SCOTCH: Slicing and Coupling based Test to Code trace Hunter

Abdallah Qusef¹, Gabriele Bavota¹, Rocco Oliveto², Andrea De Lucia¹, David Binkley³

¹University of Salerno, Fisciano (SA), Italy
²University of Molise, Pesche (IS), Italy
³Loyola University Maryland, Baltimore, USA

aqusef@unisa.it, gbavota@unisa.it, rocco.oliveto@unimol.it, adelucia@unisa.it, binkley@cs.loyola.edu

Abstract—In this demonstration we present an Eclipse plug-in, called SCOTCH (Slicing and COupling based Test to Code trace Hunter), that uses dynamic slicing and conceptual coupling to recover the traceability links between unit tests and tested classes. Empirical evaluation showed that SCOTCH identifies traceability links between unit-tests and tested classes with a high accuracy.

Index Terms—Slicing, Conceptual Coupling, Traceability.

I. INTRODUCTION

In agile development, unit tests are written by developers to help understand a problem and, when using test first design, they are written ahead of any new code. In addition, to maintain an effective regression suite, unit tests are continually updated to reflect changes in the production code [1]. This makes unit tests an important source of documentation, especially when performing maintenance tasks where they can help a developer to comprehend production code as well as identify failures. Beyond comprehension and maintenance, traceability information can play an important role in preserving consistency during refactoring. Indeed, code refactoring is followed by refactoring of the related unit tests [2], [3]. All these activities are easily supported when traceability links between unit tests and tested classes are explicitly maintained. However, in the absence of such links these tasks become considerably more difficult.

We demonstrate SCOTCH (Slicing and COupling based Test to Code trace Hunter), an Eclipse plug-in that supports the developer in the identification of dependencies between unit tests and tested classes. The traceability recovery process behind SCOTCH is based on the following assumptions:

1) unit tests are classes that implement a test suite and the unit-test methods implement the different test cases of the test suite [5]. In each unit-test method, the actual outcome is compared to the expected outcome (oracle) using an assert statement. Unfortunately, assert statements are also used to verify the testing environment before verifying the actual outcome [6]. Since each method of the unit test implements a specific test case, we consider only the last assert statement of each unit test method, we conjecture that such a statement is specialised to verify the behavior of the tested classes [4];

2) unit tests are related to different types of classes, i.e., mock objects, helper classes, and tested classes [6].

Mock objects and helper classes are used to create the testing environment, while tested classes are the actual classes under test. We conjecture that the textual content of the unit tests is closer to the tested classes than to the helper classes or mock objects.

Based on these assumptions, SCOTCH uses Dynamic Slicing [7] to identify all the classes that affect the results of the last assert statement in each unit-test method. Moreover, SCOTCH exploits Conceptual Coupling [8] to discriminate between the actual tested classes and the helper classes.

II. SCOTCH OVERVIEW

In this section we explain the traceability recovery engine behind SCOTCH and its integration in the Eclipse IDE.

A. Traceability Recovery Engine

SCOTCH identifies the set of tested classes using a two steps process. The first step exploits dynamic slicing to identify an initial set of candidate tested classes, called the Starting Tested Set (STS). In particular, the STS is computed by slicing the last assert statement in each method of a unit test class. Previous studies indicate that since a unit-test method implements a test case, generally the results of the last assert statement in a unit-test method are affected by methods belonging to a tested class [4]. These considerations suggest using the last assert statement in a unit test method as the slicing criterion. In particular, we employ backward dynamic slicing [7] to identify an initial set of classes by finding all the method invocations that affect the last assert statement in each unit-test method. However, the set of classes identified using dynamic slicing may include many helper classes as well as classes belonging to mock objects or instances of classes from standard libraries (e.g., String). The latter are removed from the STS through a stop-class list, while the helper classes and mock objects are identified and pruned-out from the STS in the second step of the process.

In this second step, the STS is filtered exploiting the conceptual coupling between the identified classes and the unit test under analysis resulting in the Candidate Tested Set (CTS). We conjecture that the unit test should be semantically related to the classes under test (in particular, their textual similarity should be higher than the textual similarity between unit tests and helper classes, which are used more uniformly across all
(SCOTCH) provides a checkbox that allows the developer to trace (or not) the link between the unit test end the candidate tested class. For each unit test, SCOTCH shows the classes belonging to its STS as well as to its CTS. The checkbox related to the classes in the CTS are automatically checked to suggest to the developer which are (based on our approach) the tested classes. Once the developer has finished his or her classification, the traced links are stored in an XML file.

