Establishing Multilevel Test-to-Code Traceability Links

Robert White
University College London
London, UK

Jens Krinke
University College London
London, UK

Raymond Tan
University College London
London, UK

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 and exploiting a synergistic flow of information between the method and class levels. An evaluation of TCtracer using four large, well-studied open source systems demonstrates that, on average, we can establish test-to-function links with a mean average precision (MAP) of 78% and test-class-to-class links with an MAP of 93%.

ACM Reference Format:
Robert White, Jens Krinke, and Raymond Tan. 2020. Establishing Multilevel Test-to-Code Traceability Links. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380921

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 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 [2, 9, 34]. 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 reduces the risk of desynchronisation between the tests and code, an issue which 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. Industrial need for the automated establishment of test-to-code traceability links is demonstrated by Ståhl et al. [31] 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.

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.

Most previous work on test-to-code traceability (see Parizi et al. [27] for an overview) has focused on the class level, where test classes are linked to their tested classes [6, 7, 15, 20, 29, 32]. Not much work has been done on the method level [3, 16, 17], where individual unit tests are linked to their tested functions, despite being shown to be helpful for developers [17]. 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 dynamic information and ranking potential links, instead of the static information that has typically been used before to generate sets of (unranked) 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 which 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 [17] is not an accurate technique of establishing test-to-code traceability links.

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380921

In this paper, we present TCtracer, an approach and implementation 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, including two new call-based statistical techniques.

TCtracer also combines these techniques to produce a single score that performs better overall than any individual technique. In addition, 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 [33], at both the method and class levels, from four non-trivial and well-studied subject projects[1]. Our findings show that, on average, using our combined technique, we can achieve an increase in effectiveness over existing techniques at both the method level and class level. The main contributions of this paper are:

- An approach to test-to-code traceability that utilises an ensemble of techniques and a multilevel flow of information.
- A comparative evaluation of each technique at both the method and class levels.
- An evaluation of the benefit gained by utilising multilevel information.
- A manually curated ground truth dataset [33] of test-to-function and test-class-to-class links.

2 MOTIVATION

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). 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 [23, 32]. 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 [11] 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-two relationship, the names do not match the naming convention and NC would not be able to recover these links.

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 getter must be called to get the changed state so that it can be compared to an oracle. This causes LCBA to incorrectly identify the getter 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 [2, 6]. However, none of them have been shown to be 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 [1].

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 observes which artefacts are executed while a test runs, creating candidate links between test artefacts and tested artefacts. It then assigns scores to the candidate links. 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 test and the tested artefacts.

As we are establishing links on the method- 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 each function call 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. For each test, this information is stored in an object, henceforth referred to as a hit spectrum. We then apply an ensemble of traceability techniques to the method level, using the information in the collected hit spectra. 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

[1]Evaluation artefacts available at github.com/RRGWhite/icse20-main-1069-data.

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 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. A ninth score is also computed that combines the scores for all the individual techniques. 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.

3.1 Techniques

As discussed in Section 2, existing test-to-code traceability techniques have weaknesses which we try to overcome with new techniques. Despite their weaknesses, we selected two established techniques, Naming Conventions (NC) [32] and Last Call Before Assert (LCBA) [32] 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 [21], which all utilise name similarity. Two statistical call-based techniques (SCTs) based on Tarantula fault localisation [19] and Term Frequency–Inverse Document Frequency (TFIDF) [24] are also included in the new techniques.

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 [17, 23]. 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 [6, 23, 28, 29], 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.

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 function and 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 in a similar fashion 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 eight 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. However, this is currently merely an intuition and will be tested in future work by measuring the contribution of each technique to the effectiveness of the combined score.

All of our techniques utilise dynamic trace information which allows us to avoid common problems associated with static techniques, such as over-approximation and the inability to reason about references and dependencies that are resolved at run-time.

We have discarded a series of other techniques. Fixture Element Types (FET) [32] and SCOTCH+ [29] cannot be applied on method-level and Static Call Graph (SCG), Lexical Analysis (LA), and Co-Evolution (Co-Ev) have been discarded because of their low precision and recall [20, 27, 32].

