Department of Computer Science
University College London

Automated Recommendation, Reuse, and Generation of Unit Tests for Software Systems

Robert White

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of

University College London.

Department of Computer Science

University College London

February 16, 2022

I, Robert White, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work. The papers that have been published or submitted for publication are listed by chapter below along with a summary of my contributions to the papers.

Chapter 3:

1. R. White, J. Krinke, R. Tan, Establishing Multilevel Test-to-Code Traceability Links, in *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, 2020.

   Contribution: I designed the approach in collaboration with Dr. Krinke after using a student project as a starting point, which had initial implementations for some of the techniques. I extended the approach in multiple ways, including adding multilevel information and technique combination. I built the tool and designed and conducted the experiments. I wrote the paper with input from Dr. Krinke.

2. R. White, J. Krinke, TCTracer: Establishing Test-to-Code Traceability Links Using Dynamic and Static Techniques, in *Empirical Software Engineering (EMSE) - Submitted*, 2021.

   Contribution: I designed the extended approach in collaboration with Dr. Krinke and designed and conducted all the experiments. I wrote the paper with input from Dr. Krinke.

Chapter 4:

1. R. White, J. Krinke, E. Barr, F. Sarro, C. Ragkhitwetsagul, Artefact Relation Graphs for Unit Test Reuse Recommendation, in *Proceedings of the 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2021.

   Contribution: I designed the approach and evaluation in collaboration with Dr. Krinke and Dr. Barr. I conducted the experiments. Dr. Krinke, Dr. Sarro, and Dr. Ragkhitwetsagul assisted with the manual evaluation and Dr. Krinke

and Dr. Sarro assisted with preparation and enrolment for the user study. I wrote the paper with input from all authors.

Chapter 6:

1. R. White, J. Krinke, TestNMT: Function-to-Test Neural Machine Translation, in *Proceedings of the 4th ACM SIGSOFT International Workshop on NLP for Software Engineering, ser. NL4SE*, 2018.
   Contribution: I designed the approach and evaluation and designed and conducted the experiments. I wrote the paper with input from Dr. Krinke.

Chapter 7:

1. R. White, J. Krinke, ReAssert: Deep Learning for Assert Generation, in *arXiv preprint*, 2020.
   Contribution: I designed the approach. The evaluation was designed in collaboration with Dr. Krinke. I conducted the experiments. I wrote the paper with input from Dr. Krinke.

In addition, I co-authored the following published paper, which is not included in this thesis.

1. N. Bafatakis, N. Boecker, W. Boon, M. C. Salazar, J. Krinke, G. Oznacar, R. White, Python Coding Style Compliance on Stack Overflow, in *In Proceedings of the 16th Working Conference on Mining Software Repositories*, 2019.
   Contribution: I assisted with the design of the evaluation and contributed to the writing of the paper.

Date: February 16, 2022
Name: Robert White
Signature:

Abstract

This thesis presents a body of work relating to the automated discovery, reuse, and generation of unit tests for software systems with the goal of improving the efficiency of the software engineering process and the quality of the produced software.

We start with a novel approach to test-to-code traceability link establishment, called TCTracer, which utilises multilevel information and an ensemble of static and dynamic techniques to achieve state-of-the-art accuracy when establishing links between tests and tested functions and test classes and tested classes. This approach is utilised to provide test-to-code traceability links which facilitate multiple other parts of the work.

We then move on to test reuse where we first define an abstract framework, called Rashid, for using connections between artefacts to identify new artefacts for reuse and utilise this framework in Relatest, an approach for producing test recommendations for new functions. Relatest instantiates Rashid by using TCTracer to establish connections between tests and functions and code similarity measures to establish connections between similar functions. This information is used to create lists of recommendations for new functions.

We then present an investigation into the automated transplantation of tests which attempts to remove the manual effort required to transform Relatest recommendations and insert them into another project.

Finally, we move on to test generation where we utilise neural networks to generate unit test code by learning from existing function-to-test pairs. The first approach, TestNMT, investigates using recurrent neural networks to generate

whole JUnit tests and the second approach, ReAssert, utilises a transformer-based architecture to generate JUnit asserts.

In total, this thesis addresses the problem by developing approaches for the discovery, reuse, and utilisation of existing functions and tests, including the establishment of relationships between these artefacts, developing mechanisms to aid automated test reuse and learning from existing tests to generate new tests.

Impact Statement

The work presented in this thesis has multiple impacts both inside and outside academia. Inside academia, we have furthered the state-of-the-art in several areas of research and provided to the research community a set of tools, data sets, and learnings that will facilitate further research and assist other researchers in building upon our achievements and outcomes. In the field of traceability research, we have created an open source tool, TCTracer which implements our novel approach to test-to-code traceability link establishment and is the first tool to establish links at both the method and class levels. This tool would be useful by both traceability researchers as a basis for further work or as a comparison point to any new approach. Additionally, this tool is useful for researchers who do not work in traceability research but need to use traceability links as part of their approach to solving other problems. For these researchers, this tool would save a lot of effort and could improve the results of their research. In addition to the TCTracer tool, we also provide a new manually created ground truth data set of test-to-code traceability links which other researchers can use for their evaluations. As the creation of these data sets is time-consuming and error-prone, this is a contribution to the community that can save others a lot of time and effort. In the field of reuse, we have developed a general framework for the discovery of connections between software systems artefacts in the Rashid framework. This is a generalised framework that could be utilised by researchers in new ways, such as for the recommendation of other artefacts types or revealing dependencies between artefacts. In the field of test generation, we have introduced a new approach and data sets for generating test code using machine learning. As this is a relatively new field of research, other

researchers can now use this work as a starting point to build future approaches and, in a similar fashion to our traceability data sets, utilise the new data sets that we provided to compare any new approach to our results. The experience of performing this research also created a set of learnings which we have passed on to the community as takeaway lessons. Again, this saves other researchers a lot of time and effort and lays the foundations for future work. Outside of academia, the tools developed in this work could be put to good use in industry. The goal of our tools, in general, is to assist software developers in a way that saves time and effort, and therefore cost, while improving the resulting software. The realisation of this goal, therefore, is very valuable to industry and there is a great demand for tools that assist the automation of testing, especially if those approaches fit into industry-standard practices such as continuous integration and delivery. We have shown that our work is applicable in this fashion, for example by demonstrating how our tools TCTracer and Relatest integrate into these types of development pipelines.

Acknowledgements

Many people have greatly contributed to my achievements over the course of this PhD and have provided me with the environment and opportunities needed to succeed both professionally and personally.

First and foremost, I would like to thank my supervisor, Dr Jens Krinke, for his expert guidance and for making my PhD an extremely enjoyable and collaborative experience. Working with Jens has been an absolute pleasure and I have learnt a great deal about software engineering and research from my time under his fantastic supervision. I greatly appreciated having the opportunity to work with him on the formulation and evaluation of novel solutions to interesting and difficult problems and our work together was always productive and enjoyable. Next, I wish to thank Dr Earl Barr for his excellent guidance as my second supervisor. Working with Earl has always been extremely enjoyable and his advice and technical contributions were very insightful. Earl's contributions have helped greatly in bringing this work to fruition as well as in assisting my personal professional development. I would also like to thank Dr Federica Sarro and Dr Chaiyong Ragkhitwetsagul for their fantastic contributions to the work on test reuse, Dr David Clark for his diligence and expert feedback as the examiner for my first year and transfer vivas, and Dr Dean Mohamedally for providing me with great teaching opportunities, such as the trips to Tokyo and Paris.

Also, I would like to thank all of the CRESTies with whom I spent many lunches and evenings in the pub talking about every topic imaginable and having a great deal of fun. These thanks extend to all CRESTies but I would like to specifically mention David Kelly, Dan Bruce, Leonid Joffe, Carlos Calderon,

Giovani Guizzo, David Landsburg, Profir Partachi, James Callan, and Rebecca Moussa. This fun and talented group of individuals helped make my time at UCL especially memorable.

Beyond my professional sphere, I have received immense support and encouragement from my family who have played the most vital role in keeping me happy and productive. Above all, I would like to thank my parents Clive and Daniela White for their incredible love and support. The assistance they have given me, both practically and emotionally, is without parallel and has, without a doubt, been my greatest source of support. I would also like to thank my sister Holly White and my aunt and uncle Celia and David Pratt for all of the wonderful time spent together, being a great source of joy, and for providing me with excellent well-grounded advice, even when confronted with alien topics. I also want to extend a special thank you to my nephew Max Clode for being a bundle of energy who always brings a smile to my face. I am forever grateful to all of my family and this thesis is dedicated to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software development is a costly, time-consuming, and error-prone activity. Tools that reduce the amount of development effort required and assist in the creation of high-quality software can, therefore, have a large positive effect on both the cost and outcome of the software development process. Testing is an aspect of development that can be especially onerous as the process of creating and maintaining unit tests is constant, complex, and often disliked by developers, frequently resulting in software that has a low level of test coverage. This thesis tackles the problem of how to alleviate these issues by assisting developers to create and maintain unit tests in an efficient manner. This is crucially important as maintaining a high level of good quality test coverage is required to produce robust software but has a large cost in terms of developer time, and therefore, a large monetary cost. Previous work has shown that to maintain a high level of unit test coverage, tests must be created at the same time as the tested code as retroactively creating unit tests is rarely done and only partially successful when attempted [Klammer and Kern, 2015]. Therefore, by automating parts of the unit test creation process and making it easier for developers to develop tests alongside implementation code, we hope to improve the overall efficiency of the software engineering process and the robustness of the resulting software. The work presented in this thesis aims to reduce the manual effort required to develop software and improve the quality of the resulting software through the development of novel approaches to assist discovery, reuse, and generation of unit tests and the implementation of these approaches in a set of tools.

To assist the discovery of unit tests for reuse, we first present TCTracer, a multilevel approach to test-to-code traceability link establishment which allows us to determine, with state-of-the-art precision, the tests which test each function and the test classes which test each class. This work was motivated by our need to establish test-to-code traceability links for use with our approaches to test reuse and test generation but also by the general demand in academia and industry for tools to accurately perform this task, as shown through case studies and developer interviews [Ståhl et al., 2017]. While previous work has always only focused on one level, usually the class level, and only utilised a single technique at once, our approach establishes links at both the method and the class levels and utilises an ensemble of new and existing techniques. This gives our approach better precision and applicability than previous work with our evaluation of TCTracer showing 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 a MAP of 92%.

For test reuse, we utilise the test-to-code links produced by TCTracer and Rashid, our framework for the discovery of relationships between artefacts to create Relatest, an approach for generating recommendations of existing tests to test new functions. Our evaluation finds that by using the recommendations produced by Relatest we can achieve an average of 58% reduction in developer effort (measured in tokens), for 75% of functions, resulting in an overall saving of 43% of the effort required to create tests. Additionally, a user study revealed that, on average, developers needed 10 minutes less to develop a test when given Relatest recommendations and all user study participants reported that the recommendations were useful. Given that Relatest requires some manual effort from developers to transform the recommended tests, we also developed an approach for automatically transplanting tests from one context to another context utilising a multistage approach incorporating genetic improvement (GI). Our evaluation shows that this results, on average, in a passing transplanted test for 21% of functions and a compiling test for 31% of functions. Our evaluation also revealed this approach has the potential to create new tests which reveal real-world faults not revealed by

EvoSuite [Fraser and Arcuri, 2013].

