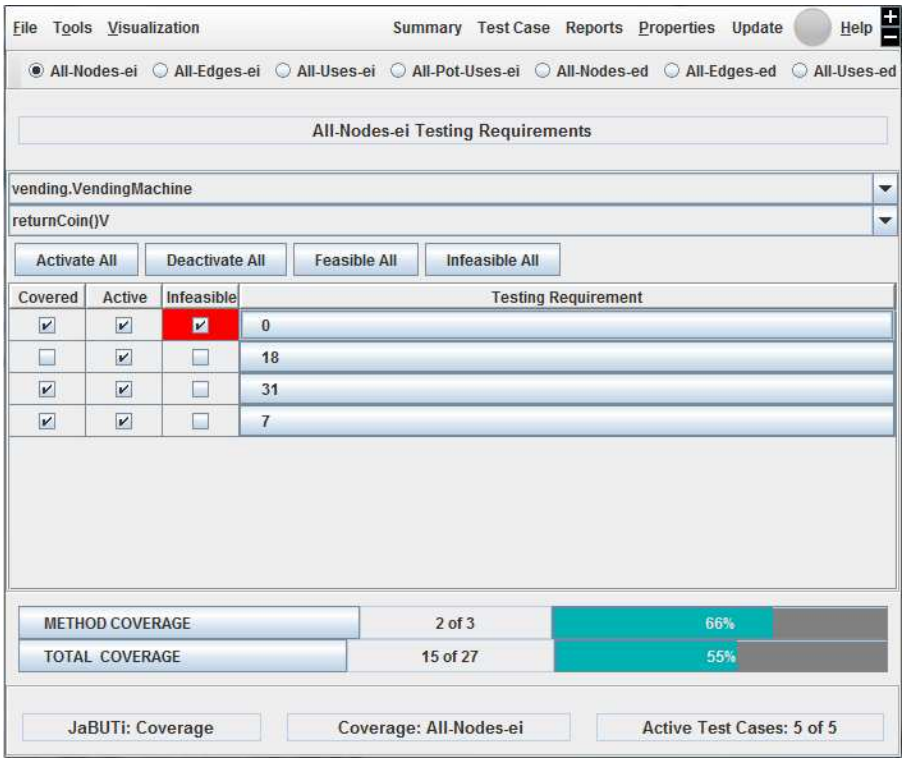


Figure 31: Dispenser.dispense() infeasible requirements erroneously identified.

4 How to use JaBUTi as a Slicing Tool

The JaBUTi slicing tool is available through the Tools → Slicing Tool menu option. The slicing tool implements only a simple heuristic based on control-flow information. As a future work, additional heuristics will be implemented considering not only control-flow but also data-flow information.

By changing to the slicing tool, the tester has to choose, among the test cases, the ones that cause the fault and the ones that do not reveal the fault. Based on the execution path of the failed and succeed test cases, the tool highlights the part of the code more probable to contain the fault.

JaBUTi uses a simple dynamic slice criterion, based on control-flow information, to identify a subset of statements more probable to contain the fault. The idea is (1) to compute the failed set F_S of $\mathcal{BG}$ nodes (the execution path) of a failed test case, which includes all statements executed by the failed test case, (2) to compute the succeed set S_S of $\mathcal{BG}$ nodes considering succeed test case(es), and (3) to calculate the difference and the intersection of these sets to prioritize the statements executed by the failed test case.