The stored traceability information can be used to (i) help the user to identify all tested classes for a given a unit test and (ii) to alert the developer when he or she modifies, e.g., refactorers, a class related to one or more unit test. In this case SCOTCH shows a warning message suggesting that the developer check if the unit test should be modified consistently with the changes applied to its tested class (see Figure 2).

III. Conclusion

In this demonstration we presented an Eclipse plug-in, named SCOTCH that supports a developer in the recovery of traceability links between unit tests and tested classes. In prior work [10], we compared the accuracy of SCOTCH with approaches based on naming conventions (NC) [11], Last Call Before Assert (LCBA) [11], and data-flow analysis (DFA) [4]. The results indicate that SCOTCH is not only the most accurate, but that it has the highest stability over the experimented systems as well. These results highlight its usefulness as a feature within a software development environment such as Eclipse.

REFERENCES

APPENDIX A
DEMO OVERVIEW

This research tool demo will present the critical aspects of the functionality provided by SCOTCH. We will first demonstrate the traceability recovery functionality and then explain the fundamental underlying concepts of the tool. We demo the functionality of SCOTCH using the AgilePlanner system\(^1\).

The demo will then show how the traceability information can be used by SCOTCH to highlight inconsistency between the production code and related unit test. The demo will conclude by addressing any questions raised by the audience.

A. Recovering Traceability Links

To demo the traceability recovery functionality of SCOTCH we consider the unit test `testSendFromServerToOnlyOneClient` from the class `NetworkCommunicationTest` shown in Figure 3. In order to identify the class(es) tested by such a unit test, the developer selects this unit test in the Eclipse Package Explorer and presses a button in the view to start the recovery process. Note that SCOTCH also supports performing recovery on sets of tests. In particular, the developer can select a package (or a project) and SCOTCH will automatically recover links for each unit test found automatically by detecting all the classes that inherit from class `TestClass`.

The recovery results are reported in the view shown in Figure 4. The view shows all the classes that are retrieved from the dynamic slice taken with respect to the last assert statement in a unit test. In the example, SCOTCH slices with respect to the assert statement `assertNull(client2.messageReceived())`, which yields the initial set of classes `Message`, `MessageDataObject`, `XMLSocketServer` and `XMLSocketClient`.

The view also reports, for each class retrieved by slicing, the CCBC value between the class and the unit test under analysis. From this list, SCOTCH automatically checks as “tested” those classes having high conceptual coupling with the unit test under analysis. In our example, the tool checks as tested the classes `XMLSocketServer` and `XMLSocketClient`, which correctly identifies the tested classes.

However, for each candidate link SCOTCH provides a checkbox option that allows the user to confirm (or not) the link between the unit test and the candidate tested class. Once the user has finished its classification, the traced links are stored in XML files.

Finally, note that a stop-class list (i.e., a list of classes from standard libraries such as `java.*`, `javax.*`, `org.junit.*`) is used by SCOTCH to filter the set of classes identified by slicing. The developer can setup this list according to his or her application domain and requirements, by selecting the stop-class list button in the SCOTCH view (see the upper left of Figure 4).

\(^1\)http://ase.cpsc.ucalgary.ca/index.php/AgilePlanning/AgilePlanner
Fig. 7. SCOTCH: consistency checking

B. Highlighting Inconsistency between Unit Test and Tested Classes

The traceability links, stored in XML files, can be exploited to support the development process by helping to maintain the consistency between the modified production code and its related unit tests. To illustrate such a functionality, during the demo we simulate a modification scenario on the class `Converter`, which is tested by the unit test `ConversionTest` (see Figure 5).

Suppose the developer modifies the `Converter` class as shown in Figure 6 by changing the method `fromXML` to call the method `marshal` instead of the method `unmarshal`. Such a change might affect the correctness of the related unit tests. For this reason, SCOTCH alerts the developer by giving a warning message to check if the related unit test (i.e., `ConversionTest`) might be needed to modify in order to reflect the changes applied to its tested class. This alert is shown in Figure 7.