3.1.1 *Naming Conventions.*

As naming conventions can change between projects [32], 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:

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

3.1.2 *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:

$$\text{score}(t, f) = \begin{cases} 1, & \text{if } f \text{ is last return before an assert in } t \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

3.1.3 *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:

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

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.

$$\text{score}(t, f) = \frac{|\text{LCS}(n_t, n_f)|}{|n_f|} \tag{5}$$

Levenshtein Distance. The Levenshtein distance [21], 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:

$$\text{score}(t, f) = 1 - \left(\frac{\text{Levenshtein}(n_t, n_f)}{\max(|n_t|, |n_f|)} \right) \tag{6}$$

3.1.4 *Tarantula.*

Tarantula [19] 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:

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

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[2]. Using this assumption we can derive our traceability score equation from Equation 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.

3.1.5 *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

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

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:

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

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

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

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

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.

3.2 Score Scaling

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

3.2.1 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:

$$\mathrm{score_d}(t, f) = \mathrm{score}(t, f) \cdot \gamma^{(dist(t,f)-1)} \tag{12}$$

Where $\mathrm{score_d}$ 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.

3.2.2 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 the different tests. Therefore, we normalise the scores so that the largest score within a test is 1:

$$\mathrm{score_n}(t, f) = \frac{\mathrm{score_d}(t, f)}{\max(\{\mathrm{score_d}(t, f') \mid f' \in t\})} \tag{13}$$

Where $\mathrm{score_n}$ is the normalised score. Normalisation allows us to define a threshold around the top-ranked link.

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

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	0.80
LCS – Both (LCS-B)	[0, 1]	Yes	0.45
Levenshtein (Leven)	[0, 1]	Yes	0.35
Last Call Before Assert (LCBA)	0 or 1	–	–
Tarantula	[0, 1]	Yes	0.95
TFIDF	[0, 1]	Yes	0.90
Combined	[0, 1]	Yes	0.80

3.3 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.

3.3.1 Method-Level Prediction. The process starts by executing each of our nine individual traceability techniques at the method level, resulting in a matrix of scores for each technique:

$$\mathbf{M} \in \mathbb{R}^{|\mathrm{T}|\times|\mathrm{F}|} \tag{14}$$

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 Equation 13.

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. Equation 15 defines how each set of method level traceability links are constructed.

$$\mathrm{LM} = \{(t, f) \in \mathrm{T} \times \mathrm{F} \mid \mathbf{M}_{tf} \geq \tau\} \tag{15}$$

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

3.3.2 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:

$$\mathbf{C} \in \mathbb{R}^{|\mathrm{TC}|\times|\mathrm{FC}|} \tag{16}$$

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 $(c_t, c_f) \in (\mathrm{TC} \times \mathrm{FC})$.

In a similar fashion to the method level, $\mathbf{C}$ is used to compute sets of class level traceability links using Equation 17.

$$\mathrm{LC} = \{(c_t, c_f) \in \mathrm{TC} \times \mathrm{FC} \mid \mathbf{C}_{c_t c_f} \geq \tau\} \tag{17}$$

3.3.3 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 **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:

$$\mathbf{EM}(c_t, c_f) \in \mathbb{R}^{|\mathrm{t}(c_t)| \times |\mathrm{f}(c_f)|} \tag{18}$$

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:

$$\mathbf{EM} \in \mathbb{R}^{|\mathrm{TC}| \times |\mathrm{FC}|} \tag{19}$$

Therefore, each element of **EM** is the score for a given test-class-to-class pair $(c_t, c_f) \in (\mathrm{TC} \times \mathrm{FC})$ that is derived from method level information. All rows in **EM** are normalised using Equation 13.

The scores in **EM** are then used to produce a set of class level predicted links using Equation 20.

$$\mathrm{LEM} = \{(c_t, c_f) \in \mathrm{TC} \times \mathrm{FC} \mid \mathbf{EM}_{c_t c_f} \geq \tau\} \tag{20}$$

3.3.4 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 $\mathbf{AM} \in \mathbb{R}^{|\mathrm{T}| \times |\mathrm{F}|}$. 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 Equation 21.

$$\mathbf{AM}_{tf} = \mathbf{M}_{tf} \cdot \mathbf{C}_{\mathrm{c}(t)\mathrm{c}(f)} \tag{21}$$

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

From **AM**, the set of augmented method level traceability link predictions are produced using Equation 22.

$$\mathrm{LAM} = \{(t, f) \in \mathrm{T} \times \mathrm{F} \mid \mathbf{AM}_{tf} \geq \tau\} \tag{22}$$

4 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 traceability links by the techniques described in Section 3.