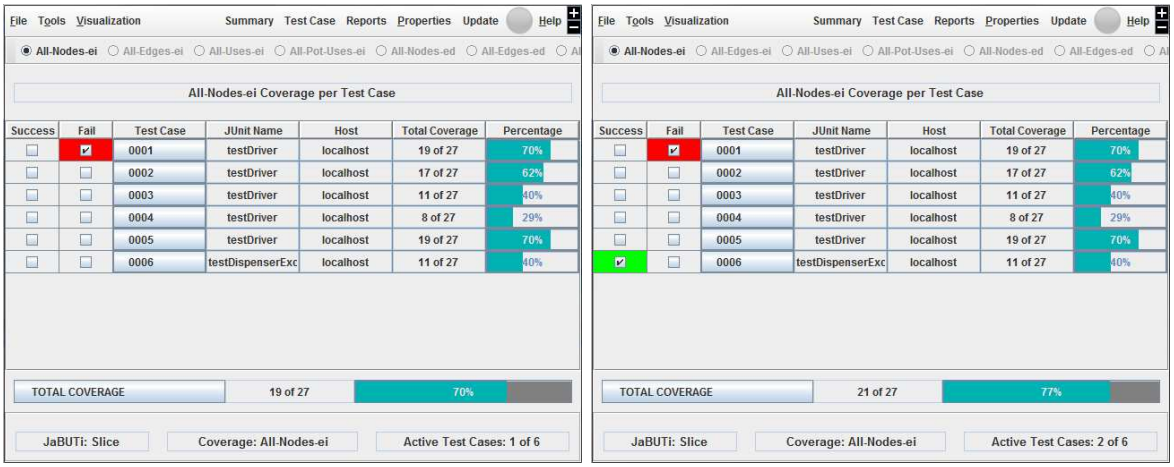
Using such an approach, instead of the complete set of $\mathcal{BG}$ nodes N (which represents all statements of a method), the tester has only to consider the subset of $\mathcal{BG}$ nodes present in F_S, since the other $\mathcal{BG}$ nodes contains the statements not executed by the failed test case and cannot contain the fault. Moreover, considering the subset of nodes executed by the succeed test cases, the most probably location of the fault is in the statements executed by the failed test case but not executed by the succeed test cases, i.e., the subset $F_S \setminus S_S$. Thus, such a subset of statements has a highest priority and should be analyzed first. If the fault is not located on this subset, due to data dependence that can lead to a data-flow dependent fault, the tester has to analyze the other statements executed by both the failed and by the succeed test cases, i.e., the subset $F_S \cap S_S$. In this way, we can provide an incremental search for the fault, trying to reduce the time and the cost to locate the fault in a program.

For example, consider the $\mathcal{BG}$ presented in Figure **??**. N is the complete set of $\mathcal{BG}$ nodes ($N = \{0, 15, 34, 43, 54, 54.82, 60, 60.82, 74, 74.82, 79, 91, 97\}$). Suppose a failed test case that goes though $\mathcal{BG}$ nodes $F_S = \{0, 34, 15, 34, 43, 54, 54.82, 91, 97\}$ and a successful test case that goes trough $\mathcal{BG}$ nodes $S_S = \{0, 34, 43, 60, 60.82, 91, 97\}$. The most probably locations for the fault are in the statements in nodes 15, 54 or 54.82, since they are only executed by the failure test case ($F_S \setminus S_S$). If the fault is not located on such statements, it will be found in the other statements that compose the $\mathcal{BG}$ nodes 0, 34, 43, 91 and 97 ($F_S \cap S_S$). All the other $\mathcal{BG}$ nodes have not to be analyzed.

The approach described above is based only on control-flow information. The success of such an approach depends on which test cases are selected as the successful test cases. It is important to select successful test cases that execute the related functionalities of the program as the ones executed by the failure test case. In this way, the difference between the two sets F_S and S_S will result in subset with a few $\mathcal{BG}$ nodes, reducing the number of statements that have to be analyzed first.

In our Vending Machine example, test case 0001 reveals the fault and test case 0006 does not. The tester can first select only the failed test case, as illustrated in Figure 32(a), and checks the execution path of this test case as illustrated in Figure 33(a). The red lines indicate the statements that are executed by the failed test case and the white lines indicates the statements not executed. Since there is a fault, it must be located among these red statements.