As our approaches for test reuse and transplantation do not always produce a working test for all functions, we also tackle the problem of unit test generation for arbitrary functions. In pursuit of this we have established two approaches to unit test code generation, TestNMT, for the generation of whole JUnit tests in an approximate form, and ReAssert which focusses on the generation of JUnit test asserts only. For both of these approaches, we adapt neural networks from the domain of natural language processing and are a break from traditional unit test generation tools which utilise random generation, search-based, or analysis-based methods. The motivation for this approach firstly comes from the idea that tests should be able to be translated into functions similarly to how one language is translated into another in typical neural machine translation tasks. Secondly, motivation is provided by the fact that test generation is subject to the oracle problem and, therefore, the hardest parts of tests to generate, including inputs and oracles may be best learnt from existing manually written tests. Our evaluation of TestNMT resulted in a maximum BLEU score of 21.2, a maximum ROUGE-L score of 38.67, and demonstrated that TestNMT is capable of generating approximate tests that are easy to adapt to working tests. Our evaluation of ReAssert shows up to 44% of generated asserts for a single project match exactly with the ground truth, increasing to 51% for generated asserts that compile with 71% of the generated asserts being unique.

The primary benefit of utilising these approaches is an increase in development productivity as reusing and generating artefacts is almost always faster than creating new artefacts from scratch; thus reducing the cost and time of software development. Reuse also improves the quality of the resultant software as reused artefacts are, by definition, more mature and well tested than any freshly created artefacts. Therefore we assume that, on average, reused artefacts are of higher quality than freshly created artefacts and that software that was constructed using a large number of reused artefacts should be of higher quality than a system that is mostly comprised of new artefacts. This delta in quality may apply to correctness

or a non-functional property, such as security or performance. As a secondary benefit, reuse and generation can also increase the happiness and job satisfaction of developers by reducing the amount of tedious and repetitive tasks they have to perform, such as writing large numbers of similar unit tests.

1.1 Problem Statements of the Thesis

The problem addressed by this work is how to create and maintain, in a time-efficient manner, a collection of effective unit tests when developing software systems. This top-level problem is addressed by decomposing it into a set of sub-problems: establishing test-to-code traceability links, identifying existing tests that make good candidates for reuse, recommending reuse candidates to developers and transplanting tests for reuse into a new environment, or learning from existing test-to-code traceability links to generate new tests for functions. To establish traceability links we address the problem of how to determine which functions are tested by which tests and which classes are tested by which test classes. To identify and recommend reuse candidates we address the problem of how to search a large corpus of artefacts, determine the most suitable artefacts for reuse in a given context, and recommend them to a developer. To tackle transplantation we tackle the problems of how to extract tests and their dependencies from their environment, implant them in a new environment, and adapt them to properly function in their new environment. To generate new tests we address the problem of how to use existing tests to learn to generate new tests.

1.2 Goal and Objectives

The goal of this thesis is to present a set of novel techniques and tools to assist with the creation and maintenance of a large high-quality suite of unit tests when developing software systems. The objectives of these techniques and tools include achieving the following:

1. Establishing of traceability links between tests and tested code.

2. Aiding reuse of mature well-tested artefacts, especially tests.

3. Generation of new test code for untested code.

1.3 Contributions

The contributions of this thesis are:

1. An approach to test-to-code traceability that utilises an ensemble of techniques using dynamic and static information and a multilevel flow of information.

2. A comparative evaluation of traceability techniques at both the method and class levels with multilevel information, technique combination methods, and technique combination weighting.

3. A manual investigation into the causes of false positive and false negative test-to-code traceability links.

4. A manually curated ground truth dataset [White and Krinke, 2020a] of test-to-function and test-class-to-class links.

5. RASHID, an abstract framework for the reuse of artefacts using artefact relation graphs.

6. RELATEST, a realisation of RASHID for recommending existing tests to be reused to test new functions.

7. An evaluation of the effectiveness of RELATEST using token-based edit distance, a manual investigation, and a user study.

8. An approach to test transplantation utilising genetic improvement, evaluated with transplantation success rates and fault detection capability.

9. TESTNMT, an approach to unit test generation utilising function-to-test neural machine translation with recurrent neural networks (RNNs), evaluated with edit distance and translation metrics.

10. ReAssert, a project-based deep learning approach for the generation of unit test asserts implemented for JUnit and evaluated using lexical accuracy and dynamic analysis with Reformer, a new state-of-the-art transformer-based model and two RNN-based models from previous work.

11. A comparative evaluation of ReAssert with all three models using lexical accuracy and uniqueness versus a previous approach, Atlas.

12. Takeaway messages for researchers and practitioners concerning the construction of data sets when applying sequence to sequence learning for code generation.

1.4 Thesis Organisation

Chapter 2 presents the background, firstly for the research areas where we make our contributions, and secondly for the research areas which are utilised to make those contributions. Chapter 3 presents TCTracer, our approach and implementation of a multilevel test-to-code traceability establishment technique. Chapter 4 presents Rashid, an abstract framework for the reuse of artefacts using artefact relation graphs and Relatest, a instantiation of Rashid, which utilises the links established by TCTracer to make unit test recommendations for query functions. Chapter 5 presents our approach to automated transplantation of tests with a focus on the transplantation of the recommendations produced by Relatest. Chapter 6 presents TestNMT, an approach for the generation of new approximate unit tests for query functions. Chapter 7 presents ReAssert, a project-based approach to the generation of asserts for JUnit tests, and its implementation using two recurrent neural network models and one transformer-based model. Chapter 8 presents a general discussion of the results, observations, and take away messages from the work along with the conclusion of the thesis and the potential future work.

Chapter 2

Background

In this chapter, we first describe the state of the art in the research areas of traceability, code reuse, code transplantation, and test code generation as these are the areas where the novel contributions of the thesis are made. Next, the background for the research areas that are utilised to make our contributions, specifically code similarity and graph edge prediction, are discussed where we establish the state of the art in each specific area and how it is utilised by this work. Finally, we conclude by motivating the contributions and demonstrate that the contributions are an improvement over the current state of the art.

2.1 Traceability

The establishment and maintenance of traceability links between artefacts in software systems is a research area that has received much attention from the community and traceability techniques can be broadly categorised by the types of links that they find in terms of the types of artefacts being linked. In this background, we focus on techniques that find test-to-code, natural-language-to-code, and natural-language-to-natural-language links. Examples of natural language artefacts include requirements documentation, regulatory codes, and design documents.

Establishing and maintaining traceability links between tests and tested code has received significant attention as these test-to-code 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. [2014] describe these benefits in detail and [Parizi et al., 2014] present an overview of the achievements and challenges of test-to-code traceability.

Rompaey and Demeyer [2009] investigates multiple methods for attempting to establish traceability links between tests and their tested functions, specifically Naming Conventions (NC), Fixture Element Types (FET), Static Call Graphs (SCG), Last Call Before Assert (LCBA), Lexical Analysis (LA), and Co-Evolution (Co-Ev). The findings from this study show that none of these techniques alone performs well enough to effectively solve the problem and that a combination of techniques is required. The sensitivity of the results to the subject system is also highlighted as there are multiple variables related to the development practices used by the project that can heavily influence the results. Despite these issues, the study highlights NC and LCBA as the best overall performing techniques, with LA, Co-Ev, and SCG performing poorly in most circumstances. 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.

An alternative approach to establishing test-to-code traceability links is presented in Hurdugaci and Zaidman [2012] where dynamic call graphs are used. In this approach, the running test code is instrumented to build dynamic call graph information which is used to establish many-to-many relationships between tests and functions by simply linking any test and function pair if the function is called by the test. These links are utilised by a tool, TestNForce, that alerts users as to which unit tests need to be changed when a function is changed. Unfortunately, although the evaluation states that developers found the tool useful, it does not provide concrete results on the precision and recall of the discovered links. This makes assessing the effectiveness of this technique for establishing traceability links difficult.

At the class level, SCOTCH+ (Source Code and Concept based Test to Code traceability Hunter) is a test-to-code traceability system introduced by Qusef et al.

[2014] that uses a hybrid approach. 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. While SCOTCH+ achieves better accuracy than LCBA or NC, the current implementation of RELATEST does not use it because it operates at the class level, not the method level.

At the method level, 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. 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.

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.

The majority of other test-to-code traceability work is based on the assumption that a test should be similar to the tested code by some code similarity measure. 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. 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. [2019a] 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., 2019b] 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 and Porubän [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.

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] focus 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.

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 provide the ability to see the differences in the predicted links between techniques. This further demonstrates the potential applicability of test-to-code traceability links and the appetite for their usage.

Other 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.

Establishing traceability links that include natural language artefacts accounts for the bulk of traceability research. This is partly because in safety-critical systems the maintenance of traceability links between regulatory codes and their implementation in the system is often mandated by law. Other use cases for

links with natural language artefacts, such as maintaining consistency between documentation and implementation, also contribute to the existing work. The techniques that have recently primarily been utilised in pursuit of establishing these links are based on NLP and machine learning but IR techniques also have a long history of being applied to traceability recovery, including probabilistic and vector space models (VSM).

Marcus and Maletic [2003] presents an approach to establishing traceability links between natural language artefacts and code artefacts by applying LSI to the artefacts to encode them in vector space. The natural language artefact vectors are then compared with the code vectors via cosine similarity and two artefacts are considered linked if their cosine similarity is greater than a given threshold. The threshold of 0.7 is selected, corresponding to a 45-degree angle between the vectors, as this threshold yielded the best results of 71% precision at 42% recall. These results are good, however, the evaluation only includes a single project with only 119 natural language artefacts. Therefore, the generality of the results presented in this work is in question. In general, for establishing traceability links, LSI suffers from the problem that it only encodes lexical information; whereas code implements the semantics described by a natural language document. This means that when a code artefact is linked to a natural language artefact but does not share significant lexical information with the natural language artefact, LSI will not be able to recover the link.

McMillan et al. [2009] presents a graph-based approach to establishing documentation-to-documentation links. The approach uses textual and structural analysis to first build code-to-code and code-to-documentation links which are then used to construct a traceability link graph (TLG). The TLG is used to predict documentation-to-documentation links by inferring edges in the graph through a neighbourhood analysis. RELATEST also utilises a graph-based approach, however, RELATEST uses the composition of unipartite and bipartite graphs through the RASHID framework to make cross-domain recommendations, and is, therefore, tackling a different problem with a more flexible approach.

Most recently, machine learning has been brought to bear on the problem of establishing traceability links. Asuncion et al. [2010] presents a system that uses topic modelling, specifically Latent Dirichlet Allocation (LDA), to build and maintain traceability links between a wide variety of artefact types such as requirements, design documents, issue reports, test cases, and implementation code. Unlike most previous work, Asuncion et al. [2010] make a distinction between retrospective traceability, where traceability links are discovered in batch over an existing set of artefacts, and prospective traceability, where the links are established online at artefact creation time. This approach uses prospective traceability to build the initial links between artefacts, by monitoring the behaviour of developers in their development environments, and retrospective traceability to enhance the set of links using Latent Dirichlet Allocation. Due to the mixture of link establishment methods, this approach is not suitable for all traceability establishment tasks, particularly in situations where developers cannot be monitored at artefact creation time or the task is to establish links for a legacy system. However, the work also provides a simple one-to-one comparison of the precision and recall attained by the LDA versus standard LSI, which shows LDA winning on both precision and recall.

