An Eclipse plug-in for Test-to-Code Traceability Recovery

Abdallah Qusef*, Gabriele Bavota*, Rocco Oliveto*, Andrea De Lucia*, and David Binkley†

*University of Salerno, Fisciano (SA), Italy,
†University of Molise, Pesche (IS), Italy
‡Loyola University Maryland, Baltimore, USA
{aqusef@unisa.it, gbavota@unisa.it, rocco.oliveto@unimol.it, adelucia@unisa.it, binkley@cs.loyola.edu}

Abstract. Developers write and maintain unit tests continually in order to reflect changes in the production code and maintain an effective regression test suite. In addition, during refactoring developers must ensure that all unit tests continue to pass, so unit tests often need to be refactored together with the source code. Traceability links between application code and respective unit tests are extremely useful to preserving consistency during refactoring. Unfortunately, support for identifying and maintaining traceability links between unit tests and tested classes in contemporary software engineering environments and tools is not satisfactory. This paper presents SCOTCH (Slicing and COupling based Test to Code trace Hunter), a tool implemented as an Eclipse plug-in that uses dynamic slicing and conceptual coupling to automatically establish the traceability links between unit tests and application code. The evaluation of SCOTCH on several software systems highlights its usefulness as a feature within a software development environment.

Keywords: Slicing, Conceptual Coupling, Traceability.

1 Introduction

Software Traceability is defined by IEEE\(^1\) as the degree to which a relationship can be established between two or more products of the development process [1]. Thus, traceability information helps software engineers to understand the relationships and dependencies among various software artefacts. For this reason, it provides important insights during software development and evolution, and aids in program comprehension, maintenance, impact analysis, and reuse of existing software [17].

In agile development, unit tests are written by developers to help understand a problem and, when using test driven development, are written ahead of any new code. In addition, unit tests are continually updated to reflect changes in the production code and thus maintain an effective regression suite [2]. This activity

\(^1\)IEEE: Institute of Electrical and Electronics Engineers -www.ieee.org
can be easily supported when traceability links between unit tests and tested classes are explicitly maintained. For example, during refactoring, traceability information can play an important role in preserving the consistency between the changes applied to the production code and the behavior tested by the unit tests. Indeed, code refactoring is often followed by unit test refactoring [3, 4]. In such a scenario, the presence of the traceability links between unit tests and the classes under test facilitates the refactoring process [15]. Unfortunately, these traceability links are often not documented and thus no explicit link is available.

To solve this problem we present SCOTCH (Slicing and COupling based Test to Code trace Hunter), an Eclipse plug-in that supports a developer in the identification of dependencies between unit test and tested classes. The traceability recovery process behind SCOTCH was presented in [11] and is composed of two steps. In the first step SCOTCH uses dynamic slicing [12] to identify all the classes that affect the result of the last assert statement in each unit-test. This set of classes will also include support classes (i.e., mock objects and helper classes), besides the actual tested classes. Then, in the second step, Conceptual Coupling [16] is exploited to discriminate between the actual tested classes and the support classes.

The paper is organized as follows. Section 2 discusses related work, Section 3 presents the proposed traceability recovery approach, while Section 4 presents SCOTCH, highlighting its use in practical environments. Finally, Section 5 provides some evaluation results and concluding remarks.
2 Related Work

Today’s integrated development environments offer little support to a developer in need of linking unit tests with the related classes under test. For example, the Eclipse Java environment\(^2\) suggests, when creating a unit test using the pre-defined wizard (see Figure 1), that the developer provides the corresponding class under test. Moreover, as shown in Figure 1, Eclipse offers a search-referring-tests menu entry that retrieves all unit tests that call a selected class or method. However, no functionality is provided to support the software engineer in the recovery of traceability links between unit tests and the classes under test.

To address these issues and to mitigate the Eclipse drawbacks, Bouillon et al.\(^7\) present a JUnit Eclipse plug-in that uses Static Call Graphs\(^8\) and Java’s annotation string construct to identify, for each unit test, the classes under test. Most other approaches proposed in literature are not integrated into an IDE.