After having observed the failed test case execution path, the tester can enable one or more additional succeed test cases (Figure 32(b)), such that the tool can identify, among the statements



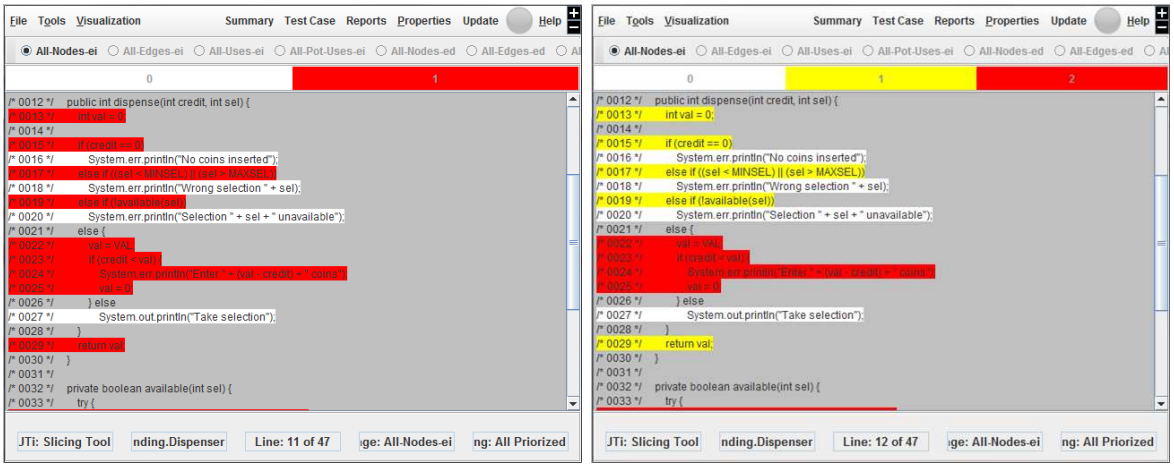
(a) (b)

Figure 32: Slicing tool test case selection: (a) only the failed test case, and (b) a failed and a succeed test cases.

executed ny the failed test case, the dice more probable to contain the fault (statements only executed by the failed test case) and the ones less probable (the ones executed for both the failed and the succeed test cases.) Figure 33(b) shows in yellow (weight 1) the statements touched by test cases number 0001 and 0006 and in red (weight 2) the statements only executed by test case number 0001.

The rationale of such an approach is that, in theory, the fault is more probable to be located at the statements executed only by the failed test case. Although, it can happen that the fault is localized in other than the red statements, but the red points can be used at least as a good starting point, trying to reduce the search space for fault localization.

Once the fault is located and corrected, the testing activity is restarted by creating a new project to revalidate the corrected program. In this case, previous test sets can be used to revalidate the behavior of the new program and also to check if new faults were not introduced by the changes.



(a) (b)

Figure 33: Highlighted execution path: (a) fail test case execution path, and (b) fail test case execution path intersected by the success test case execution path.

5 How to use the JaBUTi's Static Metrics Tool

Adicionalmente, na Seção 5.3, a métrica que avalia a complexidade ciclomática de McCabe [17] de cada método também foi implementada para identificar os métodos de "maior" complexidade dentro de uma dada classe.

5.1 LK's Metrics Applied to Classes

5.1.1 NPIM – Number of public instance methods in a class

Métrica calculada através da contagem do número de métodos de instância públicas na classe.

5.1.2 NIV – Number of instance variables in a class

Métrica calculada através da contagem do número de variáveis de instância na classe, o que inclui as variáveis *public*, *private* e *protected* disponíveis para as instâncias.

5.1.3 NCM – Number of class methods in a class

Métrica calculada através da contagem do número de métodos *static* na classe.

5.1.4 NCV – Number of class variables in a class

Métrica calculada através da contagem do número de variáveis *static* na classe.

5.1.5 ANPM – Average number of parameters per method

Métrica calculada através da divisão entre o somatório do número de parâmetros de cada método da classe pelo número total de métodos da classe.

Variação: número máximo de parâmetros em um método da classe.

5.1.6 AMZ – Average method size

Métrica calculada através da divisão entre a soma do número de linhas de código dos métodos da classe pelo número de métodos na classe (soma dos métodos instância e classe).

Variações: média do número de instruções do bytecode e tamanho do bytecode.

