

Evaluating test-to-code traceability recovery methods through controlled experiments

Abdallah Qusef¹, Gabriele Bavota¹, Rocco Oliveto^{2,*†}, Andrea De Lucia¹ and David Binkley³

¹*University of Salerno, via Ponte don Melillo, 84084 Fisciano SA, Italy*

²*University of Molise, C.da Fonte Lappone, 86090 Pesche IS, Italy*

³*Loyola University Maryland, 4501 North Charles Street, Baltimore, USA*

SUMMARY

Recently, different methods and tools have been proposed to automate or semi-automate test-to-code traceability recovery. Among these, Slicing and Coupling based Test to Code trace Hunter (SCOTCH) exploits slicing and conceptual coupling to identify the classes tested by a JUnit test. However, until now the evaluation of test-to-code traceability recovery methods has been limited to experiments assessing their tracing accuracy rather than the actual support these methods provide to a software engineer during traceability recovery tasks. Indeed, a research method or tool has a better chance of being transferred to practitioners if it is supported by empirical evidence. In this paper, we present the results of two controlled experiments carried out to evaluate the support given by SCOTCH during traceability recovery, when compared with other traceability recovery methods. The results show that SCOTCH is able to suggest a higher number of correct links with higher accuracy, thus sensibly improving the performances of software engineers during test-to-code traceability recovery tasks. Copyright © 2012 John Wiley & Sons, Ltd.

Received 9 February 2012; Revised 2 July 2012; Accepted 4 September 2012

KEY WORDS: traceability recovery; slicing; conceptual coupling; empirical studies

1. INTRODUCTION

Test-to-code traceability (relationships between tests and the code under test) is an indispensable asset during software development and evolution. For example, unit tests represent 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, test-case traceability can play a key role in preserving consistency during refactoring [1]. Indeed, often refactoring of the code is followed by refactoring of the related unit tests [1, 2]. This later step is greatly simplified and can be largely automated given a mapping between unit tests and the corresponding tested code.

Unfortunately, traceability links are rarely explicitly maintained or even documented [1, 3]. As indicated by a recent study, we performed on 18 software projects (two industrial and 16 open source); of the 637 JUnit classes considered, we found indications of the actual tested classes in the comments of only 146 (23%) [4]. This is true regardless of the test origin. For example, with model-based testing, tests are created based on software interface descriptions, whereas in test-driven development, tests are written prior to the code. Unfortunately, well before the code is in the hands of a maintenance engineer, the traceability links have been lost. This shortcoming is in part because

*Correspondence to: Rocco Oliveto, University of Molise, C.da Fonte Lappone, 86090 Pesche IS, Italy.

†E-mail: rocco.oliveto@unimol.it

maintaining an effective regression suite in light of changes in production code requires updating test-to-code traceability links [5]. However, there is a lack of support for this activity in contemporary software engineering environments and tools [3].

To help a software engineer maintain and identify test-to-code traceability information, we presented Slicing and Coupling based Test to Code trace Hunter (SCOTCH) [6]. SCOTCH presently focuses on the links between unit tests and the class under test. Thus, it takes a JUnit test as input and uses dynamic slicing and conceptual coupling to suggest a list of candidate tested classes. Experiments with three software systems indicate that, in comparison with its predecessors, namely approaches based on naming conventions (NC) [7] and Last Call Before Assert (LCBA) [7], SCOTCH is not only more accurate but also has wider applicability and higher stability [6].

We integrated SCOTCH into Eclipse,[‡] a widely used integrated development environment, to facilitate technology transfer. However, integration alone is insufficient. Indeed, a research method or tool has a better chance of being adopted by practitioners if its usefulness is investigated through empirical study [8]. Indeed, until now, the evaluation of SCOTCH (and, in general, that of test-to-code traceability recovery methods) has been limited to assessing its tracing accuracy, rather than the actual support it provides to a software engineer during traceability recovery tasks. Thus, user studies are needed to understand to what extent SCOTCH can support a software engineer during test-to-code traceability. This kind of empirical analysis is an important part of technology transfer, a known difficult process [9, 10], because empirical evidence provides important support for the adoption of software techniques [11–14]. This paper presents results from experiments into the practical impact that SCOTCH brings.

We conducted a set of controlled experiments to statistically analyze to what extent SCOTCH eases the test-to-code traceability recovery task of a software engineer. The experiments involved a total of 32 Bachelor's students from the University of Salerno and the University of Molise, who had to identify the classes tested by a set of JUnit tests with the support of a traceability recovery tool. Each subject performed two traceability recovery tasks on two different systems, namely ArgoUML[§] (an open source system) and eXVantage[¶] (an industrial system). In each task, subjects were given links recovered using SCOTCH or one of two base-line techniques (NC or LCBA). The results indicate that SCOTCH provides superior support to a software engineer when compared with other traceability recovery methods, as it allows the software engineer to identify more correct links with higher accuracy.

The paper is organized as follows. Section 2 discusses related work, whereas Section 3 presents SCOTCH and its implementation in Eclipse, together with an assessment of its traceability recovery performance on three software systems. Section 4 describes the design of the controlled experiments, whereas Section 5 reports and statistically analyzes the results. Section 6 discusses threats to validity, whereas conclusion and directions for future work are reported in Section 7.

2. RELATED WORK

Several methods have been proposed to recover traceability links between software artifacts of different types. The proposed approaches can be classified according to the method adopted to derive traceability links, such as event-based [15], scenario-based [16–18], rule-based [19], dynamic analysis-based [20, 21], data mining-based [22], and textual analysis-based [23–27]. In this section, we discuss related literature, focusing on approaches that identify test-to-code traceability information and on empirical studies conducted to evaluate the support given by test-to-code traceability recovery tools.

2.1. Approaches to support test-to-code traceability recovery

There is scant existing tool support for establishing and maintaining test-case traceability links. Today's integrated development environments offer little support to the developer trying to browse

[‡]<http://www.eclipse.org>

[§]<http://argouml.tigris.org/>

[¶]<http://www.research.avayalabs.com/>

between unit tests and the classes under test. For example, the Eclipse Java environment^{||} provides a wizard to create unit tests and suggests that the developer indicates the corresponding class under test. Also, Eclipse offers a search-referring test menu entry that retrieves all unit tests that call a selected class or method. To improve upon these features, Bouillon *et al.* [28] present a JUnit Eclipse plug-in that uses static call graphs [29] 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 [30] proposed two approaches to connect test cases and code. The first uses name matching with some manual linking to map code functions to requirement-model functions, whereas the second links test cases and code functions using time stamps. Van Rompaey *et al.* [7] compared four approaches: NC, LCBA, Latent Semantic Indexing (LSI) [31], and co-evolution. NC simply identifies the classes under test by removing the string 'Test' from the 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 with methods 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 tests and the classes of the systems, whereas co-evolution assumes that the tested class co-evolves with its unit test. Although, in general, the latter two approaches have been shown successful in identifying traceability links between different types of artifacts [32, 27] and in identifying logical coupling between source code components [22, 33], Van Rompaey *et al.* [7] show that these approaches are not effective in identifying test-to-code traceability links. The results indicate that NC is the most accurate, whereas LCBA has higher applicability (consistency). Nevertheless, these two approaches have important limitations. NC assumes that developers follow specific naming conventions, which force the identification of a single class as tested class. However, such assumptions are not always followed in practice, especially in industrial contexts [34]. LCBA's limitations come from its use of the static call graph. Because it returns the classes associated with the last called method before the 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 [34].

To overcome these limitations, we proposed the use of data flow analysis (DFA) to recover test-to-code traceability links [34]. The DFA-based 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 data dependences. This approach ignores control dependences and other aspects, such as aliasing, interprocedural flow, and inheritance. Empirical results indicate that the approach identifies tested classes with higher precision than NC and LCBA but fails to retrieve a sensible number of tested classes (and thus has low recall).

Slicing and Coupling based Test to Code trace Hunter [6] shares with the aforementioned approaches the use of assert statements to derive test-to-code traceability links. It uses dynamic slicing [35, 36] to identify the set of classes that affect the last assert statement. SCOTCH identifies a larger number of correct classes than its predecessors, in part, because the use of slicing improves recall [6]. However, the set identified by dynamic slicing is an overestimate of the set of tested classes. This negatively impacts SCOTCH's precision. Therefore, conceptual coupling is used to discriminate between the actual tested classes and the helper classes. This filtering further improves the accuracy of the approach. In this paper, we empirically evaluate through controlled experiments the support provided by SCOTCH to a software engineer during test-to-code traceability recovery.

2.2. Empirical evaluation of traceability recovery tools

Our work is closely related with papers investigating the way a software engineer uses traceability recovery tools. Several approaches based on IR methods have been proposed to identify traceability links between different types of artifacts [24, 27]. Such methods recover traceability links on the basis of the similarity between the text contained in the software artifacts. The conjecture is that artifacts having a high textual similarity probably share several concepts, so they are likely good

^{||}<http://www.eclipse.org/jdt/>

candidates to be traced from one to another. Thus, a similarity threshold is used to consider as candidate traceability links only the pairs of artifacts with similarity above a given threshold.

Few user studies have considered the actual support given by a tool to the end user while performing traceability recovery. Antoniol *et al.* [24] presented the results of a preliminary study where they compare the IR-based approaches against a ‘grep’ brute force traceability link recovery demonstrating the benefits of a more sophisticated technology, such as an IR method.

De Lucia *et al.* [32] integrated a traceability recovery tool (based on LSI) in Advanced Artefact Management System (ADAMS), a fine-grained artifact management system [37]. To validate the traceability recovery method and tool, the authors performed a case study where about 150 users allocated in 17 software development projects used the tool. Each project team included undergraduate students with development roles and Masters students responsible for project and quality management. Using the traceability features of ADAMS, the graduate students were also in charge of maintaining up-to-date traceability between the software artifacts produced by their team. In particular, they had the option of tracing links manually or using tool support. At the end of the experiment, the authors analyzed the links traced by students to determine if the traceability recovery tool was used during the development process. They observed that almost all the links traced by the students were traced with the tool support. Moreover, at the end of the experiment, students evaluated ADAMS through a questionnaire. The analysis of the answers revealed that students found the tool useful during the traceability recovery process.