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 Figure 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 [35] 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 build the hit spectrum 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.

In the final phase, TCTracer computes the sets of predicted links described in Section 3 using the hit spectra, the LCBA 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.

5 EVALUATION

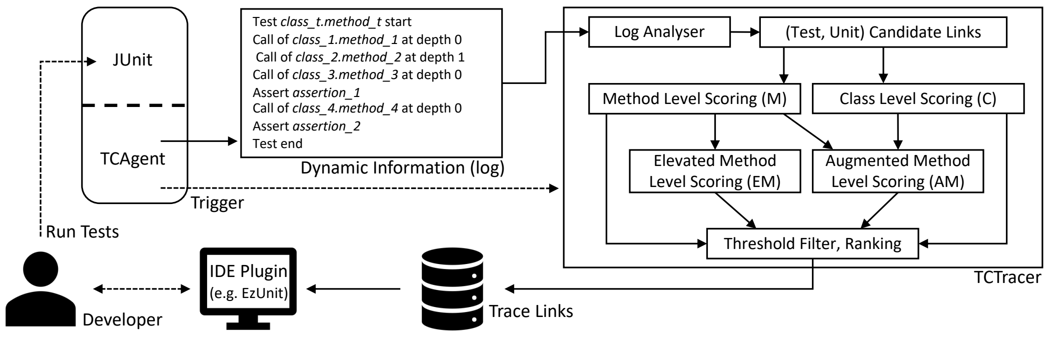
This section presents our research questions, the design of the experiments carried out to answer these questions, the results, and a discussion of the findings.

5.1 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 [12], Commons Lang [13], and JFreeChart [22]. 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. After this process, 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 total, the method level ground truth contains 138 oracle links.

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 verified by a judge. 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+ [29], 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 requires Java 8 or


JUnit
TCAgent
Run Tests
Trigger
Dynamic Information (log)
Log Analyser
(Test, Unit) Candidate Links
Method Level Scoring (M)
Class Level Scoring (C)
Elevated Method Level Scoring (EM)
Augmented Method Level Scoring (AM)
Threshold Filter, Ranking
TCTracer
Developer
IDE Plugin (e.g. EzUnit)
Trace Links

```
Test class_t.method_t start
Call of class_1.method_1 at depth 0
 Call of class_2.method_2 at depth 1
Call of class_3.method_3 at depth 0
Assert assertion_1
Call of class_4.method_4 at depth 0
Assert assertion_2
Test end
```

Figure 1: Integration of TCTRACER into JUnit.

Table 2: Subject statistics.

Project	Version	Num. of Functions	Num. of Tests	Instruction Coverage
Apache Ant	1.9.5	10477	1830	50%
Commons IO	2.5	1246	994	89%
Commons Lang	3.7	3111	3061	95%
JFreeChart	1.0.19	9053	2244	52%

newer. The only ground truth links that we were able to use were for Apache Ant [10] 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. Other previous work [6] 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. Information about the subjects is given in Table 2.

Evaluation Measures. The evaluation measures we selected are: precision, recall, F1 score, mean average precision (MAP), and area under the precision-recall curve (AUC) [24]. 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 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 [8]. All scores are presented as integer percentages for the sake of readability.

Rompaey et al. [32] 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.

5.2 RQ1 (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 Equation 15.

Findings. From the results for RQ1, shown in Table 3, we see that, on average, the combined score is the most desirable as it performs best for MAP and AUC. This means that it is good at ranking predictions, is consistent when changing thresholds, and could benefit from a further optimised threshold selection. For precision alone, NC is the best, while LCS-B is best for recall.

5.3 RQ2 (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 Equation 17.

Findings. From the results for RQ2, shown in Table 4, we see that the results are similar to RQ1: the combined score again performs best for MAP and AUC, but this time is also joint best for recall with LCS-B. For pure precision, NC wins again.

Table 3: RQ1 – Method level traceability.

	Technique	Prec.	Recall	MAP	F1	AUC
Commons IO	NC	**100**	5	7	9	–
	NCC	86	43	44	57	–
	LCS-U	64	81	71	72	63
	LCS-B	54	**86**	69	66	54
	Leven	67	57	62	62	59
	LCBA	40	33	30	36	–
	Tarantula	53	64	64	58	48
	TFIDF	55	64	64	59	55
	Combined	69	81	**75**	**75**	**67**
Commons Lang	NC	**100**	10	17	19	–
	NCC	90	49	54	63	–
	LCS-U	78	69	79	73	76
	LCS-B	70	**79**	79	75	74
	Leven	84	53	69	65	73
	LCBA	83	68	63	75	–
	Tarantula	74	**79**	82	77	78
	TFIDF	82	74	79	78	82
	Combined	86	76	**85**	**80**	**87**
JFreeChart	NC	**100**	19	21	32	–
	NCC	**100**	30	32	46	–
	LCS-U	20	68	73	31	17
	LCS-B	33	78	76	47	58
	Leven	74	62	71	**68**	52
	LCBA	53	73	74	61	–
	Tarantula	33	**78**	**76**	47	42
	TFIDF	53	76	74	62	54
	Combined	23	70	74	35	**60**
Average	NC	**100**	11	15	20	–
	NCC	92	40	43	55	–
	LCS-U	54	73	74	59	52
	LCS-B	53	**81**	75	63	62
	Leven	75	57	67	65	61
	LCBA	59	58	55	57	–
	Tarantula	53	74	74	60	56
	TFIDF	63	71	73	**66**	64
	Combined	59	76	**78**	63	**71**

Table 4: RQ2 – Class level traceability.

	Technique	Prec.	Recall	MAP	F1	AUC
Apache Ant	NC	**100**	**89**	**92**	**94**	–
	NCC	88	88	89	88	–
	LCS-U	65	86	86	74	68
	LCS-B	51	88	81	65	77
	Leven	87	83	87	85	77
	LCBA	50	72	62	59	–
	Tarantula	49	56	58	52	40
	TFIDF	51	56	58	54	42
	Combined	83	86	89	85	82
Commons IO	NC	**100**	69	72	81	–
	NCC	98	94	96	96	–
	LCS-U	70	94	91	80	89
	LCS-B	55	96	83	70	92
	Leven	100	94	97	97	96
	LCBA	51	75	71	61	–
	Tarantula	74	81	83	77	67
	TFIDF	74	81	83	77	68
	Combined	96	**96**	**98**	**96**	**96**
Commons Lang	NC	**100**	75	82	86	–
	NCC	95	86	**93**	**91**	–
	LCS-U	77	86	90	81	78
	LCS-B	63	**88**	88	74	76
	Leven	95	86	**93**	**91**	**85**
	LCBA	51	73	70	60	–
	Tarantula	50	63	66	56	41
	TFIDF	34	60	59	44	34
	Combined	90	86	**93**	88	84
JFreeChart	NC	**100**	85	91	**92**	–
	NCC	73	**86**	84	79	–
	LCS-U	56	**86**	79	68	62
	LCS-B	58	**86**	81	69	**86**
	Leven	99	**86**	**92**	**92**	**86**
	LCBA	31	82	67	45	–
	Tarantula	69	77	77	73	66
	TFIDF	67	77	78	72	66
	Combined	99	**86**	**92**	**92**	**86**
Average	NC	**100**	80	84	88	–
	NCC	88	88	90	88	–
	LCS-U	67	88	86	76	74
	LCS-B	57	**89**	83	69	83
	Leven	95	87	92	**91**	86
	LCBA	46	75	67	56	–
	Tarantula	60	69	71	64	54
	TFIDF	57	69	69	62	53
	Combined	92	**89**	**93**	90	**87**

5.4 RQ3 (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 Equation 20.

Findings. From the results shown in Table 5 we see that TFIDF slightly beats the combined score for MAP, F1 score, and AUC. NC wins again on precision and this time LCS-B is best for recall.

5.5 RQ4 (Augmented Method Level):

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

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 select the best overall technique from the method level, the combined technique, and compare its performance to the augmented combined technique from the link sets produced using Equation 22.

Findings. The results for RQ4, shown in Table 6, show that, on average, it is better to use augmented information as precision, F1 score, and AUC are significantly better using augmented information, while recall is only slightly worse.

Table 5: RQ3 – Class level using method level information.

	Technique	Prec.	Recall	MAP	F1	AUC
Apache Ant	NC	71	31	32	43	–
	NCC	59	38	37	46	–
	LCS-U	53	69	69	60	53
	LCS-B	48	73	71	58	52
	Leven	64	64	67	64	54
	LCBA	70	70	71	70	–
	Tarantula	73	70	74	71	59
	TFIDF	**75**	**75**	**77**	**75**	**65**
	Combined	64	73	75	69	62
Commons IO	NC	86	38	39	52	–
	NCC	**90**	56	57	69	–
	LCS-U	80	83	82	82	79
	LCS-B	78	**88**	**86**	82	79
	Leven	84	79	81	82	79
	LCBA	73	63	66	67	–
	Tarantula	82	77	79	80	77
	TFIDF	81	79	80	80	78
	Combined	85	85	**86**	**85**	**80**
Commons Lang	NC	**93**	58	64	71	–
	NCC	**93**	68	74	79	–
	LCS-U	82	79	84	81	75
	LCS-B	77	**81**	84	79	73
	Leven	85	78	84	81	75
	LCBA	83	71	78	76	–
	Tarantula	82	75	81	79	71
	TFIDF	87	79	**86**	**83**	**77**
	Combined	83	79	84	81	75
JFreeChart	NC	76	63	69	69	–
	NCC	77	65	70	71	–
	LCS-U	62	75	74	68	59
	LCS-B	47	**77**	72	58	57
	Leven	70	65	69	68	58
	LCBA	81	76	**79**	78	–
	Tarantula	73	63	67	68	58
	TFIDF	**82**	76	**79**	**79**	**72**
	Combined	70	75	75	72	65
Average	NC	**82**	47	51	59	–
	NCC	80	57	60	66	–
	LCS-U	69	77	77	72	67
	LCS-B	62	**80**	78	69	65
	Leven	76	72	75	74	67
	LCBA	77	70	73	73	–
	Tarantula	77	72	75	74	66
	TFIDF	81	77	**81**	**79**	**73**
	Combined	76	78	80	77	71

5.6 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 a smaller ground truth and a smaller set of projects. Based on the pre-study, we selected thresholds that generalised well. 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.

Table 6: RQ4 – Method level using multilevel information.

Technique	Prec.	Recall	MAP	F1	AUC
Commons IO					
Unaugmented	69	**81**	75	**75**	67
Augmented	**72**	79	**79**	**75**	**74**
Commons Lang					
Unaugmented	**86**	**76**	**85**	**80**	87
Augmented	85	73	82	79	**88**
JFreeChart					
Unaugmented	23	**70**	**74**	35	60
Augmented	**65**	**70**	**74**	**68**	**68**
Average					
Unaugmented	59	**76**	**78**	63	71
Augmented	**74**	74	**78**	**74**	**77**

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.

We consider the current thresholds to be sufficiently good and general. However, one can observe that the results for a technique can vary a lot between projects and a developer may want to vary the thresholds specific to a project or rely on the ranking of candidate links instead.

5.7 Discussion

The results reveal some insights that allow us to draw conclusions about the relative effectiveness of the techniques and the differences between the projects. First, we compare the naming conventions techniques, NC and NCC. For RQ1 and RQ2, NC has perfect precision. 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 low recall for NC in RQ1 and RQ3, where NCC ends up performing significantly better for F1 score and MAP, as it trades-off a small amount of precision for much more recall. For RQ2, NC and NCC end up being even in F1 score as it is easier for developers to maintain traditional naming conventions at the class level, explaining the good recall for NC at this level. However, their low recall on the method level (RQ1) make them unsuitable for that level – an observation also made by Madeja et al. [23]. When comparing LCS-U and LCS-B, we see that LCS-U usually performs better for F1 score and MAP but is generally equivalent or worse for AUC. This suggests that either LCS-U is more sensitive to the threshold or that it may be possible to find a better-optimised threshold value for LCS-B.

LCBA performs poorly in general for RQ1 and RQ2 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 value of a computation. Therefore, the results 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.

Overall, the two LCS techniques and Levenshtein are the most consistently well-performing from the set of individual techniques, but which one performs best is project dependent. The combined score, however, is consistently strong and is the best on average for MAP and AUC. Where the combined technique struggles for F1 score, most notably for JFreeChart in RQ1, the poor performance is due to a difference in optimal thresholds between projects; the combined threshold is best at approximately 0.95 for JFreeChart and 0.80 (the selected value) for the other projects. MAP and AUC confirm this as the differential in those results is much smaller and they are the least affected by the threshold. This is true for MAP as it takes into account the rankings of the true positives, which the combined score consistently ranks highly, even if it also produces false positives at lower ranks due to a low threshold. AUC is independent of the threshold entirely. The 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. This intuition holds even in situations where specific techniques are expected to be very strong, such as naming conventions at the class level.

In terms of multilevel information, RQ4 shows that method level traceability performance is improved when augmented with class level information. The AUC results show that this improvement is significant but may require threshold optimisation to fully realise.

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.

We attempted to compare our results to results from previous work. However, the only two previous works on method level [3, 17] 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 Rompaey et al.'s [32], 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 [6, 7, 20] 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.

As shown in Figure 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 [3], allowing a developer to navigate between tests and tested code quickly. TCTRACER is also easy to integrate into a standard continuous integration process [30]. 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 [4] and Jenkins [18]. 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, a developer can change code and corresponding tests at the same time, ensuring their co-evolution. In addition, 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. 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 [5].

5.8 Threats to Validity

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. The second threat comes from the method by which the ground truth was established. 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. However, all judgements were validated by more than one judge. For the method level ground truth, three judges were used and full inter-rater agreement was achieved. 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 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 using a threshold, the results are based on the set thresholds. While we attempted to choose good general thresholds, different thresholds may lead to different results, observations, and conclusions.

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.

6 RELATED WORK

Establishing and maintaining traceability links between tests and their tested functionality has received significant attention as traceability links have multiple applications in the software engineering process: 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. [29] describe these benefits in detail and Parizi et al. [27] 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 [25, 26] but this approach has not seen significant adoption.

At the method level, EzUnit [3] is a framework that allows developers to annotate tests with links to the method-under-test. To do so, it performs a 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 [17] 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. [16] 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.

SCOTCH+ (Source code and Concept based Test to Code traceability Hunter) is a traceability system introduced by Qusef et al. [29] that achieves better accuracy and provides more benefit to developers than LCBA or NC [28]. 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.

Other test-to-code traceability work is based on the assumption that a test should be similar to a tested unit. Kicsi et al. [20] explore the usage of Latent Semantic Indexing (LSI) over source code to establish traceability links between test classes and tested classes. 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. [7] 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 [6] in the same way and report that word embeddings perform best in terms of precision and recall.

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 et al. [23] 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.

Closest to our work is the work by Rompaey and Demeyer [32] who 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.

Gergely et al. [14] 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 for produce the final traceability links.

Ståhl et al. [31] 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.

7 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 and enhances them by combining them and applying them to both the method level and class level, making 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 four 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. This makes TCtracer the most effective approach for test-to-code traceability to date.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 81.

[2] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. 2002. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering* 28, 10 (2002), 970–983.

[3] Philipp Bouillon, Jens Krinke, Nils Meyer, and Friedrich Steimann. 2007. EZUNIT: A Framework for Associating Failed Unit Tests with Potential Programming Errors. In *Agile Processes in Software Engineering and Extreme Programming*. Springer Berlin / Heidelberg. https://doi.org/10.1007/978-3-540-73101-6_14

[4] Travis CI. [n.d.]. *Travis CI*. Retrieved 2020-01-13 from https://travis-ci.org/

[5] Jane Cleland-Huang. 2012. Traceability in agile projects. In *Software and Systems Traceability*. Springer, 265–275.

[6] Viktor Csuvik, András Kicsi, and László Vidács. 2019. Evaluation of Textual Similarity Techniques in Code Level Traceability. In *International Conference on Computational Science and Its Applications*. Springer, 529–543.

[7] Viktor Csuvik, András Kicsi, and László Vidács. 2019. Source Code Level Word Embeddings in Aiding Semantic Test-to-Code Traceability. In *10th International Workshop on Software and Systems Traceability*. 29—-36. https://doi.org/10.1109/SST.2019.00016

[8] Jesse Davis and Mark Goadrich. 2006. The relationship between Precision-Recall and ROC curves. In *Proceedings of the 23rd International Conference on Machine Learning*. ACM, 233–240.

[9] Andrea De Lucia, Fausto Fasano, and Rocco Oliveto. 2008. Traceability management for impact analysis. In *2008 Frontiers of Software Maintenance*. IEEE, 21–30.

[10] The Apache Software Foundation. [n.d.]. *Apache Ant*. Retrieved 2019-08-13 from https://ant.apache.org/

[11] The Apache Software Foundation. [n.d.]. *Apache Commons Collections*. Retrieved Accessed: 2018-07-30 from https://commons.apache.org/proper/commons-collections/

[12] The Apache Software Foundation. [n.d.]. *Apache Commons IO*. Retrieved 2019-08-18 from https://commons.apache.org/proper/commons-io/

[13] The Apache Software Foundation. [n.d.]. *Apache Commons Lang*. Retrieved 2018-07-30 from https://commons.apache.org/proper/commons-lang/

[14] Tamás Gergely, Gergő Balogh, Ferenc Horváth, Béla Vancsics, Árpád Beszédes, and Tibor Gyimóthy. 2019. Differences between a static and a dynamic test-to-code traceability recovery method. *Software Quality Journal* 27, 2 (2019), 797–822.

[15] Malcom Gethers, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2011. On integrating orthogonal information retrieval methods to improve traceability recovery. In *27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 133–142.

[16] Mohammad Ghafari, Carlo Ghezzi, and Konstantin Rubinov. 2015. Automatically identifying focal methods under test in unit test cases. In *15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 61–70. https://doi.org/10.1109/SCAM.2015.7335402

[17] Victor Hurdugaci and Andy Zaidman. 2012. Aiding software developers to maintain developer tests. In *16th European Conference on Software Maintenance and Reengineering*. IEEE, 11–20.

[18] Jenkins. [n.d.]. *Jenkins*. Retrieved 2020-01-13 from https://jenkins.io/

[19] James A Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. IEEE, 467–477.

[20] András Kicsi, László Tóth, and László Vidács. 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*. 8–14. https://doi.org/10.1145/3194104.3194106

[21] Vladimir I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. 707–710.

[22] Object Refinery Limited. [n.d.]. *JFreeChart*. Retrieved 2018-07-30 from http://www.jfree.org/jfreechart/

[23] Matej Madeja and Jaroslav Porubän. 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

[24] Christopher Manning, Prabhakar Raghavan, and Hinrich Schütze. 2010. Introduction to information retrieval. *Natural Language Engineering* 16, 1 (2010), 100–103.

[25] Reza Meimandi Parizi, Asem Kasem, and Azween Abdullah. 2015. Towards Gamification in Software Traceability: Between Test and Code Artifacts. In *Proceedings of the 10th International Conference on Software Engineering and Applications*. SCITEPRESS - Science and Technology Publications, 393–400. https://doi.org/10.5220/0005555503930400

[26] Reza Meimandi Parizi. 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, 193–200. https://doi.org/10.1109/SERA.2016.7516146

[27] Reza Meimandi Parizi, Sai Peck Lee, and Mohammad Dabbagh. 2014. Achievements and Challenges in State-of-the-Art Software Traceability Between Test and Code Artifacts. *IEEE Transactions on Reliability* 63, 4 (dec 2014), 913–926. https://doi.org/10.1109/TR.2014.2338254

[28] Abdallah Qusef, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and David Binkley. 2013. Evaluating test-to-code traceability recovery methods through controlled experiments. *Journal of Software: Evolution and Process* 25, 11 (Nov 2013), 1167–1191. https://doi.org/10.1002/smr.1573

[29] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley. 2014. Recovering test-to-code traceability using slicing and textual analysis. *Journal of Systems and Software* 88 (2014), 147–168.

[30] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. 2017. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access* 5 (2017), 3909–3943.

[31] Daniel Ståhl, Kristofer Hallén, and Jan Bosch. 2017. Achieving traceability in large scale continuous integration and delivery deployment, usage and validation of the eiffel framework. *Empirical Software Engineering* 22, 3 (2017), 967–995.

[32] Bart Van Rompaey and Serge Demeyer. 2009. Establishing traceability links between unit test cases and units under test. In *13th European Conference on Software Maintenance and Reengineering*. IEEE, 209–218.

[33] Robert White and Jens Krinke. 2020. *TCTracer Evaluation Data – International Conference on Software Engineering 2020 [Data set]*. https://doi.org/10.5281/zenodo.3637597

[34] Stefan Winkler and Jens von Pilgrim. 2010. A survey of traceability in requirements engineering and model-driven development. *Software & Systems Modeling* 9, 4 (2010), 529–565.

[35] Rafael Winterhalter. [n.d.]. *Byte Buddy*. Retrieved 2019-08-19 from https://bytebuddy.net