Cleland-Huang et al. [2010] provide a comparison between traditional traceability methods, represented by a Probabilistic Network [Cleland-Huang et al., 2007], and a machine learning classifier. This work shows that the traditional method outperforms the classifier when establishing links that are relatively easy to find, whereas the ML classifier performs better on links that are difficult to establish. This showed that while still nascent, the machine learning techniques have the potential to move traceability forward.

This potential is further realised in the recent work Guo et al. [2017] that utilises deep learning for the establishment of traceability links between regulatory codes and the design documents that specify the implementation of those codes in the system. This system utilises word embeddings and a recurrent neural network with an attention model to encode the artefacts into semantic vectors, which can then be compared to predict links between the artefacts. The evaluation of this

technique compares it to the more traditional semantic similarity techniques of LSI and VSM, where it is shown to achieve higher levels of precision at high levels of recall. The model is, however, very sensitive to the quantity of training data and the distribution of the data between the classes. This study shows that deep learning does have a large potential for the establishment of traceability links if a large quantity of good quality training data can be obtained.

2.2 Code Search and Recommender Systems for Reuse

Code reuse is a common practice in software development as developers look to solve problems and save time by using existing code from other projects and the web. Due to the prevalence of reuse, several research projects have focused on developing approaches to discover and reuse existing code by performing code search and making recommendations.

The simplest approach to code search is the traditional approach to plain text search, string-based pattern-matching. However, this approach is very limited as it does not deal well with complex code snippets that may contain multiple lines and is also deficient in that it requires the user to know the exact text of the code that they are looking for. String-based pattern-matching is therefore of limited use for making source code recommendations.

To overcome the limitations of string-based pattern-matching, some code search tools such as SCRUPLE [Paul and Prakash, 1994] adopted similar techniques to static analysis tools such as GENOA [Devanbu, 1992] which use the components of a compiler front-end (lexer, parser, type checker etc.) to construct internal representations of the code. These representations can then be searched using a pattern language that allows the user to define queries at the syntactic level, essentially defining a pattern that will match to a subtree of an AST. This allows the user to retrieve code snippets that are relevant and lexically diverse. However, these techniques require the developer to manually provide the query patterns and therefore demand the programmer have some a priori knowledge of what they are

searching for. RELATEST avoids this by having the search guided by the relationships between existing artefacts through the RASHID framework.

Other approaches use textual Information Retrieval (IR) methods to search existing code and make suggestions. One such example is Prompter, an IDE plugin that searches Stack Overflow discussions and recommends code fragments for developers [Ponzanelli et al., 2014]. Prompter starts by using the IDE context, such as the currently displayed code, to formulate a query for a Stack Overflow search, the results of which are displayed to the developer. However, a primary weakness of this system is that the user must manually extract the code snippets from the discussions and there is a high probability the contained code snippets will not be executable without extensive modification. This is because there is a high probability the code snippets are not complete and do not come directly from a working system. In comparison, RELATEST operates on complete functions from existing projects.

Test-Driven Code Search (TDCS) is another approach to code search which uses test suites to define the desired behaviour and test the code fragments that are returned by a code search tool, ensuring that they can be executed in the context of the target system and providing information as to the correctness of the returned code. CodeGenie [Lemos et al., 2011] is an example of a tool that utilises this approach to make code recommendations in an eclipse plugin. The code search engine Sourcerer [Bajracharya et al., 2014] is used to perform the code search. To make a recommendation, CodeGenie takes a test class as input, performs feature extraction on the test class, and uses these features to perform a Sourcerer code search. Once a result is found, the candidate functionality is extracted from its current context via slicing and then presented in the plugin for testing. The features extracted from the provided test class include keywords, identifiers, and the interface definition of the required code which are extracted by analysing the AST of the test class. CodeGenie, and TDCS in general, however, are only applicable to a Test Driven Development (TDD) scenario as a test class is required to search for the desired associated functionality and RELATEST currently works in the opposite

direction, finding tests for existing functions. TDCS may become useful in the future if RELATEST is extended to cover the TDD scenario.

Mcmillan et al. [2013] describe a tool, Portfolio, which takes the concept of recommendation and extends it beyond individual code snippets to redefine it as the search for a body of code that completely implements a high-level requirement. It is argued that this is necessary as in most cases programmers are not just searching for a single function, but for an implementation of a requirement, which likely consists of multiple functions. Therefore, all the relevant functions and an example of how they are called is required for the recommendation to be useful. Portfolio achieves this using a combination of techniques, specifically natural language processing (NLP), PageRank [Brin and Page, 1998], and spreading activation network (SAN) algorithms [Crestani, 1997]. To enable the usage of these techniques, static call graphs are constructed for all the code in the corpus. This allows the PageRank matrix to be constructed over the transitions in the static call graphs and facilitates the second ranking by SAN to be performed. Given these components, the process starts with an initial keyword search to return relevant functions; SAN is then used to expand the returned function to a set of linked functions and the PageRank score over the transitions is computed. The SAN and PageRank scores are then combined into the final score for the located functions, which is used for ranking the recommendations. The evaluation shows Portfolio outperforming the now defunct Google Code Search and Koders code search engines in confidence, precision, and normalised discounted cumulative gain, with consistently higher means and lower variance. In terms of applicability to RELATEST this approach could be useful as it returns not just single functions, but chains of functions which are more likely to be the complete solution to the problem the programmer is looking to solve. Therefore, the function chains could make good candidate organs for transplant. The downside of this approach is the pre-processing work required to construct the static call graphs and the PageRank matrix over the entire corpus, which gets expensive as the corpus grows large. There is also a runtime cost associated with the online computation of the SAN for each query but this should be negligible for

almost all queries.

Machine learning has also been utilised in the field of code search, such as in the work presented by Niu et al. [2017] which applies a learning-to-rank scheme for ranking results from code search engines. One of the benefits of this technique is that it can be integrated into any code search engine that ranks its results. The evaluation compares the learning-to-rank technique with the standard ranking from Codota[1], a commercial code search tool and the best performing ranking engine for API usage example code. In this analysis, learning-to-rank demonstrated a 35% and 48% improvement in normalised discounted cumulative gain and expected reciprocal rank. The downside of these techniques is the need to gather the training data, however, once the data is gathered the training itself can be done once offline.

A closely related work to the current implementation of RELATEST is Erfani et al. [2013] which presents work that is also concerned with the recommendation of unit tests based on the similarities between functions. Erfani et al. build clone classes using the clone detector NiCad [Roy and Cordy, 2008] and then identify tests that cover any of the clones in the class. These tests are then recommended to uncovered clones in the class. RELATEST on the other hand uses the Jaccard index on a function's bag of tokens, a more general notion of function similarity, to define sets of similar functions. For a newly written function, RELATEST suggest the tests that execute similar functions. One reason for the large difference in the number of recommendations made is that the number of clone classes Erfani et al. [2013] discover which contain both covered and uncovered functions is small. In contrast, we can control how many recommendations are made by RELATEST by adjusting the function similarity threshold. Further, the quality of the recommendations made by Erfani et al. [2013] is incomparable to RELATEST as they neither evaluate quality nor provide examples.

[1] https://www.codota.com/

2.3 Code Transplantation

Automated transplantation of artefacts between systems is a more nascent area of research than code search. However, in recent years there have been significant advancements in this area, in part due to the maturation of genetic programming techniques that lend themselves well to the task of transplantation.

Harman et al. [2013] present the Localise, Abstract, Target, Interface, Instantiate, and Verify (LATIIV^{R}) framework which describes a high-level approach for performing code transplantation, which can be specialised for specific situations depending on the techniques used to implement the individual steps.

A concrete implementation of an almost fully automated functionality transplantation system, μTrans, can be found in Barr et al. [2015]. μTrans first takes a manually specified entry point for the functionality in the donor and utilises program slicing to constructing an organ that contains the functionality for transplant and a vein that builds and initialises the execution environment expected by the functionality. Genetic programming is then used, guided by a test suite from the donor, to adapt the organ so that it can execute in the context of the host system. Observational slicing is then used to reduce the organ and explore mappings from host variables to the parameters of the organ. The organ is then implanted into the host and validated by additional testing. This approach forms the basis of the transplantation approach described in chapter 5 for use with Relatest, however, as Relatest is concerned with the transplantation of tests instead of functionality, there are some significant differences, discussed in Section 2.7.

Zhang and Kim [2017] presents a system, Grafter, that also performs a form of transplantation. In the case of Grafter, the transplanted code is untested fragments that are clones of tested fragments. The untested fragments are transplanted on top of their corresponding tested fragment to allow them to be tested by the existing tests. While the overall concept has issues, such as the transplanted fragment no longer being representative of the original fragment, Grafter does provide an algorithm for adapting the organ fragment for integration into the target location. The adaptation algorithm used by Grafter is claimed to be sound with

respect to the compilation of the target code and provides mechanisms for handling variable name variation, method call variation, variable type variation, expression type variation, and recursion.

The fact that GRAFTER is sound with respect to compilation is positive, however, there are issues with aspects of the algorithm which threaten its usefulness. One such issue is the notion of the structural equivalence of types which is applied to deal with translating the types in the transplanted fragment to match the target context. This technique assumes that two types can be substituted for each other if they contain the same number of fields and the types of all fields match or are themselves structurally equivalent. Firstly, this is a broad definition of equivalence that will often produce unwanted behaviour and secondly, if there is no matching type in the patient for a type in the organ, the technique simply fails. Another issue is the handling of variable name variation where the transplanted code references variables that are not present in the patient scope. Here GRAFTER uses the Levenshtein distance between the variable names to attempt to match variable references in the transplanted fragment to variables in the patient scope. However, the similarity between variables names may not be a good indicator of semantic similarity.

In comparison, RELATEST only has to deal with a subset of the issues that GRAFTER deals with since RELATEST will be transplanting whole tests, instead of arbitrarily sized fragments. This difference makes the problem of matching variables easier as all local variable declarations will be included in the transplanted code; only global variable references will become disconnected.

Existing work can also provide some insight into the best sources for donor artefacts. Barr et al. [2014] examine the plastic surgery hypothesis which tests the extent to which new code for a system can be constructed from code fragments that already exist in the system. The evaluation of this hypothesis compares how suitable code taken from the same project is versus other projects, and discovers that the same project has a significantly higher likelihood of already containing the new code than other projects. This result complements the results in section 4.3 which

show that good recommendations are more likely to come from within-project, and reinforces the need to include the patient project in the corpus.

Petke et al. [2017] also utilise the transplantation of code fragments, in this case for assisting a genetic improvement algorithm. Here different variations of the same program, each of which has been optimised for different purposes act as a bank of genetic material that can be utilised by a genetic improvement algorithm to improve the original code. The success of this technique also gives evidence for the intuition that programs that have similar purposes should contain good candidate organs for each other in the same way that specialised variants of programs contain good organs for each other. This demonstrates that there may be value in categorising candidate donor systems in an attempt to select donors that are in the same category as the host system.

2.4 Code Similarity

Source similarity and the closely related task of clone detection has generated a great deal of work from the research community. This is due to the usefulness of being able to determine the similarity between code fragments; a technique that is helpful in multiple tasks including refactoring, bug fixing, and plagiarism detection [Ragkhitwetsagul et al., 2017].