De Lucia *et al.* [38] also conducted a controlled experiment and a replication to statistically analyze how the tracing accuracy of a software engineer is affected by the use of an IR-based traceability recovery tool. The experiment involved 32 Masters students at the University of Salerno, Italy, who had to perform (with and without tool support) two traceability recovery tasks on a software repository of a completed project. The results demonstrated that the use of a traceability recovery tool significantly improves the tracing accuracy of the software engineer. In particular, it significantly reduces the percentage of false positives, although it does not significantly help to recover more correct links (better recall). Moreover, it was observed that the tool significantly reduces the time spent by a software engineer tracing links.

A recent controlled experiment evaluated an IR-based traceability recovery process [39]. In particular, the authors compared the tracing performances achieved by subjects using the ‘one-shot’ process (where the full-ranked list of candidate links is proposed) and the incremental process (where a similarity threshold is used to cut the ranked list, and the links are classified step-by-step). The analysis of the results showed that, in general, the incremental process improves the tracing accuracy and reduces the effort to analyze the candidate links [39].

We share with the aforementioned works the need to statistically analyze how the tracing accuracy of a software engineer is affected by the use of SCOTCH in order to verify whether the improvement in the correctness of the classes identified by SCOTCH really represents an advantage for the software engineer performing test-to-code traceability recovery. For this reason, we have conducted a set of controlled experiments aiming to provide such a statistical evidence.

3. TEST-TO-CODE TRACEABILITY WITH SLICING AND CONCEPTUAL COUPLING

This section presents the traceability recovery method SCOTCH [6] and its implementation in Eclipse.

3.1. The traceability recovery method

Identification of tested classes begins with a unit-test method. In general, each method includes two types of statements: non-assertion statements 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’ [40]). In practice, this is common because developers use assert statements to verify the testing environment before verifying the behavior of tested classes.

Slicing and Coupling based Test to Code trace Hunter identifies the set of tested classes using the two steps highlighted (by dashed boxes) in Figure 1. In the first step, SCOTCH executes each test method in a JUnit class (test suite) and identifies the last assert statement instance in each execution trace. Thus, these assert-statement instances are used as the *slicing criterion*, the starting point of a slice. Previous study indicates that the last assert statement tends to be affected by methods of the tested class [34]. Note that using only the last executed assert statement also reduces the retrieval of helper classes and also reduces slicing time. SCOTCH then computes the backward dynamic slices [35, 36] with respect to the slicing criteria by using the *JavaSlicer*** tool. *JavaSlicer* relies on the approach described in [41] to compute dynamic dependence graphs for executable Java programs. In our approach, *JavaSlicer* is used to produce the traces of JUnit tests executions and to compute dynamic backward slices on them. Note that dynamic slicing is preferred to static slicing because the natural way to link a unit test to the related code is by executing the test. The slices collectively capture the set of statements that affect the computation of the final assert-statement instance in each execution trace. SCOTCH then extracts the set of classes with method invocations encountered in the backward dynamic slices. This set is likely an over approximation because it may contain classes belonging to standard libraries (e.g., *String*) and (unwanted) mock objects. All such classes are false positives that should be removed from the set of CTS. The first of these are easily removed using a *stop-class list* (i.e., a list of classes from standard libraries such as *java.**, *javax.**, *org.junit.**), whereas the others are pruned out by SCOTCH's second step. We referred to the classes retrieved by slicing and then stop-list filtered as the Starting Tested Set (STS), which is the output of step 1.

Figure 2 shows a fragment of code from the JUnit class *RemoveElementsTest* taken from the eXVantage project. The JUnit class is related to the class *ConnectedGraph* where it tests the *removeElement* method. In this JUnit class, there are two assert statements: the *fail* statement at line 43 and the *assertTrue* statement at line 45. Analyzing the code, it is easy to see that during the execution of the *testRemoveElement* method, only one of these two statements is executed. Suppose that no errors are detected in the tested class. Thus, the generation of a *DisconnectedGraphException* is expected (and thus the *assertTrue* statement is executed). In this case, SCOTCH will identify *assertTrue* as the last executed assert statement. Thus, the dynamic slice will include all the statements affecting *assertTrue* (i.e., those on lines 44, 42, 39, 34, 27, 28, 25, and 24). The classes extracted from this slice are *DisconnectedGraphException*, *Set*, *EdgeElement*, *NodeElement*, *ConnectedGraph*, and *ConnectedGraphFactory*. The set of classes retrieved using slicing includes the class *Set* (i.e., *java.util.Set*) belonging to the standard library. Thus, it is pruned out by the stop-class list.

In addition to the tested class *ConnectedGraph*, the STS includes several helper classes (which also affect the last assert statement). These classes negatively impact the accuracy of the result. The second step of SCOTCH uses conceptual coupling to prune-out unwanted classes and thus improve accuracy. The filtering of the STS is based on the conjecture that unit tests will be semantically related to the classes under test (in particular, their textual similarity will 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) [42]. CCBC uses LSI [31], 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. Having such a representation is possible to compute the conceptual coupling between two methods m_i and m_j as the cosine of the angle between their corresponding vectors [42]:

$$\text{CCM}(m_i, m_j) = \frac{\vec{m_i} \cdot \vec{m_j}}{\|\vec{m_i}\| \cdot \|\vec{m_j}\|}$$

where $\vec{m_i}$ and $\vec{m_j}$ are the vectors corresponding to the methods m_i and m_j , respectively, and $\|\vec{x}\|$ represents the Euclidean norm of the vector x [43]. Then, the conceptual coupling between two classes c_i and c_j is defined as follows:

**<http://www.st.cs.uni-saarland.de/javaslicer>

††We used the LSI implementation provided by the R platform to compute the CCBC in SCOTCH.

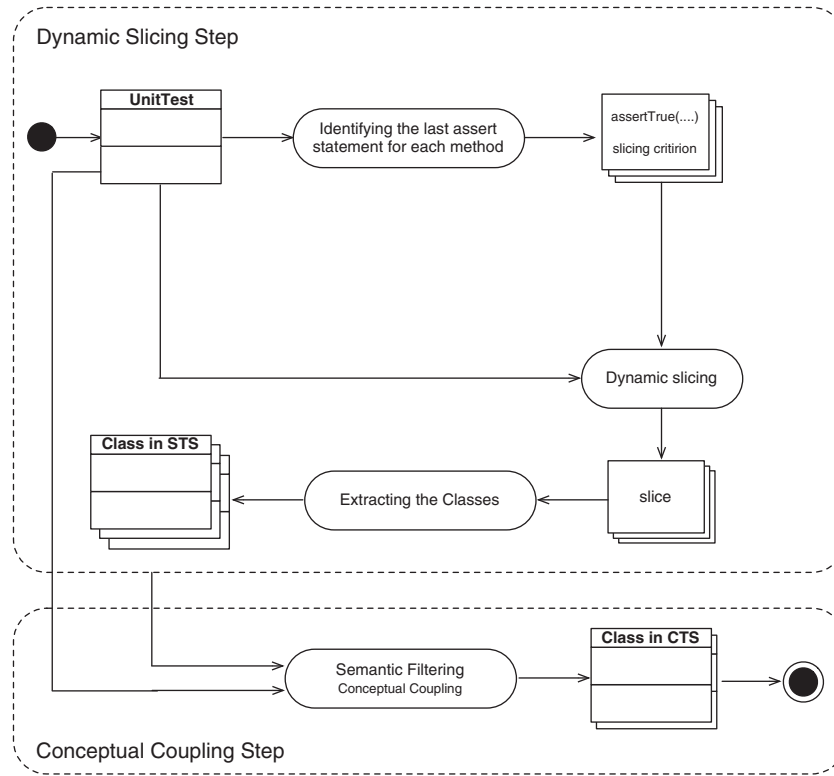


Figure 1. Slicing and Coupling based Test to Code trace Hunter traceability recovery process [6].

$$CCBC(c_i, c_j) = \frac{\sum_{m_h \in c_i} \sum_{m_k \in c_j} CCM(m_h, m_k)}{|c_i| \times |c_j|}$$

where $|c_i|$ ($|c_j|$) is the number of methods in c_i (c_j). Thus, $CCBC(c_i, c_j)$ is the average of the coupling between all unordered pairs of methods from class c_i and class c_j .

We use CCBC to rank the classes of the STS 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, a threshold is used to truncate the ranked list and identify the top coupled classes as the set of CTS. Defining a ‘good’ threshold 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 [24] 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 lower than $\lambda\%$ of the conceptual coupling between the unit test and the top ranked class. The parameter λ has been empirically estimated and good results are achieved by setting $\lambda = 0.95$ [6].

Continuing the example shown in Figure 2, computing the CCBC between the unit test shown in Figure 2 and the classes in the STS identified by slicing produces 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.

```

public class RemoveElementsTest extends TestCase
{
    public void testRemoveElement () throws Exception {
        ConnectedGraphFactory factory = ConnectedGraphFactory.newFactory();
        ConnectedGraph graph = factory.newGraph();
        NodeElement node1 = graph.getStartNode();
        NodeElement node2 = graph.newNode();
        NodeElement node3 = graph.newNode();
        NodeElement node4 = graph.newNode();
        NodeElement node5 = graph.newNode();

        EdgeElement edge1, edge2, edge3, edge4, edge5;
        edge1 = graph.newEdge(node1, node2);
        edge2 = graph.newEdge(node2, node3);
        edge3 = graph.newEdge(node3, node4);
        edge4 = graph.newEdge(node4, node5);
        edge5 = graph.newEdge(node5, node3);

        Set<GraphElement> removedElement = null;

        try {
            removedElement = graph.removeElement(edge2);
            fail("DisconnectedGraphException expected");
        } catch (DisconnectedGraphException e) {
            assertTrue(true);
        }
    }
}

```

Figure 2. Fragment of *RemoveElementsTest* from eXVantage.

3.2. Tracing accuracy evaluation

This section summarizes results from a previous empirical evaluation of SCOTCH's tracing accuracy [6] where we compared SCOTCH with the use of NC, LCBA, and DFA. The interested reader can find more details about the experimental design and results in our prior work [6].

3.2.1. Design. The experiment was conducted on two industrial systems, AgilePlanner (32 JUnit classes) and eXVantage (17), and one open source system, ArgoUML (75). Thus, the total number of JUnit classes involved in the experiment was 124, and for each of them, we used the four techniques, NC, LCBA, DFA, and SCOTCH, to identify the classes under test. Then, to evaluate and compare the accuracy of each technique, we used two well-known IR metrics, recall and precision [43]:

$$\text{recall} = \frac{|\text{correct} \cap \text{retrieved}|}{|\text{correct}|} \quad \text{precision} = \frac{|\text{correct} \cap \text{retrieved}|}{|\text{retrieved}|}$$

where *correct* and *retrieved* represent the set of correct links and the set of links retrieved by the traceability recovery method, respectively. Because the two aforementioned metrics measure two different concepts, we assessed the global accuracy of the recovery methods using the F-measure (the harmonic mean of precision and recall):

$$\text{F-measure} = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}}$$

Note that to evaluate the experimented techniques, we need to know the set of correct links (i.e., *correct*). However, we did not find any documentation describing the actual dependencies between unit tests and code classes. This demonstrates that such links are not explicitly maintained, and they might be retrieved when needed during software evolution. Thus, to create an oracle for the experiments, three PhD students of the University of Salerno manually identified the links

between the unit tests and the tested classes. Students individually analyzed the code 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.

3.2.2. Results. First, we analyzed the effectiveness of SCOTCH and how the conceptual coupling threshold affects its accuracy. Figure 3 shows the F-measure using different values of λ , 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 (because of 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.

Having identified SCOTCH's best configuration, we compared its accuracy with the other techniques (i.e., NC, LCBA, and DFA). Figure 4 compares the F-measure achieved by SCOTCH (with $\lambda=0.95$) and those for the three benchmarking techniques. Striking in Figure 4 is how NC provides very good accuracy for ArgoUML, where naming conventions are strictly followed, whereas in the other two projects, NC performs much worse. Next, LCBA retrieves generally more classes (resulting in a higher recall) than NC (except on ArgoUML). However, in several cases, LCBA fails to correctly retrieve tested classes because the classes with methods called before the assert statements are not the tested classes. Finally, DFA has an accuracy slightly better than LCBA [34], but it generally fails to retrieve the correct tested class when the assert statement does not contain 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 because the dynamic slicing method also exploits control dependencies that are completely ignored by DFA.

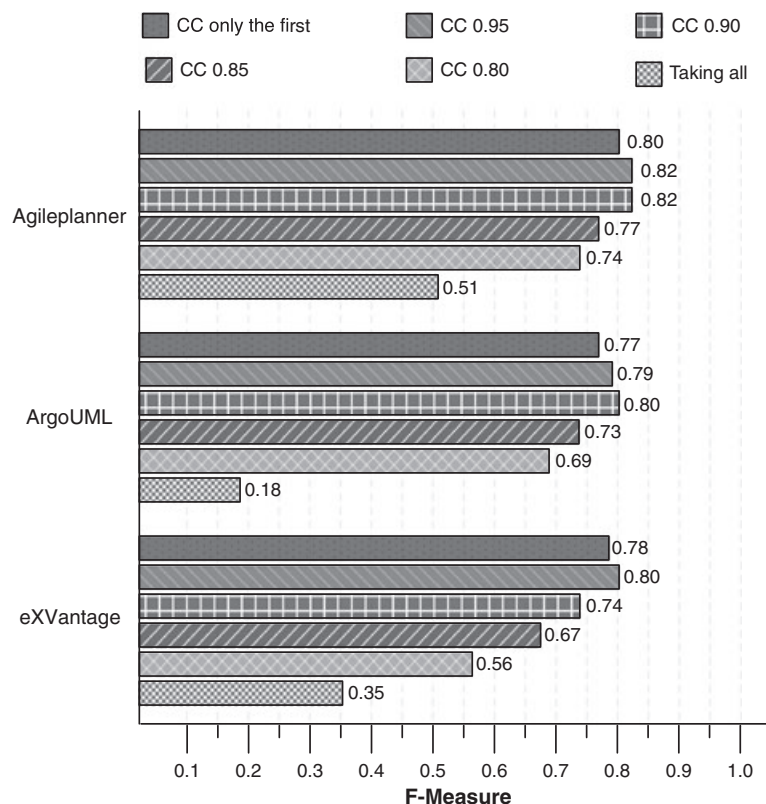


Figure 3. Results of the proposed approach using different thresholds.

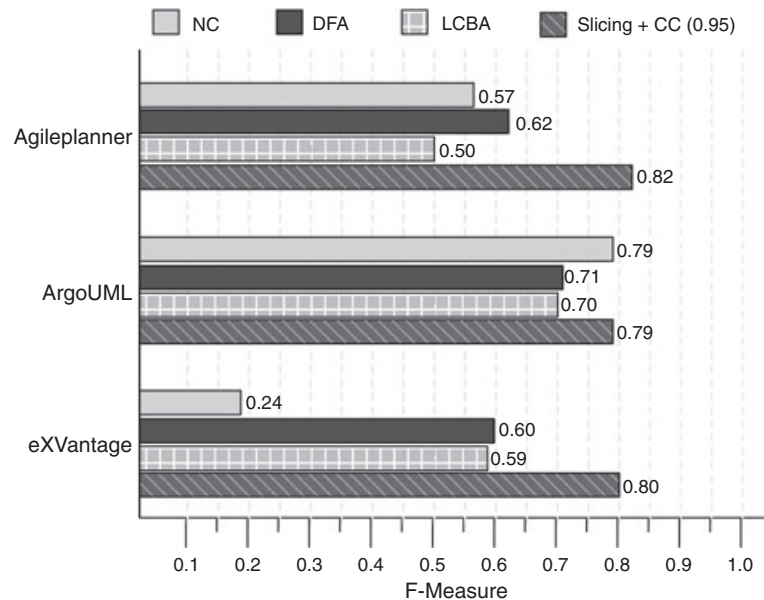


Figure 4. Accuracy of the experimented traceability recovery methods.

For all three systems, Figure 4 indicates the new approach that provides the best performance. With AgilePlanner and eXVantage (where naming conventions are badly applied), SCOTCH significantly outperforms NC, LCBA, and DFA. For ArgoUML, SCOTCH overcomes LCBA and DFA, and it is able to obtain the same performances as NC. Note that ArgoUML represents the best case for NC because 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.

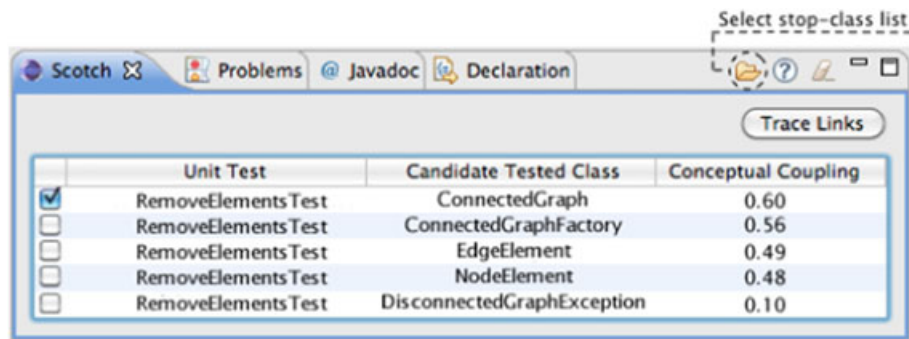
3.3. Integrating the traceability recovery method in Eclipse

Slicing and Coupling based Test to Code trace Hunter has been implemented as an Eclipse plug-in. The plug-in contributes a new view to the Eclipse workbench that allows a developer to recover traceability links between unit tests and tested classes. Suppose that the developer wants to identify the classes tested by the JUnit class reported in Figure 2. To this aim, she selects this unit test in the Eclipse Package Explorer and presses a button in the view to start the recovery process. Note that the developer can also select a package (or a project), and SCOTCH will recover links for each unit test automatically identified by detecting all the classes that inherit from class *TestClass*.

The results are then reported in the view shown in Figure 5. This view shows all the classes of the STS. In the example, SCOTCH slices with respect to the assert statement *assertTrue(true)*, which yields the initial set of classes shown in Figure 5. In addition, for each candidate link, SCOTCH provides a checkbox that allows the developer to trace (or not) the link between the unit test and each candidate tested class. Furthermore, SCOTCH highlights the classes of the CTS (output of step 2) by checking the appropriate checkbox to suggest to the developer which classes are truly tested classes. In the example, SCOTCH checks the class *ConnectedGraph*, which correctly identifies the tested class. Once the developer has finished the classification, the traced links are stored in an XML file. Finally, the interface allows the developer to customize the *stop-class list* used by SCOTCH to filter the set of classes identified by slicing by selecting the *stop-class list* button in the SCOTCH view (see the upper left of Figure 5).

4. EXPERIMENT DESIGN

Although the previous study summarized in Section 3.2 [6] showed that SCOTCH is more accurate in identifying test-to-code traceability links, it stopped short in demonstrating SCOTCH's usefulness to a software engineer during the traceability recovery process. For this reason, in this paper, we present



Unit Test	Candidate Tested Class	Conceptual Coupling
☒ RemoveElementsTest	ConnectedGraph	0.60
☐ RemoveElementsTest	ConnectedGraphFactory	0.56
☐ RemoveElementsTest	EdgeElement	0.49
☐ RemoveElementsTest	NodeElement	0.48
☐ RemoveElementsTest	DisconnectedGraphException	0.10

Figure 5. Slicing and Coupling based Test to Code trace Hunter: traceability recovery view.

results from two controlled experiments that complement our previous study as they aim to evaluate SCOTCH when used by software engineers during traceability recovery.

In the following sections, we provide details on the design of these experiments. The description follows a template originating from the Goal-Question-Metric paradigm [44] as described by Wohlin *et al.* [45].

4.1. Definition and context

The goal of our study is to deeply analyze the support given by SCOTCH when used by software engineers during the identification of the classes tested by a JUnit test. In particular, we aim at investigating whether the improvement in the correctness of the classes identified by SCOTCH as compared with NC and LCBA lead to better human performances. The tracing performances of subjects were measured in terms of correct links traced and false positive links incorrectly traced (tracing errors).

We executed two experiments with the same design but different recovery techniques. The first experiment compares SCOTCH with NC, whereas the second one compares SCOTCH with LCBA. Both experiments were conducted at the University of Salerno (Italy) and replicated at the University of Molise (Italy) with Bachelor's students attending the software engineering course. Table I reports the number of subjects involved by location and experiment. Within each experiment, all the subjects were from the same class with comparable academic backgrounds but different demographics. All the students had knowledge of software development and testing, as well as software artifact traceability.

Subjects were asked to identify the classes tested by a set of JUnit tests from two software systems: ArgoUML and eXVantage. We chose these two systems because they exhibit different characteristics. In particular, naming conventions are generally applied only in ArgoUML, whereas with eXVantage, such conventions are generally not followed, and there are several JUnit classes that test more than one class. Moreover, ArgoUML is an open source system, whereas eXVantage is an industrial one.

Table II shows the size of the two subject systems in terms of classes and KLOC (thousands of lines of code). The table also shows the number of unit tests (total and usable in our experiments) and their corresponding KLOC. Some unit tests were not usable because they are not related to actual classes under test (for example, they test mock objects, network settings, and complex database queries).

Each experiment was performed in a controlled laboratory setting where subjects were asked to identify the classes under test for 10 JUnit tests randomly selected from the usable tests of ArgoUML and 10 randomly selected from the usable tests of eXVantage.** The number of selected JUnit tests for the experiment is low as compared with the number of JUnit tests in the subject systems (especially for ArgoUML). However, each experiment was limited to 3 h to limit subject fatigue. This meant that subjects had, on average, 9 min per JUnit test. It is not practical to perform such experiments using substantially larger sets of JUnit tests.

Finally, each experiment considered the four pairings of the two systems (ArgoUML and eXVantage) with the two experimented techniques. SCOTCH was used in both experiments where it was compared with NC and LCBA, respectively. These two methods provide a baseline for analyzing the effectiveness

**We used the *Random Java* class to write a simple program selecting the 20 JUnit classes used in the experiments.

Table I. Number of subjects involved in the experiments by location and experiment.

Experiment	University of Salerno (First experiment)	University of Molise (Replication)	Total
1. SCOTCH versus NC	8	8	16
2. SCOTCH versus LCBA	8	8	16

SCOTCH, Slicing and Coupling based Test to Code trace Hunter; NC, naming conventions; LCBA, Last Call Before Assert.

Table II. Characteristics of the experimented systems.

Experimental		Classes	KLOC	Unit tests			
				Total		Usable	
				Number	KLOC	Number	KLOC
ArgoUML	Open source UML modeling tool	1430	124	163	12	75	7
eXVantage	Novel testing and generation tools	348	28	30	5	17	4

KLOC, thousands of lines of code.

of the novel traceability recovery technique, SCOTCH. As detailed in Section 4.3, a subject performed two of the four possible pairings in two laboratory sessions.

4.2. Hypothesis formulation and variable selection

We performed a single factor experiment, where the technique used to identify the classes tested by a JUnit test represents the main factor. This factor is denoted as *Technique* and has two values:

1. SCOTCH: subjects performed the assigned traceability recovery task using SCOTCH;
2. Control: subjects performed the assigned traceability recovery task using one of the techniques NC or LCBA.

It is worth noting that with both treatments, subjects were free to enrich the set of links identified by the tool with *manually identified* links.

To better assess the effect of *Technique*, it was necessary to control other factors (called co-factors) that may impact the results achieved by the subjects and be confounded with the effect of the main factor. In the context of our study, we identify the following co-factors:

- *System*: ArgoUML or eXVantage and
- *Session*: Session₁ or Session₂, the two laboratory sessions.

The goal of our study is to analyze how the use of SCOTCH affects the tracing performances of subjects during traceability link identification (i.e., correct links traced and false positive links incorrectly traced). Thus, we formulated the following null hypotheses related to our research questions:

- H_{0_c} : SCOTCH does not increase the percentage of correct links traced by the software engineer;
- H_{0_e} : SCOTCH does not reduce the percentage of links erroneously traced by the software engineer (false positives).

When the null hypothesis can be rejected, it is possible to formulate an alternative hypothesis, which admits a positive effect of SCOTCH over the control traceability recovery method (i.e., NC or LCBA) [45]. Note that these two hypotheses are one-tailed because we were interested in testing whether or not retrieval performance improves with the use of SCOTCH.

We again use recall and precision; although in Section 3.2, these metrics are used to evaluate the accuracy of a traceability recovery method, here, they are used to measure the ability of subjects to correctly trace JUnit tests onto tested classes; thus,

$$\text{recall} = \frac{|correct \cap traced|}{|correct|} \% \quad \text{precision} = \frac{|correct \cap traced|}{|traced|} \%$$

In particular, recall measures the percentage of tested classes correctly traced by a subject, whereas precision measures the percentage of traced classes that are correct. Therefore, recall and precision are the dependent variables used to test our null hypotheses H_{0_r} and H_{0_e} , respectively.

Clearly, as with the experiment summarized in Section 3.2 [6], we need to know the set of correct links to evaluate the subject's performances. We used the same oracle as used in the previous experiment.

4.3. Experiment design and procedure

During each experiment, subjects were divided into four groups. Table III shows the assignment of the four combinations of system and recovery technique to each group (A–D) in the two laboratory sessions. This arrangement follows a counter-balanced experimental design, which ensures that each subject works with each of the two systems and two of the three recovery techniques. Each subject analyzed the selected JUnit tests in a random order to combat learning effects. Finally, the chosen design supports the use of two-way and three-way analysis of variance (ANOVA) tests [46] to analyze the effects of multiple factors.

The two experiments follow the same design. In particular, *Control* in Table III represents the benchmark traceability technique (NC in the first experiment and LCBA in the second). The experimental design allows to assess the support given by SCOTCH in comparison with the other two techniques without comparing NC and LCBA head-to-head, which was out of the scope of the paper.

In each experiment, subjects worked individually. Before the experiments, we presented them a description of the goals of traceability recovery. We also showed a presentation with detailed instructions on the task to be performed. During the experiment, each subject was provided with the following material:

- handouts of the introductory presentation;
- an online copy of the source code for each system;
- a spreadsheet file with the CTS identified by the particular traceability recovery technique. A spreadsheet file was used to avoid confounding the results with how well subjects could use each tool's interface;
- the survey questionnaire shown in Table IV.

Table III. Experiment design (*Control* is instantiated as NC in the first experiment and LCBA in the second one).

	Group			
	A	B	C	D
Session ₁	ArgoUML/Control	ArgoUML/SCOTCH	eXVantage/Control	eXVantage/SCOTCH
Session ₂	eXVantage/SCOTCH	eXVantage/Control	ArgoUML/SCOTCH	ArgoUML/Control

SCOTCH, Slicing and Coupling based Test to Code trace Hunter.

Table IV. Post-experiment questionnaire.

Session ID: 1 ☐ or 2 ☐		Scale from yes to no				
Id	Question	5	4	3	2	1
Q1	I had sufficient time to complete the task	☐	☐	☐	☐	☐
Q2	The domain of the system was clear to me	☐	☐	☐	☐	☐
Q3	The objectives of the experiment were clear to me	☐	☐	☐	☐	☐
Q4	The task was perfectly clear to me	☐	☐	☐	☐	☐
Q5	I experienced major difficulties in performing the task	☐	☐	☐	☐	☐
	(Overall I found recovers a difficulty task)					
Q6	I found the list of proposed tested classes useful	☐	☐	☐	☐	☐

The survey questionnaire was filled out by each subject at the end of each session. Thus, each subject filled out two survey questionnaires. Each questionnaire was composed of questions expecting closed answers on a Likert scale [47] from 1 (strongly agree) to 5 (strongly disagree). Questions Q1 to Q5 aim to assess qualitative aspects of the identification process such as if the system's domain was clear or if the task was clear, whereas question Q6 investigates the usefulness of the classes proposed by the traceability recovery technique.

Before the experiment, we also collected demographic information, such as years at the university, number of passed exams, subject's grade point average, and number of programming exams passed. We also asked subjects whether they had previous work experience, whether they had experience with JUnit, and their number of years of Java experience.

Subjects had one and a half hours for each *Session*. After the task was complete, they submitted the list of identified tested classes for each JUnit test and completed the survey questionnaire. Once collected, all the data (lists of tested classes and survey questionnaires) were statistically analyzed to test our hypotheses. In all our statistical tests, we rejected the null hypotheses for p -values < 0.05 (i.e., we accept a 5% chance of rejecting a null hypothesis when it is true [48]).

We used Student's t -test [48] to investigate our null hypotheses, whereas to analyze the effect of co-factors and their interaction with the main factor, we used the three-way ANOVA [48]. Such an analysis was performed on the whole data set as in similar experiments [45, 49]. This analysis was possible because the experimental design, material, and procedure were exactly the same for the two experiments (i.e., SCOTCH versus NC and SCOTCH versus LCBA). The interaction between factors was also analyzed using interaction plots (line graphs where the means on the dependent variable for each level of one factor are plotted over all the levels of the second factor). The resulting profiles are parallel when there is no interaction and nonparallel when an interaction is present [46]. The chosen experimental design supports the use of these tests.

5. EXPERIMENTAL RESULTS

In this section, we report the analysis of the results achieved in the controlled experiments. Table V reports the descriptive statistics of the dependent variables (recall and precision) for both experiments grouped by *System* and *Technique*. The p -values for Student's t -tests are reported in Table VI. The table also reports descriptive statistics of the impact on subjects of the different techniques used. For each subject, we calculated the difference between the results achieved when tracing links on a system using SCOTCH and the results achieved by the same subject on the other system using a baseline approach, that is, NC or LCBA.

Table V. Descriptive statistics of the dependent variables.

System	Technique	H_{0_c} : recall (%)			H_{0_e} : precision (%)		
		Mean	Median	Standard deviation	Mean	Median	Standard deviation
Both	NC	51.1	50.0	21.0	56.2	55.8	13.2
	LCBA	56.1	60.0	23.8	50.2	52.1	18.2
	SCOTCH	80.9	81.8	13.8	82.1	86.2	15.5
ArgoUML	NC	52.3	54.6	15.9	55.2	50.0	11.3
	LCBA	39.8	31.8	18.1	42.1	39.3	18.7
	SCOTCH	75.1	78.4	14.3	76.4	80.0	19.7
eXVantage	NC	50.0	46.7	26.2	57.2	62.0	15.6
	LCBA	72.5	73.3	16.5	58.3	59.6	14.5
	SCOTCH	86.7	86.7	10.8	88.0	87.9	5.8

SCOTCH, Slicing and Coupling based Test to Code trace Hunter; NC, naming conventions; LCBA, Last Call Before Assert.

Table VI. Student's *t*-test *p*-values and descriptive statistics of differences by subjects.

Test	H_{0_c} : recall				H_{0_e} : precision			
	Mean	Median	Standard deviation	<i>p</i> -value	Mean	Median	Standard deviation	<i>p</i> -value
SCOTCH versus NC	30.3	33.5	24.4	<0.0001	27.3	34.0	23.4	<0.0001
SCOTCH versus LCBA	24.3	20.3	29.7	<0.0001	30.7	24.5	23.6	<0.0001

SCOTCH, Slicing and Coupling based Test to Code trace Hunter; NC, naming conventions; LCBA, Last Call Before Assert.

5.1. Analysis of the percentage of correct links traced

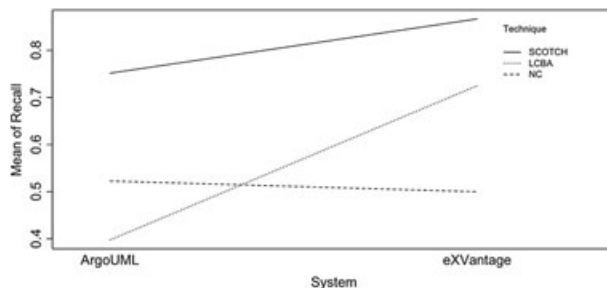
Concerning the influence of *Technique* on the percentage of correct links traced (*recall*), the *t*-tests revealed in both experiments that the *recall* achieved by the subjects using SCOTCH is significantly higher than the *recall* achieved by the subjects when using either of the two control techniques. In particular, the average difference in terms of *recall* between SCOTCH and the two control techniques is about 30%. Thus, we can reject the null hypothesis H_{0_c} , when considering both NC or LCBA as benchmarks.

We manually analyzed the links retrieved by the subjects using the different approaches to determine if the better performances achieved using SCOTCH has a symptomatic cause.

The better recall achieved using SCOTCH with respect to the recall achieved using NC clearly depends on the JUnit classes where naming conventions are not used. Moreover, we found that sometimes not only naming conventions are not used but the name assigned to the class is totally useless and thus provides no assistance to the subjects when trying to identify the tested class(es). An example is the JUnit class *TestExtensionMechanismHelper* implemented in ArgoUML to test the class *Model* in the production code. The better recall achieved using SCOTCH with respect to the recall achieved using LCBA is mainly due to JUnit classes where the class invoked just before an assert statement is not the actual tested class but a helper class. In these cases, most of the subjects were not able to identify the correct class and thus achieved lower recall.

Turning to the influence of the co-factors *System* and *Session* on recall, the ANOVA test confirmed the statistically significant effect of *Technique* ($p < 0.0001$) and also found no significant effect of *Session* ($p = 0.66$). The ANOVA did find a statistically significant effect in the co-factor *System* ($p = 0.002$) as well as a statistically significant interaction between *System* and *Technique* ($p = 0.03$).

To better understand the effect of the *System* on the recall, Figure 6 shows the interaction plot between *Technique* and *System* on recall. The interaction plot highlights the high applicability of SCOTCH. As we can see, subjects using SCOTCH achieved very high recall on both systems. A different situation can be observed by comparing the recall achieved by subjects using NC and LCBA where *System* plays an important role. In particular, on ArgoUML, the number of correct links detected by the subjects using NC is higher than that obtained by using LCBA. The reason is that in ArgoUML, naming conventions are generally applied, and NC is able to suggest more

Figure 6. *Technique* and *System* interaction on recall.

correct links than LCBA. Conversely, with eXVantage, the number of correct links detected by the subjects using LCBA is higher than that obtained by using NC. With this system, naming conventions are not applied, and there are several JUnit tests with more than one tested class. For such systems, most of the links suggested by NC are false positives; thus, the technique provides poor support when identifying tested classes.

Evident in Figure 6 is also an interaction between NC and the other two techniques. An interaction arises when the influence of one variable depends on another. Both SCOTCH and LCBA have the same slope line indicating no interaction between them. The slope of this line also reinforces eXVantage being the easier system for subjects to analyze. However, NC's line has a different (almost zero) slope. The apparent cause is a combination of two things: first, ArgoUML is harder for subject to analyze, and second, it rigorously follows naming conventions. These two combine to 'buoy-up' subject performance with NC on ArgoUML. If, instead, the harder system had been the one that did not use naming conventions, then NC's line could be expected to be steeper than the other two.

We conclude that SCOTCH not only provides the best support in terms of percentage of correct links traced but the support provided is also less influenced by the application context (i.e., the system that the technique is applied to).

5.2. Analysis of the percentage of links erroneously traced

Concerning the second null hypothesis, *t*-tests reveal that the precision achieved by the subjects using SCOTCH is statistically higher than the precision achieved by subjects tracing links using NC or LCBA. The data in Table V show that, on average, subjects are able to increase their precision by over 25% when using SCOTCH. Thus, we reject the null hypothesis H_{0c} for both NC and LCBA.

In this case, manual inspection of the links retrieved by the subjects with the different techniques highlights some reasons why SCOTCH led to better performances. The better precision achieved when using SCOTCH with respect to NC is again due to the JUnit classes where naming conventions are not used. In these cases, subjects using NC are clearly not able to correctly trace tested class based on the suggestions of tool, thus achieving a lower precision. On the other hand, the higher precision achieved when using SCOTCH with respect to LCBA is mainly the kind of JUnit tests that contain a high number of assert statements. For these JUnit tests, LCBA generally overestimates the set of tested classes, leading subjects to tracing errors.

ANOVA analysis for precision confirmed the statistical significance of *Technique* ($p < 0.0001$) and revealed no influence of *Session* ($p = 0.92$). The test again revealed a statistically significant effect of *System* on precision ($p < 0.009$) and no significant interaction between *System* and *Technique* ($p = 0.48$).

The analysis of the interaction plot between *Technique* and *System* on precision shown in Figure 7 helps to understand the effect of *System* on the dependent variable precision. In particular, the interaction plot highlights how SCOTCH simply outperforms both NC and LCBA, which is evident as the line for SCOTCH is higher than the lines for LCBA and NC. This is an impressive result, demonstrating that even in a context that is ideal for NC, SCOTCH is able to provide better support.

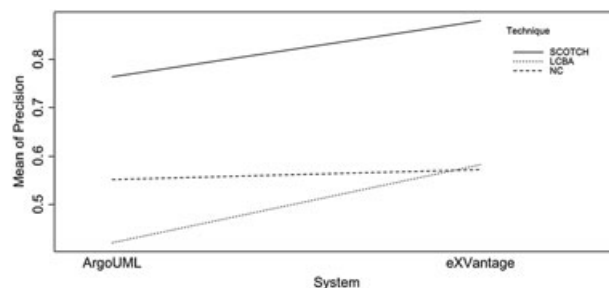


Figure 7. *Technique* and *System* interaction on precision

In addition, the results highlight better precision for subjects using NC with ArgoUML when compared with those using LCBA. On eXVantage, the *precision* achieved by subjects using NC is comparable with the precision achieved by subjects using LCBA. At first, this appears as a strange result given the superior performance of LCBA with eXVantage [6]. The reason is twofold. First, the number of links suggested by LCBA is higher than the number of links suggested by NC because the latter technique assumes a one-to-one relationship between JUnit tests and tested classes. A higher number of suggested classes leads to higher likelihood of a mistake in classifying the classes. Second, because with eXVantage, naming conventions are not applied, very often the tested class suggested by NC is a nonexistent class (i.e., the suggested class is not a class of the system). For example, the JUnit test *DGTest* from eXVantage tests the class *DependencyGraph*, whereas NC suggests *DG*, which is a nonexistent class. In such a scenario, it is very easy for the subject to classify the suggested classes as false positives. Conversely, using LCBA, all suggested classes are classes of the system; here, it is harder to classify a suggested link as a correct link or a false positive increasing the likelihood of a tracing error (e.g., a subject classifies as correct a link suggested by LCBA that is actually a false positive).

As observed on Figure 6 for the recall, Figure 7 shows interaction between NC and the other two techniques with respect to the precision values reached by the subjects. In particular, with SCOTCH and LCBA, subjects achieved higher precision on eXVantage, confirming that ArgoUML is a harder system to perform test-to-code traceability recovery. On the contrary, subjects using NC achieved almost constant performances among the two systems because the higher difficulties encountered on the ArgoUML system are mitigated by the application of the naming conventions.

In conclusion, our results highlight the support provided by SCOTCH during the identification of the classes tested by a JUnit test as well as its higher applicability when compared with NC and LCBA.

5.3. The role of the traceability recovery method

The results achieved in our controlled experiments indicated that the improvement in the correctness of the classes identified by SCOTCH as compared with NC and LCBA lead to better human performances. However, during the experiments, we allowed subjects to trace links manually aiming at enriching the set of links identified by the tool. Thus, we need to analyze the data to determine if the differences are due to one of the tools or to an exceptional manual effort on the part of the subjects.

Table VII shows the average number of links correctly traced by the subjects with tool support as well as manually. As we can see, SCOTCH provides the best support to the software engineer by supplying the most complete set of links to be traced. Indeed, subjects using SCOTCH were only able to slightly enrich the set of correct links with manually traced links. The situation with the other two recovery techniques is different. For ArgoUML, exploiting the suggestions provided by LCBA subjects traced a much lower number of correct links (2.6) than the subjects that exploited the suggestions provided by NC (4.8) or SCOTCH (8.1). Thus, for this system, the support provided

Table VII. Average number of links traced by subjects.

		ArgoUML			eXVantage		
		NC	LCBA	SCOTCH	NC	LCBA	SCOTCH
Correctly traced	Tool + manual	5.8	4.4	8.3	7.5	10.9	13.0
	Tool only	4.8	2.6	8.1	2.6	10.0	12.1
	Manual only	1.0	1.8	0.2	4.9	0.9	0.9
Wrongly traced	Tool + manual	4.8	6.3	3.6	5.5	9.3	1.8
	Tool only	1.1	4.9	1.7	1.4	5.3	0.7
	Manual only	3.6	1.4	1.9	4.1	4.0	1.1
Suggested link discarded	Correctly	1.4	4.0	6.4	5.6	6.1	0.4
	Wrongly	2.8	0.5	0.8	0.4	2.6	0.9

On ArgoUML, there are 11 correct links (in total), whereas on eXVantage, the total number of correct links is 15. SCOTCH, Slicing and Coupling based Test to Code trace Hunter; NC, naming conventions; LCBA, Last Call Before Assert.

by LCBA is quite poor requiring the subjects to manually trace about 40% of the correct links reported. The situation is similar with eXVantage, except LCBA and NC reverse roles. In fact, on this system, NC provided a much poorer support than the other techniques, requiring subjects to manually trace about 65% of the correct links identified. This result emphasizes the poor support provided by NC when naming conventions are not applied.

Also, interesting are the results achieved when analyzing the tracing errors made by subjects (i.e., when they traced a link that is actually not correct). Table VII shows that when using LCBA, subjects produced the highest number of tracing errors, whereas, as expected, the higher accuracy of NC in suggesting the tested class resulted in subjects making few tracing errors. From this point of view, SCOTCH also provided excellent support comparable with that of NC. In fact, the number of tracing errors made by subjects when using SCOTCH is comparable with the number of tracing errors made using NC. Such a result, combined with the analysis of links correctly traced, confirms the high accuracy and applicability of SCOTCH and highlights its valuable support to a software engineer during the traceability recovery task.

The data shown in Table VII also indicates that the tracing errors made when tracing links manually is generally higher than those when tracing links with the support of NC or SCOTCH. Such a result further highlights the high accuracy of SCOTCH. In fact, as said before, subjects performed few tracing errors when using NC because the wrong suggestions made by NC are generally very easier to discard (we observed that when the suggestion of NC is wrong, the suggested class is generally a nonexistent class). It is important to note that subjects using LCBA made a higher number of tracing errors with tool support than when tracing links manually. This emphasizes the poor support given by LCBA indicating that the suggestions provided by such a technique might misguide subjects into making tracing errors.

The analysis of the tracing errors made by subjects also highlights the difficulty of manually identifying the class tested by a JUnit test and recalls the need for at least a semi-automatic approach to support the software engineer in the identification of such classes.

We also analyzed the links suggested by the recovery techniques and discarded (i.e., classified as false positives) by subjects. In such a situation, suggested links could be correctly discarded (i.e., they are actually false positives) or wrongly discarded (i.e., they are actually correct). In general, subjects correctly classified the suggested links except when using NC on ArgoUML and when using LCBA on eXVantage. Although this result could be reasonable with LCBA on eXVantage, it is quite strange for NC because the suggestions provided by such a technique on ArgoUML are quite accurate. We deeply analyzed the correct links suggested by NC that were classified as false positives by the subjects. We observed that these tracing errors generally involved two JUnit tests, namely *testActionCollaborationDiagram.java* and *TestActionStateDiagram.java*. The classes tested by these two JUnit tests are *ActionCollaborationDiagram.java* and *ActionStateDiagram.java*, respectively, which are correctly suggested by NC. However, we observed that in these two tests, there are several assert statements (used to verify the testing environment) that involved other classes (e.g., *UMLDiagram*) and not the tested classes. Thus, subjects were at times playing ‘assertion roulette’ and classified such classes as tested classes discarding the (correct) suggestion by NC.

To better assess subject classification accuracy, we used a metric widely used to evaluate classification results [50]:

$$\text{accuracy} = \frac{|correctlyTraced \cup correctlyDiscarded|}{|traced \cup discarded|}$$

where *correctlyTraced* and *correctlyDiscarded* are the sets of links suggested by the traceability recovery method that were correctly traced and discarded by the subject, respectively. Table VIII reports the descriptive statistics (i.e., mean, median, and standard deviation) of the accuracy achieved by each subjects grouped by *System* and *Technique*. Overall, we can consider the classification accuracy made by subjects acceptable, indicating that the achieved results were not influenced by bad classifications made by subjects.

Table VIII. Classification accuracy (%) achieved by subjects.

	ArgoUML			eXVantage		
	NC	LCBA	SCOTCH	NC	LCBA	SCOTCH
Mean	61.3	55.2	85.3	82.5	67.2	88.8
Median	65.0	58.3	88.2	90.0	66.7	92.9
Standard deviation	11.3	16.0	12.5	23.1	5.2	10.4

SCOTCH, Slicing and Coupling based Test to Code trace Hunter; NC, naming conventions; LCBA, Last Call Before Assert.

Finally, it is worth noting that on ArgoUML subjects correctly classified as false positives, on average, more than six links were suggested by SCOTCH. This means that SCOTCH suggested a higher number of false positives as compared with the other two techniques. To understand this result, it is necessary to recall that on ArgoUML, there is usually a one-to-one relationship between a JUnit test and its tested class (e.g., the 10 selected JUnit tests are associated with only 11 correct links). Because SCOTCH often retrieves more than one tested class (because of the use of slicing), it must, at times, discard a higher number of suggested classes as compared with NC or LCBA. However, the additional effort required to discard false positives is paid for the higher number of correct links suggested with respect to NC and LCBA.

5.4. Survey questionnaire analysis

Considering the questionnaire responses, Figure 8 shows box plots of the answers (grouped by *Technique*) given by subjects in the survey questionnaires filled out after each *Session*. Overall,

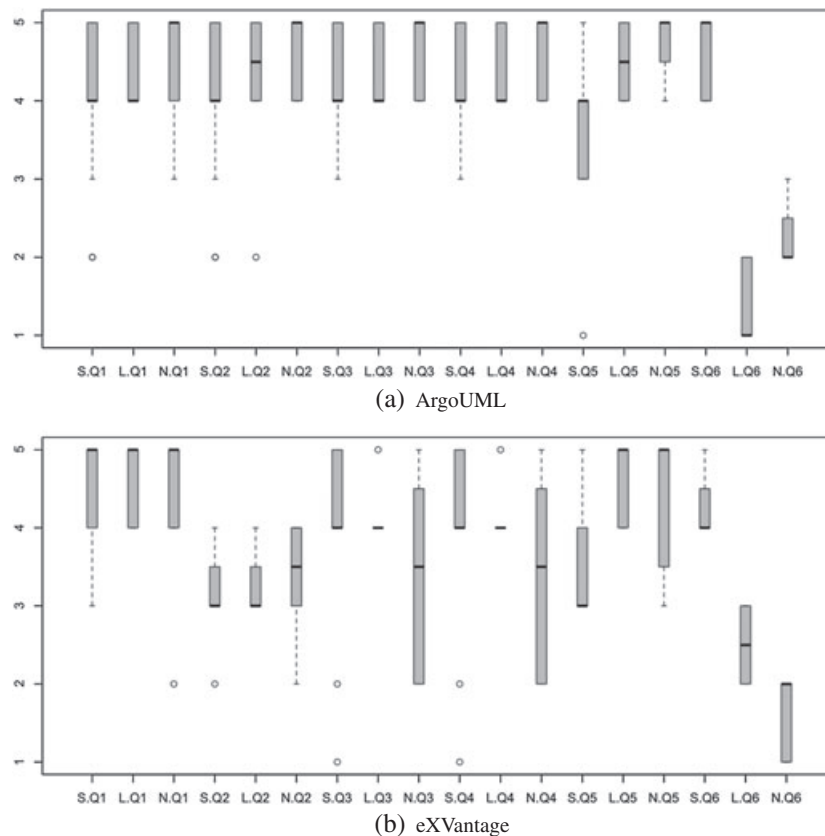


Figure 8. Box plot of answers by *Technique* (S=SCOTCH, L=LCBA, N=NC): (a) ArgoUML and (b) eXVantage.

subjects had sufficient time to perform the task in both experiments (Q1) where no significant differences emerged between systems. The objectives (Q3) and the laboratory tasks (Q4) were clear. Regarding the system domain knowledge (Q2), the results show that during the controlled experiment, subjects had a perception of good domain knowledge in ArgoUML because students have used ArgoUML or similar UML modeling tools in class (the median of the answer is 4.5), whereas the scenario is quite different in eXVantage (here, the median is 3.0). However, for both systems, the domain knowledge is acceptable. Finally, subjects experienced no major difficulties during the laboratory sessions (Q5) where again no differences emerged between systems.

Turning to the final question related to the usefulness of the set of suggested links (Q6), on both systems, the subjects agree that the list of tested classes suggested by SCOTCH is much more useful than the lists suggested by either NC or LCBA. As expected, subjects also indicated that the list provided by NC is more useful than that provided by LCBA for ArgoUML.

6. THREATS TO VALIDITY

This section discusses the achieved results focusing attention on threats that could affect their validity [45]. We consider first construct validity, then internal validity, external validity, and finally conclusions validity.

6.1. Construct validity

An important construct validity concern in our study is the metrics used to evaluate the recovery accuracy of the software engineers. The chosen metrics, recall and precision, are widely used in IR, reverse engineering, and traceability recovery experiments where they have reflect traceability recovery performance.

Interactions between different techniques were mitigated by the experimental design, which was chosen to mitigate learning effect experienced by subjects between sessions. Subjects worked, over the two sessions, on different systems with different traceability recovery techniques.

Even if the experiment design tries to mitigate the learning effect, there is still the risk that, across the lab sessions, subjects might have learned how to improve their tracing accuracy. We tried to limit this possibility by means of a preliminary training phase. Moreover, the factor *Session* has been considered as a co-factor in an ANOVA analysis, where it was not significant. Finally, there was no abandonment, and everything was clear as shown in the survey.

6.2. Internal validity

One possible issue related to internal validity concerns the possible information exchange among the subjects between the laboratories. To avoid such a threat, the experimenters monitored all the subjects to avoid collaboration and communication between them.

Our experiments required that the traceability recovery task was completed without interruption to keep variables under control. On the other hand, traceability management is often conducted in several sessions where software engineers and application domain experts incrementally recover links and also change their opinions after discussion; thus, better results (in particular, better recall) could likely be achieved if the traceability recovery tasks were performed across multiple traceability recovery sessions.

The accuracy of the oracle used in our experiments could affect our results. We were not able to have the traceability matrices provided by the original developers. For this reason, we asked three PhD students to manually recover the traceability links. To increase the confidence in the evaluation process, students were not aware of the experimental goals and of the experimented recovery techniques. 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.

6.3. External validity

The subjects involved in our study are bachelor's students taking a software engineering course. When experimenting with students, there is a threat to external validity (i.e., generalization of the results). However, the selected subjects represent a population specifically trained on software development technologies and software engineering methods. In particular, all the students had knowledge of software development and testing, as well as software artifact traceability. To avoid social threats because of evaluation apprehension, students were not evaluated on their performance. Moreover, subjects were not aware of the experimental hypotheses.

Table IX lists the seven demographic variables collected and their average values (overall and broken out by the baseline technique used). Note that the last two variables are booleans, and the table reports the number of 'Yes' responses. The only significant difference is that subjects using LCBA as the baseline had a higher average number of years of Java experience.

We statistically analyzed the potential for demographic influences using linear mixed-effects regression models, in which backward elimination of statistically nonsignificant terms ($p \geq 0.05$) yields the final model. A series of linear mixed-effects regression models for recall and precision were constructed that initially included the demographic variables. None proved significant. In particular, the number of years of Java experience is not significant in any of the final models. Thus, there is support in the data to affirm that there are no unwanted demographic influences.

Another threat is represented by the variance in the domain knowledge of subjects. Such a situation could influence our experiments because it might be difficult to identify the tested classes given limited domain knowledge. To mitigate this threat, two weeks before the experiment, we gave subjects access to the source code of both systems with the aim of enriching their domain knowledge. Moreover, some students have used ArgoUML or similar UML modeling tools during the software engineering course. However, although they are familiar with some of ArgoUML's functionality, they had not looked at the code.

The dominant threat to external validity in our study is related to the systems used. To mitigate this threat, we selected an open source system ArgoUML and an industrial project eXVantage. This makes sure that the results achieve have broad applicability. However, this type of experiment has to be conducted in a controlled environment and in a limited amount of time. For this reason, it is not easy to use larger repositories. Finally, note that some JUnit tests from the object systems were discarded because they are not related to actual classes under test (for example, they test mock objects, network settings, and complex database queries). A similar choice on the AgilePlanner and ArgoUML projects was made in previous work [7,34]. Moreover, Table X compares the size, in terms of lines of code, number of methods, the cohesion, in terms of lack of cohesion of methods (LCOM), and the coupling, in terms of coupling between object, of the *usable* and *discarded* JUnit classes. Moreover, we report these metrics for the ten JUnit classes *selected* in our controlled experiments. These JUnit characteristics are shown to verify if substantial differences exist between the *usable*, *discarded*, and *selected* JUnit tests (e.g., there is the risk that only simple unit tests were selected in our evaluation). From the analysis of Table X, no substantial differences in terms of coupling are apparent, whereas it is clear that the *usable* JUnit classes are, in general, bigger and

Table IX. Demographic analysis.

Overall	Average for		<i>p</i> -value	Demographic
	LCBA	NC		
2.7	2.5	2.8	0.0680	Number of years at the university
8.3	8.4	8.1	0.3761	Number of passed exams
25.2	25.2	24.9	0.3306	Average exam score
3.1	3.3	3.0	0.2338	Number of programming exams passed
1.6	1.9	1.3	0.0287	Years of Java experience
3	2	1	(Count)	Have you ever programmed at work?
2	1	1	(Count)	Do you have experience with JUnit?

NC, naming conventions; LCBA, Last Call Before Assert.

Table X. Characteristics of the discarded, usable, and selected JUnit tests.

System	Set	LOC			NOM			LCOM			CBO		
		Mean	Median	Standard deviation	Mean	Median	Standard deviation	Mean	Median	Standard deviation	Mean	Median	Standard deviation
ArgoUML	Discarded (88)	47	24	65	4	4	3	9	3	16	5	4	3
	Usable (75)	83	47	129	5	4	4	12	4	28	5	4	5
eXVantage	Selected (10)	72	44	82	4	4	3	9	3	13	4	4	3
	Discarded (13)	102	88	99	3	2	3	6	3	8	11	13	9
	Usable (17)	228	122	273	10	4	12	16	6	14	8	8	5
	Selected (10)	189	104	109	7	4	5	14	5	9	9	7	5

LOC, lines of code; NOM, number of methods; LCOM, lack of cohesion of methods; CBO, coupling between object.

have a slightly worse cohesion than the *discarded* ones (LCOM is an inverse cohesion measure where the higher the LCOM value, the lower the class cohesion). Note that this is an expected result because the usable JUnit classes test classes in the production code that are generally more complex than mock objects or simple queries tested by the discarded JUnit tests. As for the ten JUnit tests *selected* for each system in the controlled experiments, their characteristics are almost inline with the *usable* JUnit tests testing classes of the production code.

6.4. Conclusion validity

Concerning conclusion validity, attention was paid not to violate assumptions made by statistical tests. The interaction between different factors has been analyzed by ANOVA tests. ANOVA is quite robust to deviations from normality. Finally, survey questionnaires, mainly intended to obtain qualitative insights, were designed using standard question format and scales [47].

7. CONCLUSION

This paper presented the results of a set of controlled experiments carried out to evaluate the support given by SCOTCH during the identification of the classes tested by a JUnit test. The experiments, involving a total of 32 subjects, were performed on the open source system ArgoUML and the industrial system eXVantage. They compared SCOTCH with two existing tools: NC, based on naming conventions, and LCBA, based on the last call before an assert. With the support of a traceability recovery tool, each subject was charged to identify the classes tested by a set of JUnit tests randomly selected from two subject systems. The achieved results indicate that SCOTCH provides better support to a software engineer when compared with the other traceability recovery methods, as it allows the software engineer to identify a higher number of correct links with higher accuracy.

As it always happens with empirical studies, replications in different contexts, with different subjects and objects, are the only ways to corroborate our findings. Replicating the experiment with professional software engineers and using other repositories is part of our future work agenda. Moreover, there are a number of directions to improve the performance of SCOTCH. In particular, we plan to investigate the use of other heuristics to use with or in place of CCBC.

ACKNOWLEDGMENTS

We are very grateful to Annibale Panichella of the University of Salerno, Italy, for his valuable support during the execution of the experiment at the University of Salerno. Special thanks go to the students who were involved in the two experiments as subjects. We would also like to thank anonymous reviewers for their careful reading of our manuscript and useful comments. Their comments have helped us to substantially revise, extend, and improve the original version of this paper. Finally, Dave Binkley is supported by NSF grant CCF 0916081.

REFERENCES

1. Van Deursen A, Moonen L. The video store revisited – thoughts on refactoring and testing. *Proceedings of the International Conference on eXtreme Programming and Flexible Processes in Software Engineering*, Alghero, Sardinia, Italy, 2002; 71–76.
2. Hayes JH, Dekhtyar A, Janzen DS. Towards traceable test-driven development. *Proceedings of the 5th Workshop on Traceability in Emerging Forms of Software Engineering*, IEEE CS Press: Vancouver, British Columbia, Canada, 2009; 26–30.
3. Van Rompaey B. Developer testing as an asset during software evolution: a series of empirical studies. PhD Thesis, University of Antwerp March 2009.
4. Bavota G, Qusef A, Oliveto R, De Lucia A, Binkley D. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. *Proceedings of the 28th IEEE International Conference on Software Maintenance*, IEEE Pres: Lago di Garda, Italy, 2012; To appear.
5. Beck K, Gamma E. Test infected: Programmers love writing tests. *Java Report* 1998; 3(7):51–56.
6. Qusef A, Bavota G, Oliveto R, Lucia AD, Binkley D. Scotch: Test-to-code traceability using slicing and conceptual coupling. *Proceedings of the 27th International Conference on Software Maintenance*, IEEE CS Press: Williamsburg, VA, USA, 2011; 63–72.

