Empirical Software Engineering (2022) 27:67
https://doi.org/10.1007/s10664-021-10079-1

TCTracer: Establishing test-to-code traceability links using dynamic and static techniques

Robert White[1] · **Jens Krinke**[1]

Accepted: 8 November 2021

Abstract
Test-to-code traceability links model the relationships between test artefacts and code artefacts. When utilised during the development process, these links help developers to keep test code in sync with tested code, reducing the rate of test failures and missed faults. Test-to-code traceability links can also help developers to maintain an accurate mental model of the system, reducing the risk of architectural degradation when making changes. However, establishing and maintaining these links manually places an extra burden on developers and is error-prone. This paper presents TCTracer, an approach and implementation for the automatic establishment of test-to-code traceability links. Unlike existing work, TCTracer operates at both the method level and the class level, allowing us to establish links between tests and functions, as well as between test classes and tested classes. We improve over existing techniques by combining an ensemble of new and existing techniques that utilise both dynamic and static information and exploiting a synergistic flow of information between the method and class levels. An evaluation of TCTracer using five large, well-studied open source systems demonstrates that, on average, we can establish test-to-function links with a mean average precision (MAP) of 85% and test-class-to-class links with an MAP of 92%.

Keywords Software testing · Traceability · Software development · Software engineering

1 Introduction

Unit testing is an integral part of software development, however, to fully realise the benefits of unit testing, it is necessary to maintain an accurate picture of the relationships between the

Communicated by: Gabriele Bavota

✉ Jens Krinke
j.krinke@ucl.ac.uk

Robert White
robert.white.13@ucl.ac.uk

[1] UCL Computer Science, University College London, London, UK

Published online: 17 March 2022

tests and the tested code. Traceability links provide an intuitive mechanism for modelling these relationships.

Once established, test-to-code traceability links can improve the software engineering process in several ways, including making changes to the system safer, facilitating the reuse of artefacts, and aiding program comprehension (De Lucia et al. 2008; Antoniol et al. 2002; Winkler and von Pilgrim 2010). Changes to the system become safer as, when a developer makes a change to a piece of tested code, they can use the traceability links to easily discover which tests also need to be changed, and vice-versa. This helps to promote the co-evolution of code as it highlights to the developer code that needs to evolve along with a change. This is important as previous work has shown that test repair and test modification is a common and important task (Pinto et al. 2012) and that co-evolution is desirable but typically does not happen consistently over the course of a project (Zaidman et al. 2011). This work has shown that testing is often done in short intense periods between periods of increasing test stagnation. Co-evolution, therefore, is often not consistent in practice and the utilisation of automated test-to-code traceability link establishment could help to improve this and reduce the risk of desynchronisation between the tests and code, an issue that can cause test failures and prevent the discovery of new faults. While developers can use fault localisation techniques to discover which functions may be causing test failures, traceability links have the benefit of being bidirectional, so developers can start from a function and find the corresponding tests. Traceability links are also used in regression test suite optimisation in continuous integration (Elsner et al. 2021) to identify and execute tests that are potentially affected by a change and where executing the full test suite would be too expensive. This parallel between test-to-code traceability link establishment and regression test case selection is also noted by Soetens et al. (2016) who discovered that existing test-to-code traceability techniques, such as naming conventions, fixture element types, static call graphs, and LCBA can work well but are very situational.

Industrial need for the automated establishment of test-to-code traceability links is demonstrated by Ståhl et al. (2017) through case studies and developer interviews. The developer interviews were focused on themes and the theme that encompasses this work, 'Test Results and Fault Tracing', attracted the most number of relevant statements, with interviewees stating, for example, that it was 'particularly important' and 'super crucial'. Using trace links to 'drill down' when troubleshooting failed tests was specifically mentioned. The developers also made clear that automation is crucial as manual traceability handling is a major blocker for more frequent deliveries of software. Traceability is also gaining importance due to the recent growth of machine learning for software engineering, where traceability links have been used to build corpora of training data. Watson et al. (2020), White and Krinke (2018, 2020) are examples of work that utilise test-to-code traceability links for building a training corpus for neural networks that generate test code for a given function. In this use case, test-to-code traceability links are used to train sequence to sequence machine learning models to generate test code using a function as input. Therefore, a large, high-quality data set of test-to-code traceability links is required to train and test the model. As the performance of the model is dependent on the size and quality of the data set, developing approaches for automatically and accurately establishing traceability links can produce larger data sets and reduce the amount of noise, thus improving the ability of the models to solve these problems.

While there has been an effort on some projects to have developers manually maintain traceability links, this practice is not common as it creates extra work for developers. Instead, developers often employ naming conventions, e.g., matching the names of test

classes with the names of tested classes, with 'Test' appended. In most instances, where projects have attempted to manually maintain traceability links, these have been at the class level where the number of links is more manageable and the relationships between test artefacts and tested artefacts are usually simple. Therefore, to avoid creating extra work for the developers and the errors associated with the manual maintenance of traceability links, the research community has focused on developing approaches for the automatic establishment of traceability links.

The difficulty in establishing test-to-code links lies in the fact that not all code executed by a test is part of the code that is being tested. This is because many tests will call functions that are not considered to be amongst the functions under test, such as helper functions, getters and setters, or functions that initialise the state of an object before the functions under test are invoked. Therefore, simply considering all executed code as tested code (Hurdugaci and Zaidman 2012) is not an accurate technique of establishing test-to-code traceability links.

In this paper, we present TCTracer, an approach and implementation[1] which aims to overcome the weaknesses of existing test-to-code traceability link establishment approaches by employing a wide range of techniques that utilise information from dynamic call traces and static information. TCTracer also joins these techniques to produce a combined score that performs better overall than any individual technique. In addition, unlike previous work, TCTracer is applied to both the method level and the class level which allows us to establish links between individual tests and their tested functions as well as whole test classes and their tested classes. TCTracer uses its multilevel aspect to create a flow of information between the levels that can improve effectiveness.

Our approach is evaluated using a manually curated ground truth (White and Krinke 2021), at both the method and class levels, from five non-trivial and well-studied subject projects.[2] Our findings show that, on average, using our combined technique, we can achieve an increase in effectiveness over existing techniques at both the method and the class levels. At the class level, our findings reveal that static naming techniques alone can produce results equivalent to the combined score.

In addition to this evaluation, we conduct experiments to assess an alternative technique for combining scores using machine learning (ML), the effect of weighting techniques during combination, and a manual investigation into the causes of false negatives and false positives.

This is an extension to our previous work (White et al. 2020) where we first introduced TCTracer. We build on the previous work by incorporating static techniques, investigating alternate combination methods and technique weighting schemes, expanding the ground truth, and performing a more in-depth analysis of the accuracy of the approach.

The main contributions of this paper are:

- An approach to test-to-code traceability that utilises an ensemble of techniques using dynamic and static information and a multilevel flow of information.
- A comparative evaluation of each technique at both the method and class levels and across information types.
- An evaluation of the benefit gained by utilising multilevel information.

[1] Available at https://github.com/RRGWhite/tctracer.

[2] Evaluation artefacts available at https://doi.org/10.5281/zenodo.4608587.

- An evaluation of two methods for combining individual techniques and the effect of weighting individual techniques prior to combination
- A manual investigation into the causes of false positive and false negative links
- An updated manually curated ground truth dataset (White and Krinke 2021) of test-to-function and test-class-to-class links.

The paper is structured as follows: Section 2 provides the background and motivation for this work by presenting a review of previous work which explores the current state-of-the-art and highlights the weaknesses of existing approaches which TCTRACER addresses. Section 3 presents an overview of the approach used by TCTRACER. Section 4 provides the details of the techniques used while Section 5 describes how we utilise the techniques within our approach to generate the predicted traceability links. Section 6 describes the implementation of TCTRACER and TCAGENT, the Java Virtual Machine agent used for the collection of the dynamic trace information. Section 7 presents the evaluation, including the experimental setup, research questions, and results and Section 8 presents a discussion of these results and other findings and discussion points, including the main takeaways. Section 9 describes the internal and external threats to validity and ethical considerations. Section 10 discusses additional related work that does not form part of the background in Section 2. Finally, Section 11 presents the overall conclusion of the work.

2 Background

Establishing and maintaining traceability links between tests and their tested functionality has received significant attention from the research community as traceability links have multiple applications in the software engineering process, such as determining which test cases need to be rerun after a change has been made, maintaining consistency during refactoring, and providing a form of documentation. Test-to-code traceability can, for example, help to locate the fault that causes a test case to fail. Qusef et al. (2014) describe these benefits in detail and (Parizi et al. 2014) presents an overview of the achievements and challenges of test-to-code traceability. Prior research has investigated the use of gamification to improve manual maintenance of traceability links (Parizi 2016; Meimandi Parizi et al. 2015) but this approach has not seen significant adoption.

Most previous work on test-to-code traceability (see Parizi et al. (2014) for an overview) has focused on the class level, where test classes are linked to their tested classes (Van Rompaey and Demeyer 2009; Qusef et al. 2014; Gethers et al. 2011; Kicsi et al. 2018; Csuvik et al. 2019a, b).

Van Rompaey and Demeyer (2009) is the closest work to ours as they investigate six traceability techniques to link test classes to classes-under-test over three projects from which they extracted a ground truth of 59 links. They report perfect precision and recall for the use of naming conventions, but report very low precision and recall for using similarity (LSI) between test classes and classes-under-test. Rompaey and Demeyer investigate mostly static techniques and only use tracing to establish LCBA. While they only investigate on the class level, we investigate dynamic techniques on the class and the method level over much larger ground truths.

SCOTCH+ (Source code and Concept based Test to Code traceability Hunter) is a traceability system introduced by Qusef et al. (2014) that achieves better accuracy and provides more benefit to developers than LCBA or NC (Qusef et al. 2013). SCOTCH+ applies dynamic slicing to identify a set of candidate tested classes which it then filters using a

textual coupling analysis called Close Coupling between Classes (CCBC) and name similarity (NS) scores.

Kicsi et al. (2018) explore the usage of Latent Semantic Indexing (LSI) over source code to establish traceability links between test classes and tested classes by assuming that a test class should be lexically similar to its tested class. They extract a ground truth from five open source systems by extracting only the links between test classes and tested classes that follow (exact) naming conventions. They report that the ground truth link is ranked top between 30% and 62% and is present in the top 5 between 57% and 89%, suggesting a low recall (precision is not investigated). Csuvik et al. (2019b) replaced LSI with word embeddings within the same approach and report better precision when using word embeddings (no investigation of recall has been done). They also compare LSI, word embeddings and TF-IDF (Csuvik et al. 2019a) in the same way and report that word embeddings perform best in terms of precision and recall.

Not much work has been done on the method level (Bouillon et al. 2007; Hurdugaci and Zaidman 2012; Ghafari et al. 2015), where individual unit tests are linked to their tested functions, despite being shown to be helpful for developers (Hurdugaci and Zaidman 2012).

EzUnit (Bouillon et al. 2007) is a framework that allows developers to annotate tests with links to the method-under-test. To do so, it performs static analysis and identifies the methods called by a test which are suggested for annotation. EzUnit highlights the linked methods when an error in the test occurs. A similar tool is TestNForce (Hurdugaci and Zaidman 2012) which links tests to methods-under-test. Like our approach, tracing is used to identify the methods that are called by a test. No further filtering is done and their approach will thus include a large number of utility methods leading to low precision. Ghafari et al. (2015) also work at the method level where they break down test cases into sub-scenarios for which they attempt to establish the tested function, termed the focal method. This is done using static data flow analysis. The results for this technique are promising, however, two of the four subjects used for the evaluation are very small (130 and 43 tests), while the other two are still smaller than our smallest subject. As it is easier to achieve higher precision and recall on smaller projects, due to fewer candidate links, the results cannot be directly compared to those presented in this paper.

Our work is the first to address both the class level and the method level simultaneously. This allows us to construct both types of links and utilise a cross-level flow of information to improve overall performance. This gives our approach a more accurate and fine-grained view of the relationships between the artefacts. Our work also distinguishes itself from previous work by utilising both dynamic and static information and ranking potential links, instead of the purely static information that has typically been used before to generate sets of (unranked) links.

Therefore, the development of a new approach to test-to-code traceability establishment is motivated primarily by the fact that all existing techniques have some weaknesses that make them unsuitable for use as a general solution. One of the most common techniques for establishing traceability links, naming conventions (NC), is a good example of this. This approach relies on using the naming conventions for test artefacts (unit tests or test classes) to identify their links to tested artefacts (functions or classes). For example, JUnit 3 required a prefix of 'test' to identify test methods. The specific conventions used may vary between projects, however, the standard convention is that a test artefact should share the same name as the artefact that it is testing, with *test* prepended or appended (Van Rompaey and Demeyer 2009; Madeja and Porubän 2019). For example, a function named *union* will be considered to be tested by a test named *testUnion*. However, this technique is not effective if the project

does not adhere to the naming conventions and can have poor recall even for projects that do. This is because it assumes a one-to-one relationship between test artefacts and tested artefacts when this is not always the case. The Commons Collections project[3] provides a real-world example of this, where the function *disjunction* is tested by the tests *testDisjunctionAsUnionMinusIntersection* and *testDisjunctionAsSymmetricDifference*. As this is a one-to-many relationship, the names do not match the naming conventions and NC would not be able to recover these links. While test-to-code traceability based on name similarity has good accuracy on the class level, as developers usually follow naming conventions for the test classes, on the method level there exist various guidelines on how to name a test method. Madeja and Porubán (2019) investigated 5 popular Android projects and found that only 49% of tests contain the full name of the method-under-test in the test name and that 76% of tests contain a partial name of a method-under-test in the test name.