5.1.7 UMI – Use of multiple inheritance

Como a herança múltipla não se aplica à linguagem JAVA será definida uma variação à esta métrica.

Variação: número de interfaces implementadas pela classe (*Number of interfaces implemented* - NINI).

5.1.8 NMOS – Number of methods overridden by a subclass

Métrica calculada através da contagem do número de métodos definidos na subclasse com o mesmo nome de métodos de sua superclasse.

5.1.9 NMIS – Number of methods inherited by a subclass

Métrica calculada através da contagem do número de métodos herdados pela subclasse de suas superclasses.

5.1.10 NMAS – Number of methods added by a subclass

Métrica calculada através da contagem do número de novos métodos adicionados pelas subclasses.

5.1.11 SI – Specialization index

Métrica calculada através da divisão entre o resultado da multiplicação de NMOS e DIT (métrica de CK) pelo número total de métodos.

5.2 CK's Metrics Applied to Classes

5.2.1 NOC – Number of Children

Métrica calculada através da contagem do número de subclasses imediatas subordinadas à classe na árvore de hierarquia.

5.2.2 DIT – Depth of Inheritance Tree

É o maior caminho da classe à raiz na árvore de hierarquia de herança. Interfaces também são consideradas, ou seja, o caminho através de uma hierarquia de interfaces também pode ser o que dá a profundidade de uma classe.

Variação: como a representação de programa utilizada não inclui todas as classes até a raiz da árvore de hierarquia, será utilizado o caminho da classe até a primeira classe que não pertence à estrutura do programa.

5.2.3 WMC – Weighted Methods per Class

Métrica calculada através da soma da complexidade de cada método. Não se define qual tipo de complexidade pode ser utilizada, assim serão aplicadas as seguintes variações:

- Utiliza-se o valor 1 como complexidade de cada método; assim WMC_1 é o numero de métodos na classe;
- Utiliza-se a métrica CC para a complexidade de cada método; esta métrica será chamada WMC_CC;
- Utiliza-se a métrica LOCM para a complexidade de cada método; esta métrica será chamada WMC_LOCM;
- Utiliza-se o tamanho do método (número de instruções) para a complexidade de cada método; esta métrica será chamada WMC_SIZE;

5.2.4 LCOM – Lack of Cohesion in Methods

Métrica calculada através da contagem do número de pares de métodos na classe que não compartilham variáveis de instância menos o número de pares de métodos que compartilham variáveis de instância. Quando o resultado é negativo, a métrica recebe o valor zero. Os métodos estáticos não são considerados na contagem, uma vez que só as variáveis de instância são tomadas.

Variações:

- Considerar só a coesão entre métodos estáticos; esta métrica será chamada LCOM_2;
- Considerar a coesão de métodos estáticos ou de instância; esta métrica, chamada LCOM_3 pode ser calculada como $LCOM\_ - LCOM\_2$;

5.2.5 RFC – Response for a Class

Métrica calculada através da soma do número de métodos da classe mais os métodos que são invocados diretamente por eles. É o número de métodos que podem ser potencialmente executados em resposta a uma mensagem recebida por um objeto de uma classe ou por algum método da classe. Quando um método polimórfico é chamado para diferentes classes, cada diferente chamada é contada uma vez.

5.2.6 CBO – Coupling Between Object

Há acoplamento entre duas classes quando uma classe usa métodos e/ou variáveis de instância de outra classe. Métrica calculada através da contagem do número de classes às quais uma classe está acoplada de alguma forma, o que exclui o acoplamento baseado em herança. Assim, o valor CBO de uma classe A é o número de classes das quais a classe A utiliza algum método e/ou variável de instância.

5.3 Another Metrics Applied to Methods

5.3.1 CC – Cyclomatic Complexity Metric

Métrica que calcula a complexidade do método, através dos grafos de fluxo de controle que descreve a estrutura lógica do método.

Os grafos de fluxo consistem de nós e ramos, onde os nós representam comandos ou expressões e os ramos representam a transferência de controle entre estes nós.