The approaches to clone detection can be broken down into categories: text-based, token-based, tree-based, and graph-based. Text-based clone detectors include NiCad [Roy and Cordy, 2008], a tool that detects clones using TXL [Cordy, 2006] with flexible pretty-printing to standardise the formatting of program text, removing noise and making it easier to search. Island grammars to are used to extract and compare potential clones. While lightweight, since NiCad is text-based, it has limited capability to detect type 2 and type 3 clones where the clones are significantly different lexically but may be very close syntactically.

CCFinderX [Kamiya et al., 2002] is an example of a token-based tool. CCFinderX first lexes the code into a sequence of tokens and then applies a rule-based transformation to this token sequence. This transforms the code into a regular form

in which clones can be detected using suffix trees. The token-based representation allows the detection of syntactic clones that would have been missed by lexically driven detectors.

Deckard [Jiang et al., 2007] is a tree-based tool that uses a vector space model to detect similar subtrees. This is applied to clone detection by detecting similar subtrees in an abstract syntax tree (AST). The parse trees are first constructed from the source, processed to produce a set of characteristic vectors that capture the syntactic information, and then clustered to detect clones. The evaluation shows Deckard to have good accuracy and scalability.

In addition to the above, LSI has also been used to perform clone detection [Ragkhitwetsagul et al., 2017, Mills and Haiduc, 2017]. However, all of these tools perform approximately equivalently or worse than the token set ratio similarity method that is currently used by RELATEST, as empirically shown by Ragkhitwetsagul et al. [2017]. Moreover, a comparison of their performance on three clone benchmarks shows that they offer lower search accuracy and performance than Siamese [Ragkhitwetsagul et al., 2017]. A full survey of clone detection techniques can be found in Ragkhitwetsagul et al. [2017].

Another similarity measurement is the normalized compression distance (NCD). NCD is a practical analogue derived from the universal distance metric known as normalized information distance (NID). NCD is based on real-world compressors [Li et al., 2004] and, therefore, can measure the distance for objects of all kinds, such as files [Axelsson, 2010].

NLP also provides an opportunity for measuring the similarity between code, as exemplified in the recent work Zilberstein and Yahav [2016]. This tool, SIMON, uses NLP and large-scale code repositories to establish semantic similarity between snippets of code by utilising type information and natural language descriptions associated with the code. Code snippets that are similar to a query snippet are discovered using the following process: First, a database of code snippet to natural language description pairs is searched to find a code snippet that is semantically similar to the query snippet. The similarity of the natural language descriptions is

then used to infer the semantic similarity between the code snippets. The evaluation presented shows the system performing extremely well, at over 85% precision, recall, and accuracy. However, this result should be approached with caution as the test query snippets were generated by mutating snippets that already existed in the database and, therefore, may be easier to match than snippets from the wild. Despite this concern, the results do still appear to be very good. The largest drawback of this system is that it requires a database of code snippet to natural language description pairs, which is difficult to build and maintain.

The use of vector representations is another NLP technique that presents a clear opportunity for discovering the similarity between code fragments. Vector representations for similarity were first introduced in word2vec [Mikolov et al., 2013], a technique for embedding the syntactic and semantic information of words into vectors. These vectors can be directly compared and operated on to determine the similarities of words or expose the relationships between words. Word2vec was designed for use in natural language contexts, where it has been successfully applied in many systems. Due to this success, a recent work [Alon et al., 2018] has taken the concept and applied it to source code, creating code2vec; a tool that can take a code fragment of arbitrary length and create a fixed-length vector representation that captures the syntactic and semantic information encoded in the fragment. These vectors can be compared in the same fashion as word vectors, such as via cosine similarity, to determine the similarity of the source fragments. The evaluation of code2vec reports precision, recall, and F1 score of 63%, 54%, and 58% respectively. This means that the results for code2vec are not as good as that claimed by SIMON [Zilberstein and Yahav, 2016], but code2vec has the advantage of not needing an expensive database (in terms of construction and maintenance) of code fragment to natural language description pairs; needing only a sufficiently large training set of code fragments, which is easier to obtain.

2.5 Bipartite Edge Prediction

Bipartite edge prediction is a graph theory technique that has been used to tackle many different problems that involve creating links between two or more classes of objects and is used in RELATEST to determine which tests should be recommended to which functions after we model the relationships between the artefacts as a graph using the RASHID framework. There are multiple approaches to bipartite edge prediction, including neighbourhood methods, path methods, and machine learning. The majority of these methods are general graph edge prediction techniques that are not specific to bipartite edge prediction, however, they can be used as such by only accepting predictions made on bipartite edges.

Preferential attachment [Newman, 2001a] is a neighbourhood method that calculates the edge prediction score for a pair of vertices as the product of their degrees. Common Neighbours [Newman, 2001b] is another neighbourhood method that calculates the edge prediction score for a pair of vertices as the number of shared neighbours between the vertices. The Jaccard's Coefficient [Salton and McGill, 1986] neighbourhood method uses the number of shared neighbours between the vertices divided by the total number of neighbours for both vertices. Adamic/Adair [Adamic and Adar, 2003] is a variation on common neighbours which weights the effect of neighbour vertices to be inversely proportional to the log of their degree.

The common path methods include an adapted version of page rank [Brin and Page, 1998] that calculates the significance of each vertex and makes predictions under the assumption that it is beneficial to link to significant vertices. Rooted page rank [Liben-Nowell and Kleinberg, 2007] adapts the page rank method to calculate the edge prediction scores between two vertices as the probability of visiting one vertex during a random walk from the other vertex. PropFlow [Lichtenwalter et al., 2010] is another path-based method that calculates edge prediction scores by starting with a source vertex with score 1 and propagating the score outwards, at each step assigning an equal proportion of the score to each of the vertices encountered. For example, if the source vertex has 4 neighbours, each of the

neighbours will be assigned a score of 0.25 on the first step. On the second step, these scores will again be propagated to the neighbours of those vertices and so on. Where a vertex already has a score, any incoming score will be added to the existing score. This mechanism encodes the intuition that vertices that have more than one path to the source vertex should be favoured when deciding which vertices should have an edge to the source vertex.

Machine learning has been applied to edge prediction using unsupervised, semi-supervised, and supervised techniques. [Benchettara et al., 2010] presents a supervised learning approach to edge prediction that uses the Preferential Attachment, Common Neighbours, Adamic/Adar, and Jaccard Coefficient scores between pairs of vertices as the features to learn from. As these features are not specific to bipartite edge prediction, they firstly compute these features over only the bipartite graph and label these direct features, then compute the features over the graphs projected over each of the two vertex sets, labelled indirect features. These two feature sets are then combined to make the final bipartite edge predictions. The evaluation of this technique, however, is not complete as it does not compare the results with other common techniques, instead only demonstrating that using a combination of the direct and indirect features is better than using only the direct features.

Davis et al. [2011] present firstly, a new bipartite edge prediction technique derived from the standard neighbourhood methods called multi-relational link prediction (MRLP) and secondly, a supervised learning approach. The goal of MRLP is to adapt neighbourhood and path methods to specifically target bipartite edge prediction. This is achieved by discovering incomplete 3-node subgraphs, named partial triads, and calculating the probability that the partial triad should be completed by each edge type. This is done by assigning a weighting for each edge type to the common neighbours of the two non-neighbour vertices based on their own connected edge types and the edge types of the path between the two vertices and the common neighbour. The supervised learning approach presented by Davis et al. [2011] use the High-Performance Link Prediction (HPLP) system

[Lichtenwalter et al., 2010] to capture graph features for each pair of unconnected vertices. These features include the scores of typical neighbourhood and path methods, such as Common Neighbours, Jaccard's Coefficient, Adamic/Adair, and PropFlow. These features are used to learn to classify potential edges as one of the edge types or no edge. The evaluation presents a comprehensive comparison of the MRLP and its associated supervised learning approach with the standard neighbourhood and path methods: Preferential Attachment, Common Neighbours, Adamic/Adar, Jaccard Coefficient, Page Rank, Rooted Page Rank, and PropFlow techniques. The effectiveness of each technique is evaluated over three large datasets and reveals that on average MRLP and Jaccard Coefficient are the most effective out of the non-learning methods and the supervised learning method slightly outperforms both of them.

Another more recent machine learning approach to bipartite edge prediction that is particularly applicable to RELATEST is cross-graph learning. Liu and Yang [2015] presents a semi-supervised transductive learning technique that utilises cross-graph information to make bipartite edge predictions. This work formulates the edge prediction problem as a label propagation problem over the product graph between the two unipartite graphs. In this formulation, each vertex in this product graph represents a possible bipartite edge between the two unipartite graphs; the vertices representing known edges are labelled and semi-supervised learning is applied to determine the labels of the other vertices. Liu and Yang [2016] extends Liu and Yang [2015] to allow application of the technique to multipartite graphs with more than two component unipartite graphs. The cross-graph information is encoded in a Tucker decomposition [Tucker, 1966] of the product graph over the unipartite graphs, which is then used to carry out semi-supervised learning over the total bipartite edge set. The evaluation of this technique does not compare its performance to neighbourhood or path methods but does demonstrate superior performance to other machine learning techniques, specifically, tensor factorisation, one class nearest neighbour, ranking support vector machines, and low-rank tensor kernel machines. This technique was selected to be implemented as it uses semi-

supervised transductive learning and therefore should be more resilient to sparsity in the training data compared to inductive supervised learning techniques.

2.6 Unit Test Code Generation

Prior to the application of the machine learning techniques presented in this thesis, unit test generation was done primarily by traditional test suite generation tools. These tools can be split into distinct categories depending on the general approach used, including random generation, search-based, and analysis-based approaches.

The primary examples of tools that use approaches based on random generation are Randoop [Pacheco and Ernst, 2007], Nighthawk [Andrews et al., 2007], JCrasher [Csallner and Smaragdakis, 2004], and CarFast [Park et al., 2012]. The most well-used tool in this category, Randoop, uses feedback-directed random test generation. Randoop implements this by iteratively building sequences of method calls via random selection until a user-specified contract is violated or an implicit failure oracle is triggered, such as a crash or exception. Sequences that violate a contract are output as contract-violating tests and sequences that exhibit normal behaviour are output as regression tests. Randoop can also generate regression assertions by asking users to annotate the methods for which they wish assertions to be generated. These are regression asserts as they utilise the current output of the method as the oracle. Nighthawk generates tests by randomly building sequences of method calls using a set of parameters that include the number of method calls to make for each test, the number of tests to generate, and the frequency with which each method should be called. This set of parameters is used as the genome for a genetic algorithm that is applied to maximise a coverage-based fitness function. JCrasher analyses classes to randomly generate inputs to methods that obey the type system and uses these inputs to construct sequences of method calls. Crashes, specifically unexpected exceptions, are used as the implicit oracle that a failure has occurred. As not all exceptions necessarily represent a program failure but could instead be an expected result of a pre-condition violation, JCrasher uses a heuristic to attempt to determine when an exception is considered unexpected and,

therefore, is a failure. CarFast employs an approach that mixes static analysis with random generation in an attempt to maximise coverage more quickly than typical purely random approaches. To do this, CarFast first performs a static analysis to determine which branches contain the most lines of code and then uses constraint-based selection to select inputs that guide the execution towards these branches.

Search-based tools include EvoSuite [Fraser and Arcuri, 2013] and eToc [Tonella, 2004], which utilise forms of meta-heuristic search guided by coverage goals. The most widely studied and best-developed tool in this category, EvoSuite, utilises a genetic algorithm with multiple branch coverage goals to concurrently evolve a whole test suite. The EvoSuite approach starts with an initial randomly generated population of test suites and applies the genetic algorithm to this population using the coverage goals as its primary fitness function with a secondary goal of minimising test suite size. After the genetic search budget has been expended, regression test assertions are then added using mutation analysis [Fraser and Zeller, 2012]. These asserts are considered to be regression asserts only as they use the current behaviour of the class under test as the oracle. As a final step, the tests are minimised by iteratively removing statements until coverage is reduced, ensuring that any statements which do not contribute to the coverage goal are not included.

jCUTE [Sen and Agha] and Symbolic Pathfinder [Păsăreanu and Rungta, 2010] are examples of analysis-based tools which attempt to generate inputs that cover the largest number of code paths. jCUTE uses concolic execution, a mixture of concrete and symbolic execution, iteratively, where first a random concrete input is generated and a path constraint is constructed as the program executes with that concrete input. The tool then backtracks and uses a constraint solver on the path constraint to generate a new input. This process continues until all the paths have been explored. Symbolic Pathfinder predominantly utilises dynamic symbolic execution with constraint solvers to try to find inputs that cover the most branches but support is also provided for concolic execution with a given input. As these analysis-based tools are complicated to use, rely on path constraints being solvable, and have difficulty scaling, they are less widely used and less well studied than the

other categories of test generation tools.

2.7 Conclusion

The previous work on test-to-code traceability provides a good foundation by establishing a baseline of several straightforward approaches to the establishment of these links. However, there is much room for further novel contributions and improvements to be made. The investigation by Ståhl et al. [2017] 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. Rompaey and Demeyer [2009] investigate mostly static techniques, using tracing only to establish LCBA and only investigate establishing links on the class level. We improve on this by utilising both static and dynamic techniques in concert on both the class and the method levels. We also utilise much larger ground truths for the evaluations. For Ghafari et al. [2015], while the results presented are promising, two of the four subject projects used for the evaluation are very small (130 and 43 tests) and 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 our work. In Hurdugaci and Zaidman [2012], like our approach, tracing is used to identify the methods that are called by a test. However, no further filtering is done and their approach will thus include a large number of utility methods leading to low precision.

The previous work on code search and recommendation demonstrates that the contribution to artefact discovery provided by the Rashid framework is novel as no existing work, to the best of our knowledge, has defined a general model for using the within-domain and cross-domain relationships between artefacts to find appropriate existing artefacts for reuse. Erfani et al. [2013] is the closest known work to the Rashid approach, however, the approach presented by Erfani et al. [2013] is extremely limited and does not generalise. Further, the quality of the recommendations produced cannot be determined and the number of recommenda-

tions produced is very low.

The previous work that relates most closely to the transplantation contribution is the work of Barr et al. [2015] which uses a similar approach to transplantation. However, the contribution made by RELATEST in this area is novel as RELATEST is concerned with the transplantation of tests and thus tackles a different research problem. Achieving the successful transplantation of tests requires some solutions not given by the existing work, such as how to guide a genetic search for transplantation without the assistance of a test suite, which will be the primary contribution of the transplantation component of RELATEST. RELATEST is also an improvement over the existing work in terms of the level of automation achieved as the artefact search component, facilitated through the RASHID framework, automatically finds the tests to transplant and therefore removes the need for the manual identification of an entry point.

The previous work on test code generation has led to a diversity of tools for the generation of unit tests, however, with the advent of machine learning for software engineering we have seized the opportunity to employ neural networks for this task for the first time. Additionally, the previous tools all focus primarily on things other than the generation of meaningful oracles and asserts. The typical goals for the existing tools are maximising test coverage, executing the most code paths, or exposing faults in other ways, such as generating exceptions and crashes. Therefore, for even the most well developed and studied examples of these tools which have some form of assert generation, such as EvoSuite and Randoop, the asserts they generate are often trivial or not meaningful, contributing to the relatively high rate of missed faults in real-world projects [Shamshiri, 2015, Shamshiri et al., 2015]. For example, while Randoop can generate asserts, it requires the user to explicitly annotate the methods which they wish assertions to be created for. Also, only regression assertions can be generated, and the diversity of these assertions is limited. These weaknesses are also present in the EvoSuite asserts as they use mutation analysis with the current behaviour of the software and can not determine if an assert is trivial. The problem of test generation tools producing poor asserts

also has been reported by developers [Almasi et al., 2017] with quotes such as *"... poor assertions, sometimes there is an assertion and sometimes there is not? The assertions are mostly checking for simple stuff like list size and so on"*. We address these shortfalls by targeting ReAssert specifically for the generation of meaningful and diverse asserts that can reveal real-world faults. One of the key motivations for our approach of using neural networks to learn from existing test code is that we can learn the patterns of asserts and oracles explicitly written by developers, potentially helping to side step the oracle problem and produce asserts that developers would write themselves.

In addition to the techniques that make up the contributions, Relatest utilises a selection of techniques from other research areas to perform its tasks of finding and transplanting tests; specifically test-to-code traceability, code similarity, and bipartite edge prediction. For test-to-code traceability, the LCBA and NC techniques have been selected as Rompaey and Demeyer [2009] shows those techniques performing the best in a combined evaluation of six different techniques. To compute the similarity between two fragments of code, the 3-gram Jaccard Index was selected as a recent investigation [Ragkhitwetsagul et al., 2017] show that textual similarity measurements can perform well on source code with modifications and using Ragkhitwetsagul et al.'s framework [Ragkhitwetsagul et al., 2017], we have confirmed that using the Jaccard index over 3-grams as implemented in the Java String Similarity library performs better than most of the 30 algorithms as given in the paper. For bipartite edge prediction, a selection of techniques from the literature were tested but, so far, none have outperformed the triangle method used by Relatest. The details of this investigation are provided in subsection 4.3.7.

Chapter 3

Establishing Test-to-Code Traceability Links

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 [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. White and Krinke [2018], Watson et al. [2020], White and Krinke [2020b] 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.

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 [Rompaey and Demeyer, 2009, Qusef et al., 2014, Gethers et al., 2011, Kicsi et al., 2018, Csuvik et al., 2019b,a]. 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]. 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.

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 chapter, 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, 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, 2020a], 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

[1] Available at `https://github.com/RRGWhite/tctracer`.
[2] Evaluation artefacts available at `https://doi.org/10.5281/zenodo.4608587`.

the accuracy of the approach.

The main contributions of this chapter 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.
- 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, 2020a] of test-to-function and test-class-to-class links.

3.1 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). 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 [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.

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., 2019b]. 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

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

of the individual techniques, collectively overcomes their weaknesses. This is the approach utilised in TCTracer and presented in this chapter.

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.

3.2 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 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 3.6.2.5. 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.

3.3 Techniques

As discussed in Section 3.1, 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) [Rompaey and Demeyer, 2009] and Last Call Before Assert (LCBA) [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 3.3.1) and static (Section 3.3.2) 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., 2019b], 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 3.1. 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) [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 evaluations performed in previous work have shown a precision and recall of below 50% [Rompaey and Demeyer, 2009].

3.3.1 Dynamic Techniques

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

3.3.1.1 Naming Conventions

As naming conventions can change between projects [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*.

$$\text{score}(t, f) = \begin{cases} 1, & \text{if } n_t \text{ equals } n_f \\ 0, & \text{otherwise} \end{cases} \tag{3.1}$$

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:

$$\text{score}(t, f) = \begin{cases} 1, & \text{if } n_f \text{ substring of } n_t \\ 0, & \text{otherwise} \end{cases} \tag{3.2}$$

3.3.1.2 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:

$$\text{score}(t, f) = \frac{|\text{LCS}(n_t, n_f)|}{\max(|n_t|, |n_f|)} \tag{3.3}$$

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.

$$\text{score}(t, f) = \frac{|\text{LCS}(n_t, n_f)|}{|n_f|} \tag{3.4}$$

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:

$$\text{score}(t, f) = 1 - \left(\frac{\text{Levenshtein}(n_t, n_f)}{\max(|n_t|, |n_f|)} \right) \tag{3.5}$$

3.3.1.3 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:

$$\text{score}(t, f) = \begin{cases} 1, & \text{if } f \text{ is last return before an assert in } t \\ 0, & \text{otherwise} \end{cases} \tag{3.6}$$

3.3.1.4 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:

$$\text{suspiciousness}(e) = \frac{\frac{\text{failed}(e)}{totalfailed}}{\frac{\text{passed}(e)}{totalpassed} + \frac{\text{failed}(e)}{totalfailed}} \tag{3.7}$$

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 Equation 3.8:

$$\text{score}(t, f) = \frac{1}{\frac{|\{t' \in T : f \in t'\}|-1}{|T|-1} + 1} \tag{3.8}$$

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.

3.3.1.5 Term Frequency–Inverse Document Frequency (TFIDF)

Term frequency–inverse document frequency (TFIDF) [Manning et al., 2010] 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:

$$\text{score}(t, f) = \text{tf}(t, f) \cdot \text{idf}(f) \tag{3.9}$$

The usual definition of the term frequency (tf) function does not match the

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

test/function scenario. Thus, tf and idf are defined as:

$$\mathrm{tf}(t, f) = \ln\left(1 + \frac{1}{|\{f' \in F : f' \in t\}|}\right) \tag{3.10}$$

$$\mathrm{idf}(f) = \ln\left(1 + \frac{|T|}{|\{t' \in T : f \in t'\}|}\right) \tag{3.11}$$

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.

3.3.2 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.

3.3.2.1 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 because 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.

$$\text{score}(t, f) = \begin{cases} 1, & \text{if } n_t \text{ equals } n_f \wedge\ n_{tc} \text{ equals } n_{fc} \\ 0, & \text{otherwise} \end{cases} \tag{3.12}$$

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.

$$\text{score}(t, f) = \begin{cases} 1, & \text{if } n_f \text{ substring of } n_t \wedge\ n_{tc} \text{ substring of } n_{fc} \\ 0, & \text{otherwise} \end{cases} \tag{3.13}$$

3.3.2.2 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 3.3.1. 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.

3.3.3 Score Scaling

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

3.3.3.1 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:

$$\text{score}_\text{d}(t, f) = \text{score}(t, f) \cdot \gamma^{(dist(t,f)-1)} \tag{3.14}$$

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.

3.3.3.2 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:

$$\text{score}_\text{n}(t, f) = \frac{\text{score}_\text{d}(t, f)}{\max(\{\text{score}_\text{d}(t, f') \mid f' \in t\})} \tag{3.15}$$

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

Table 3.1: 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

In the end, we focus on thirteen individual techniques, shown in Table 3.1. 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. The thresholds (τ) are used as the prediction boundary to determine which scores predict a link. Specifically, scores above the threshold predict a link between the test and the function, whereas scores below the threshold predict there is no link between them. The selection and application of the thresholds is discussed further in Section 3.6.4.

3.4 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.

3.4.1 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:

$$\mathbf{M} \in \mathbb{R}^{|\mathrm{T}|\times|\mathrm{F}|} \tag{3.16}$$

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 Equation 3.15.

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 3.1), and consider scores above the threshold as positive link predictions. Equation 3.17 defines how each set of method level traceability links are constructed.

$$\mathrm{LM} = \{(t, f) \in \mathrm{T}\times\mathrm{F} \mid \mathbf{M}_{tf} \geq \tau\} \tag{3.17}$$

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

3.4.2 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:

$$\mathbf{C} \in \mathbb{R}^{|\mathrm{TC}|\times|\mathrm{FC}|} \tag{3.18}$$

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

Similarly to the method level, $\mathbf{C}$ is used to compute sets of class level traceability links using Equation 3.19.

$$\text{LC} = \{(c_t, c_f) \in \text{TC} \times \text{FC} \mid \mathbf{C}_{c_t c_f} \geq \tau\} \tag{3.19}$$

3.4.3 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:

$$\mathbf{EM}(c_t, c_f) \in \mathbb{R}^{|\text{t}(c_t)| \times |\text{f}(c_f)|} \tag{3.20}$$

Where $\text{t}(c_t)$ is the set of tests in test class c_t, $\text{f}(c_f)$ 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 $(t, f) \in (\text{t}(c_t) \times \text{f}(c_f))$.

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:

$$\mathbf{EM} \in \mathbb{R}^{|\text{TC}| \times |\text{FC}|} \tag{3.21}$$

Therefore, each element of $\mathbf{EM}$ 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 $\mathbf{EM}$ are normalised using Equation 3.15.

The scores in $\mathbf{EM}$ are then used to produce a set of class level predicted links

using Equation 3.22.

$$\text{LEM} = \{(c_t, c_f) \in \text{TC} \times \text{FC} \mid \mathbf{EM}_{c_t c_f} \geq \tau\} \tag{3.22}$$

3.4.4 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 Equation 3.23.

$$\mathbf{AM}_{tf} = \mathbf{M}_{tf} \cdot \mathbf{C}_{\text{c}(t)\text{c}(f)} \tag{3.23}$$

Where $\text{c}(f)$ returns the class containing function f.

From **AM**, the set of augmented method level traceability link predictions are produced using Equation 3.24.

$$\text{LAM} = \{(t, f) \in \text{T} \times \text{F} \mid \mathbf{AM}_{tf} \geq \tau\} \tag{3.24}$$

3.5 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 3.3.1.

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

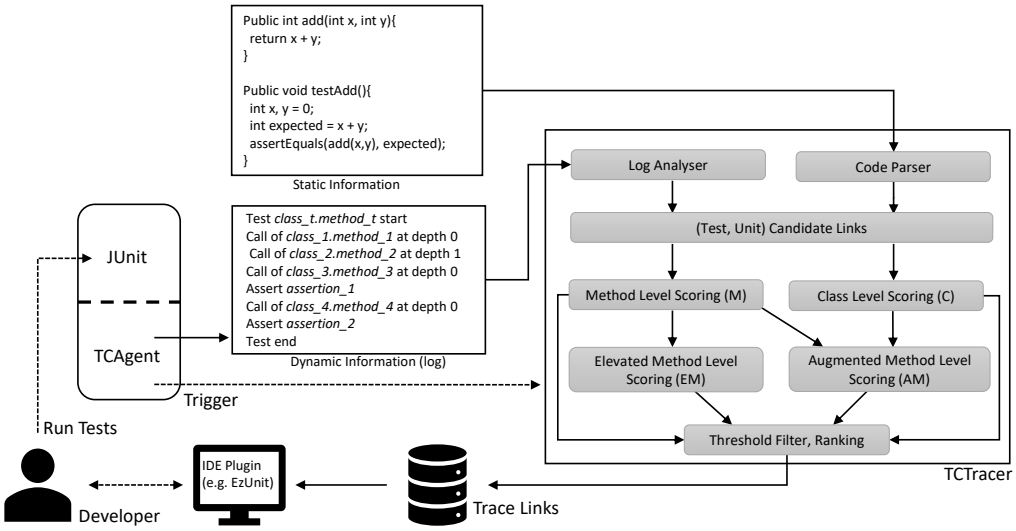

Figure 3.1: Integration of TCTracer into JUnit.

access to the bytecode of Java classes and allows for them to be transformed before being loaded by the JVM. As shown in Figure 3.1, 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.

The main challenge of working with static information is the number of test-

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

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 3.4 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.

3.6 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.

3.6.1 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. To perform the selection, we extracted all the tests from the subjects, assigned each test a number from 1 to the number of tests, and then used a random number generator to select 100 tests in total. To make the 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

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

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 of 20 test classes was drawn from the set of all test classes by assigning every test class a number from 1 to the number of test classes and then using a random number generator to select the test classes for the sample uniformly at random. The tested classes for the test classes in this sample were the 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 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

[10] `https://github.com/google/gson`
[11] `https://ant.apache.org/`

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 3.2.

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., 2019b] 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

Table 3.2: Subject statistics.

Project	Ver.	Num. Func.	Num. Tests	Instr. Coverage	Num. Method Level Ground Truth Links	Num. 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	-

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 [Davis and Goadrich, 2006]. All scores are presented as integer percentages for the sake of readability.

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.

3.6.2 Research Questions

In the following section, 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 3.7.

3.6.2.1 Research Question 1 (Method Level)

How effective are our techniques at the method level?

This research question investigates how effective each of the techniques is 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 3.17.

Findings. From the results for RQ1, shown in Table 3.3, 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 precision alone, NC is the best, while LCS-B is best for recall. When comparing variants, the dynamic techniques consistently outperform the static techniques.

3.6.2.2 Research Question 2 (Class Level)

How effective are our techniques at the class level?

This research question investigates how effective each of the techniques is 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 3.19.

Findings. From the results for RQ2, shown in Table 3.4, 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.

3.6.2.3 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 Equation 3.22.

Table 3.3: 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
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

Table 3.4: 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
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

Findings. From the results shown in Table 3.5 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.

3.6.2.4 Research Question 4 (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 compute the evaluation measures over the link sets produced using Equation 3.24.

Findings. The results for RQ4, shown in Table 3.6, 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.

3.6.2.5 Research Question 5 (Technique Combination)

As described in section 3.4, 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.

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

Table 3.5: 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
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

Table 3.6: 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
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

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 work, 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 3.6.1, 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 Table 3.7 and Table 3.8, 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 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 under-performs no weighting,

Table 3.7: 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

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.

3.6.2.6 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

Table 3.8: 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

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 3.9.

Findings. The results for RQ6, shown in Table 3.10, 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.

Table 3.9: 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.

Table 3.10: 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

3.6.3 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

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

Table 3.11: 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

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 3.11, 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.

3.6.4 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 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.

3.6.5 Call Depth Discounting Analysis

As discussed in Section 3.3.3.1, 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 3.12, 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. One such example is the function *XmlStreamWriter.detectEncoding(char[], int, int)* which is tested by multiple tests that call it through other forwarding functions. *XmlStreamWriterTest.testLatin7Encoding()* is one such test that first calls *XmlStreamWriterTest.checkXmlWriter()* which then calls an overloaded version of itself which calls *XmlStreamWriterTest.checkXmlContent()* which then calls *XmlStreamWriter.write(char[], int, int)* which in turn calls the tested function. 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 3.12 shows the number

Table 3.12: 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 3.13: 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%)

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 3.13 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.

3.7 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.

3.7.1 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 Porubän [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. 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.

3.7.2 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.

3.7.3 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.

3.7.4 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 3.6.2.5, 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.

3.7.5 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 3.6.5.

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.

3.7.6 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 [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.

3.7.7 Traceability Integration

As shown in Figure 3.1, 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, 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].

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

3.7.8 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 3.6.1. As Trautsch et al. [2020] conclude, 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.

3.7.9 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] use a static version of LCBA which in our evaluation only achieved 63% precision and 62% recall. Another approach [Villmow et al., 2021] use 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.

3.8 Threats to Validity

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

3.8.1 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.

While our work 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 3.6.1. It would be interesting to separate integration tests and unit tests in our evaluation, however, Trautsch et al. [2020] observe 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.

3.8.2 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.

3.8.3 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]. The work presented in this chapter was performed in line with research ethics at UCL [UCL Research Ethics Committee].

3.9 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. [2014] describe these benefits in detail and [Parizi et al., 2014] present 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.

At the method level, 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 chapter.

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.

Other test-to-code traceability work is based on the assumption that a test should be similar to a tested unit. 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. 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. [2019a] 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., 2019b] 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 and Porubän [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.

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.

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 for producing the final traceability links.

Ståhl et al. [2017] focus 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. 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.

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 a 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 difference in predicted links between different techniques. This further demonstrates the potential applicability of test-to-code traceability links and the appetite for their usage.

3.10 Conclusion

In this chapter, 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.

Chapter 4

Reuse of Unit Tests

Reuse is a core pillar of software development that delivers multiple benefits including an increase in development productivity and a reduction in the cost and development time of software projects. Reuse can also improve the quality of the resultant software as reused artefacts tend to be more mature and well tested than newly created artefacts. For example, developers often create new tests from existing tests by copying and adapting them. However, reuse opportunities are often missed due to the cost of discovering suitable artefacts to reuse.

Discovering relationships between artefacts facilitates reuse as software artefacts are linked to each other by their relationships, for example, a test for a function, or a design artefact for a requirement. Revealing new connections between existing artefacts can, therefore, be used to discover situations where artefacts may be reused. For example, an existing test may be discovered and adapted to test a new function, or an existing function may be discovered that can be adapted, or even used directly, to fulfil a new requirement. The discovered relationships may also be useful for traceability establishment, such as discovering which regulatory codes are relevant to a particular requirement. Development artefacts form groups that have both internal connections between artefacts of the same type, and cross-group connections between artefacts of different types.

This chapter presents RASHID[1], an abstract framework for facilitating the reuse of software artefacts by modelling the relations between artefacts of two different

[1]Rashid (meaning "guide"): the Arabic name of Rosetta, the town where the Rosetta Stone was discovered.

types using what we define as an artefact relation graph: a graph with a bipartite subgraph connecting the artefacts of the two types. We also present a tool, RELATEST, that instantiates RASHID and utilises the traceability links established by TCTRACER (presented in Chapter 3) to partially automate the reuse of existing unit tests.

Test reuse was selected to serve as the practical example for this approach as many modern software systems struggle to maintain a high level of test coverage, especially as projects grow in size and complexity. This is in part due to the heavy burden that writing many unit tests places on the developers and the de-prioritisation of this work in the presence of tight release deadlines. The failure to maintain an adequate level of test coverage can result in large numbers of bugs going undetected for long periods of time, jeopardising the correctness and reliability of the system.

RELATEST assists in test reuse by using RASHID to discover existing tests that are closely related to a new function and can therefore be recommended and adapted to test the new function. When the recommended test is close to the needed test, RELATEST saves time and avoids introducing new faults that might have been introduced if the developer had written a new test from scratch.

RELATEST also provides a unique benefit over test generation tools in that the recommended tests contain human written elements that are difficult or impossible for test generation tools to produce, such as oracles and specific test inputs.

An investigation of the quality of RELATEST's recommendations showed that developers consider 65% of its recommendations to be useful and, when using the token-based edit distance to known tests as a proxy for effort, represent a 58% reduction in developer effort versus writing tests from scratch. When considering only top-ranked recommendations, the chance of being considered useful by a developer rises to 91% and the reduction in developer effort rises to 66%. When considering the rate of functions that receive recommendations, this results in a 43% reduction in the total effort required to create tests. A user study revealed that, on average, developers needed 10 minutes less to develop a test when given RELATEST recommendations and all developers reported that the recommendations were useful. We demonstrate the applicability of RELATEST in a real-world application

by using RELATEST to create twelve tests for a large open source project, yielding an average 45% saving of effort.

The main contributions of this chapter are:

- RASHID, an abstract framework for the reuse of artefacts using artefact relation graphs.
- RELATEST, a instantiation of RASHID to recommend existing tests to be reused for new functions.
- An evaluation of the effectiveness of RELATEST which shows a 43% overall reduction in effort to create tests across a project.

Our evaluation artefacts are available at:
`https://figshare.com/s/2e54394880818d6d32dc`.

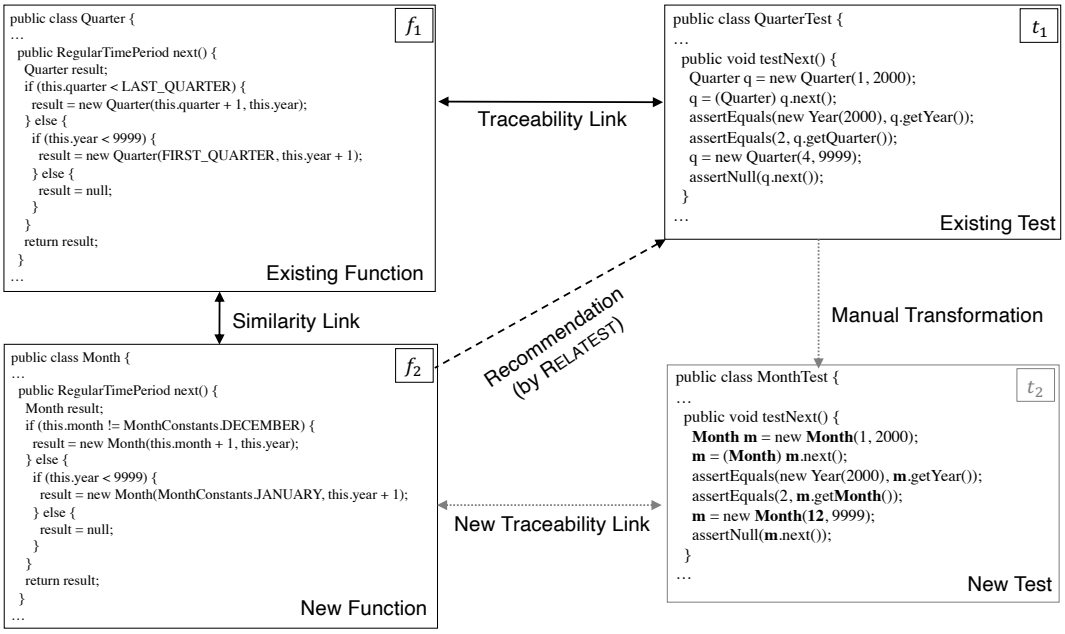

Figure 4.1: Unit Test Recommendation using RELATEST.

4.1 Approach

This section presents the approach of RASHID and its realisation in RELATEST. The overall idea of RASHID is to use existing relationships between the artefacts of two domains and the domain-intrinsic relationships between the artefacts to recommend

new relationships across the two domains. For example, if two artefacts are related across domains, then a relationship to similar artefacts is recommended.

4.1.1 Example

Figure 4.1 shows how test reuse through RELATEST can be applied to a real-world example from the JFreeChart system. In this example, the developer has just implemented a new function (f_2) implementing `next` for a new class and requires a new unit test for it. Without RELATEST, the developer would need to either write the new test from scratch or manually search through a potentially very large codebase to find an appropriate test to reuse: both time consuming and repetitive tasks. RELATEST streamlines this process by finding f_1, a function similar to f_2 (actually implementing `next` in a different class) in the corpus and exploiting the traceability link between f_1 and t_1 (the unit test `testNext`) to recommend t_1 to the developer as the starting point for a test for f_2 (the new unit test). In Figure 4.1 the developer needs only to make a few edits to transform t_1 (the old unit test) into the new test t_2 to test f_2 and all but one of the changes (marked in bold) are simple type replacements or variable renames to match the new type.

4.1.2 Artefact Reuse Framework

Figure 4.2 illustrates RASHID, which utilises an artefact relation graph to model the intra- and inter-domain relationships for a set of artefacts over two distinct domains. The artefact relation graph is defined as $G = (V_1, V_2, E_1, E_2, E_B, E_P)$ where V_1 and V_2 are the sets of vertices representing the artefacts in domain 1 and domain 2 respectively; E_1 is the edge set that contains the edges between the vertices in V_1, and E_2 is the edge set that contains the edges between the vertices in V_2. E_1 and E_2 model the intra-domain relationships between the artefacts. E_B is the edge set that contains the edges of the bipartite subgraph, that is the edges between the vertices in V_1 and the vertices in V_2. E_B models the inter-domain connections between the artefacts. E_P is a set of predicted bipartite edges that we will construct using E_1, E_2, and E_B.

To construct E_1, E_2, and E_B, we define three binary relations (R_1, R_2, R_B), one

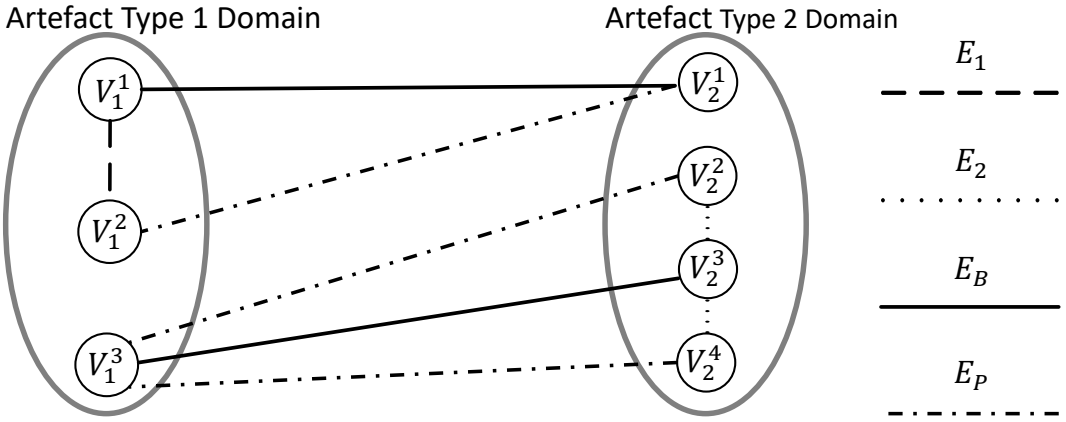

Figure 4.2: Artefact Reuse Framework.

for each of the three edge sets, such that two vertices will be connected by an edge if they satisfy the relation. For E_1: $(v, v') \in E_1$ if vR_1v' $(v, v' \in V_1)$. For E_2: $(v, v') \in E_2$ if vR_2v' $(v, v' \in V_2)$, and for the bipartite edges E_B: $(v, v') \in E_B$ if vR_Bv' $(v \in V_1, v' \in V_2)$. In the case of R_1 and R_2, the relation is defined over pairs of the same specific type of artefact. For example, for code artefacts, the relation may be similarity, where the vertices representing two functions satisfy the relation if the code similarity between the artefacts is above a certain threshold.

The relation R_B that constructs the bipartite edge set E_B is over pairs of artefacts of different types and will therefore be defined in terms of a traceability technique that establishes links between artefacts of the two types, for example naming conventions [Rompaey and Demeyer, 2009] for test-to-function links, or a tracing network for requirement-to-design-element links [Guo et al., 2017]. The relation would then be satisfied if the artefacts represented by the two vertices were identified as linked by the traceability technique.

After E_1, E_2, and E_B have been constructed, Rashid predicts a set of new edges E_P. The edges in E_P can reveal inter-domain artefact connections that could not be discovered otherwise. The newly revealed connections present opportunities to reuse artefacts but can also be considered as predicted traceability links and therefore used as a method for combating the missing link problem in traceability.

Since the edge prediction is being performed over a bipartite subgraph, the edge prediction techniques that can be applied are not limited to only those applicable in strictly bipartite graphs, as the non-bipartite edges provide extra

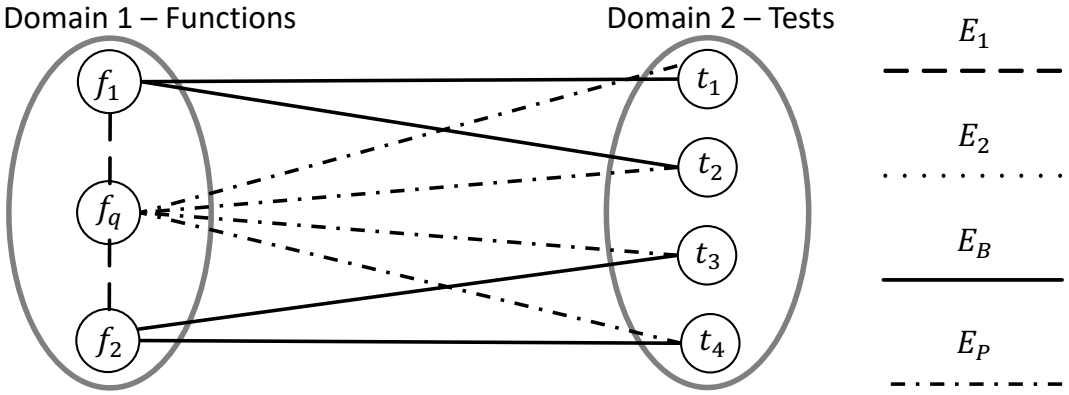

Figure 4.3: Example framework instantiation for RELATEST.

information. This means that almost any edge prediction method can be used, including neighbourhood methods, such as Common Neighbours and Jaccard's coefficient [Nigam and Chawla, 2016], path methods, such as Page Rank [Liben-Nowell and Kleinberg, 2007], or supervised machine learning [Benchettara et al., 2010]. The edge prediction method used in Figure 4.2 is a neighbourhood method that predicts a bipartite edge in E_P where the addition of that edge creates a semi-bipartite 3-vertex clique, consisting of two vertices from the same domain and one vertex from the other domain. This clique forms an inter-domain triangle and this method will be referred to as the triangle method hereafter. Simplified, an edge (v, v') is added to E_P if $vR_1v''R_Bv'$.

4.1.3 RELATEST

Figure 4.3 shows how the RELATEST approach uses RASHID to make test recommendations for functions. RELATEST utilises a code corpus consisting of a set of unit tests $T = \{t_1, t_2, ...t_n\}$, where n is the total number of test functions, and a set of functions $F = \{f_1, f_2, ...f_m\}$, where m is the total number of functions. This corpus is used to build a set of traceability links L that maps tests to tested functions:

$$L = \{(t, f) \in T \times F \mid t \text{ tests } f\} \tag{4.1}$$

When we want to make recommendations for a new function, hereafter referred to as the query function f_q, we instantiate RASHID so that V_1 contains a vertex for each function in $F \cup \{f_q\}$, and V_2 contains a vertex for each test in T. The relation

R_1 for building the edge set E_1 is a similarity relation, where R_1 is satisfied by a pair of vertices ($v \in V_1, v' \in V_1$) if the function represented by v and the function represented by v' are similar to each other above a certain threshold, formally vR_1v' if $\text{sim}(A(v), A(v')) \geq \tau$, where $A(v)$ returns the artefact represented by vertex v, $\text{sim}(a_1, a_2)$ returns the code similarity between artefacts a_1 and a_2, and τ is a similarity threshold. The technique used for implementing the similarity function is dependent on the implementation and is discussed in Section 4.2.2. E_2 is assigned the empty set as in this instance we do not use intra-domain edges in the test domain.

The relation R_B used to construct the inter-domain edges E_B is defined using the traceability link set L so that a pair of vertices (v, v') where $v, v' \in V_1 \cup V_2$ satisfy the relation if the artefacts represented by those vertices are linked in L, formally, vR_Bv' if $(A(v), A(v')) \in L$.

Now that we have E_1, E_2, and E_B, we apply the triangle method for bipartite edge prediction, as described in Section 4.1.2, to construct our set of predicted edges E_P. The set E_P links all functions to the tests of similar functions.

We use the completed artefact relation graph to discover recommendations by constructing a ranked list of recommendations $R(f_q)$ for our query function f_q. First, we build a list $S(f_q)$ which ranks the functions that are neighbours of f_q in the graph in descending order of similarity to f_q so that:

$$\underset{1 \leq i < |S(f_q)|}{\forall} \text{sim}(f_q, S(f_q)_i) \geq \text{sim}(f_q, S(f_q)_{i+1}) \tag{4.2}$$

Where $\text{sim}(f_1, f_2)$ is the similarity between f_1 and f_2.

Given the list of similar functions $S(f_q)$ and the inter-domain edges, the recommendation list is built by iterating through the elements of $S(f_q)$ and for each member $f \in S(f_q)$ constructing the set of tests T_f that are neighbours with f:

$$T(f) = \{t \in T \mid \exists (V(f), V(t)) \in (E_B \cup E_P)\} \tag{4.3}$$

Where $V(a)$ returns the vertex representing artefact a.

The elements in $T(f)$ are added to $R(f_q)$ until $T(f)$ has no more elements or

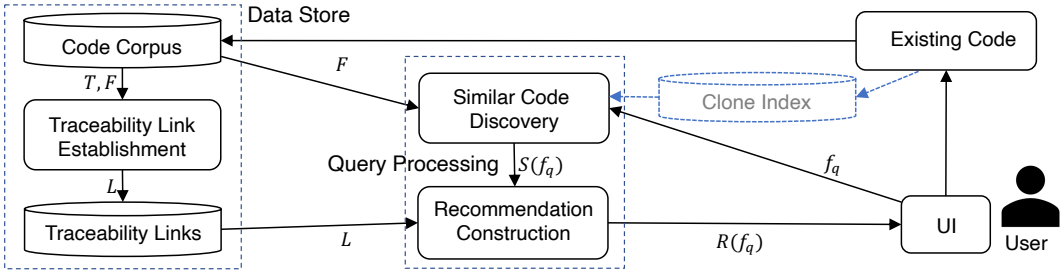

Figure 4.4: System diagram. T is all tests from the corpus, F is all functions from the corpus, L is all traceability links, f_q is the query function, $S(f_q)$ is the list of functions similar to the query function, and $R(f_q)$ is the list of test recommendations for f_q.

because we also want to place an upper bound k on the size of the recommendation list, if $|R(f_q)| = k$. Otherwise, the next element in $S(f_q)$ is selected and the process continues. Once this process terminates, the recommendation list $R(f_q)$ is presented to the user.

4.2 Implementation

Figure 4.4 shows the overall architecture of RELATEST. The core of RELATEST is comprised of two component groups; the data store and the query processing. The data store builds and maintains the corpus of existing functions and tests, establishes test-to-function traceability links from the code corpus, and stores those links in the traceability link database. Query processing takes the query function, discovers similar functions, and builds the test recommendations. The UI allows the user to specify the locations of the existing code to add to the corpus, input the query function, and view the recommendations. To discover similar functions, one may rely on clone detectors that build a clone index over existing code.

One key distinction between the data store and query processing is that the data store is constructed once and only updated when the user changes the existing code. The query processing components are invoked for every individual query.

4.2.1 Data Store: Traceability Link Establishment

The correct selection and configuration of the traceability link establishment method is of high importance as it has a large impact on the performance of the

system. Given this, we selected a state-of-the-art test-to-code traceability tool, TCTracer [White et al., 2020], to generate the traceability links. TCTracer implements a set of traceability techniques that use dynamic information from both the method and class levels gathered during the execution of test suites to create links between tests and tested functions along with test classes and tested classes. In Relatest, we utilise the method level links produced using the LCS-B technique which utilises the distances between the names of tests and the names of executed functions to make predictions as to which functions are tested functions. The distance is measured based on the longest common subsequence (LCS) in both names. We chose to use LCS-B as it has high recall with good precision and high recall is important for Relatest because if we do not have any traceability links for a test we cannot use it as a recommendation.

The first time Relatest is run on a new project, the output from TCTracer is parsed and then stored in the traceability link database. The traceability database is refreshed only when a project is added or removed from the corpus, ensuring that the expensive operations involved in building the traceability links are only executed when necessary.

4.2.2 Query Processing: Similar Code Discovery

The measure that we have used to implement the similarity function $\text{sim}(f_1, f_2)$ is based on the Jaccard index [Jaccard, 1901]. This measure was selected as the Jaccard index is a widely accepted language-agnostic measure of similarity and a recent investigation [Ragkhitwetsagul et al., 2017] show that textual similarity measurements can perform well on source code with modifications. Using the framework from Ragkhitwetsagul et al. [2016], Ragkhitwetsagul et al. [2017], we have confirmed that using the Jaccard index over 3-grams as implemented in the Java String Similarity library[2] performs better than most of the 30 algorithms as given in the paper (the framework reports a precision of 0.891 and an accuracy of 0.884).

The pair-wise Jaccard index method finds every existing function in the corpus

[2] `https://github.com/tdebatty/java-string-similarity`

that is a member of at least one traceability link and adds it to $S(f_q)$ if the similarity to the query function is greater than or equal to a certain threshold τ (function $f \in L$ is added to $S(f_q)$ if $\text{sim}(f, f_q) \geq \tau$). $S(f_q)$ is then ranked by these similarity scores.

Pair-wise Jaccard index may become too expensive for a large corpus of many projects so one may substitute the pair-wise comparison with a scalable clone detector to create a clone index from which similar functions can be retrieved instantly. The corpus used in the evaluation is sufficiently small that the substitution with a clone detector is not necessary. Moreover, using Jaccard we establish a baseline which is not affected by implementation details of a clone detector as different clone detectors may produce very different results.

4.2.3 Query Processing: Recommendation List Construction

Given the ranked list of similar functions $S(f_q)$ that has been constructed by the similar code discovery, the recommendation list $R(f_q)$ is constructed using the algorithm in Figure 4.5. The function $append(R(f_q), t)$ appends test t to the end of the list $R(f_q)$ and $truncate(R(f_q), k)$ truncates the list $R(f_q)$ to length k. This algorithm uses the ranked list of similar functions to the target function and the set of traceability links to build a recommendation list by iterating through the similar functions and, for each function, adding the tests linked to that function to the recommendation list. If necessary, the recommendation list is truncated to the given maximum length at the end of this process.

4.3 Evaluation

We now present our research questions, the design of the experiments carried out to answer them, and the results.

Experimental Setup To carry out the experiments, we first establish the sets of test-to-function traceability links for all projects using TCTracer, as described in Section 4.2.1. We then take the functions that are a member of at least one traceability link as the query functions – the functions we are making recommendations for. The maximum recommendation list size was set to five as it has been shown that the average person is only able to reason about five to nine different items

Input: The set of traceability links L, The ranked list of existing similar functions $S(f_q)$, The max length of recommendation list k
Result: A list of recommendations $R(f_q)$ for the query function
$R(f_q) \leftarrow \emptyset$;
$i \leftarrow 0$;
$(s_1, ..., s_n) \leftarrow S(f_q)$;
while $|R(f_q)| < k$ **do**
  foreach $t \in \{t \in T \mid (s_i, t) \in L\}$ **do**
    $append(R(f_q), t)$;
  end
  $i \leftarrow i + 1$;
end
if $|R(f_q)| > k$ **then**
  $truncate(R(f_q), k)$;
end

Figure 4.5: Recommendation List Construction.

at a given time [Miller, 1956]. We chose to stay at the bottom of that range as test recommendations can be relatively complex and may take some time for a developer to assess.

Corpus For our corpus, we selected four well known open source projects that are written in Java and utilise the JUnit testing framework as subject projects: Commons Collections, Commons IO, Commons Lang, and JFreeChart. These subjects were selected as they are well known, widely used, and sufficiently large to demonstrate the applicability of RELATEST to real-world systems. Three of the subject systems have been used in the TCTRACER evaluation [White et al., 2020] and we added Commons Collections to increase the size and diversity of the subject sample. To filter out empty tests, we define a minimum test length θ in terms of lines of source code and set it to three.

Table 4.1 gives the following information about the subjects:

- Version (Ver.): The version that was used.
- Number of Functions (F): The total number of functions.
- Number of Tests (T): The total number of JUnit tests.

Table 4.1: Subject statistics.

Project	Ver.	F	T	IC	BC	$\|F_Q\|$
Commons Collections	4.1	4132	2819	84%	78%	1063
Commons IO	2.4	1187	1430	89%	87%	690
Commons Lang	3.7	3169	3556	95%	90%	2033
JFreeChart	1.0.19	9061	2502	52%	45%	2179

- Instruction Coverage (IC): The total instruction coverage provided by the JUnit tests – measured with Jacoco[3].
- Branch Coverage (BC): The total branch coverage provided by the JUnit tests – measured with Jacoco.
- Number of Queries ($|F_Q|$): Number of query functions for each project.

4.3.1 RQ1 (Intra-project Recommendations):

What performance is achieved by the tool when recommending tests from the same project as the query function?

We perform an investigation to evaluate the usefulness of the recommendations in an intra-project scenario, where the recommendations come from the same project as the query function. The evaluation uses the token-based Levenshtein edit distance between the recommendations and the tests linked to the query function to determine the usefulness of the recommendations. To compute this, we tokenise the code using JavaParser and calculate the Levenshtein edit distance between the two tests being compared.

To perform the evaluation we use the following process for each project: First we construct the set of test-to-function traceability links L as shown in Equation (4.1). We then use L to construct the set of functions F_Q that are a member of at least one traceability link $F_Q = \{f \mid \exists (f,t) \in L\}$, these functions are used as the query functions. Then, for each query function $f_i \in F_Q$, we construct the set of test recommendations $T_R(f_i)$ for that function and the set of linked tests $T_L(f_i)$ for that function $T_L(f_i) = \{t \mid (f_i,t) \in L\}$. Then for each member of $T_R(f_i)$ the average edit

[3] `https://www.jacoco.org/jacoco/`

distance to the members of $T_L(f_i)$ is computed. This provides a measure of how close to the actual tests each test recommendation is, and therefore also provides an approximation of the amount of manual effort that would be required for a developer to turn the recommendation into a working, useful test. To avoid biasing the results, we ensure that $T_R(f_i)$ does not contain any elements of $T_L(f_i)$ by skipping any tests in $T_L(f_i)$ when we build the recommendation list. Table 4.1 shows the number of query functions ($|F_Q|$) for each project.

Table 4.2 presents the results for the following measures:

- Rank (Rank): The rank in the recommendation lists that the recommendations come from.
- The Number of Recommendations (NR): The number of test recommendations made.
- Average Recommended Test Length (ARTL): The average length in tokens of the recommended tests.
- