SCOTCH: Test-to-Code Traceability using Slicing and Conceptual Coupling

Abdallah Qusef\(^1\), Gabriele Bavota\(^1\), Rocco Oliveto\(^2\), Andrea De Lucia\(^1\), David Binkley\(^3\)
\(^1\)University of Salerno, Fisciano (SA), Italy
\(^2\)University of Molise, Pesche (IS), Italy
\(^3\)Loyola University Maryland, Baltimore, USA
aqusef@unisa.it, gavota@unisa.it, rocco.oliveto@unimol.it, adelucia@unisa.it, binkley@cs.loyola.edu

Abstract—Maintaining traceability links between unit tests and tested classes is an important factor for effectively managing the development and evolution of software systems. Exploiting traceability links helps in program comprehension and maintenance by ensuring consistency between unit tests and tested classes during maintenance activities. Unfortunately, it is often the case that such links are not explicitly maintained and thus they have to be recovered manually during software evolution. A novel automated solution to this problem, based on dynamic slicing and conceptual coupling, is presented. The resulting tool, SCOTCH (Slicing and COupling based Test to Code trace Hunter), is empirically evaluated on three systems: an open source system and two industrial systems. The results indicate that SCOTCH identifies traceability links between unit test classes and tested classes with a high accuracy and greater stability than existing techniques, highlighting its potential usefulness as a feature within a software development environment.

Keywords—Slicing, Conceptual Coupling, Traceability, Empirical Studies.

I. INTRODUCTION

Unit tests are continually updated during software development to reflect changes in production code and thus maintain an effective regression suite [1]. This makes unit tests an important source of documentation, especially when performing maintenance tasks where they can help a developer comprehend production code as well as identify failures. In addition to comprehension and maintenance, traceability information can play an important role preserving consistency during refactoring. Indeed, often refactoring of the code must be followed by refactoring of the related unit tests [2], [3]. The presence of test-case traceability links (links between unit tests and the classes under test) facilitate the refactoring process [2] in addition to other comprehension and evolution tasks.

Despite its importance, support for identifying and maintaining traceability links between unit tests and tested classes in contemporary software engineering environments and tools is not satisfactory. One of the better examples is the XUnit framework [5], [6], which provides a consistent and useful set of test objects used to execute unit tests. Unfortunately, it maintains no pre-defined structure nor explicit links between code and test cases. As a result, developers who use XUnit even given the frameworks shortcomings, must compensate for the inadequate traceability by using costly manual detection and maintenance of the traceability links. This cost is especially high when the task must be done frequently due to the iterative nature of modern software development.

These considerations highlight the need for automatic and even semi-automatic approaches to support the developer in the identification of dependencies between unit test and tested classes. In this paper, we propose a novel approach, SCOTCH (Slicing and COupling based Test to Code trace Hunter), based on the following assumptions:

1) unit tests are maintained in classes that collectively implement the unit test suite. Each unit-test in the suite is method that implements a test case [5]. In each unit-test method, the actual outcome is compared to the expected outcome (oracle) using an assert statement. However, assert statements may also be used to verify the testing environment before verifying the actual outcome [7]. Since each method of the unit test implements a specific test case, we assume that the last assert statement in a unit-test is the one that compares the actual outcome with the oracle [4];

2) in addition to the classes under test, unit tests interact with different types of classes, such as mock objects and helper classes [7]. These other classes are used to create the testing environment. 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 two assumptions we use dynamic slicing [8], [9] to identify all the classes that affect the result of the last assert statement in each unit-test method. Dynamic slicing is preferred to static slicing [10], because it allows more precise identification of the classes tested by a particular test case. However, the set of classes uncovered using dynamic slicing is an overestimate of the set of tested classes (because, for example, it includes helper classes). Thus, we use conceptual coupling [11] to help discriminate between the actual tested classes and the helper classes.

The accuracy of SCOTCH is empirically evaluated on three systems: the open source system ArgoUML\(^1\), and

\(^1\)http://argouml.tigris.org/
two industrial systems, AgilePlanner\textsuperscript{2} and eXVantage\textsuperscript{3}. As a benchmark, we compare the accuracy of SCOTCH with approaches based on naming conventions (NC) \cite{vanRompaey2010}, Last Call Before Assert (LCBA) \cite{vanRompaey2010}, and data-flow analysis (DFA) \cite{song2012}. The results indicate that SCOTCH is not only the most accurate, but also has the highest stability.

The paper is organised as follows. Section II discusses related work, while Section III presents the proposed traceability recovery approach. Section IV provides details on the design of the case study and reports on the results achieved. Section V discusses the results and threats to validity. Finally, Section VI provides some concluding remarks.

II. RELATED WORK

Work illustrating the complexity of the test-case traceability link problem includes that of Bruntink \textit{et al.} \cite{bruntink2012}. Their work shows how classes often depend on other helper classes. Furthermore it demonstrates that such classes require more test code and are thus more difficult to test. To improve the testability of complex classes, they suggest using “cascaded test suites,” where a test of a complex class uses the tests of its required classes to set up the complex test scenario.

There is scant existing tool support for establishing and maintaining test-case traceability links. For example, today’s integrated development environments offer little support to the developer trying to browse between unit tests and the classes under test. For example, the Eclipse Java environment\textsuperscript{4} suggests, when creating a unit test via a wizard, that the developer provides the corresponding class under test. Also, Eclipse offers a search-referring-tests menu entry that retrieves all unit tests that call a selected class or method. To improve upon these features, Bouillon \textit{et al.} \cite{bouillon2012} present a JUnit Eclipse plug-in that uses Static Call Graphs \cite{bouillon2012} to identify for each unit test the classes under test. Moreover, it uses Java annotations to identify traceability links from information within a comment string.

Several other approaches, not tied to Eclipse, have been considered. Sneed \cite{sneed2012} proposed two approaches to connect test cases and code. The first uses name matching with some manual linking to map code functions to requirements model functions, while the second links test cases and code functions using time stamps. Van Rompaey \textit{et al.} \cite{vanRompaey2010} compare four approaches: NC, LCBA, Latent Semantic Indexing (LSI) \cite{vanRompaey2010}, and Co-evolution. NC simply identifies the classes under test by removing the string “Test” from the method name of the unit test. LCBA exploits the static call graph to identify the set of tested classes. In particular, the tested classes identified by LCBA are the classes called in the statement that precedes an assert statement. LSI is an Information Retrieval (IR) technique that identifies the tested classes based on the textual similarity between the unit test and the classes of the systems, while Co-evolution assumes that the tested class co-evolves with the unit test. Although in general the latter two approaches have been shown successful in identifying traceability links between different types of artefacts \cite{vanRompaey2010}, \cite{vanRompaey2010} and in identifying logical coupling between source code components \cite{vanRompaey2010}, \cite{vanRompaey2010}, van Rompaey \textit{et al.} show that these approaches are not effective in identifying test-case traceability links. The results indicate that NC is the most accurate, while LCBA has a higher applicability (consistency). Nevertheless, these two approaches have important limitations. NC assumes that developers follow required naming conventions and thus implicitly that a single class is tested by each unit test. However, such assumptions are not always valid in practice, especially in industrial contexts \cite{song2012}. LCBA’s limitations come from is use of the static call graph. Because it returns the last called class before the last assert statement, LCBA falls short when, right before the assert statement, there is a call to a state-inspector method from a class that is not the class under test \cite{song2012}.

To overcome these limitations, the authors proposed the use of data flow analysis (DFA) to recover test-case traceability links \cite{song2012}. The approach identifies, as the tested classes, a set of classes that affect the result of the last assert statement in each unit-test using a simple reachability analysis that exploits only data dependence, while ignoring control dependences and other aspects, such as aliasing, inter-procedural flow, and inheritance. Empirical results indicate that the approach identifies tested classes with higher precision than NC and LCBA, while fails to retrieve a sensible number of tested classes (has low recall).

SCOTCH shares with the aforementioned approaches the use of assert statements to derive test-case traceability links. However, it uses dynamic slicing \cite{singh1990}, \cite{bagchi1992} to identify, with higher accuracy, the set of classes that affect the last assert statement. To the best of our knowledge, while program slicing has been used in many applications (e.g., testing, debugging, program comprehension, software maintenance and optimization) \cite{singh1990}, it has been never used to improve test-case traceability. As shown in Section IV, SCOTCH is able to identify a larger number of classes than its predecessors in part because the use of slicing improves recall. However, the set identified by dynamic slicing is an overestimate of the set of tested classes. This negatively impacts SCOTCH’s precision. Here conceptual coupling is used to discriminate between the actual tested classes and the helper classes. This filtering improves the accuracy of the approach.
III. SCOTCH OVERVIEW

SCOTCH identifies the set of tested classes using the two steps highlighted in Figure 1. The first step exploits dynamic slicing to identify an initial set of candidate tested classes, called the Starting Tested Set (STS). In the second step, the STS is filtered exploiting the conceptual coupling between the identified classes and the unit test resulting in the Candidate Tested Set (CTS). These two steps are detailed in the following subsections.

A. Identifying the Starting Tested Set

In the first step, SCOTCH identifies an initial set of candidate tested classes by taking the dynamic slice with respect to the final assert statement in a unit-test method. In general, each unit-test method consists of two kinds of statements: non-assertion and assertion statements. The assertion statements compare the actual outcome with the expected (oracle) outcome. Thus, it is likely that the computation of an assert statement is affected by a tested class. However, a unit-test often contains more than one assert statement (leading to “Assertion Roulette” [7]). In practice, this is common because developers use assert statements to verify the testing environment before verifying the behavior of the tested classes. Previous studies indicate that the results of the last assert statement in a unit-test method are affected by methods of the tested class [4].

These considerations suggest using only the last assert statement in a unit test method as the starting point for the slice, referred to as the slicing criterion. In particular, we employ backward dynamic slicing [8], [9] to identify an initial set of classes by finding all the method invocations that affect the last assert statement in each unit-test method. Using only the last assert statement reduces the retrieval of helper classes and also reduces slicing time. Moreover, dynamic slicing is preferred to static slicing since the natural way to link the unit tests to the related code is by executing them.

The recovery process first parses the code of each unit test to identify its last assert statement, which is used as the dynamic slicing criterion. The set of statements contained in the backward dynamic slice taken with respect to this statement is analyzed to extract the classes that affect the last assert statement. These classes include (unwanted) mock objects and classes that belong to standard libraries (e.g., String). All such classes are false positives that should be removed from the set of candidate tested classes. To remove the latter, a stop-class list (i.e., a list of classes from standard libraries such as java.*, javax.*, org.junit.*) is used. The resulting set of classes represents the STS. Step 2 attempts to remove the remaining unwanted classes.

Figure 2 shows a fragment of code from the unit test RemoveElementsTest taken from the eXVantage system. The test is related to the class ConnectedGraph where it tests the removeElement method. The dynamic slice taken with respect to the last assert statement (line 11) includes all statements affecting this assert statement (i.e., statements 10, 8, 6, 5, 4, 3, 2, and 1). Analyzing this slice, it is possible to identify a set of classes that affects the computation of the last assert statement. In the example these classes, the STS, are DisconnectedGraphException, EdgeElement, NodeElement, ConnectedGraph, and ConnectedGraphFactory. In addition to the tested class ConnectedGraph, this set includes several helper classes (which also affect the last assert statement). These classes negatively impact the accuracy of the result. The next section presents a technique that prunes-out helper classes from the STS and thus improves the accuracy of our approach.

B. Identifying the Candidate Tested Set

Our filtering of the STS is based on the conjecture that unit test will be semantically related to the classes under test (in particular, their textual similarity should be higher than the similarity between unit tests and helper classes, which are used more uniformly across the system). In other
words, the semantic information captured in the unit test by comments and identifiers is closer to a tested class than a helper class. This closeness can be measured using the Conceptual Coupling Between Classes (CCBC) [11]. CCBC uses Latent Semantic Indexing [17], an advanced IR technique, to represent each method of a class as a real-valued vector in a space defined by the vocabulary (words) 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 between the unit test and a class in the STS, the higher the likelihood that the class is a tested class. For this reason, once the classes in the STS are ranked according to their conceptual coupling with the unit test, a threshold is used to cut the ranked list and identify the top coupled classes that represent the candidate tested classes. Defining a “good” threshold 
\( \lambda \) a priori is challenging, because it depends on the quality of the class in terms of identifiers and comments as well as on the number of tested classes. For this reason, we use a scaled threshold \( t \) [23] based on the coupling between the unit test and the top class in the ranked list:

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

where \( CCBC_{C_t} \) 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 lower than \( \lambda \% \) of the conceptual coupling between the unit test and the top ranked class.