A métrica pode ser calculada das seguintes formas:

- O número de regiões do grafo de fluxo corresponde à Complexidade Ciclomática;
- A Complexidade Ciclomática, V(G), para o grafo de fluxo G é definida como:

$$V(G) = E - N + 2$$

  onde, V(G) mede os caminhos linearmente independentes encontrados no grafo, E é o número de ramos do grafo de fluxo e N o número de nós.

- A Complexidade Ciclomática, V(G), para o grafo de fluxo G é definida como:

$$V(G) = P(G)+1$$

onde, P(G) é o número de nós predicativos contidos no grafo de fluxo G. Os nós predicativos são comandos condicionais (*if*, *while*, ...) com um ou mais operadores booleanos (*or*, *and*, *nand*, *nor*). Um nó é criado para cada nó 'a' ou 'b' de um comando `IF a OR b`.

Outras formas de se calcular a Complexidade Ciclomática são:

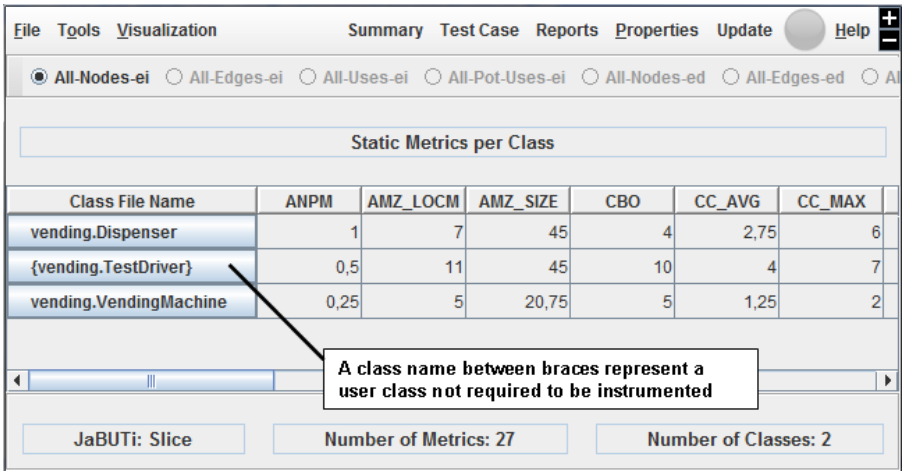
- V(G) = número de regiões de G;
- V(G) = número de ramos - número de nós + 2;
- V(G) = número de nós predicativos + 1.

Variações: a métrica é aplicada a métodos mas pode ser aplica à classe através da soma (ver WMC_CC), da média (que será chamada CC_AVG) e do máximo (CC_MAX) entre os métodos.

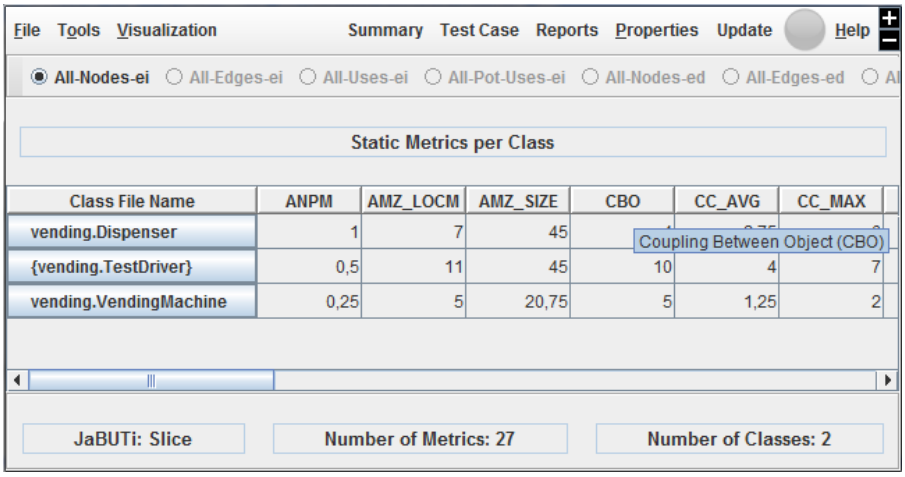
5.4 The Static Metrics GUI

The set of static metrics implemented in JaBUTi can be accessed by the Visualization → Complexity Metrics menu option. Figure 34(a) illustrates the window that will be displayed, considering the vending.jbt project file. Each line corresponds to a given user class and each column is the resultant value of a given static metric applied to such a class. Observe that not only the classes under testing are displayed but also the other user classes required by the base class. A user class not required to be instrumented appears between braces. The idea is to allow the tester to visualize the metrics with respect to all possible classes that can be instrumented such that he/she can decide later to select such classes to be tested.

By moving the mouser cursor onto the head of a given column, it is displayed a tip, giving a brief description of the corresponding metric of such a column. For example, Figure 34(b) shows the tip for the CBO metrics, responsible to measure the coupling between objects.



(a) Metrics view



(b) Metrics tool tip

Figure 34: Static Metrics Tool's graphical interface.

6 License

GNU Free Documentation License
Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the

publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of

the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering

more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified

   Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for
   public access to a Transparent copy of the Document, and likewise
   the network locations given in the Document for previous versions
   it was based on. These may be placed in the "History" section.
   You may omit a network location for a work that was published at
   least four years before the Document itself, or if the original
   publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications",
   Preserve the Title of the section, and preserve in the section all
   the substance and tone of each of the contributor acknowledgements
   and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document,
   unaltered in their text and in their titles. Section numbers
   or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section
   may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements"
   or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or
appendices that qualify as Secondary Sections and contain no material
copied from the Document, you may at your option designate some or all
of these sections as invariant. To do this, add their titles to the
list of Invariant Sections in the Modified Version's license notice.
These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains
nothing but endorsements of your Modified Version by various
parties--for example, statements of peer review or that the text has
been approved by an organization as the authoritative definition of a
standard.

You may add a passage of up to five words as a Front-Cover Text, and a
passage of up to 25 words as a Back-Cover Text, to the end of the list
of Cover Texts in the Modified Version. Only one passage of
Front-Cover Text and one of Back-Cover Text may be added by (or
through arrangements made by) any one entity. If the Document already
includes a cover text for the same cover, previously added by you or
by arrangement made by the same entity you are acting on behalf of,
you may not add another; but you may replace the old one, on explicit
permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License
give permission to use their names for publicity for or to assert or
imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this
License, under the terms defined in section 4 above for modified
versions, provided that you include in the combination all of the

Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the

GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

    Copyright (c) YEAR YOUR NAME.

References

[1] H. Agrawal. Dominators, super block, and program coverage. In *SIGPLAN – SIGACT Symposium on Principles of Programming Languages – POPL'94*, pages 25–34, Portland, Oregon, January 1994. ACM Press.

[2] H. Agrawal, J. Alberi, J. R. Horgan, J. Li, S. London, W. E. Wong, S. Ghosh, and N. Wilde. Mining system tests to aid software maintenance. *IEEE Computer*, 31(7):64–73, July 1998.

[3] K. Beck and E. Gamma. JUnit cookbook. web page, 2002. Available on-line: `http://www.junit.org/` [01-20-2003].

[4] S. R. Chidamber and C. F. Kemerer. A metric suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[5] M. Dahm. Byte code engineering with the BCEL API. Technical Report B-17-98, Freie Universität Berlin – Institut für Informatik, Berlin – German, April 2001. Available on-line at: `http://bcel.sourceforge.net/` [04-13-2002].

[6] Inc. GrammaTech. Dependence graphs and program slicing. White Paper, March 2000. Available on-line: `http://www.grammatech.com/research/`.

[7] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. In *Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 154–163, New York, NY, December 1994. ACM Press.

[8] P. M. Herman. A data flow analysis approach to program testing. *Australian Computer Journal*, 8(3):92–96, November 1976.

[9] J. R. Horgan and S. A. London. Data flow coverage and the C language. In *Symposium Software Testing, Analysis, and Verification*, pages 87–97, Victoria, British Columbia, Canada, October 1991. ACM Press.

[10] ILOG, Inc. Ilog jviews component suite. web page, 2003. `http://www.ilog.com/products/jviews/`.

[11] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2 edition, 1999.

[12] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice Hall, Englewood Cliffs, New Jersey, 1994. Object-Oriented Series.

[13] J. C. Maldonado. *Potential-Uses Criteria: A Contribution to the Structural Testing of Software*. Doctoral dissertation, DCA/FEE/UNICAMP, Campinas, SP, Brazil, July 1991. (in Portuguese).

[14] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[15] A. Orso, M. J. Harrold, D. Rosenblum, G. Rothermel, H. Do, and M. L. Soffa. Using component metacontent to support the regression testing of component-based software. In *IEEE International Conference on Software Maintenance (ICSM'01)*, pages 716–725, Florence, Italy, November 2001. IEEE Computer Society Press.

[16] P. Piwowarski, M. Ohba, and J. Caruso. "Coverage measurement experience during function test". In *Proceedings of the 15th International Conference on Software Engineering*, pages 287–301, Baltimore, MD, May 1993.

[17] R. S. Pressman. *Software Engineering – A Practitioner's Approach*. McGraw-Hill, 5 edition, 2000.

[18] J. B. Rainsberger. JUnit: A starter guide. Web Page, 2003. Available on-line: `http://www.diasparsoftware.com/articles/JUnit/jUnitStarterGuide.html`.

[19] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.

[20] M. Roper. *Software Testing*. McGrall Hill, 1994.

[21] S. Sinha and M. J. Harrold. Analysis of programs with exception-handling constructs. In *ICSM'98 – International Conference on Software Maintenance*, pages 348–357, Bethesda, MD, November 1998.

[22] S. Sinha and M. J. Harrold. Criteria for testing exception-handling constructs in Java programs. In *International Conference on Software Maintenance*, pages 265–274, Oxford, England, August 1999. IEEE Computer Society Press.

[23] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[24] C. T. Utsonomia. Estudo e aplicação de métricas para sistemas orientados a objeto. Exame geral de qualificação, Universidade Estadual do Parana, Departamento de Informática, Curitiba, PR, August 2002.

[25] S. R. Vergilio. *Critérios Restritos: Uma Contribuição para Aprimorar a Eficácia da Atividade de Teste de Software*. PhD thesis, DCA/FEEC/UNICAMP, Campinas, SP, July 1997.

[26] A. M. Vincenzi, W. E. Wong, M. E. Delamaro, A. S. Simão, and J. C. Maldonado. Structural testing of object-oriented programs. Technical report, ICMC/USP, 2003. (in preparation).

[27] A. M. R. Vincenzi, M. E. Delamaro, J. C. Maldonado, and W. E. Wong. Java bytecode static analysis: Deriving structural testing requirements. In *2nd UK Software Testing Workshop – UK-Softest'2003*, pages –, Department of Computer Science, University of York, York, England, September 2003. University of York Press. (accepted for publication).

[28] M. Weiser. *Program Slicing: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Method*. Phd thesis, The University of Michigan, Ann Arbor , Michigan, 1979.

[29] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

[30] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[31] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set size and block coverage on fault detection effectiveness. In *Fifth IEEE International Symposium on Software Reliability Engineering*, pages 230–238, Monterey, CA, November 1994.

[32] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. "Effect of test set minimization on fault detection effectiveness". *Software-Practice and Experience*, 28(4):347–369, April 1998.

[33] J. Zhao. Analyzing control flow in Java bytecode. In *16th Conference of Japan Society for Software Science and Technology*, pages 313–316, September 1999.

[34] J. Zhao. Dependence analysis of Java bytecode. In *24th IEEE Annual International Computer Software and Applications Conference (COMPSAC'2000)*, pages 486–491, Taipei, Taiwan, October 2000. IEEE Computer Society Press.