Last Call Before Assert (LCBA) is another existing technique that has severe limitations. LCBA operates on the assumption that the function which returned last before an assert is called is the function that the assert is testing. However, this assumption is often incorrect. One common example of this is when the purpose of a tested function is to change the state of an object. In this case, to check that the function has performed the correct operation, a state checking function must be called to get the changed state so that it can be compared to an oracle. This causes LCBA to incorrectly identify the state checking function as the tested function. Even if the tested function does directly return the value that needs to be checked, this value will often not be checked by an assert immediately after being returned. This could be because the test needs to call helper functions before the assert, possibly to establish the oracle.

Finally, textual similarity measures based on information retrieval techniques have also been used in an attempt to recover test-to-code traceability links, with varying degrees of success (Antoniol et al. 2002; Csuvik et al. 2019a). However, none of them are sufficient on their own as techniques designed for natural language do not directly translate to code. This is due to the bimodality of code which leads to the possibility that two code snippets may be closely related semantically but completely different lexically, or vice-versa (Allamanis et al. 2018).

Given these inherent weaknesses in the individual existing techniques, there is a strong motivation to design a new approach that, while exploiting the strengths of the individual techniques, collectively overcomes their weaknesses. This is the approach utilised in TCTracer and presented in this paper.

A secondary motivation for the development of a new approach to test-to-code traceability stems from the fact that existing work has only focused on either the method level or the class level. As both levels can provide useful information to a developer, we were motivated to develop a single approach that worked at both levels simultaneously. This resulted in the multilevel aspect of TCTracer, which in turn facilitated the use of multilevel information flow to further increase the effectiveness of the approach.

3 Approach

Our approach utilises dynamic call traces and static information to create candidate links between test artefacts and tested artefacts. It assigns scores to the candidate links using an

[3] https://commons.apache.org/proper/commons-collections/

ensemble of techniques and these scores are used to rank the candidates and predict which of them are true test-to-code traceability links. The predicted links can then be used, e.g., in an IDE, to navigate between tests and the tested artefacts.

We utilise dynamic information as it provides us with the call traces showing which functions were executed by which tests, thus providing a natural filtering that serves as a starting place for establishing traceability links. However, as dynamic analysis requires the system-under-analysis to be executed, gathering dynamic information is not possible in all scenarios, such as where a large and diverse corpus of code is being used. For example, if an approach uses a corpus that includes the top 1000 GitHub projects, having to build and execute every project would be prohibitively time-consuming. In this scenario, static information is the only practical information source and we, therefore, incorporated techniques that only require static information to determine the usefulness of the approach in this scenario.

As we are establishing links on the method-level as well as on the class-level, we use the terms *function* or *method-under-test* when referring to a tested method and the terms *tested class* or *class-under-test* for the class-level. Moreover, on the class-level, a class-under-test is tested by one or more *test classes*, and on the method-level, a method-under-test is tested by one or more *test methods*.

Our multilevel approach starts by dynamically collecting information about the function calls made by each test, specifically, which function was called and the depth in the call stack of the function call relative to the calling test and the set of functions that were executed immediately before an assert. Static information, which consists of the fully qualified names (FQNs) and bodies of all classes, test classes, functions, and tests is also collected by parsing the source code of the project-under-analysis. We then apply an ensemble of traceability techniques to the method level, using the collected dynamic and static information. This results in a set of test-to-function scores for each technique, each of which encodes the likelihood that a given function is the tested function for a given test that calls it. We refer to these scores collectively as the method level information. The same process is then applied at the class level, where sets of test-class-to-class scores are established using the same techniques, providing us with the class level information. At this stage, we create a cross-level flow of information by utilising the method level information for class level predictions and the class level information to augment the method level predictions.

To compute our scores we start with the techniques which utilise the dynamic information, for which we selected two existing test-to-code traceability techniques and formulated six new techniques. Six of the techniques produce a score in the interval $[0, 1]$ for every possible link, indicating the likelihood that the link is correct, while the other two produce binary scores. For the techniques which utilise static information, we selected the dynamic techniques which were applicable and modified them to work with static instead of dynamic information. We also compute a combined score for all the individual techniques. The method of combining scores is explored in Section 7.2.5. These scores are used to rank the candidate links so that those ranked highest are most likely to be true traceability links. Thresholds are then applied to construct the sets of predicted links.

We describe our techniques in the following section where, for simplicity, we will present them at the method level. To apply them on the class level, test classes are used instead of test methods and tested classes instead of tested functions.

4 Techniques

As discussed in Section 2, existing test-to-code traceability techniques have weaknesses that we try to overcome with new techniques. Despite their weaknesses, we selected two established techniques, Naming Conventions (NC) (Van Rompaey and Demeyer 2009) and Last Call Before Assert (LCBA) (Van Rompaey and Demeyer 2009) because they perform well in certain situations. The new techniques formulated for TCTRACER include four string-based techniques: a variant of Naming Conventions (NCC), two variants of Longest Common Subsequence (LCS-B and LCS-U), and using the Levenshtein edit distance (Levenshtein 1966), which all utilise name similarity. Two statistical call-based techniques (SCTs) based on Tarantula fault localisation (Jones et al. 2002) and Term Frequency–Inverse Document Frequency (TFIDF) (Manning et al. 2010) are also included in the new techniques. All the mentioned techniques will be discussed in their dynamic (Section 4.1) and static (Section 4.2) variants.

The original NC was selected for our technique ensemble as it should have high precision, especially in projects where the naming conventions are strictly followed and is a common method by which developers identify tests for a given method during development (Hurdugaci and Zaidman 2012; Madeja and Porubän 2019). LCBA was selected as it can perform well in certain circumstances, specifically when the tests conform to the style of using an assert to test the returned value from a function immediately after the function is called. As both NC and LCBA are well-established techniques for test-to-code traceability recovery (Qusef et al. 2013, 2014; Madeja and Porubän 2019; Csuvik et al. 2019a), they also make good candidates to serve as comparison points for our other techniques.

NCC requires that the name of the test *contains* the name of the tested artefact. It was included in the technique ensemble as it utilises the strengths of NC but should achieve higher recall as it can establish many-to-one relationships between functions and tests, as opposed to the solely one-to-one relationships that are discoverable with traditional NC. This helps to alleviate some of the problems with traditional NC, as discussed in Section 2. LCS-B and LCS-U compute the ratio of the name lengths and the length of the longest common subsequence of the names of the test and the tested artefact. They were used as they utilise the same intuitions as NC and NCC respectively but instead of producing a binary score, they produce a real-valued score that indicates how close to satisfying NC/NCC the potential link is. This is useful as there are instances where NC/NCC are not satisfied but are very close to being satisfied, for example, in the case of NC, if there are extra words before or after the name of the function or, in the case of NCC, if the name of the function is abbreviated or has grammatical differences in the name of the test case. In these instances, the real-valued scores of LCS-B and LCS-U are more useful than the binary scores of NC and NCC as we can still determine if a test and a function are likely related. We include the normalised Levenshtein distance between the names as a technique as it provides a different view of name similarity to the longest common subsequence which is used in the LCS-B and LCS-U techniques. For these naming techniques, we use the simple method or class names as using FQNs causes the scores to have less difference between them. For example, all the FQNs in Commons Lang share the common prefix *org.apache.commons.lang3* and many share even longer prefixes. Using the FQNs, therefore, squashes the distribution of the naming scores towards the high end making it more difficult to distinguish correct links from incorrect links.

We include the Tarantula technique as, intuitively, the task of recovering test-to-code traceability links is similar to the task of fault localisation as, if a function is causing a test to

fail, it is likely that the function and the test should be linked. Therefore, our intuition is that by adapting a well-known fault localisation technique to traceability we may find an effective method of recovering test-to-code trace links. The inclusion of the TFIDF technique is motivated similarly to Tarantula in that we view the task of determining the relevance of terms to a document as being analogous to the task of determining which functions are most relevant to a test case and therefore which functions are most likely to be the targets of that test. As TFIDF is a standard, well-tested method of establishing term relevance, we adapted this method to test-to-code traceability.

All of the above techniques will be evaluated to identify individual strengths and weaknesses and compared to the established techniques NC and LCBA to establish if their known weaknesses can be overcome. We also include all techniques in a combined score as we believe that each technique has the potential to provide at least some information that cannot be wholly obtained using any other technique.

All of our techniques utilise dynamic trace information and, where possible, we have adapted the techniques to create variations that use static information. We opted to only adapt the naming-based techniques to use static information as to use the call-based techniques we would have to use static call graphs which are inherently an over-approximation with regards to polymorphic function calls that are resolved at run-time. This results in very low precision when using them for traceability techniques.

We have discarded a series of other techniques. First, Fixture Element Types (FET) (Van Rompaey and Demeyer 2009) and SCOTCH+ (Qusef et al. 2014) cannot be applied on method-level and Static Call Graph (SCG). Second, Lexical Analysis (LA), and Co-Evolution (Co-Ev) have been discarded because of their low precision and recall (Van Rompaey and Demeyer 2009; Parizi et al. 2014; Kicsi et al. 2018).

4.1 Dynamic Techniques

In this subsection we describe the techniques that use dynamic information to compute traceability scores.

4.1.1 Naming Conventions

As naming conventions can change between projects (Van Rompaey and Demeyer 2009), we have selected two techniques for traceability recovery using naming conventions: *traditional* and *contains*.

Traditional Naming Conventions (NC). NC establishes links by considering a function to be linked to a test if the name of the test is the same as the function after the word *test* has been removed from the test name. For example, a function named *union* will be considered to be tested by a test named *testUnion*.

$$\text{score}(t, f) = \begin{cases} 1, & \text{if } n_t \text{ equals } n_f \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

where n_t and n_f are the names of t and f respectively, after the word *test* has been removed from the name of test t.

Naming Conventions – Contains (NCC). NCC is a derivative of traditional NC which replaces the requirement that the test name must match the function name exactly, with the more relaxed requirement that the test name only needs to contain the function name.

Therefore, NCC considers a function to be linked to a test if the name of the test contains the name of the function, after removing *test* from the test name. A positive NCC result is counted as a score of 1 while a negative NCC result is counted as 0:

$$\text{score}(t, f) = \begin{cases} 1, & \text{if } n_f \text{ substring of } n_t \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

4.1.2 Name Similarity

Name similarity is a variation of the Naming Conventions approach and is based on the premise that developers, following established naming conventions, give unit tests names that are similar to or match the name of the function. Our hypothesis is that name similarity measures have the potential to perform better than the existing NC approach as they are less strict on exact matches and allow for slight variations in name, for example, due to grammatical reasons. For instance, a method named *clone* would not be identified under NC for a test named *testCloning*, whereas it would be possible under name similarity measures for *clone* to be assigned a high traceability score with *testCloning*. We consider the name for a method to be simply the name of the method in lower case without the class name and with the string *test* removed from test names when performing comparisons. For example, for the fully qualified method name *com.example.ExampleClass.testComputeScore(boolean)*, we perform name similarity comparisons on *computescore*.

To compute the name similarity, we use two well-established techniques, *Longest Common Subsequence (LCS)* and *Levenshtein Distance*.

To establish the LCS similarity, we compare the length of the longest common subsequence to the length of the function and test name. The longest common subsequence techniques give function names that have more characters in common with (and in the same order as) a test name a higher score.

Longest Common Subsequence – Both (LCS-B) In the first LCS variant, we maximise the score at 1 when the method and function names coincide exactly (aligned with the behaviour of the NC approach), that is, when $n_t = n_f$ and $\text{LCS}(n_t, n_f) = n_t$. We divide the length of the LCS by the greater of the length of the two strings as follows:

$$\text{score}(t, f) = \frac{|\text{LCS}(n_t, n_f)|}{\max(|n_t|, |n_f|)} \tag{3}$$

Longest Common Subsequence – Unit (LCS-U) In the second variant, we divide the length of the LCS by the length of the function name only. This variant is more closely aligned with the behaviour of the NCC approach, with the score maximised at 1 when the function name is contained in the test name.

$$\text{score}(t, f) = \frac{|\text{LCS}(n_t, n_f)|}{|n_f|} \tag{4}$$

Levenshtein Distance The Levenshtein distance (Levenshtein 1966), often known as edit distance, measures the distance between two strings by measuring the minimum number of edits it takes to transform one string into the other. Under this technique, the distances between the function names and test names are computed and links with the lowest Levenshtein distance are awarded the highest scores. We first normalise the Levenshtein distance

by dividing it by the length of the longest string and then take the compliment so that higher scores are given to closer strings:

$$\text{score}(t, f) = 1 - \left(\frac{\text{Levenshtein}(n_t, n_f)}{\max(|n_t|, |n_f|)} \right) \tag{5}$$

4.1.3 Last Call Before Assert (LCBA)

LCBA attempts to establish traceability links by working on the assumption that the function returned last before an assert is called is the function that the assert is testing. Therefore, LCBA will establish links between a test and every function that is returned last before an assert that appears in that test. In TCTRACER, if an LCBA link is established between a test and a function it is counted as a traceability score of 1 while no LCBA link is counted as a score of 0:

$$\text{score}(t, f) = \begin{cases} 1, & \text{if } f \text{ is last return before an assert in } t \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

4.1.4 Tarantula

Tarantula (Jones et al. 2002) is an automatic fault localisation technique that assigns a suspiciousness value to code, with higher suspiciousness values indicating a higher probability of the code in question being responsible for the fault. The use of automatic fault localisation is based on the idea that it would point to the most relevant entity if the current test fails. The suspiciousness of a code entity e is defined as follows:

$$\text{suspiciousness}(e) = \frac{\frac{\text{failed}(e)}{totalfailed}}{\frac{\text{passed}(e)}{totalpassed} + \frac{\text{failed}(e)}{totalfailed}} \tag{7}$$

where failed(e) is the number of tests that executed e and failed, *totalfailed* is the number of tests that failed in total, passed(e) is the number of tests that executed e and passed, and *totalpassed* is the number of tests that passed in total.