Sneed\(^{19}\) proposed a name matching and manual linking based approach that first maps code functions to requirements and then links test cases to code functions by associating test case time stamps with code function stamps.

Rompaey and Demeyer compare four approaches: Naming Convention (NC), Last Call Before Assert (LCBA), Latent Semantic Indexing (LSI)\(^9\), and Co-evolution\(^{18}\). NC relies on the name of a unit test to identify the class under test by removing the prefix “Test” from the name of the unit test. LCBA exploits the static call graph to identify the set of tested classes as those called by the statement that precedes an assert statement. LSI computes the textual similarity between the unit test and the classes of the systems to retrieve tested class, while Co-evolution assumes that the tested class co-evolves with the unit test. Rompaey and Demeyer show that these approaches are not effective in identifying traceability links between unit tests and classes under test\(^{18}\). The results indicate that NC is the most accurate, while LCBA has a higher applicability (consistency). Nevertheless, these two approaches have important limitations. With regards to NC, it is based on the assumption that naming conventions are used to name the unit test and that a single class is tested by each unit test. However, such assumptions are not always valid in practice, especially in industrial contexts\(^{10}\). As for LCBA, it fails when, right before an assert statement, there is a call to a state-inspector method from a class that is not the class under test\(^{10}\).

To overcome such limitations, we analyzed the last assert statements in unit tests and performed a simple reachability analysis that takes into account only data dependence to recover traceability links between unit tests and classes under test\(^{10}\). This approach ignores control dependences and other issues, such as aliasing, inter-procedural control and data flow, and inheritance. The results achieved indicate that on one hand the approach is able to identify tested classes with higher precision than NC and LCBA, while, on the other hand, it fails to retrieve a sensible number of tested classes (low recall).

\(^2\) http://www.eclipse.org/jdt/
In [11] we introduced SCOTCH, a novel approach to traceability link recovery. SCOTCH exploits dynamic slicing [12, 13] to identify the set of classes that affect the last assert statement. It is able to identify a larger number of classes as compared to the approach proposed in [10], since it considers several aspects that are completely ignored in [10]. The use of slicing helps in improving recall but the set of identified classes is an overestimate of the set of tested classes because it also includes helper classes. This negatively impacts the approach’s precision. To address this shortcoming, conceptual coupling is used to discriminate between the actual tested classes and helper classes and thus improve the accuracy of the approach.

3 The Traceability Recovery Method

The traceability recovery process behind SCOTCH is based on the following assumptions:

1. unit tests are classes whose methods implement the tests of a test suite [5]. In each test, the actual outcome is compared to the expected outcome (oracle) using an assert statement. However, assert statements are also used to verify the testing environment before verifying the actual outcome [15]. Since each
method of the unit test implements a specific test case, we assume that the last assert statement in the unit-test method compares the actual outcome with the oracle [10]. This means that tested classes affect the result of the last assert statement in each unit-test method;

2. unit tests are related to different types of classes including mock objects, helper classes, and tested classes [15]. Support classes (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 identifies the set of tested classes using the two steps shown in Figure 2. In the first step dynamic slicing is exploited to identify an initial set of candidate tested classes, called the Starting Tested Set (STS). Previous studies indicate that in general for a method implementing a unit-test, the results of the last assert statement in the method are affected by methods belonging to the tested classes [10]. Thus, we use the last assert statement in a unit test as the slicing criterion (starting point for the slice). In particular, we employ backward dynamic slicing [12] to identify the initial set of classes by finding all the method invocations that affect the last assert statement in each unit-test. However, the set of classes identified using dynamic slicing may include many helper classes as well as classes belonging to mock objects and those from standard libraries (e.g., String). The latter are removed from the STS through a stop-class list (i.e., a list of classes from standard libraries such as java.*, javax.*, org.junit.*), while the helper classes are identified and pruned-out of the STS in the second step of the process.

Indeed, we conjecture that a unit test is semantically related to the classes under test (in particular, their textual similarity is higher than the similarity between unit tests and helper classes, which are used more uniformly across all tests). In other words, the semantic information captured in the unit test by comments and identifiers is closer to that of the tested classes than to that of the helper classes. This closeness can be measured using the Conceptual Coupling Between Classes (CCBC) [16]. CCBC uses Latent Semantic Indexing (LSI), an advanced Information Retrieval (IR) technique [9] to represent each method of a class as a real-valued vector that spans a space defined by the vocabulary extracted from the code.

We use CCBC to rank the classes in the slice according to their coupling with the unit test. The higher the rank, the higher the likelihood that the class is a tested class. Thus, once the classes in the STS are ranked according to their conceptual coupling with the unit test, a threshold is used to truncate the ranked list and identify the top coupled classes that represent the candidate tested classes. Defining a “good” threshold a priori is challenging, because it depends on the quality of the classes in terms of identifiers and comments as well as on the number of tested classes. For this reason, we use a scaled threshold $t$ [17] based on the coupling between the unit test and the top class in the ranked
list:

\[ t = \lambda \cdot CCBC_{c_1} \]

where \( CCBC_{c_1} \) is the conceptual coupling between the unit test and the top class in the ranked list and \( \lambda \in [0, 1] \). The defined threshold is used to remove from the STS classes that have a conceptual coupling with the unit test lower than \( \lambda \% \) of the conceptual coupling between the unit test and the top ranked class. The resulting set is called the Candidate Tested Classes (CTS).

4 Integrating SCOTCH in the Eclipse IDE

SCOTCH contributes a new view to the Eclipse workbench that allows a developer to recover traceability links between unit tests and tested classes. The developer can select a unit test, a package, or an entire software project in the Eclipse Package Explorer, and press the button in the view to start the recovery process. If the developer selects a module (i.e., a package or a project) SCOTCH automatically identifies the unit test classes and then recovers links for each unit test. The results are then reported in the view shown in Figure 3. Indeed, for each candidate link, SCOTCH provides a checkbox that allows the developer to trace (or not) the link between the unit test and the candidate tested class. For each unit test, SCOTCH shows the classes belonging to its STS as well as to its CTS. The checkbox for 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.

In the following subsections we first demonstrate the traceability recovery functionality and then explain the fundamental underlying concepts of the tool and how the traceability information can be used by SCOTCH to highlight inconsistency between the production code and related unit test. Finally, we describe the SCOTCH architecture.
4.1 Recovering Traceability Links

To explain the traceability recovery functionality of SCOTCH we consider the unit test `testSendFromServerToOnlyOneClient` from the class `NetworkCommunicationTest` shown in Figure 4. 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 the 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 `TestCase`.

The recovery results are reported in the view shown in Figure 3. 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 can-

```java
public class NetworkCommunicationTest extends TestCase {
    ... ...
    @Test
    public void testSendFromServerToOnlyOneClient() throws Exception {
        XMLSocketServer server = new XMLSocketServer(5053, null);
        Thread.sleep(1000);
        MockClientCommunicator client1 = new MockClientCommunicator();
        MockClientCommunicator client2 = new MockClientCommunicator();
        XMLSocketClient nc1 = new XMLSocketClient("localhost", 5053, client1);
        XMLSocketClient nc2 = new XMLSocketClient("localhost", 5053, client2);
        Thread.sleep(1000);
        Message msg = new MessageDataObject(99);
        server.sendToSingleClient(msg, 2);
        nc1.breakConnection();
        nc2.breakConnection();
        assertTrue(client1.messageReceived() != null);
        assertTrue(client1.messageReceived().getMessageType() == msg.getMessageType());
        assertNull(client2.messageReceived());
        server.kill();
        Thread.sleep(1000);
    }
    ... ...
}
```

Fig. 4. Fragment of NetworkCommunicationTest from AgilePlanner
didate 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 3).

4.2 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 modified production code and its related unit tests. To illustrate this functionality, we consider a modification to the class Converter, shown at the top of Figure 5.

Suppose the modification of class Converter updates the method fromXML by replacing the call to unmarshal with a call to marshal. The resulting code is
shown in the bottom of Figure 5. Further suppose that class \textit{Converter} is tested by the unit test \textit{ConversionTest} as indicated in Figure 6.

The change to method \textit{fromXML} may affect the correctness of the unit test \textit{ConversionTest}. Thus, as illustrated in Figure 7, SCOTCH alerts the developer by giving a warning message advising the user to check if the related unit tests might need modification to reflect the changes made in one or more of the tested classes.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{warning.png}
\caption{SCOTCH: consistency checking}
\end{figure}

\subsection{4.3 Architecture}

The plug-in is decomposed into four modules, namely \textit{View}, \textit{Link Retriever}, \textit{XML Manager}, and \textit{Consistency Manager} (see Figure 8). The module \textit{View} implements the presentation layer of the plug-in. In particular, as shown in Figure 3, it supports the visualization of the links that SCOTCH discovers. This view extends the \textit{ViewPart} view defined in Eclipse.

The traceability recovery process used by SCOTCH is implemented in the \textit{Link Retriever} module. This module takes as input the set of unit tests selected by the user (that can be a single unit test, a package, or an entire software project) and uses the JavaSlicer\textsuperscript{3} tool to identify the STS for each unit test. Then, using an internal LSI-based IR-Engine [9], the Link Retriever module computes the conceptual coupling between each unit test and the classes contained in its STS in order to compute the CTS, which is then shown to the user.

The links traced by the user are stored in XML files using the \textit{XML Manager} module that is in charge of managing all the operations performed on the produced XML files.

Finally, the \textit{Consistency Manager} module maintains consistency between the changes applied to the classes in the production code and the behavior tested by the related unit tests. To this end, this module analyzes the operations performed by the user on the production code and check if the performed changes potentially affect the correctness of the related unit tests. The link between the production code and the unit tests are provided by the \textit{XML Manager} module.

\textsuperscript{3} http://www.st.cs.uni-saarland.de/javaslicer
Fig. 8. SCOTCH architecture

5 Conclusion

Several traceability recovery tools have been proposed in the past. Although in general different approaches (e.g., those based on IR and Co-evolution) have been successful in identifying traceability links between different types of artefacts, the authors have shown that these approaches are not effective in identifying traceability links between unit tests and the classes under test [18, 10].

In this paper we presented SCOTCH, an Eclipse plug-in that supports a developer in the recovery of traceability links between unit tests and tested classes. In prior work [11], we compared the accuracy of SCOTCH with benchmark tools based on naming conventions (NC) [18], Last Call Before Assert (LCBA) [18], and data-flow analysis (DFA) [10] using three programs as a testbed (Agile-Planner, eXVantage, and ArgoUML). We assessed the accuracy of each recovery methods using the F-measure (the harmonic mean of precision and recall) [20]. For SCOTCH the threshold parameter value 0.95 was used for λ because it provides the best average performance on these three programs.

Figure 9 compares the F-measures attained by SCOTCH and the three benchmark tools. Overall, SCOTCH provides the best performances. In more detail, for AgilePlanner and eXVantage (where naming conventions are badly applied) SCOTCH significantly outperforms NC, LCBA, and DFA, while on ArgoUML SCOTCH overcomes LCBA and DFA and is able to obtain the same performances as NC. It is worth noting that SCOTCH not only obtained the higher accuracy between the experimented tools, but also provided the most stable performances across systems having different characteristics. These results highlight SCOTCH’s usefulness as a feature within a software development environ-
ment such as Eclipse. For a deeper description of the empirical evaluation of the SCOTCH traceability recovery engine see [11].

To the best of our knowledge SCOTCH is the first traceability recovery tool that relies on slicing and conceptual coupling to recover the traceability links between unit tests and application code. Indeed, SCOTCH helps developers detect traceability links during software development. Moreover, SCOTCH helps reduce consistency errors that are possible if tested classes are modified without corresponding modification to their unit tests. It does this with a pop-up message alerting the developer of the potential problem.

Future work will be devoted to additional experiments and assessment of the traceability recovery tool. A second future direction regards the granularity of the traceability links recovered and their semantics. In particular, we plan to adapt SCOTCH in order to recover the traceability links between unit tests and the methods under test.

![graph](image)

**Fig. 9.** Accuracy of the experimented traceability recovery methods.

### References