7. Van Rompaey B, Demeyer S. Establishing traceability links between unit test cases and units under test. *Proceedings of the 13th European Conference on Software Maintenance and Reengineering* 2009; 209–218.
8. Pfleeger SL, Menezes W. Marketing technology to software practitioners. *IEEE Software* 2000; **17**(1):27–33.
9. Redwine, Jr ST, Riddle WE. Software technology maturation. *Proceedings of 8th International Conference on Software Engineering*, IEEE: London, UK, 1985; 189–200.
10. Zolkowitz M, Wallace D, Binkley D. Experimental validation of new software technology, chap. 12. *SE Empirical Validation*, Natalia J, Ana M (eds.). Molex, 2003.
11. Raghavan SA, Chand DR. Diffusing software-engineering methods. *IEEE Software* 1989; **6**:81–90.
12. Kitchenham B, Pickard L, Pfleeger SL. Case studies for method and tool evaluation. *IEEE Software* 1995; **12**:52–62.
13. Pfleeger SL, Menezes W. Marketing technology to software practitioners. *IEEE Software* 2000; **17**:27–33.
14. Boogerd C, Moonen L. Assessing the value of coding standards: An empirical study. *Proceedings of the 24th International Conference on Software Maintenance*, IEEE CS Press: Beijing, China, 2008; 277–286.
15. Cleland-Huang J, Chang CK. Event-based traceability for managing evolutionary change. *IEEE TSE* 2003; **29**(9):796–810.
16. Egyed A. A scenario-driven approach to trace dependency analysis. *IEEE Transactions on Software Engineering* Feb 2003; **29**(2):116–132.
17. Egyed A. A scenario-driven approach to traceability. *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, IEEE Computer Society, 2001; 123–132.
18. Ziftci C, Krueger I. Tracing requirements to tests with high precision and recall. *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, IEEE Computer Society, 2011; 472–475.
19. Zisman A. Software and Systems Traceability, chap. Using Rules for Traceability Creation. Springer-Verlag, 2012; 147–172.
20. Cleland-Huang J, Settini R, BenKhadra O, Berezanskaya E, Christina S. Goal-centric traceability for managing non-functional requirements. *Proceedings of the 27th international conference on Software engineering*, ICSE '05, ACM, 2005; 362–371.
21. Egyed A, Grünbacher P. Automating requirements traceability: Beyond the record and replay paradigm. *Proceedings of the 17th International Conference on Automated Software Engineering*, IEEE CS Press: Edinburgh, UK, 2002; 163–171.
22. Gall H, Hajek K, Jazayeri M. Detection of logical coupling based on product release history. *Proceedings of the 14th IEEE International Conference on Software Maintenance*, IEEE CS Press: Bethesda, Maryland, USA, 1998; 190–198.
23. Antoniol G, Canfora G, Casazza G, De Lucia A, Merlo E. Tracing object-oriented code into functional requirements. *Proc. of 8th IEEE IWPC*, IEEE CS Press: Limerick, Ireland, 2000; 79–87.
24. Antoniol G, Canfora G, Casazza G, De Lucia A, Merlo E. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering* 2002; **28**(10):970–983.
25. Asuncion HU, Asuncion AU, Taylor RN. Software traceability with topic modeling. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, ACM, 2010; 95–104.
26. De Lucia A, Marcus A, Oliveto R, Poshyvanyk D. Information Retrieval Methods for Automated Traceability Recovery, chap. 4. *Software and Systems Traceability* (Jean Cleland-Huang, Orlena Gotel, and Andrea Zisman eds), Springer Press, 2012; 71–98.
27. Marcus A, Maletic JI. Recovering documentation-to-source-code traceability links using latent semantic indexing. *Proceedings of 25th International Conference on Software Engineering*, Portland, Oregon, USA, 2003; 125–135.
28. Bouillon P, Krinke J, Meyer N, Steimann F. Ezunit: A framework for associating failed unit tests with potential programming errors. *Proceedings of the 8th International Conference on Agile Processes in Software Engineering and eXtreme Programming*, Como, Italy, 2007; 101–104.
29. Ryder B. Constructing the call graph of a program. *IEEE Transactions on Software Engineering* 1979; **5**(3):216–226.
30. Sneed HM. Reverse engineering of test cases for selective regression testing. *Proceedings of the 8th Working Conference on Software Maintenance and Reengineering*, IEEE CS Press: Tampere, Finland, 2004; 69.
31. Deerwester S, Dumais ST, Furnas GW, Landauer TK, Harshman R. Indexing by latent semantic analysis. *Journal of the American Society for Information Science* 1990; **41**(6):391–407.
32. De Lucia A, Fasano F, Oliveto R, Tortora G. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology* 2007; **16**(4):13.
33. Gall H, Jazayeri M, Krajewski J. CVS release history data for detecting logical couplings. *Proceedings of the 6th International Workshop on Principle of Software Evolution*, IEEE CS Press: Helsinki, Finland, 2003; 13–23.
34. Qusef A, Oliveto R, De Lucia A. Recovering traceability links between unit tests and classes under test: An improved method. *Proceedings of the 26th IEEE International Conference on Software Maintenance*, IEEE CS Press: Timisoara, Romania, 2010.
35. Korel B, Laski J. Dynamic program slicing. *Information Processing Letter* October 1988; **29**:155–163.
36. Korel B, Laski JW. Dynamic slicing of computer programs. *Journal of Systems and Software* 1990; **13**(3):187–195.
37. De Lucia A, Fasano F, Francese R, Tortora G. ADAMS: an artefact-based process support system. *Proceedings of the 16th International Conference on Software Engineering and Knowledge Engineering*, KSI Press: Banff, Alberta, Canada, 2004; 31–36.
38. De Lucia A, Oliveto R, Tortora G. Assessing IR-based traceability recovery tools through controlled experiments. *Empirical Software Engineering* 2009; **14**(1):57–93.
39. De Lucia A, Oliveto R, Tortora G. IR-based traceability recovery processes: an empirical comparison of “one-shot” and incremental processes. *Proceedings of 23rd International Conference on Automated Software Engineering*, ACM Press: L'Aquila, Italy, 2008; 39–48.

40. Van Deursen A, Moonen L, Bergh A, Kok G. Refactoring test code. Technical Report, CWI (Centre for Mathematics and Computer Science), Amsterdam 2001.
41. Agrawal H, Horgan JR. Dynamic program slicing. *SIGPLAN Not* Feb June 1990; **25**:246–256.
42. Poshyvanyk D, Marcus A, Ferenc R, Gyimóthy T. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering* 2009; **14**(1):5–32.
43. Baeza-Yates R, Ribeiro-Neto B. Modern Information Retrieval. Addison-Wesley, 1999.
44. Basili V, Caldiera G, Rombach DH. The Goal Question Metric Paradigm, Encyclopedia of Software Engineering. John Wiley and Sons, 1994.
45. Wohlin C, Runeson P, Host M, Ohlsson MC, Regnell B, Wesslen A. Experimentation in Software Engineering - An Introduction. Kluwer, 2000.
46. Devore JL, Farnum N. Applied Statistics for Engineers and Scientists. Duxbury, 1999.
47. Oppenheim AN. Questionnaire Design, Interviewing and Attitude Measurement. Pinter Publishers, 1992.
48. Conover WJ. Practical Nonparametric Statistics. 3rd edn., Wiley, 1998.
49. Ricca F, Di Penta M, Torchiano M, Tonella P, Ceccato M. How developers' experience and ability influence web application comprehension tasks supported by uml stereotypes: A series of four experiments. *IEEE Transactions on Software Engineering* 2010; **36**(1):96–118.
50. Sebastiani F. Machine learning in automated text categorization. *ACM Computing Surveys* 2002; **34**(1):1–47.

AUTHORS' BIOGRAPHIES:



Abdallah D. Qusef was born in Kuwait on 25 February 1979. In 2001, he received his Bachelor's degree in Computer Science from University of Jordan (Jordan). He completed his graduate studies from Al-Balqa Applied University (Jordan) in 2005. He received his PhD in Computer Science from the University of Salerno (Italy) in 2012. Currently, he has research fellow at the University of Salerno. He is member of IEEE. His research interest is in software engineering, in particular, requirements engineering, software maintenance and evolution, software traceability, agile software development, refactoring, software testing and empirical software engineering.



Gabriele Bavota was born in Napoli (Italy) on 19 November 1985. He received (cum laude) his Laurea in Computer Science from the University of Salerno (Italy) in 2009 defending a thesis on Traceability Management advised by Prof. Andrea De Lucia and Dr. Rocco Oliveto. He is currently a PhD student at the Department of Mathematics and Informatics of the University of Salerno under the supervision of Prof. Andrea De Lucia. His research interests include refactoring of software systems, traceability management, information retrieval, software maintenance and empirical software engineering. He serves and has served on in the organizing and program committees of international conferences in the field of software engineering. In particular, he was the web chair of WCRE 2012 and he will be publicity co-chair of ICPC 2013. He is student member of IEEE.



Rocco Oliveto received (cum laude) his Laurea in Computer Science from the University of Salerno (Italy) in 2004. From October 2006 to February 2007, he has been a visiting student at the University College London, UK. He received his PhD in Computer Science from the University of Salerno (Italy) in 2008. From July 2009 to September 2009, he has been a visiting researcher at the Polytechnique of Montreal, Canada. From July 2009 to September 2009, he has been a visiting researcher at the Polytechnique of Montreal, Canada. From 2008 to 2010, he was research fellow at the Department of Mathematics and Informatics of the University of Salerno. From 2005 to 2010, he is also adjunct

professor at the Faculty of Science of the University of Molise (Italy). He is member of the Software Engineering Lab at the University of Salerno. In 2011, he joined the STAT Department of the University of Molise where he is currently assistant professor. His research interests include traceability management, information retrieval, software maintenance and evolution, and empirical software engineering. He has published about 40 papers on these topics in international journals, books, and conference proceedings. He serves and has served on in the organizing and program committees of international conferences in the field of software engineering. In particular, he was the program co-chair of TEFSE 2009, the Traceability Challenge Chair of TEFSE 2011, the Industrial Track Chair of WCRE 2011, the Tool Demo Co-chair of ICSM 2011 and he will be the program co-chair of WCRE 2012. Dr. Oliveto is member of IEEE, ACM, and IEEE-CS Awards and Recognition Committee.



Andrea De Lucia received his Laurea degree in Computer Science from the University of Salerno, Italy, in 1991, his MSc degree in Computer Science from the University of Durham, UK, in 1996, and his PhD degree in Electronic Engineering and Computer Science from the University of Naples “Federico II”, Italy, in 1996. He is a full professor of software engineering and the director of the International Summer School on Software Engineering at the University of Salerno. Previously, he was with the Department of Engineering and the Research Centre on Software Technology (RCOST) at the University of Sannio. He is actively consulting in industry and has been involved in several research and technology transfer projects conducted in cooperation with industrial partners. His research interests include software maintenance, program comprehension, reverse engineering, reengineering, migration, global software engineering, software configuration management, workflow management, document management, empirical software engineering, visual languages, web engineering, and e-learning. He has published more than 150 papers on these topics in international journals, books, and conference proceedings and has edited books and journal special issues. He serves on the editorial board of *Journal of Software: Evolution and Process* and other international journals and on the organizing and program committees of several international conferences in the field of software engineering. Prof. De Lucia is a senior member of the IEEE and the IEEE Computer Society. He was also an at-large member of the executive committee of the IEEE Technical Council on Software Engineering (TCSE) and committee member of the IEEE Real World Engineering Project (RWEP) Program.



Dr. David Binkley is a professor of Computer Science at Loyola University Maryland where he has worked since earning his doctorate from the University of Wisconsin in 1991. In 2006 he co-chaired the program at the International Conference on Software Maintenance and has been active in that community every since. While on leave from Loyola in 2008, he joined the researchers at the Crest Centre to work, among other things, several testing projects. Dr. Binkley's current NSF-funded research focuses on semantics-based software engineering tools, application of information retrieval techniques in software engineering, and improved techniques for software testing.