To obtain the traceability score for a given test-to-function pair, where the test executes the function, we compute the suspiciousness of the function with respect to the test, assuming that the test under consideration fails and all others pass.[4] It is a heuristic to identify the methods that are most specific to the current test. Tarantula decreases the suspiciousness of methods executing during passing tests – in our case, passing tests that execute the method. Using this heuristic we can derive our traceability score equation from (8):

$$\text{score}(t, f) = \frac{1}{\frac{|\{t' \in T : f \in t'\}| - 1}{|T| - 1} + 1} \tag{8}$$

where T is the set of all tests in the test suite and $f \in t'$ indicates that function f is executed by test t'. For pairs where the test t does not execute the function f, a score of 0 is assigned.

[4] A model under which all tests executing the function fail is not suitable as the Tarantula suspiciousness would then be 100%.

4.1.5 Term Frequency–Inverse Document Frequency (TFIDF)

Term frequency–inverse document frequency (TFIDF) is a measure traditionally used in information retrieval to determine how significant a term is to a document. TFIDF takes into account the prevalence of the term in the document and in the corpus as a whole, with the intuition being that if a term is frequent in a particular document but not frequent in the rest of the corpus, that term must carry a high significance to the document and carries useful information about the semantics of the document. We apply this to the domain of test-to-code traceability by having tests take the role of the documents and functions take the role of the terms. This expresses the intuition that if a function is executed frequently by a particular test and infrequently by other tests, it is likely that the test is testing the function. We define our traceability score using TFIDF as:

$$\text{score}(t, f) = \text{tf}(t, f) \cdot \text{idf}(f) \tag{9}$$

The usual definition of the term frequency (tf) function does not match the test/function scenario. Thus, tf and idf are defined as:

$$\text{tf}(t, f) = \ln\left(1 + \frac{1}{|\{f' \in F : f' \in t\}|}\right) \tag{10}$$

$$\text{idf}(f) = \ln\left(1 + \frac{|T|}{|\{t' \in T : f \in t'\}|}\right) \tag{11}$$

where T is the set of all tests in the test suite and F is the set of all functions in the system. The tf function measures how the information of a test is spread over the called functions and the idf function measures how common the function is over all tests.

4.2 Static Techniques

In this section, we describe the techniques we selected to adapt to using static information and the changes that we had to make to them.

4.2.1 Naming Conventions

For the static versions of naming conventions, we use the same variants that we use for our dynamic versions, namely the *traditional* and *contains* variants. However, in contrast to how they are used in the dynamic approach, when using them statically we must utilise both the function name and the class name. This is due to the fact that in the dynamic approach we are only using the names of functions that have been executed, whereas in the static approach we are using all the functions in the project. Therefore, if we were to use only the function name in the static approach, we would likely have a very low precision as it is often the case that multiple classes contain functions of the same name. This effect is most obvious when examining commonly overloaded functions such as *toString*. In this instance, using only the simple name would result in any test for a *toString* function being linked to all *toString* functions in the project instead of just the one belonging to the appropriate class.

Static Naming Conventions (Static NC). Similar to the dynamic approach, we compare function and test names after the word *test* has been removed from the test name. However, we now also incorporate the class name and perform the same comparison with the test class

name. Therefore, we now link a test to a function if the test and function names match and the test class and functions class names match.

$$\text{score}(t, f) = \begin{cases} 1, & \text{if } n_t \text{ equals } n_f \wedge n_{tc} \text{ equals } n_{fc} \\ 0, & \text{otherwise} \end{cases} \tag{12}$$

where n_t and n_f are the names of t and f respectively and n_{tc} and n_{fc} are the names of the classes containing t and f respectively, after the word *test* has been removed from the names of the test and test class.

Static Naming Conventions – Contains (Static NCC). Similar to static NC, we adapt the NCC technique from the dynamic version to incorporate the class names for the static version. Therefore, static NCC considers a function to be linked to a test if the name of the test contains the name of the function and the name of the test class contains the name of the functions class, after removing *test* from the test name and test class name.

$$\text{score}(t, f) = \begin{cases} 1, & \text{if } n_f \text{ substring of } n_t \wedge n_{tc} \text{ substring of } n_{fc} \\ 0, & \text{otherwise} \end{cases} \tag{13}$$

4.2.2 Static Name Similarity

We use the same name similarity techniques in our static approach as in our dynamic approach, namely LCS-B, LCS-U, and Levenshtein distance. The way the scores are computed remains unchanged from the dynamic techniques, as described in Section 4.1. However, in the case of the static techniques, we use the FQNs of the functions and the tests, instead of just the simple names and we remove the word *test* from anywhere it appears in the whole FQN of the test or test class. We use the FQN because, like the static naming conventions techniques, we have to account for the fact that multiple classes are likely to have functions of the same name. Using the FQNs accounts for this, as well as for the situation where different packages may contain classes of the same name.

4.3 Score Scaling

Our approach utilises two techniques for scaling traceability scores which can be applied independently as well as composed together.

4.3.1 Call Depth Discounting

Tests often do not invoke the tested functions directly, for example when a public method delegates the actual implementation to a private method. The TCTracer approach utilises the intuition that the relative depth between a test and a function in the call stack can serve as an indicator of if the function is tested by the test. We hypothesise that functions that are closer to a test in the call stack are more likely to be the tested functions than functions that are far away. Therefore, we utilise a relative call depth discount factor $\gamma \in [0, 1]$, which discounts the traceability score for a test-to-function pair in proportion to the distance between them in the call stack:

$$\text{score}_\text{d}(t, f) = \text{score}(t, f) \cdot \gamma^{(dist(t,f)-1)} \tag{14}$$

Table 1 Traceability techniques, their score range (Score), if the technique is normalised (N), and the used threshold (τ).

Technique	Score	N	τ
Naming Conventions (NC)	0 or 1	–	–
Naming Conv. – Contains (NCC)	0 or 1	–	–
LCS – Unit (LCS-U)	[0, 1]	Yes	0.75
LCS – Both (LCS-B)	[0, 1]	Yes	0.55
Levenshtein (Leven)	[0, 1]	Yes	0.95
Last Call Before Assert (LCBA)	0 or 1	–	–
Tarantula	[0, 1]	Yes	0.95
TFIDF	[0, 1]	Yes	0.90
Static Naming Conventions (Static NC)	0 or 1	–	–
Static Naming Conv. – Contains (Static NCC)	0 or 1	–	–
Static LCS – Unit (Static LCS-U)	[0, 1]	Yes	1.0
Static LCS – Both (Static LCS-B)	[0, 1]	Yes	1.0
Static Levenshtein (Static Leven)	[0, 1]	Yes	0.995

where score_d is the discounted score, score is the non-discounted score, and $dist(t, f)$ is the distance between the test and the function in the call stack. We subtract one from the distance so as to apply no discount to functions that are called directly by the test.

4.3.2 Normalisation

The computed scores can be used to rank the possible links to called functions within a test directly, using the top-ranked link as the most likely link. However, the actual distribution of scores can vary between techniques and between tests. Therefore, we normalise the scores so that the largest score within a test is 1:

$$\text{score}_\text{n}(t, f) = \frac{\text{score}_\text{d}(t, f)}{\max(\{\text{score}_\text{d}(t, f') \mid f' \in t\})} \tag{15}$$

where score_n is the normalised score. Normalisation allows us to define a threshold around the top-ranked link.

In the end, we focus on thirteen individual techniques, shown in Table 1. NC, NCC variants and LCBA are binary, i.e., they produce scores of either 1 or 0 which are used directly. The eight other non-binary techniques are normalised and use call depth discounting.

5 Link Prediction

To construct link predictions, we first apply our traceability techniques to the method level and class level individually. The techniques can be directly applied to the class level by using the test classes instead of test methods and tested classes instead of tested methods. The information extracted from each level is then propagated between levels to produce another set of links at each level. The propagation is done by utilising method level scores in the computation of class level scores and class level scores in the computation of method level scores.

5.1 Method-Level Prediction

The process starts by executing each of our individual traceability techniques at the method level, resulting in a matrix of scores for each technique:

$$\mathbf{M} \in \mathbb{R}^{|\mathrm{T}| \times |\mathrm{F}|} \tag{16}$$

where T is the set of all tests in the system and F is the set of all functions. Each element of $\mathbf{M}$ is the traceability score for a given test-to-function pair $(t, f) \in (\mathrm{T} \times \mathrm{F})$.

Another matrix is then constructed for the combined technique by averaging over all the individual technique matrices and normalising the rows, using (15).

Each of these nine matrices is used to build sets of predicted test-to-function traceability links. To convert the real-valued scores into boolean link/no-link predictions we introduce a set of thresholds, one for each technique (shown in Table 1), and consider scores above the threshold as positive link predictions. (17) defines how each set of method level traceability links are constructed.

$$\mathrm{LM} = \{(t, f) \in \mathrm{T} \times \mathrm{F} \mid \mathbf{M}_{tf} \geq \tau\} \tag{17}$$

where $\mathbf{M}_{tf}$ is the score for the given test-to-function pair and τ is the threshold for the technique.

5.2 Class-Level Prediction

We now move to the class level where, in the same way as the method level, we apply our individual traceability techniques and combine them, resulting in nine matrices, one for each technique:

$$\mathbf{C} \in \mathbb{R}^{|\mathrm{TC}| \times |\mathrm{FC}|} \tag{18}$$

where TC is the set of all test classes in the system and FC is the set of all non-test classes. Each element of $\mathbf{C}$ is the traceability score for a given test-class-to-class pair $(c_t, c_f) \in (\mathrm{TC} \times \mathrm{FC})$.

Similarly to the method level, $\mathbf{C}$ is used to compute sets of class level traceability links using (19).

$$\mathrm{LC} = \{(c_t, c_f) \in \mathrm{TC} \times \mathrm{FC} \mid \mathbf{C}_{c_t c_f} \geq \tau\} \tag{19}$$

5.3 Method- to Class-Level Propagation

Given the method level and class level score matrices, we can now propagate information across levels. First, we elevate the method level information to the class level by extracting scores from $\mathbf{M}$ and organising them into class level pairs. This allows us to use them for computing class level traceability scores. To do this, for each test-class-to-class pair (c_t, c_f), we construct a matrix $\mathbf{EM}(c_t, c_f)$ to hold the relevant method level information:

$$\mathbf{EM}(c_t, c_f) \in \mathbb{R}^{|\mathrm{t}(c_t)| \times |\mathrm{f}(c_f)|} \tag{20}$$

where $\mathrm{t}(c_t)$ is the set of tests in test class c_t, $\mathrm{f}(c_f)$ is the set of functions in class c_f. Each element of $\mathbf{EM}(c_t, c_f)$ is the method level traceability score for a given test-to-function pair $(t, f) \in (\mathrm{t}(c_t) \times \mathrm{f}(c_f))$.

To obtain the traceability score for the test-class-to-class pair, the method-level scores in $\mathbf{EM}(c_t, c_f)$ are summed along both dimensions, resulting in a scalar score.

This process is executed for each test-class-to-class pair in the system and the produced scores are used to create a symmetric matrix that holds the scores for all pairs:

$$\mathbf{EM} \in \mathbb{R}^{|\mathrm{TC}| \times |\mathrm{FC}|} \tag{21}$$

Therefore, each element of **EM** is the score for a given test-class-to-class pair $(c_t, c_f) \in (\text{TC} \times \text{FC})$ that is derived from method level information. All rows in **EM** are normalised using (15).

The scores in **EM** are then used to produce a set of class level predicted links using (22).

$$\text{LEM} = \{(c_t, c_f) \in \text{TC} \times \text{FC} \mid \mathbf{EM}_{c_t c_f} \geq \tau\} \tag{22}$$

5.4 Class- to Method-Level Propagation

To propagate information from the class level to the method level, we take the method level information in **M** and augment it with the class level information in **C**, creating a new matrix $\mathbf{AM} \in \mathbb{R}^{|\text{T}| \times |\text{F}|}$. For each test-to-function pair (t, f), the augmentation is performed by first finding the test-class-to-class pair (c_t, c_f) that corresponds to the test-to-function pair, i.e., the test class c_t that contains the test t and the tested class c_f that contains the function f. We then take the score for the method level pair from **M** and the score for the class level pair from **C** and multiply them to produce the augmented method level score for **AM**, as shown in (23).

$$\mathbf{AM}_{tf} = \mathbf{M}_{tf} \cdot \mathbf{C}_{\text{c}(t)\text{c}(f)} \tag{23}$$

Where $\text{c}(m)$ returns the class containing method m.

From **AM**, the set of augmented method level traceability link predictions are produced using (24).

$$\text{LAM} = \{(t, f) \in \text{T} \times \text{F} \mid \mathbf{AM}_{tf} \geq \tau\} \tag{24}$$

6 Implementation

TCTracer is compatible with any Java system that uses the JUnit 3, 4, or 5 test framework and is compatible with Java 8 or newer. Dynamic trace data is collected from JUnit test suite executions, which is then used for computing the dynamic traceability links by the techniques described in Section 4.1.

To collect the dynamic execution traces, TCTracer requires the system-under-analysis to be instrumented. The Java Agent API was used for this as it provides access to the bytecode of Java classes and allows for them to be transformed before being loaded by the JVM. As shown in Fig. 1, the instructions for transforming the bytecode are provided by a Java program, TCAgent, which is passed to the JVM at runtime through the *-javaagent* flag. TCAgent utilises the ByteBuddy[5] library and allows us to easily transform the bytecode of the running system to log the data that is used by TCTracer to compute the traceability links.

The execution traces are parsed to collect the dynamic information for each test and record the set of methods that were the last return before an assert was called, as is needed for LCBA. Methods that are not defined in the project-under-analysis, such as those from third-party APIs, are filtered out.

The static information is obtained by scanning for .java files in the source and test folders in the project-under-analysis and using Java Parser[6] to parse the classes and test classes. These are used to extract the functions and tests from the project which are stored in TCTracer.

[5] https://bytebuddy.net
[6] https://javaparser.org/

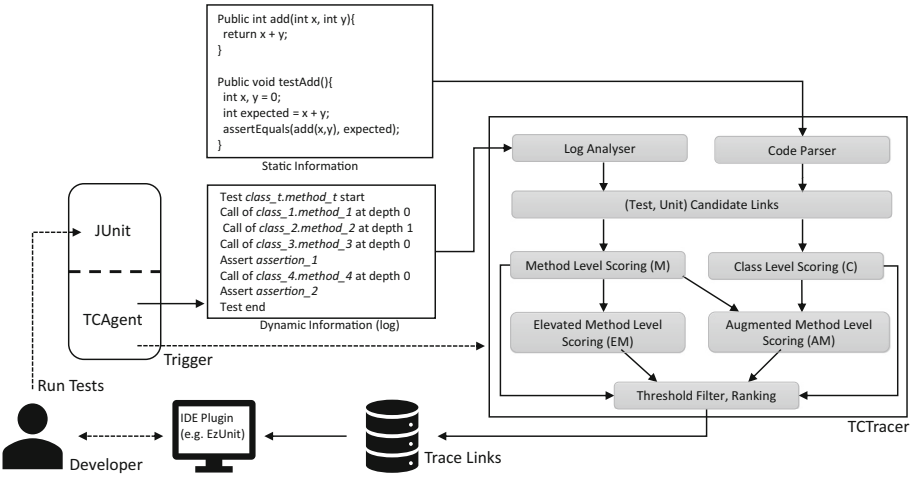

Fig. 1 Integration of TCTRACER into JUnit

The main challenge of working with static information is the number of test-to-function pairs that we need to compute scores for. This is essentially the total number of tests in the project-under-analysis multiplied by the total number of functions. This leads to a very large number of candidates pairs, even in medium-sized projects. For example, Commons Lang has a total of 9,522,771 candidate pairs. This creates a problem when it comes to link prediction as it causes the matrices to be very large and results in the time and space complexity of the analysis increasing to the point where it is intractable on even high resource computers. To work around this problem we set a threshold on the sum of the static scores, which we use to discard any candidate pair that does not meet the threshold, before progressing to the link prediction phase. This threshold was set by finding the highest value which does not have any impact on the recall and, therefore, does not filter out any true links. This does not mean that we will achieve 100% recall overall, just that recall is not lower than it would be without this threshold being applied. By doing this, we can filter out over 90% of the candidate links before the link prediction phase, effectively managing the size of the matrices, while not affecting the ability of TCTRACER to find correct links.

In the final phase, TCTRACER computes the sets of predicted links described in Section 5 using the dynamic information, the static information, and configuration parameters, such as threshold and call depth discount factor. If a ground truth is present, TCTRACER computes the evaluation metrics for each set of predicted links.

7 Evaluation

This section presents our research questions, the design of the experiments carried out to answer these questions, the results, and a discussion of the findings.

7.1 Experimental Setup

The experimental setup consists of running TCTRACER on a set of open source subjects and computing a set of evaluation measures for each subject, using a manually established ground truth.

Subjects. For our subjects, we selected three well known open source projects that are written in Java and utilise the JUnit testing framework: Commons IO,[7] Commons Lang,[8] and JFreeChart.[9] These subjects were selected as they are well known, widely used, and sufficiently large to demonstrate the applicability of TCTracer to real-world systems. For the evaluation of TCTracer, we established a ground truth for these projects at both the method level and the class level. To establish the method level ground truth, we used a team of three judges, one PhD student and two final-year undergraduate students, who each independently inspected a set of tests selected uniformly at random from the subjects and made determinations about which functions were tested by each test. In order to make these judgements, the judges looked at evidence such as which functions were called, how often they were called, how many other functions were called, how often called functions were called by other tests, the names of the tests, and which functions returned values that were then checked by an assert. After conducting this process independently, the judges collectively inspected any instances where there were disagreements and were able to reach a final, unanimous judgement, resulting in full inter-rater agreement.

In addition to our own method level ground truth, we searched for an existing ground truth that has been used in previous work to broaden our results and cross-validate our ground truth creation protocol. This resulted in the discovery of two other ground truths which contained seven projects between them. We investigated the given link sets for all seven of these projects but decided to only use one of them. The reasons for rejecting the link sets for the other projects were numerous, with all of the link sets suffering from multiple problems. The list of problems affecting these projects included a lack of random sampling, poor project selection, including interface methods as tested methods, choosing functions in base classes that were tested by many tests identical tests, choosing tests that are too similar to each other, and inaccuracies in the links. Only the links for the project Gson[10] from the TestRoutes (Kicsi et al. 2020) data set were not affected by any of these issues, allowing us to utilise them. In total, the method level ground truth contains 218 oracle links and an analysis of the method level ground truth shows the number of functions per test ranges from 1 to 12, with a median of 1 for all projects and a mean average of 1.66. The difference between the median and the mean is due to a handful of tests in each project having an unusually large number of tested functions. For example, in Gson, there is one test with 12 tested functions and another with 11 tested functions. This causes the average to be much higher at 1.89 while the median is still 1.

The class level ground truth was provided mostly by the developers as, in all three projects, a subset of the test classes contain a comment at the start of the class specifying which classes it tests. These developer provided links were extracted and then manually verified by a judge to confirm that they are still valid. To boost the number of links for the project with the least developer links, Commons IO, a random sample was drawn from the set of all test classes and the tested classes for this sample were decided by two judges in the same way as the method level sample, again resulting in full inter-rater agreement. Another class level ground truth had previously been established by SCOTCH+ (Qusef et al. 2014), which we also investigated for use. However, due to the age of the projects, they were all no longer able to be built or were incompatible with our tracing agent, TCAgent, which

[7] https://commons.apache.org/proper/commons-io/
[8] https://commons.apache.org/proper/commons-lang/
[9] http://www.jfree.org/jfreechart/
[10] https://github.com/google/gson

Table 2 Subject statistics

Project	Version	Num. Functions	Num. Tests	Instruction coverage	Num. of method level ground truth links	Num. of class level ground truth links
Apache Ant	1.9.5	10477	1830	50%	-	79
Commons IO	2.5	1246	994	89%	41	56
Commons Lang	3.7	3111	3061	95%	78	85
JFreeChart	1.0.19	9053	2244	52%	44	388
Gson	2.8.0	635	1006	83%	55	-

requires Java 8 or newer. The only ground truth links that we were able to use were for Apache Ant[11] and the results cannot be compared directly as the oldest version of Apache Ant that was compatible with TCAGENT was newer than the version used by SCOTCH+. The links that we used from SCOTCH+ were independently established by three judges with an average inter-rater agreement of 90%. In total, our class level ground truth contains 608 links. Information about the subjects and ground truth is given in Table 2.

As we use Gson at the method level, we have also investigated using it for a class level ground truth, however, the nature of this project does not lend itself to an evaluation at the class level. This is because most of the test classes test the same class *com.google.gson*. This is due to the fact the library exposes the serialisation and deserialisation methods through this class, which makes up the bulk of the libraries interface. Thus the test classes testing different aspects of serialisation and deserialisation are all linked to this single class. This makes Gson not representative of software projects in general and therefore not useful for an evaluation which needs to produce well generalised results.

In similar fashion, as we only have class level links for the Ant project, we investigated using Ant for a method level ground truth also. However, Ant is not suited to providing a method level evaluation as many of the tests are not unit testing individual functions but are testing the execution of Ant tasks. A set of Ant tasks have been pre-defined for testing purposes and the tests call into the *execute* method of the task runner to run them. The runner then runs the task and returns the output, which is then checked. This testing pattern doesn't fit in with our approach as it more closely resembles integration testing, rather than unit testing, and does not allow us to establish clear test-to-function relationships.

Some previous work (Csuvik et al. 2019a) has used naming conventions to establish a ground truth. However, as demonstrated by our work, this technique has low recall and would introduce bias. Ultimately, when creating a new ground truth, one cannot simply apply an existing traceability technique, as it causes a bias towards that type of technique.

Evaluation Measures. The evaluation measures we selected are: precision, recall, F1 score, mean average precision (MAP), and area under the precision-recall curve (AUC) (Manning et al. 2010). We selected precision and recall as they are elementary measures for evaluating the performance of a binary classifier and allow us to measure the proportion of true positives out of all positive predictions and the proportion of all positive examples that are retrieved. As precision and recall generally represent a trade-off between each other, the F1

[11] https://ant.apache.org/

score is a useful measure as it evenly weights both precision and recall, allowing us to determine which techniques best handle the trade-off. We also use the mean average precision (MAP) as it takes into account the rank of the true positives in our link prediction lists. This is useful information as it shows which techniques are better at ranking true positives higher than false positives and will also punish techniques that more often return no positives at all.

Finally, we use the area under the precision-recall curve (AUC) as it gives us a view of the performance of each technique that is threshold independent. As most of our techniques need a threshold to make predictions, the performance of these techniques can be very sensitive to the values used for their thresholds. An incorrectly chosen threshold can give the incorrect impression of the usefulness of a technique and, therefore, while we have attempted to select the best threshold for each technique, AUC gives us a general measure of the performance of these techniques that is not affected by threshold values. We selected a precision-recall (PR) curve over a receiver operating characteristics (ROC) curve because the classes in our domain are unbalanced, there are many more negative links than positive links, and PR curves exhibit better characteristics in this situation (Davis and Goadrich 2006). All scores are presented as integer percentages for the sake of readability.

Van Rompaey and Demeyer (2009) also measure applicability, i.e., the ratio of tests for which at least one link is retrieved. However, because of the normalisation that we apply, all non-binary techniques will always produce at least one link, resulting in 100% applicability.

7.2 Research Questions

In the following, we will evaluate the presented techniques according to a list of research questions:

1. How effective are our techniques at the method level?
2. How effective are our techniques at the class level?
3. What effectiveness is achieved by utilising method level information for class level traceability?
4. Can we improve method level predictions by augmenting with class level information?
5. Can we improve predictions by combining the individual technique scores into a single score?
6. What are the reasons for the occurrence of false negatives and false positives?

The six research questions and findings will be presented below. While the first five research questions are answered with a quantitative evaluation, the sixth required a qualitative evaluation. The results are discussed in Section 8.

7.2.1 Research Question 1 (Method Level)

How effective are our techniques at the method level?

This research question investigates how effective each of the techniques are for establishing test-to-function links using only method level information. To answer this question, we compute the evaluation measures over the link sets produced using (17).

Findings From the results for RQ1, shown in Table 3, we see that, on average, LCS-U is the most desirable as it performs best for F1 and AUC while only trailing the best MAP (LCS-B) by one point. This means that it is good at balancing precision and recall, is consistent when changing thresholds, and could benefit from a further optimised threshold selection. For

Table 3 RQ1 – Method level traceability

	Technique	Prec.	Recall	MAP	F1	AUC	True Pos.	False Pos.
Commons IO	NC	**100**	07	09	14	–	3	0
	NCC	94	39	45	55	–	16	1
	LCS-U	66	76	68	**70**	**63**	31	16
	LCS-B	49	**85**	**70**	63	54	35	36
	Leven	66	56	60	61	58	23	12
	LCBA	44	34	32	38	–	14	18
	Tarantula	58	68	67	63	52	28	20
	TFIDF	59	66	65	62	59	27	19
	Static NC	**100**	07	09	14	–	3	0
	Static NCC	17	39	24	24	–	16	77
	Static LCS-U	13	46	32	21	9	19	124
	Static LCS-B	18	32	30	23	11	13	60
	Static Leven	29	49	47	36	16	20	50
Commons Lang	NC	**100**	10	18	19	–	8	0
	NCC	98	53	57	68	–	41	1
	LCS-U	82	77	86	79	84	60	13
	LCS-B	67	**85**	82	75	74	66	32
	Leven	84	55	73	67	76	43	8
	LCBA	84	69	64	76	–	54	10
	Tarantula	79	**85**	**87**	81	84	66	18
	TFIDF	89	81	85	**85**	**87**	63	8
	Static NC	90	12	20	20	–	9	1
	Static NCC	26	53	40	35	–	41	116
	Static LCS-U	20	58	46	29	14	45	183
	Static LCS-B	25	38	37	31	15	30	88
	Static Leven	25	46	51	33	16	36	107
JFreeChart	NC	**100**	16	21	27	–	7	0
	NCC	92	25	32	39	–	11	1
	LCS-U	63	66	77	64	**61**	29	17
	LCS-B	30	**84**	**82**	44	60	37	87
	Leven	83	57	74	68	60	25	5
	LCBA	67	77	79	**72**	–	34	17
	Tarantula	39	77	78	52	41	34	53
	TFIDF	55	66	73	60	57	29	24
	Static NC	80	09	12	16	–	4	1
	Static NCC	56	20	20	30	–	9	7
	Static LCS-U	40	41	42	40	18	18	27
	Static LCS-B	47	39	43	43	20	17	19
	Static Leven	39	43	51	41	21	19	30

Table 3 (continued)

	Technique	Prec.	Recall	MAP	F1	AUC	True Pos.	False Pos.
Gson	NC	**100**	11	10	20	–	6	0
	NCC	80	22	20	34	–	12	3
	LCS-U	56	82	78	67	65	45	35
	LCS-B	34	**85**	**79**	49	63	47	90
	Leven	75	76	77	**76**	**66**	42	14
	LCBA	58	65	65	62	–	36	26
	Tarantula	59	69	70	64	54	38	26
	TFIDF	61	69	70	65	54	38	24
	Static NC	**100**	07	06	14	–	4	0
	Static NCC	47	16	15	24	–	9	10
	Static LCS-U	23	49	30	31	14	27	93
	Static LCS-B	17	22	18	19	9	12	59
	Static Leven	17	29	25	22	11	16	76
Average	NC	**100**	11	14	20	–	6	0
	NCC	91	35	38	49	–	20	2
	LCS-U	67	75	77	**70**	**68**	41	20
	LCS-B	45	**85**	**78**	58	63	46	61
	Leven	77	61	71	68	65	33	10
	LCBA	63	62	60	62	–	35	18
	Tarantula	59	75	75	65	58	42	29
	TFIDF	66	70	73	68	64	39	19
	Static NC	93	09	12	16	–	5	1
	Static NCC	37	32	25	28	–	19	53
	Static LCS-U	24	49	38	30	14	27	107
	Static LCS-B	27	33	32	29	14	18	57
	Static Leven	27	42	44	33	16	23	66

precision alone, NC is the best, while LCS-B is best for recall. When comparing variants, the dynamic techniques consistently outperform the static techniques.

7.2.2 Research Question 2 (Class Level)

How effective are our techniques at the class level?

This research question investigates how effective each of the techniques are for establishing test-class-to-class links, using only class level information. To answer this question, we compute the evaluation measures over the link sets produced using (19).

Findings From the results for RQ2, shown in Table 4, we see that Static Levenshtein is the most desirable overall with the best F1 score and only one point lower than the best MAP (Static LCS-B) and the best AUC (Leven). It is also evident that at the class level, the static techniques outperform the dynamic techniques with generally higher precision and recall. For pure precision, NC variants win again.

Table 4 RQ2 – Class level traceability

	Technique	Prec.	Recall	MAP	F1	AUC	True Pos.	False Pos.
Apache Ant	NC	**100**	73	75	84	–	57	0
	NCC	89	73	75	80	–	57	7
	LCS-U	65	71	73	67	62	55	30
	LCS-B	51	72	66	60	**86**	56	53
	Leven	87	68	72	76	**86**	53	8
	LCBA	50	59	52	54	–	46	46
	Tarantula	49	46	47	47	66	36	38
	TFIDF	51	46	47	49	66	36	34
	Static NC	**100**	83	86	**91**	–	65	0
	Static NCC	40	87	68	54	–	68	104
	Static LCS-U	37	87	64	52	40	68	118
	Static LCS-B	86	**88**	**91**	87	84	69	1
	Static Leven	91	87	90	89	84	68	7
Commons IO	NC	**100**	86	89	92	–	43	0
	NCC	94	90	92	92	–	45	3
	LCS-U	73	90	88	80	86	45	17
	LCS-B	61	92	80	73	**92**	46	30
	Leven	**100**	90	93	**95**	**92**	45	0
	LCBA	50	70	67	58	–	35	35
	Tarantula	74	78	80	76	64	39	14
	TFIDF	74	78	80	76	65	39	14
	Static NC	**100**	86	89	92	–	43	0
	Static NCC	98	92	**95**	**95**	–	46	1
	Static LCS-U	82	**94**	94	88	78	47	10
	Static LCS-B	92	88	90	90	84	44	4
	Static Leven	94	90	93	92	88	45	3
Commons Lang	NC	**100**	71	79	83	–	55	0
	NCC	95	81	89	88	–	63	3
	LCS-U	77	81	86	79	73	63	19
	LCS-B	63	82	84	72	71	64	37
	Leven	95	81	89	88	79	63	3
	LCBA	51	68	67	58	–	53	51
	Tarantula	50	59	63	54	38	46	46
	TFIDF	34	56	56	43	32	44	85
	Static NC	**100**	72	80	84	–	56	0
	Static NCC	84	**87**	91	86	–	68	13
	Static LCS-U	77	**87**	88	82	67	68	20
	Static LCS-B	94	86	**94**	90	82	67	4
	Static Leven	96	86	**94**	**91**	**83**	67	3

Table 4 (continued)

	Technique	Prec.	Recall	MAP	F1	AUC	True Pos.	False Pos.
JFreeChart	NC	**100**	85	91	**92**	–	329	0
	NCC	73	**86**	84	79	–	330	123
	LCS-U	56	**86**	79	68	62	332	266
	LCS-B	58	**86**	81	69	**86**	332	239
	Leven	99	**86**	**92**	**92**	**86**	332	3
	LCBA	31	82	67	45	–	314	684
	Tarantula	69	77	77	73	66	295	133
	TFIDF	67	77	78	72	66	297	145
	Static NC	**100**	85	91	**92**	–	327	0
	Static NCC	59	85	79	70	–	328	229
	Static LCS-U	46	85	72	60	40	328	381
	Static LCS-B	98	85	91	91	84	327	6
	Static Leven	98	85	91	91	84	327	6
Average	NC	**100**	76	80	86	–	121	0
	NCC	87	78	81	82	–	124	34
	LCS-U	65	77	78	70	71	124	83
	LCS-B	56	78	74	65	84	125	90
	Leven	92	76	81	83	**86**	123	4
	LCBA	46	67	59	53	–	112	204
	Tarantula	54	57	58	55	59	104	58
	TFIDF	51	56	57	53	57	104	70
	Static NC	**100**	81	86	89	–	123	0
	Static NCC	55	**87**	76	66	–	128	87
	Static LCS-U	49	**87**	72	61	56	128	132
	Static LCS-B	91	**87**	**92**	89	84	127	6
	Static Leven	94	86	91	**90**	85	127	5

7.2.3 Research Question 3 (Elevated Method Level)

What effectiveness is achieved by utilising method level information for class level traceability?

This research question investigates how each of the techniques perform for establishing test-class-to-class links when we use method level information that has been elevated to the class level. To answer this question, we compute the evaluation measures over the link sets produced using (22).

Findings From the results shown in Table 5 we see that TFIDF is the best for MAP, F1 score, and AUC by a clear margin. Static NC wins on precision and this time LCS-B is best for recall.

7.2.4 Research Question 4 (Augmented Method Level)

Can we improve method level predictions by augmenting with class level information?

Table 5 RQ3 – Elevated method level traceability

	Technique	Prec.	Recall	MAP	F1	AUC	True Pos.	False Pos.
Apache Ant	NC	53	26	26	34	–	20	18
	NCC	49	29	30	37	–	23	24
	LCS-U	56	55	55	55	42	43	34
	LCS-B	52	58	57	55	41	45	41
	Leven	64	53	55	58	43	41	23
	LCBA	70	58	59	63	–	45	19
	Tarantula	72	56	59	63	49	44	17
	TFIDF	76	**60**	**63**	**67**	**54**	47	15
	Static NC	**100**	35	36	51	–	27	0
	Static NCC	77	44	44	56	–	34	10
	Static LCS-U	30	28	29	29	14	22	52
	Static LCS-B	30	28	29	29	15	22	52
	Static Leven	30	28	29	29	15	22	52
Commons IO	NC	87	40	40	55	–	20	3
	NCC	90	56	56	69	–	28	3
	LCS-U	83	80	**80**	**82**	**78**	40	8
	LCS-B	77	**82**	79	80	76	41	12
	Leven	80	72	73	76	72	36	9
	LCBA	71	60	63	65	–	30	12
	Tarantula	82	74	76	78	74	37	8
	TFIDF	82	74	76	78	75	37	8
	Static NC	**100**	38	39	55	–	19	0
	Static NCC	**100**	60	61	75	–	30	0
	Static LCS-U	02	02	02	02	1	1	46
	Static LCS-B	09	08	09	08	2	4	43
	Static Leven	08	08	09	08	2	4	47
Commons Lang	NC	90	55	62	68	–	43	5
	NCC	91	64	71	75	–	50	5
	LCS-U	81	73	79	77	70	57	13
	LCS-B	83	**76**	**82**	79	69	59	12
	Leven	85	74	81	79	70	58	10
	LCBA	83	67	74	74	–	52	11
	Tarantula	82	69	76	75	66	54	12
	TFIDF	88	74	**82**	**81**	**72**	58	8
	Static NC	**100**	49	55	66	–	37	0
	Static NCC	96	68	76	80	–	52	2
	Static LCS-U	23	21	21	22	14	16	53
	Static LCS-B	28	24	26	26	15	19	50
	Static Leven	27	24	26	26	15	19	51

Table 5 (continued)

	Technique	Prec.	Recall	MAP	F1	AUC	True Pos.	False Pos.
JFreeChart	NC	76	63	69	69	–	243	76
	NCC	75	63	69	68	–	241	82
	LCS-U	66	69	70	67	58	264	139
	LCS-B	56	74	72	64	57	285	225
	Leven	69	64	68	66	57	246	110
	LCBA	81	76	79	78	–	292	70
	Tarantula	73	63	67	68	58	243	90
	TFIDF	83	75	78	79	72	287	57
	Static NC	**100**	**78**	**83**	**88**	–	301	0
	Static NCC	73	75	75	74	–	290	106
	Static LCS-U	31	27	29	29	19	103	230
	Static LCS-B	31	27	29	29	20	104	229
	Static Leven	34	30	32	32	22	115	220
Average	NC	76	46	49	57	–	82	26
	NCC	76	53	57	62	–	86	29
	LCS-U	72	69	71	70	62	101	49
	LCS-B	67	**72**	72	69	61	108	73
	Leven	75	66	69	70	61	95	38
	LCBA	76	65	69	70	–	105	28
	Tarantula	77	66	70	71	62	95	32
	TFIDF	82	71	**75**	**76**	**68**	107	22
	Static NC	**100**	50	53	65	–	96	0
	Static NCC	87	62	64	71	–	102	30
	Static LCS-U	21	19	20	20	12	36	95
	Static LCS-B	24	22	23	23	13	37	94
	Static Leven	25	23	24	24	14	40	93

This research question investigates if the method level traceability performance can be improved by augmenting the method level information with class level information. To answer this question, we compute the evaluation measures over the link sets produced using (24).

Findings The results for RQ4, shown in Table 6, show that, on average, LCS-U is the most desirable technique when utilising augmented scores as it has the highest F1 and AUC scores. However, the average scores for LCS-U are similar to the average scores when using the unaugmented method level technique. For pure precision TFIDF is the best technique. It can be observed that for a number of techniques the augmentation produces a drastically higher number of false positives.

7.2.5 Research Question 5 (Technique Combination)

As described in Section 5, we also compute a combined score which averages and normalises the individual technique scores. As this score takes a simple average and weights all techniques equally we refer to it as the *simple* combination method.

Table 6 RQ4 – Augmented method level traceability

	Technique	Prec.	Recall	MAP	F1	AUC	True Pos.	False Pos.
Commons IO	NC	03	**90**	18	06	–	37	1237
	NCC	12	73	54	21	–	30	217
	LCS-U	**73**	73	74	**73**	**75**	30	1
	LCS-B	50	**90**	**78**	64	59	37	37
	Leven	65	59	63	62	62	24	13
	LCBA	05	41	31	10	–	17	294
	Tarantula	64	66	68	65	59	27	15
	TFIDF	65	68	69	67	65	28	15
	Static NC	03	**90**	18	06	–	37	1237
	Static NCC	09	71	30	16	–	29	297
	Static LCS-U	14	51	38	22	10	21	125
	Static LCS-B	34	44	42	38	19	18	35
	Static Leven	27	49	47	34	18	20	55
Commons Lang	NC	03	**95**	31	07	–	74	2095
	NCC	10	78	63	17	–	61	566
	LCS-U	81	81	**89**	81	87	63	15
	LCS-B	56	90	86	69	80	70	54
	Leven	44	60	76	51	43	47	59
	LCBA	17	74	64	27	–	58	286
	Tarantula	86	79	84	83	83	62	10
	TFIDF	**90**	81	85	**85**	**89**	63	7
	Static NC	03	82	29	06	–	64	2086
	Static NCC	09	77	44	15	–	60	637
	Static LCS-U	20	59	47	30	14	46	184
	Static LCS-B	28	44	44	34	16	34	88
	Static Leven	26	47	52	33	16	37	106
JFreeChart	NC	06	77	31	11	–	35	514
	NCC	08	75	43	15	–	34	372
	LCS-U	44	75	74	55	45	33	42
	LCS-B	24	**89**	**84**	38	49	39	123
	Leven	**58**	66	74	62	**51**	29	21
	LCBA	57	70	71	**63**	–	32	23
	Tarantula	43	59	64	50	36	26	34
	TFIDF	52	59	63	55	49	26	24
	Static NC	06	73	24	10	–	32	550
	Static NCC	07	59	31	12	–	26	372
	Static LCS-U	45	45	50	45	25	20	24
	Static LCS-B	43	45	51	44	23	20	27
	Static Leven	37	48	52	42	24	21	36

Table 6 (continued)

	Technique	Prec.	Recall	MAP	F1	AUC	True Pos.	False Pos.
Gson	NC	09	80	28	16	–	44	465
	NCC	10	76	33	18	–	42	360
	LCS-U	62	78	76	69	63	43	26
	LCS-B	32	**85**	**81**	46	65	47	102
	Leven	**75**	75	76	**75**	**66**	41	14
	LCBA	24	75	69	36	–	41	129
	Tarantula	64	69	70	67	55	38	21
	TFIDF	62	69	69	66	55	38	23
	Static NC	08	78	24	15	–	43	481
	Static NCC	10	75	29	17	–	41	389
	Static LCS-U	19	73	36	30	16	39	170
	Static LCS-B	18	67	34	28	15	37	171
	Static Leven	19	71	39	30	16	39	166
Average	NC	05	86	27	10	–	48	1078
	NCC	10	76	48	18	–	42	379
	LCS-U	65	77	78	**70**	**68**	42	24
	LCS-B	41	**89**	**82**	54	63	48	79
	Leven	60	65	72	62	56	35	27
	LCBA	26	65	59	34	–	37	183
	Tarantula	65	68	72	66	58	38	20
	TFIDF	**67**	69	72	68	65	39	17
	Static NC	05	81	24	09	–	44	1089
	Static NCC	08	70	33	15	–	39	424
	Static LCS-U	25	57	42	32	16	32	126
	Static LCS-B	31	50	43	36	18	27	80
	Static Leven	27	54	47	35	19	29	91

Technique Exclusion: *Can we achieve optimal performance with a subset of the individual techniques?*

Given that we are combining a set of techniques, it is natural to ask if any of the techniques are redundant, or even harmful to performance, and if we can optimise or improve performance by removing any of the techniques from the combined score. To investigate this, we ran the method level experiments again looking at the combined technique performance when one of the individual techniques was excluded. This analysis was run once for each technique to determine what the impact of removing that technique was.

Technique Weighting: *Can we improve the performance of the combined scores by weighting individual techniques differently?*

As we are combining techniques, it is natural to investigate if there is a way we can improve the results achieved using the combined score by weighting the individual technique scores before combining them. To perform a weighting we must define a weight vector that contains a value for each of the thirteen individual techniques which we can multiply with the scores matrix before combination. This gives us an extremely large space of

possible weight vectors to choose from. There are many possible approaches for determining the values of the weight vectors as it is simply an optimisation problem to which a wide range of search-based approaches can be applied. However, before investing large amounts of time and resources into optimising the weight vector, some preliminary work is required to determine if using weightings even has the potential to deliver significant results. To test the hypothesis that weighting can significantly change the results we used a simple approach to weighting called precision-based weighting which allows us to select a weight vector that should intuitively have a good chance of being beneficial. When using precision-based weighting, we set the weight of a technique to the precision achieved by that individual technique. For example, the weight for the Levenshtein technique at the method level is set to 0.66 as it achieves a 66% precision in the RQ1 results. This provides an intuitive weight vector as the more precise a technique is, the higher is it weighted.

Machine Learning: *Can a machine learning method for technique combination outperform our standard approach?*

Choosing the right weights for combining the scores is complex due to the size of the search space. We, therefore, investigate if a machine learning method can outperform our standard approach.

As the development of machine learning techniques for combination is not a primary focus of this paper, we opted to use a very simple feedforward network consisting of just a single hidden layer with 64 units. To use our feedforward network for technique combination, we supply the vector of individual technique scores between a test and a function as the input and use a single real-valued output as the probability that the test and function form a true link. To implement this, we used the *keras.Sequential* model from TensorFlow with the mean squared error loss function and the Adam optimiser. The model was constructed with one hidden layer of 64 units and 1 unit in the output layer. We trained for 12000 steps, checkpointing every 1000 steps, and selected the checkpoint with the best accuracy for inference. The biggest challenge with this approach is obtaining a labelled data set for training and validation. As discussed in Section 7.1, creating ground truth data sets for traceability is time-consuming and error-prone. Therefore, manually creating an entire data set that is large enough to train a neural network is not feasible. Our solution to this was to augment our manually created ground truth with extra links which we extracted by assuming that links that had an NC score of 1 were true positive links, and links that had a very low sum of technique scores were true negative links. We make the first assumption as the results for RQ1 and RQ2 show that the NC technique has perfect precision. We need the second assumption as we need to label as many true negatives as true positives to have a balanced data set. Therefore, we take a sum of the scores for all the techniques and mark the lowest scoring N links as true negatives, where N is the number of links marked as true positive using the manual ground truth and the NC assumption. This gives us $2N$ total links with a 50/50 split between true positive labels and true negative labels. We drew the training and validation sets from the Commons IO, Commons Lang, and JFreeChart projects, leaving the Gson and Apache Ant projects as hold outs to check for project overfitting.

Findings The results, shown in Tables 7 and 8, reveal that, on average, the simple combined technique outperforms any individual technique. The results from the technique exclusion (not shown in the tables) revealed that removing any technique made the results worse for

Table 7 RQ5 – Method level technique combination comparison

	Technique	Prec.	Recall	MAP	F1
Commons IO	Simple	**71**	**83**	**79**	**76**
	Prec. Based Weighting	66	79	**79**	72
	FFN	**71**	37	34	48
Commons Lang	Simple	**89**	**86**	**92**	**88**
	Prec. Based Weighting	85	71	84	77
	FFN	84	72	67	77
JFreeChart	Simple	**81**	**80**	**86**	**80**
	Prec. Based Weighting	63	68	74	65
	FFN	65	**80**	64	71
Gson	Simple	**73**	84	83	**78**
	Prec. Based Weighting	72	**85**	**84**	**78**
	FFN	56	36	36	44
Average	Simple	**79**	**83**	**85**	**81**
	Prec. Based Weighting	72	76	80	73
	FFN	69	56	50	60

at least two of the projects with the exception of naming conventions, for which the removal had no effect on the results. As for precision-based weighting, at the method level it underperforms no weighting, while at the class level both approaches are essentially equivalent. The results for the machine learning based combination reveal that the standard combination approach consistently outperforms the neural network approach.

Table 8 RQ5 – Class level technique combination comparison

	Technique	Prec.	Recall	MAP	F1
Apache Ant	Simple	**79**	**90**	**92**	**84**
	Prec. Based Weighting	78	90	91	83
	FFN	60	67	70	63
Commons IO	Simple	**98**	**96**	**97**	**97**
	Prec. Based Weighting	**98**	**96**	**97**	**97**
	FFN	84	86	87	85
Commons Lang	Simple	90	85	92	87
	Prec. Based Weighting	**92**	**86**	**94**	**89**
	FFN	84	78	85	81
JFreeChart	Simple	**98**	**86**	**92**	**92**
	Prec. Based Weighting	**98**	**86**	**92**	**92**
	FFN	90	**86**	90	88
Average	Simple	91	**89**	93	**90**
	Prec. Based Weighting	**92**	90	**94**	**90**
	FFN	74	74	79	74

Table 9 RQ6 – Incorrect link reason categorisation

Category ID	Description
A	The tested function has low naming scores compared to other functions.
B	LCBA finds a non-tested function and does not find the tested function.
C	LCBA cannot find the tested function as JUnit *fail* calls are not executed and therefore not accounted for by LCBA.
D	A non-tested method is called frequently.
E	A non-tested overload of a tested function has similar scores to the tested function.
F	A non-tested or default constructor is wrongly marked as tested due to similar naming technique scores as the tested constructor.
G	A non-tested function is higher in the call stack than the tested function.
H	The test tests class functionality, not individual functions.
I	The test is named after an issue number resulting in poor naming scores.
J	The test doesn't execute the tested method.
K	The test tests an exception not the function.

7.2.6 Research Question 6 (False Negative and False Positive Analysis)

What are the reasons for the occurrence of false negatives and false positives?

Although we achieve high F1 scores using the simple combined technique there are still instances where we produce false positives and false negatives. Investigating these instances and determining the reasons for them is useful as it may reveal ways in which we can improve the approach or show opportunities for improving the software engineering process. To do this, we investigated every false positive and false negative in the Commons IO, Commons Lang, and JFreeChart projects using the simple combined score at the method level and categorised the reason for it. This was done by determining the cause of the false positive or false negative and either adding that cause to the list of categories or assigning it to the existing category if we had already encountered that cause for another example. The resulting categories are defined in Table 9.

Table 10 RQ6 – False positive and false negative analysis

Category	Commons IO		Commons Lang		JFreeChart		Totals	
	FP	FN	FP	FN	FP	FN	FP	FN
A	0	0	0	10	0	5	0	15
B	0	0	0	1	5	0	5	1
C	0	2	0	0	5	0	5	2
D	1	0	5	0	0	0	6	0
E	3	0	0	0	1	0	4	0
F	0	0	3	0	1	0	4	0
G	2	2	0	0	0	0	2	2
H	4	2	0	0	0	0	4	2
I	0	0	0	2	0	3	0	5
J	0	0	0	0	0	1	0	1
K	0	1	0	0	0	0	0	1

Table 11 Extended manual precision evaluation results

Project	True Positives	False Positives	Precision	Fleiss' kappa
Commons Net	5	20	20%	0.48
Commons Text	22	3	88%	0.62

Findings The results for RQ6, shown in Table 10, show the largest sources of false negatives are the tested function scoring poorly in the naming techniques versus some other non-tested function and the test being named after an issue number rather than a tested function. The primary source of false positives is non-tested functions being called frequently, leading to high scores from the SCTs, and the fact that *fail* calls are not captured by LCBA because they are not executed in passing tests.

7.3 Extended Manual Precision Evaluation

To further demonstrate the generalisability of the results we executed TCTRACER on two other projects: Commons Net[12] and Commons Text[13] and manually labelled a sample of 25 predicted links from each project as true positives or false positives. These links were produced by the combined technique with simple combination, as RQ5 shows this is the most effective technique and were selected uniformly at random. The links were then independently judged by two judges and the inter-rater agreement was computed using Fleiss' kappa. Once the inter-rater agreement over the original ratings had been computed the judges conferred to resolve differing judgements leaving one canonical set of judgements from which the precision could be calculated.

The results, presented in Table 11, show a very large difference between the two projects with Commons Text performing very well with 88% precision while Commons Net lags behind with only 20%. There are several reasons for this. Firstly, Commons Net contains a sizeable number of empty tests and abstract functions. TCTRACER does not currently filter these out and where one of these empty tests or abstract functions were predicted in a link, that link was necessarily a false positive. This issue is easily resolvable by simply filtering out those artefacts. Another contributing factor is the number of classes in Commons Net that have very similar names due to them implementing the same logic for different protocols. For example, *UnixFTPEntryParser*, *VMSFTPEntryParser*, *NTFTPEntryParser*, *OS2FTPEntryParser*, *OS400FTPEntryParser*, and others have very similar names, making false positives more likely. In projects where this regularly occurs, it may be possible to somewhat negate this effect by setting the thresholds more strictly to reduce the number of false positives.

7.4 Parameter Value Selection

Our approach includes tunable parameters; a threshold value for each technique and the call depth discount factor, all of which are real numbers. The current values for the thresholds and the call depth discount have been established in a pre-study with smaller ground truth and a smaller set of projects. We used the pre-study to empirically determine the threshold

[12] https://commons.apache.org/proper/commons-net/
[13] https://commons.apache.org/proper/commons-text/

for each technique by starting from zero and incrementing the threshold in steps of 0.01, each time recording the precision, recall, and F1 score. We then gathered all of the results and selected thresholds that generalised well across projects for the F1 score. We, therefore, consider the current thresholds to be sufficiently general and the best performing overall. However, the score distributions can vary between projects and a practitioner may want to alter the thresholds to match to a specific project if they have a ground truth or some other heuristic on which to base this decision. In the absence of a mechanism to measure precision and recall on a specific project, we suggest that practitioners use the given thresholds.

We also observed that a discount factor $<= 0.5$ usually gives the highest F-score and varying the factor between 0 and 0.5 does not change the results. Increasing the factor above 0.5 has only a small effect on recall and a larger negative effect on precision, lowering the F-score overall. Given these results, we selected a final discount factor of 0.5.

7.5 Call Depth Discounting Analysis

As discussed in Section 4.3.1, we incorporate the principal of call depth discounting into our approach as it encodes the assumption that the further away in the call stack a function is from its calling test, the less likely it is that the function is tested by the test. Using our evaluation, we can determine the accuracy of this assumption by looking at the depth of the known tested functions in our ground truth links. This data, as shown in Table 12, reveals that Commons IO is the only project which has more than one tested function that is not called directly by the test and, therefore, has a depth greater than zero. These results support the assumption behind call depth discounting in general. However, as Commons IO has a relatively large number of such examples, it shows the quality of this assumption can vary between projects. We, therefore, also wanted to assess how well our approach handles tested functions that have a depth greater than zero when they do occur. Table 12 shows the number of ground truth links at each depth that we discover using the combined score and shows that our approach handles tested functions at depth one and two very well as we correctly identify seven out of the eight tested functions at these depths. We do not discover the few tested functions at depth three and four as the scores are so heavily discounted at this level that we very rarely make a prediction at those depths. In general, this is the correct approach as the few counter-examples at depth three and four are outliers in the way the tests are implemented and are not representative of tests in general. A breakdown of the number of functions we predict to be the tested function at each depth is presented in Table 13 and the numbers shown here are broadly in line with the distribution of tested functions over depths in the ground truth data where the two projects that have tested functions at a depth greater than zero are the two projects that have a lower proportion of predicted functions at depth zero.

Table 12 Call depths of true positive functions (found versus total)

Project	Depth 0	Depth 1	Depth 2	Depth 3	Depth 4
Commons IO	28/31	5/6	1/1	0/1	0/2
Commons Lang	66/77	1/1	0/0	0/0	0/0
JFreeChart	35/44	0/0	0/0	0/0	0/0
Gson	46/55	0/0	0/0	0/0	0/0

Table 13 Total predicted function call depths

Project	Depth 0	Depth 1	Depth 2	Depth 3	Depth 4
Commons IO	1554 (82%)	291 (15%)	51 (3%)	9 (0.5%)	0 (0%)
Commons Lang	4136 (83%)	757 (15%)	49 (1%)	36 (0.7%)	0 (0%)
JFreeChart	3923 (95%)	179 (4%)	10 (0.2%)	6 (0.1%)	3 (0.07%)
Gson	1602 (93%)	112 (7%)	6 (0.3%)	1 (0.006%)	0 (0%)

8 Discussion

The results reveal some insights that allow us to draw conclusions about the relative effectiveness of the techniques and differences between the projects, weighting, and combination techniques.

8.1 Techniques

First, we compare the naming conventions techniques, NC and NCC. At the method level, NC has perfect precision for the dynamic variant and very high precision for the static variant. This is expected as it is unlikely that a test and function will share the same name, after the word *test* has been removed, without being linked. However, this strictness results in generally low recall for NC. In contrast, NCC trades-off some of this precision for more recall, resulting in better F1 scores for the NCC variants. However, due to their overall low recall on the method level, NC and NCC are unsuitable at that level. On method-level, other naming conventions are often followed. This observation was also made by Madeja and Porubán (2019). At the class level, NC beats NCC for F1 score as it is easier for developers to maintain traditional naming conventions at this level compared to the method level and, therefore, recall does not suffer as much with NC at the class level.

When comparing LCS-U and LCS-B, we see that LCS-U usually performs better for precision, whereas LCS-B generally performs better for recall. This could be explained by the interaction between the distribution of the scores and the way the thresholds are selected. As the two techniques produce difference score distributions and the thresholds are optimised for F1 score, the natural point at which the F1 is maximised may differ for the two techniques, with the optimal F1 being reached at a high recall/low precision point for LCS-B and a high precision/low recall point for LCS-U. This intuitively matches with the thresholds that were chosen by this process: 0.55 for LCS-B and 0.75 for LCS-U as higher threshold values are typically associated with higher precision. However, as the difference in precision is greater than the difference in recall, LCS-U is the better choice overall as evidenced by its better F1 score.

LCBA performs poorly in general but is especially bad for Commons IO in RQ1. This is an artefact of the nature of Commons IO, where the effect of many function calls is to change some state, rather than return the result of a computation. Therefore, the returns of method calls are not as frequently testable by simply comparing the return value to an oracle; instead, a further function call is required to check that the state was changed correctly. This causes many false positives for LCBA. Commons Lang is the opposite of Commons IO in this regard, as tested functions usually have their return values checked against oracles immediately after returning, resulting in a relatively high LCBA score. The differences in

scores for LCBA are due to the simplicity of the LCBA heuristic which cannot distinguish between acting and asserting methods (when the arrange–act–assert pattern is used).

Overall, LCS-U, Levenshtein, and TFIDF are the most consistently well-performing from the set of individual techniques, but which one performs best is project dependent. However, the results from RQ5 show that our simple combined score is consistently better than any individual technique for MAP, F1, and AUC at the method level. These results confirm our intuition that the benefit gained from combining the individual strengths of the techniques outweighs the negative effects of combining their weaknesses, thus giving a better result overall. Due to the strengths of the static techniques at the class level, the combined score is not always the best here for F1, but it is the best on average for AUC, indicating that its performance could be improved with a more optimised threshold.

8.2 Static vs. Dynamic Techniques

When comparing dynamic and static techniques, a mixed picture emerges due to the large differences in the method level results and class level results. At the method level, the dynamic techniques outperform the static techniques by a large margin, however, at the class level, the static techniques often perform marginally better than the dynamic techniques. This is due to the complexity of the task. At the method level, we have to match both the test class to the tested class and the test to the function, whereas at the class level we only have to do the former. This makes the method level task significantly more difficult, especially when using naming based techniques where we have issues relating to the reuse of function names across multiple classes or many classes/functions being named similar things. The dynamic techniques perform better as we are only considering executed functions as candidates and, therefore, we find less false positives. At the class level, the problem is much simpler and good naming conventions are more strictly adhered to as its is relatively straightforward to name a test class similar to the tested class. The static techniques, therefore, do not suffer the same precision loss as at the method level and can pick up slightly more recall, resulting in marginally better F1 scores overall. This is an important observation as it shows that, if only class level traceability is required, the static techniques alone are sufficient and there is no reason to go to the extra effort of using dynamic information. However, if method level traceability is required, the use of static name-based techniques alone will likely not be sufficiently accurate.

8.3 Multi-Level Information

In terms of the usefulness of multilevel information, RQ3 shows that using method level information for class level traceability produces worse results than just using class level information. RQ4 shows that method level traceability performance may be improved when augmenting with class level information, such as in the case of Commons IO where the largest F1 score is improved by three points. However, on average there is no significant improvement. As a result of the augmentation changing the distribution of scores, some of the techniques, most notably NC and NCC, change their characteristics with regards to how they balance precision and recall. However, this does not confer an overall benefit for F1 score. These results are in contrast to our previous work (White et al. 2020) where we showed that using multilevel information at the method level has a positive impact on results. This work shows that adding static techniques removes this benefit, however, it produces better results using only method level information than the previous work.

8.4 Weighted Technique Combinations

In RQ5, our technique exclusion study results confirm that all techniques contribute some useful information, with the possible exception of NC, which has no effect on the results obtained over our subject projects. One possible explanation for this is that NCC also provides all the information provided by NC, which is, therefore, redundant. However, despite this possibility, our advice to practitioners would be to include the technique as it does not incur significant cost in time or space complexity and may still provide useful information on projects outside of our set of experimental subjects.

With regards to the weighting experiments, the results seem unintuitive as weighting better techniques more should benefit the combined score. However, we selected precision as our weighting mechanic because it was a simple and intuitive way of testing the value of weighting as a concept and it is very possible this approach may not be the best way of weighting techniques. There is also the issue of thresholds. As the threshold was set based on the unweighted combination, we cannot be sure the threshold is still optimised when changing weights, as weighting changes the overall distribution of the scores. This means that to extensively search for weights, optimal thresholds would have to be recomputed every time any of the weights are changed, adding complexity. Overall, our experiments showed that weighting techniques do not seem to be beneficial.

Our experiments assessing the possibility of using neural networks to perform the technique combination show disappointing results for the performance of our neural network. However, the model we used was extremely simple as we wanted to ensure it was fast to implement and train and would provide a baseline for comparison with our standard combination approach. Therefore, it's likely the results in this work are not representative of the maximum achievable using a machine learning combination method, as it is entirely possible that larger more complex models would perform better. Also, as mentioned in Section 7.2.5, the manual creation of a labelled data set large enough to train a network on is not feasible and, therefore, we had to make some assumptions and use technique scores to label additional true positives and true negatives. This approach to creating a data set is not perfect and some noise or bias may be introduced into the data set.

8.5 Interpretation

Our investigations into the causes of false negatives and false positives show that the majority of these errors occur when our fundamental assumptions about the relationships between tests and their tested functions are subverted. These assumptions include the idea that test and tested function names should in some way be similar, tests should execute their tested functions relatively frequently, and tested functions should be high up in the call stack. These types of assumptions help us craft our techniques and achieve good performance, however, as shown by this analysis, there will always be examples where these assumptions do not hold and TCTRACER produces a false negative or false positive. Some of these assumptions can be tested, such as in the case of call depth discounting, as discussed in Section 7.5.

Finally, we gain some additional insights into the differences between subjects by utilising the two categories of techniques, naming-based and statistical call-based techniques (SCTs), to provide a new interpretation of the results: We use the naming-based techniques as a proxy for how well organised the test suites are, the SCTs at the method level as a proxy for how coherent the tests are, and the SCTs at the class level as a proxy for how cohesive the test classes are. This interpretation of the naming-based techniques flows from

the intuition that the better test suites are organised, such as by maintaining simple one-to-one relationships between tests and units-under-test, the better the naming techniques will perform. For the SCTs, this interpretation comes from the fact that they are measures of how many different units-under-test are called by an individual test unit and, thus, serve as a proxy for method level coherence and class level cohesiveness. Using this interpretation, we see that Commons Lang is the best organised and most coherent at the method level, while Commons IO is the best organised and most cohesive at the class level. Commons Lang scores poorly for the SCTs at the class level because some of its test classes are large and contain many tests. Therefore, these test classes have lots of calls to non-tested classes, introducing noise.

8.6 Comparison with Earlier Work

We attempted to compare our results to results from previous work. However, the only two previous works on method level (Bouillon et al. 2007; Hurdugaci and Zaidman 2012) suggest all called methods in a test, leading to very low precision. On the class level, we can compare our results as we have (in part) reimplemented suggested approaches, namely NC and LCBA. Our results are similar to Van Rompaey and Demeyer (2009), but direct comparison is not possible as their ground truth is not available. Moreover, their techniques do not provide any ranking over recommended links. They also evaluate combined techniques, but as their ground truth has 100% precision and recall for NC, all combinations result in lower accuracy. In comparison, our results show that a combination of techniques outperforms individual techniques.

Previous work that is based on similarity between tests and units-under-test (Kicsi et al. 2018; Csuvik et al. 2019a, b) use the NC results as a ground truth and therefore cannot be directly compared to our study, however, their precision and recall values are lower than the ones from our class-level combined approach.

8.7 Traceability Integration

As shown in Fig. 1, our approach can be easily integrated into the software development process. TCAGENT is injected into the JUnit framework to collect the necessary data which is then analysed by TCTRACER at the end of a JUnit run to generate the test-to-code traceability links which are ready to be used. TCAGENT and TCTRACER can be used inside the IDE via a framework like EzUnit (Bouillon et al. 2007), allowing a developer to navigate between tests and tested code quickly. TCTRACER is also easy to integrate into a standard continuous integration process (Shahin et al. 2017; Elsner et al. 2021). This integration is made simple by the fact that TCAGENT instruments the JUnit test suite and, therefore, the gathering of dynamic trace information happens automatically during the testing stage. All that remains is to add an extra step that executes TCTRACER. The addition of this step is easy in most modern continuous integration frameworks such as Travis CI[14] and Jenkins[15]. The gathered traceability links can then be used to backtrack from executed tests to the tested code or vice versa. Moreover, the traceability links are constantly kept up-to-date as part of the continuous integration pipeline and are readily available. For example,

[14] https://travis-ci.org/
[15] https://jenkins.io/

a developer can change code and corresponding tests at the same time, ensuring their co-evolution. Also, further analysis of the produced links can be performed as part of the continuous integration process, such as automatically alerting developers when a function has no tests even if it is covered (executed) during testing or identifying tests affected by a change for regression test optimisation (Elsner et al. 2021). Therefore, using TCTRACER to automate test-to-code traceability link capture through continuous integration can provide multiple benefits and could be especially useful in safety-critical systems that are subject to regulations requiring that traceability links are maintained (Cleland-Huang 2012).

8.8 Unit vs. Integration Testing

A further point of discussion is how TCTRACER interacts with integration tests and what the differences are between using TCTRACER for unit tests and using it for integration tests. Our approach targets traceability for unit tests we excluded some obvious integration tests from our evaluation as discussed in Section 7.1. As Trautsch et al. (2020) concludes, there is no longer a clear distinction between unit testing and integration testing in modern software testing, and the JUnit framework is often used for both. Interestingly Orellana et al. (2017) use naming conventions to distinguish unit and integration tests and Trautsch et al. (2020) use coverage information for the same purpose, thus utilising techniques which are similar to those we evaluate. Orellana et al. (2017) did find a difference between unit and integration tests with regards to the time and developer coordination needed to fix them but the findings were unintuitive as they found that unit tests took more time and coordination to fix than integration tests. Given this, it may not be easy to clearly define the differences one may find when using TCTRACER for integration tests versus unit tests. However, if we accept the fundamental assumption that integration tests test more units than unit tests and may not have as close a relationship to them, for example, not be as easily matched by name similarity, intuitively, TCTRACER may struggle to work with the same level of precision. However, this is merely conjecture and would need to be validated with experimental evidence but we are not aware of a ground truth that would allow this.

8.9 Takeaway Messages

The first key takeaway message is to use the combined score at both the method and the class levels as it is the most consistent and performs the best in the majority of cases. Secondly, we selected our thresholds for good generalisability so they should be sufficient in the general case but if a ground truth is available for the project under analysis, practitioners can tune the thresholds to their specific project. The final takeaway is that, at the class level, static (name-based) techniques alone are sufficient as adding dynamic techniques confers no benefit.

Method level traceability has recently become important for approaches that generate assert statements. However, current approaches use simple approaches with low precision or recall that may affect the quality of the recommendations. For example, Watson et al. (2020) uses a static version of LCBA which in our evaluation only achieved 63% precision and 62% recall. Another approach (Villmow et al. 2021) uses a name similarity based approach where from the called methods of a test method the most similar name is assumed to be the tested method. The authors report 94% precision over a random sample, but their approach was only able to identify a tested method for 36% of tests (an upper limit for recall). As our approach achieves significantly higher precision and recall, it has the potential to improve recommendation approaches like ATLAS or CONTEST significantly.

9 Threats to Validity

This section describes the main external and internal threats to validity.

9.1 External Threats

The main threats to validity come from the subjects and the ground truth. Firstly, the representativeness of the subjects is an external threat to validity as we have no clear evidence as to how representative these subjects are of the general population of software projects. However, the subjects that we have selected are widely used in research and by practitioners and are large enough to demonstrate the applicability of our approach to non-trivial systems.

Name-based techniques rely on projects following some form of naming convention. NC is a very simple but often used heuristic to establish links. It is based on the requirement to have the "test" prefix to identify tests in JUnit 3. However, experience and our results show that while NC on class-level performs well, it does not do so well on method-level where often other naming conventions are used (see Madeja and Porubån (2019)).

While our paper targets unit testing, JUnit is used for unit and integration testing and therefore our evaluation includes both, unit and integration tests. The presence of integration tests can be a challenge for traceability techniques as discussed in Section 7.1. It would be interesting to separate integration tests and unit tests in our evaluation, however, Trautsch et al. (2020) observes that the current definitions of unit and integration tests may need to be reconsidered.

Finally, there is a threat to generalisability as our experiments only cover Java projects that use the JUnit framework and we do not know how representative our chosen projects are. Therefore, we do not have direct evidence that this approach would apply to other languages or testing frameworks. However, in our estimation, there is nothing inherent in our approach that would prevent the application of the TCTracer approach in other scenarios.

9.2 Internal Threats

The use of manual investigation for establishing the ground truth poses an internal threat to validity as there is room for interpretation when determining which functions or classes are tested by a test or test class. However, all judgements were validated by more than one judge. For the method level ground truth, three judges were used and a full inter-rater agreement was achieved. All of the judges were students which may have introduced some bias but despite sharing the student status, the judges were from varied backgrounds with significant previous experience and there is no clear reason to believe their judgements on which tests test which functions would be different to that of an average developer. Additionally, as there was a meeting to discuss differences after each judge had independently made their judgements, the process was not entirely independent. However, the number of differences was small and the minimal changes enacted in the meeting were the result of fixing mistakes rather than convincing judges to change their judgements. At the class level, the majority of links were provided by the developers and verified by a judge, and a small number of links (12) were created by two judges, again with a full inter-rater agreement. As we are using some developer created links, there is potential for a bias to be introduced due to the selection of classes that were annotated by the developers. While a manual inspection does not reveal any obvious bias, the existence of one cannot be ruled out.

As with any approach that uses thresholds, the results are based on the chosen values for the thresholds. While we attempted to choose good general thresholds, different thresholds may lead to different results, observations, and conclusions.

9.3 Ethics

The ethics of analysing the subject systems and the extraction of traceability links have been considered and are in line with the ethics of mining software repositories (Gold and Krinke 2020, 2022). The work presented in this paper was performed in line with research ethics at UCL (UCL Research Ethics Committee 2020).

10 Related Work

In this section we discuss additional related work exploring techniques and research areas that do not form part of the background, as presented in Section 2 but are also of interest to this work, provide opportunities for future work, or have cross-over with the techniques.

Gergely et al. (2019) do not extract links between units directly, but instead, use clustering. The clustering is done with static (packaging structure) and dynamic (coverage) analysis. The two sets of traceability clusters are compared and the differences are manually analysed to produce the final traceability links.

Ståhl et al. (2017) focuses on the deployment of traceability into continuous integration and delivery systems. As part of this work, they present an investigation into existing needs and practices and propose a unified framework for integrating traceability establishment into continuous integration systems. The investigation into existing practices showed that there is a strong desire among developers for the integration of automated traceability handing into build systems which is, in large part, currently not being fulfilled. This demonstrates the demand for tools such as TCTracer. In a related work, Elsner et al. (2021) have used a subset of the techniques we presented in their evaluation of regression test optimisation approaches in a continuous integration setting. This is interesting as it demonstrates that the types of techniques we have developed for traceability link establishment have applications in other use cases.

Soetens et al. (2016) uses static and dynamic method invocations for determining which tests need to be included in a regression test case run. This problem is similar to that of traceability establishment and they experimented with some existing traceability techniques in previous work. The TCTracer approach could, therefore, also improve over these existing techniques when utilised for the regression test selection use case. Conversely, the techniques developed by Soetens et al. (2016) could be recast as traceability recovery techniques and evaluated for that use case.

A recent work (Aljawabrah et al. 2021) has also explored the visualisation of traceability links as a way of assisting developers to utilise them and providing the ability to see the difference in predicted links between techniques. This further demonstrates the potential applicability of test-to-code traceability links and the appetite for their usage.

11 Conclusion

In this paper, we have presented TCTracer, an approach and implementation for establishing test-to-code traceability links at both the method level and class level. TCTracer

utilises a wide range of new and existing test-to-code traceability link establishment techniques using dynamic and static information and enhances them by combining them and applying them to both the method level and class level. This makes TCTRACER the first approach that establishes two types of links and utilises a cross-level information flow. An empirical evaluation of TCTRACER was conducted, at both the method level and class level, with five real-world open source projects. The results show that, on average, TCTRACER is more effective at both the method level and the class level than any single existing technique and at the class level only static information is required to achieve the best performance. This makes TCTRACER the most effective approach for test-to-code traceability to date.

Acknowledgements This This work was funded through the Engineering and Physical Sciences Research Council (EPSRC), award number EP/N509577/1.

References

Aljawabrah N, Gergely T, Misra S, Fernandez-Sanz L (2021) Automated recovery and visualization of test-to-code traceability (TCT) links: an evaluation. IEEE Access 9:40111–40123

Allamanis M, Barr ET, Devanbu P, Sutton C (2018) A survey of machine learning for big code and naturalness. ACM Comput Surv (CSUR) 51(4):81

Antoniol G, Canfora G, Casazza G, De Lucia A, Merlo E (2002) Recovering traceability links between code and documentation. IEEE Trans Softw Eng 28(10):970–983

Bouillon P, Krinke J, Meyer N, Steimann F (2007) EZUNIT: A framework for associating failed unit tests with potential programming errors. In: Agile Processes in Software Engineering and Extreme Programming. Springer. https://doi.org/10.1007/978-3-540-73101-6_14

Cleland-Huang J (2012) Traceability in agile projects. In: Software and Systems Traceability. Springer, pp 265–275

Csuvik V, Kicsi A, Vidács L (2019a) Evaluation of textual similarity techniques in code level traceability. In: International Conference on Computational Science and Its Applications. Springer, pp 529–543

Csuvik V, Kicsi A, Vidács L (2019b) Source code level word embeddings in aiding semantic test-to-code traceability. In: 10th International Workshop on Software and Systems Traceability, pp 29–36. https://doi.org/10.1109/SST.2019.00016

Davis J, Goadrich M (2006) The relationship between precision-recall and ROC curves. In: Proceedings of the 23rd International Conference on Machine Learning. ACM, pp 233–240

De Lucia A, Fasano F, Oliveto R (2008) Traceability management for impact analysis. In: 2008 Frontiers of Software Maintenance. IEEE, pp 21–30

Elsner D, Hauer F, Pretschner A, Reimer S (2021) Empirically evaluating readily available information for regression test optimization in continuous integration. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp 491–504

Gergely T, Balogh G, Horváth F, Vancsics B, Beszédes Á, Gyimóthy T (2019) Differences between a static and a dynamic test-to-code traceability recovery method. Softw Qual J 27(2):797–822

Gethers M, Oliveto R, Poshyvanyk D, De Lucia A (2011) On integrating orthogonal information retrieval methods to improve traceability recovery. In: 27th IEEE International Conference on Software Maintenance (ICSM), pp 133–142

Ghafari M, Ghezzi C, Rubinov K (2015) Automatically identifying focal methods under test in unit test cases. In: 15Th international working conference on source code analysis and manipulation (SCAM). IEEE, pp 61–70. https://doi.org/10.1109/SCAM.2015.7335402

Gold NE, Krinke J (2020) Ethical mining: A case study on MSR mining challenges. In: Proceedings of the 17th International Conference on Mining Software Repositories. ACM, pp 265–276. https://doi.org/10.1145/3379597.3387462
Gold NE, Krinke J (2022) Ethics in the mining of software repositories. Empirical Software Engineering. 27(17) https://doi.org/10.1007/s10664-021-10057-7
Hurdugaci V, Zaidman A (2012) Aiding software developers to maintain developer tests. In: 16th European Conference on Software Maintenance and Reengineering. IEEE, pp 11–20
Jones JA, Harrold MJ, Stasko J (2002) Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering (ICSE). IEEE, pp 467–477
Kicsi A, Tóth L, Vidács L (2018) Exploring the benefits of utilizing conceptual information in test-to-code traceability. In: 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, pp 8–14. https://doi.org/10.1145/3194104.3194106
Kicsi A, Vidács L, Gyimóthy T (2020) TestRoutes. In: Proceedings of the 17th International Conference on Mining Software Repositories. ACM, pp 593–597. https://doi.org/10.1145/3379597.3387488
Levenshtein VI (1966) Binary codes capable of correcting deletions, insertions, and reversals. Sov Phys Doklady 10(8):707–710
Madeja M, Porubán J (2019) Tracing naming semantics in unit tests of popular GitHub Android projects. In: 8th Symposium on Languages, Applications and Technologies, (SLATE 2019). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/OASIcs.SLATE.2019.3
Manning C, Raghavan P, Schütze H (2010) Introduction to information retrieval. Nat Lang Eng 16(1):100–103
Meimandi Parizi R, Kasem A, Abdullah A (2015) Towards gamification in software traceability: Between test and code artifacts. In: Proceedings of the 10th International Conference on Software Engineering and Applications. SCITEPRESS, pp 393–400, https://doi.org/10.5220/0005555503930400
Orellana G, Laghari G, Murgia A, Demeyer S (2017) On the differences between unit and integration testing in the TravisTorrent dataset. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp 451–454. https://doi.org/10.1109/MSR.2017.25
Parizi RM (2016) On the gamification of human-centric traceability tasks in software testing and coding. In: IEEE 14th international conference on software engineering research, management and applications (SERA). IEEE, pp 193–200. https://doi.org/10.1109/SERA.2016.7516146
Parizi RM, Lee SP, Dabbagh M (2014) Achievements and challenges in state-of-the-art software traceability between test and code artifacts. IEEE Trans Reliab 63(4):913–926. https://doi.org/10.1109/TR.2014.2338254
Pinto LS, Sinha S, Orso A (2012) Understanding myths and realities of test-suite evolution. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE), vol 1. ACM, p 1. https://doi.org/10.1145/2393596.2393634
Qusef A, Bavota G, Oliveto R, Lucia AD, Binkley D (2013) Evaluating test-to-code traceability recovery methods through controlled experiments. J Softw Evol Process 25(11):1167–1191. https://doi.org/10.1002/smr.1573
Qusef A, Bavota G, Oliveto R, Lucia AD, Binkley D (2014) Recovering test-to-code traceability using slicing and textual analysis. J Syst Softw 88:147–168
Shahin M, Babar MA, Zhu L (2017) Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. IEEE Access 5:3909–3943
Soetens QD, Demeyer S, Zaidman A, Pérez J (2016) Change-based test selection: an empirical evaluation. Empir Softw Eng 21(5):1990–2032. https://doi.org/10.1007/s10664-015-9405-5
Ståhl D, Hallén K, Bosch J (2017) Achieving traceability in large scale continuous integration and delivery deployment, usage and validation of the Eiffel framework. Empir Softw Eng 22(3):967–995
Trautsch F, Herbold S, Grabowski J (2020) Are unit and integration test definitions still valid for modern Java projects? an empirical study on open-source projects. J Syst Softw 159:110421
UCL Research Ethics Committee (2020) Research Ethics at UCL. https://ethics.grad.ucl.ac.uk/
Van Rompaey B, Demeyer S (2009) Establishing traceability links between unit test cases and units under test. In: 13th European Conference on Software Maintenance and Reengineering. IEEE, pp 209–218
Villmow J, Depoix J, Ulges A (2021) CONTEST: A unit test completion benchmark featuring context. In: The First Workshop on Natural Language Processing for Programming, pp 17–25
Watson C, Tufano M, Moran K, Bavota G, Poshyvanyk D (2020) On learning meaningful assert statements for unit test cases. In: 42Nd international conference on software engineering (ICSE), Seoul. https://doi.org/10.1145/3377811.3380429
White R, Krinke J (2018) TestNMT: Function-to-test neural machine translation. In: Proceedings of the 4th ACM SIGSOFT International Workshop on NLP for Software Engineering. ACM, NL4SE, pp 30–33. https://doi.org/10.1145/3283812.3283823

White R, Krinke J (2020) ReAssert: Deep learning for assert generation. arXiv:201109784
White R, Krinke J (2021) TCTracer: Establishing test-to-code traceability links using dynamic and static techniques – evaluation data. https://doi.org/10.5281/zenodo.5206476
White R, Krinke J, Tan R (2020) Establishing multilevel test-to-code traceability links In: 42nd International Conference on Software Engineering (ICSE). ACM. https://doi.org/10.1145/3377811.3380921
Winkler S, von Pilgrim J (2010) A survey of traceability in requirements engineering and model-driven development. Softw Syst Model 9(4):529–565
Zaidman A, Van Rompaey B, van Deursen A, Demeyer S (2011) Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. Empir Softw Eng 16(3):325–364. https://doi.org/10.1007/s10664-010-9143-7