Computing the CCBC between the unit test shown in Figure 2 and the classes in the STS identified by slicing allows us to obtain the following ranking: ConnectedGraph (0.60), ConnectedGraphFactory (0.56), EdgeElement (0.49), NodeElement (0.48), and DisconnectedGraphException (0.10). Setting \( \lambda = 0.95 \), the conceptual coupling threshold is

\[
t = 0.95 \cdot 0.60 = 0.57
\]

In this way, only ConnectedGraph (the actual tested class) is retrieved as tested class. The parameter \( \lambda \) is empirically estimated in Section IV.

IV. EMPIRICAL EVALUATION

This section describes the design and the results of a case study carried out to assess the accuracy of SCOTCH. The case study was conducted following the guidelines given by Yin [24].

A. Definition and Context

The goal of the case study is to analyze the effectiveness of SCOTCH at identifying relationships between unit tests and code classes. The quality focus was on ensuring better recovery accuracy, while the perspective was of both (i) a researcher, who wants to evaluate the effectiveness of slicing and conceptual coupling in identifying relationships between unit tests and tested classes; and (ii) a project manager, who wants to evaluate the possibility of adopting the proposed method in his or her software company.

Several constraints must be taken into consideration to select the context (i.e., the software system) to be studied. First, we needed direct access to a source code repository with a considerable test suite. In addition, SCOTCH depends on the use of assert statements in unit tests (our approach cannot be applied if the test methods do not contain assert statements). Finally, the implementations are currently targeted towards systems developed in Java. To satisfy these constraints, we selected three software systems written in Java, namely AgilePlanner, eXVantage, and ArgoUML.

Table I shows the size of the three systems in KLOC (thousand of lines of code) and classes. The table also shows the number of unit tests (total and selected for the experiments) and the corresponding KLOC for each. Indeed, there are unit tests used to test mock objects, network settings, and complex queries against the database. We decided to remove such tests since they are not related to actual classes under test. Note that a similar choice on the AgilePlanner and ArgoUML projects was made in previous work [4], [12]. An example of discarded test is shown in Figure 3.

Table II reports the characteristics of the unit tests used in our experiments. The table also shows the distribution of unit tests that test various numbers of classes. For instance, in AgilePlanner there are 32 unit tests: 26 (81%) of them test only one class, 2 (6%) test two classes, while 4 (13%) test more than two classes. This data confirms that generally developers write a unit test for each class, but there are also cases where the tested classes are more than one. Table II also reports the distribution of unit tests having different number of assert statements. As seen in the table, generally unit tests contain more than two assert statements. This means that “Assertion Roulette” [7] is a real problem.

We did not find any documentation describing the actual dependencies between unit tests and code classes. The only documentation we found was some guidelines in ArgoUML that advocated a unit test reside in the same package as the
class it tests. However, an explicit and complete traceability matrix for this system is not available. This demonstrates that such links are not explicitly maintained and they might be retrieved when needed during software evolution. Thus, in order to create an oracle to assess the accuracy of the proposed traceability recovery method, three Ph.D. students of the University of Salerno manually identified the links between the unit tests and the tested classes. Students individually analyzed the code in order to associate each unit test with a set of tested classes. To avoid bias in the experiment, students were not aware of the experimental goals or of the experimental recovery methods. Once students retrieved the tested classes for each unit test, they performed an open-discussion with researchers to solve conflicts and reach a consensus on the traceability links identified.

Finally, to identify the classes that affect the results of the last assert statement, backward dynamic slices were computed using the open-source tool JavaSlicer\(^5\).

**B. Research Questions, Data Analysis and Metrics**

In the context of our case study we formulate two research questions:

- **RQ\(_1\)**: Does SCOTCH effectively identify relationships between unit tests and tested classes?
- **RQ\(_2\)**: Does SCOTCH outperform other unit test traceability recovery approaches?

To address the first research question we compared the classes retrieved by SCOTCH with those manually identified. For the second research questions, we compared the accuracy of SCOTCH with the accuracy of three other approaches: NC [12], LCBA [12], and DFA [4].

To evaluate and compare these traceability recovery techniques, we used two well-known IR metrics, recall and precision [25]:

\[
\text{recall} = \frac{|\text{cor} \cap \text{ret}|}{|\text{cor}|} \quad \text{precision} = \frac{|\text{cor} \cap \text{ret}|}{|\text{ret}|} \%
\]

\(^5\)http://www.st.cs.uni-saarland.de/javaslicer

Moreover, to provide a further comparison of the traceability recovery methods we computed the following overlap metrics [26]:

\[
\text{correct}_{m_i \cap m_j} = \frac{|\text{correct}_{m_i} \cap \text{correct}_{m_j}|}{|\text{correct}_{m_i} \cup \text{correct}_{m_j}|} \%
\]
\[
\text{correct}_{m_i \setminus m_j} = \frac{|\text{correct}_{m_i} \setminus \text{correct}_{m_j}|}{|\text{correct}_{m_i} \cup \text{correct}_{m_j}|} \%
\]

where \(\text{correct}_{m_i}\) represents the set of correct links identified by method \(m_i\), \(\text{correct}_{m_i \cap m_j}\) measures the overlap between the set of correct links identified by methods \(m_i\) and \(m_j\), and \(\text{correct}_{m_i \setminus m_j}\) measures the correct links identified by \(m_i\), but missed by \(m_j\). The latter metric gives an indication of how a traceability method contributes to enriching the set of correct links identified by another method. This information can be used to analyze the orthogonality of the techniques.

**C. Analysis of the Results**

This section discusses the results achieved in our case study. First, we analyze the effectiveness of SCOTCH and how the conceptual coupling threshold affects its accuracy. Figure 4 shows the F-measures obtained for different values of \(\lambda\), ranging from \(\lambda = 1\) where only the first class in the STS is identified to \(\lambda = 0\) where all the classes in the STS are identified. As expected, the higher the threshold the lower the F-measure (due to the higher number of retrieved classes). The analysis of the results shows that a value of \(\lambda = 0.95\) provides good results over all systems. It is worth noting that this threshold allows the technique to recover more than one class only when there are other classes having a conceptual coupling very close to the first class in the ranked list. The results achieved are very encouraging and support a positive response to our first research question. In particular, setting a relative high coupling threshold (\(\lambda = 0.95\) or \(\lambda = 0.90\)) yields an average F-measure of about 77%.

---

### Table I

**CHARACTERISTICS OF THE EXPERIMENTED SYSTEMS.**

<table>
<thead>
<tr>
<th>System</th>
<th>Description</th>
<th>Classes</th>
<th>KLOC</th>
<th>Number</th>
<th>KLOC</th>
<th>Selected</th>
<th>KLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agileplanner</td>
<td>Industrial tool supports distributed agile teams</td>
<td>299</td>
<td>24</td>
<td>32</td>
<td>4</td>
<td>32</td>
<td>4</td>
</tr>
<tr>
<td>eXVantage</td>
<td>eXtreme Visual-Aid Novel Testing and Generation tools</td>
<td>348</td>
<td>28</td>
<td>30</td>
<td>5</td>
<td>17</td>
<td>4</td>
</tr>
<tr>
<td>ArgoUML</td>
<td>Open source UML modeling tool</td>
<td>1,430</td>
<td>124</td>
<td>163</td>
<td>12</td>
<td>75</td>
<td>7</td>
</tr>
</tbody>
</table>

### Table II

**CHARACTERISTICS OF THE TESTS USED IN THE EXPERIMENTATION.**

<table>
<thead>
<tr>
<th>System</th>
<th># tests</th>
<th>Methods</th>
<th>Tested classes</th>
<th>Asserts for methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agileplanner</td>
<td>32</td>
<td>174</td>
<td>81% 6% 13%</td>
<td>40% 27% 33%</td>
</tr>
<tr>
<td>eXVantage</td>
<td>17</td>
<td>199</td>
<td>65% 29% 6%</td>
<td>38% 22% 40%</td>
</tr>
<tr>
<td>ArgoUML</td>
<td>75</td>
<td>454</td>
<td>96% 3% 1%</td>
<td>35% 20% 45%</td>
</tr>
</tbody>
</table>
The analysis of the results also reveals that filtering is crucial. This can be seen in Table III where the number of classes removed by filtering is broken down into those that were correct and those that were undesired. The overall ratio of almost 50 to 1 demonstrates how crucial the filtering step is. This is also seen in the F-measures, which are significantly worse before filtering. Indeed, using dynamic slicing we are able to identify a high number of tested classes (high recall) but we also recover many helper classes, which negatively impact precision (see Figure 4).

Regarding our second research question, Figure 5 compares the F-measure achieved by SCOTCH (with $\lambda = 0.95$) and those for the three benchmark techniques. Striking in Figure 5 is how NC provides very good accuracy for ArgoUML, where naming conventions are strictly followed, while in the other two projects it performs much worse. In some cases, NC retrieves classes that are not even in the system. This occurs when unit tests derive their names from local variables used in an assert statement or from the methods under test. Figure 2 shows an example of the latter scenario from eXVantage. In this case NC retrieves as tested class $RemoveElements$, while the actual tested class is $ConnectedGraph$. In addition, some unit tests use abbreviations for the classes under test causing NC to fail to correctly identify the tested classes. For example, the unit test $DGTest$ from eXVantage tests the class $DependencyGraph$ while the name predicts the class $DG$. This illustrates how the naming convention approach can be quite accurate but only when the naming convention is strictly followed [12].

Next, LCBA retrieves generally more classes (resulting in a higher recall) than NC except on ArgoUML, where naming conventions are strictly followed. In several cases LCBA fails to correctly retrieve tested classes since the classes called before the assert statements are not the tested classes.

Finally, DFA has an accuracy slightly better than LCBA [4]. However, DFA generally fails to retrieve the correct tested class when the assert statement does not contains any variables. This occurs, for example, when testing exception catching using “assert(True)”. It is worth noting that in all such cases SCOTCH correctly identifies the class under test since dynamic slicing also exploits control dependencies that are completely ignored by DFA.

For all three systems, Figure 5 indicates that SCOTCH provides the best performance. With AgilePlanner and eXVantage (where naming conventions are badly applied) SCOTCH significantly outperforms NC, LCBA, and DFA. For example, the unit test $NetworkCommunicationTest$ shown in Figure 6 from AgilePlanner tests the classes $XMLSocketServer$ and $XMLSocketClient$. In this case, NC retrieves $NetworkCommunication$ as tested class, LCBA analyses the last three assert statements to retrieve $MockClientCommunicator$, $Message$ and $MessageDataObject$, while DFA retrieves $MockClientCommunicator$ as tested class. It
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 client1 =
            new XMLSocketClient("localhost",5053,client1);
        XMLSocketClient client2 =
            new XMLSocketClient("localhost",5053,client2);
        Thread.sleep(100);
        Message msg = new MessageDataObject(99);
        server.sendToSingleClient(msg, 2);
        client1.breakConnection();
        assertTrue(client1.messageReceived().getMessageType() == msg.getMessageType());
        client2.messageReceived();
        server.kill();
        Thread.sleep(1000);
    }
}

Figure 6. Fragment of NetworkCommunicationTest from AgilePlanner.

Table IV
STUDENT’S T-TEST COMPARING SCOTCH WITH OTHER TECHNIQUES.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Correct Classes Mean</th>
<th>p-value</th>
<th>False Positives Mean</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCOTCH</td>
<td>0.97</td>
<td>n.a.</td>
<td>0.38</td>
<td>n.a.</td>
</tr>
<tr>
<td>NC</td>
<td>0.73</td>
<td>&lt; 0.001</td>
<td>0.27</td>
<td>0.059</td>
</tr>
<tr>
<td>DFA</td>
<td>0.84</td>
<td>0.024</td>
<td>0.48</td>
<td>0.145</td>
</tr>
<tr>
<td>LCBA</td>
<td>0.79</td>
<td>0.003</td>
<td>0.58</td>
<td>0.017</td>
</tr>
</tbody>
</table>

is worth noting that DFA does not retrieve other classes since it does not perform any inter-procedural analysis. In the STS SCOTCH initially identifies the following classes MockClientCommunicator, Message, MessageDataObject, XMLSocketServer and XMLSocketClient. The conceptual coupling based filtering reduces this set to the two classes XMLSocketServer and XMLSocketClient, which are the actual tested classes.

For ArgoUML SCOTCH improves upon LCBA and DFA and it is able to obtain the same performances as NC. Note that ArgoUML represents the best case for NC, since naming conventions are strictly enforced. Such a result highlights not only the high accuracy of SCOTCH, but also its high stability across systems with different characteristics.

The statistical tests summarized in Table IV reinforce the data presented in Figure 5. For each technique the table includes the average percentage of correct classes identified (second column) and the average percentage of false positives (fourth column). The third and fifth columns provide the p-value attained when comparing the performance of each technique to SCOTCH. For correct classes, the higher average percentage attained by SCOTCH is statistically better than all three other techniques. It achieves this while identifying fewer false positives than LCBA. As seen in the final column, there is no statistical difference between the percentages of false positives reported by SCOTCH, NC, and DFA.

Finally, Table V compares the overlap of links retrieved by SCOTCH with the three benchmark techniques. For the comparison of SCOTCH with NC, we observe that on AgilePlanner and ArgoUML the overlap is relatively high (about 70%). In these systems we also observed that about 20%—on average—of the links are correctly identified only by SCOTCH, while about 8% of the correct links are retrieved only by NC. A different scenario is seen with eXVantage, where naming conventions are not applied. In this case the overlap is very low and a large number of links are correctly identified only by SCOTCH. This result is due to the low performances of NC on eXVantage.

The overlap between SCOTCH and LCBA is relatively high on ArgoUML, while it is lower with the other two systems. On the latter systems, SCOTCH is also able to sensibly enrich the set of links retrieved by LCBA (34% and 25% of the correct links are identified by SCOTCH only), while for all the systems LCBA detected an average of about 6% of the correct links not detected by SCOTCH.

More interesting is the comparison of the proposed approach and DFA. As expected, the overlap of the correct links recovered by the two approaches is high (more than 60% on AgilePlanner and ArgoUML). In fact, there are still a few correct links recovered only by DFA (e.g., about 5% for ArgoUML and eXVantage). Nevertheless, there is still a significant number of links that are recovered only by the proposed approach (over 14%). This is because SCOTCH extends the DFA-based approach and considers several aspects completely ignored in DFA (e.g., control dependencies and inter-procedural analysis). The achieved results provide further evidence that SCOTCH is able to outperform all three previous approaches.

V. DISCUSSION AND THREATS TO VALIDITY

This section discusses the achieved results focusing on the threats that could affect their validity [24].

A. Evaluation Method

To evaluate the accuracy of the experimented traceability recovery methods, we used recall and precision, two widely used metrics for assessing traceability recovery technique [23]. In addition, the overlap metrics give a good indication on the overlap of the correct links recovered by
the different traceability methods. All the metrics can also be exploited to evaluate the accuracy improvement provided by SCOTCH as compared to NC, LCBA and DFA.

B. Oracle Accuracy

The accuracy of the oracle used to evaluate the traceability recovery methods might affect the results. We were not able to get the traceability matrices from the original developers. For this reason, we asked three Ph.D. students to manually recover the traceability links. To increase the confidence in the evaluation process students were not aware of the experimental goals or the experimented recovery technique. Even if students had a good background on development techniques and unit testing, they were not familiar with the application domain and the code of the software systems. To mitigate such a threat students individually identified the traceability links, and the identified links were validated during review meetings and open-discussion sessions made by the students and academic researchers.

C. Object Systems

Regarding the generalization of the results, an important threat is related to the repositories used in the study. To mitigate this threat, we selected three systems with different characteristics including open source and industrial projects. In particular, we use an open source systems (ArgoUML) and two industrial systems (eXVantage and AgilePlanner).

D. Naming Convention Applicability

Figure 7 shows that naming conventions are generally applied on two of the experimented systems, namely ArgoUML and AgilePlanner. The figure also highlights that on these two systems there is in general a one-to-one relationship between unit tests and tested classes. The scenario is completely different on eXVantage, where naming conventions are generally not applied and there are several unit tests where the number of tested classes is more than one. This analysis suggests a correlation between the use of conventions to name a unit test and the number of tested classes. In general, the developers correctly name the unit test when there is only one tested class. Otherwise, the name of the unit test tends not to follow any convention because it is challenging to encapsulate in one name all the tested classes. In summary, the results shown in Figure 7 suggest that generally developers write a unit test for each class and in this case they also apply naming conventions. However, there are a significant number of cases with multiple tested classes and thus where the convention is not followed.

E. The Role of the Last Assert Statement

SCOTCH identifies an initial set of tested classes by slicing with respect to the last assert statement in each unit-test method. Even if there are several assert statements, we assume that the last assert statement is the statements used to compare the actual outcome with the oracle. The results achieved in our prior work [4] and those presented in this paper support this assumption. However, we also compare
the accuracy of SCOTCH when not only the last assert statement is considered as the slicing criterion, but also when the last two and even the last three assert statements are used as the slicing criterion. The results achieved did not reveal any improvement at all in terms of recall. Indeed, on ArgoUML considering the last three assert statements we obtained a lower precision, which negatively impacts the F-measure (0.76 vs 0.79).

F. Conceptual Coupling as Filtering Strategy

We proposed the use of conceptual coupling to filter the classes in the STS in order to prune-out some helper classes retrieved by slicing. Our conjecture was that a unit test and its tested class(es) are conceptually coupled because their (domain) semantics are similar. However, other coupling metrics, such as structural coupling metrics, could also be used to filter the STS. To highlight the benefits provided by the exploited conceptual coupling measure and to provide further evidence to our conjecture, we also used Information-Flow-based coupling (ICP) [27] to filter the STS. We compared the accuracy achieved with ICP to that obtained using CCBC. Also, with ICP we considered several thresholds. In general, the best results are again achieved setting \( \lambda = 0.95 \) (the same as obtained using CCBC). The results achieved indicate that the conceptual coupling is more accurate in discriminating between helper and tested classes (see Table VI). Indeed, we observed that the number of calls performed by a unit test to a tested class is comparable to the number of calls performed to an helper classes. This does not allow ICP to efficiently discriminate between helper and tested classes.

The ability of CCBC to filter-out false positives from the STS could suggest the use of CCBC as recovery technique. For this reason, we also use CCBC alone to recover links between unit tests and code classes of the object systems. The results highlight that CCBC is suitable only as filtering technique, since its accuracy in terms of F-measure as recovery technique is very low, i.e., 18% on AgilePlanner, 14% on eXVantage, and 5% on ArgoUML.

G. JavaSlicer Limitations

During our study, we observed some limitations of the tool used to slice the unit tests. Indeed, JavaSlicer cannot slice some unit tests. Figure 8 shows a snapshot of unit test TestModelEventPump from ArgoUML where JavaSlicer fails to slice the last assert statement – assertTrue(eventcalled). In particular, in this case the tool returns an empty slice. However, we observed only a few such examples (two for AgilePlanner, four for ArgoUML, and only one for eXVantage). We decided to not exclude these cases from our analysis. Thus, we believe that more accurate results can be achieved using a more robust slicing tool.

VI. CONCLUSION AND FUTURE WORK

This paper presented SCOTCH, a novel approach for recovering traceability links between unit tests and tested classes based on dynamic slicing and conceptual coupling. SCOTCH identifies the dependencies between a unit test and the tested classes in two steps. First, it applies dynamic slicing to identify an initial set of classes that affect the unit test’s last assert statement. Then, it uses conceptual coupling to retain only the most strongly coupled classes. As the data bears out, tested classes have a stronger semantic coupling with the unit test than helper classes (e.g., classes used to create the testing environment).

The accuracy of SCOTCH has been empirically evaluated on two industrial systems, namely AgilePlanner and eXVantage, and on the open source system ArgoUML. As a benchmark, the recovery accuracy of SCOTCH was compared with the accuracy of previous approaches, namely NC, LCBA and DFA. The results of this empirical evaluation indicate that the new approach has the highest accuracy and applicability.

Future work includes experiments in different contexts to corroborate our findings. Also, we will consider other heuristics that may improve traceability link recovery. Finally, we plan to exploit the identification of traceability links in a semi-automatic approach that maintains consistency between unit tests and related source code during refactoring.
VII. ACKNOWLEDGEMENTS
Dave Binkley is supported by NSF grant CCF 0916081.

REFERENCES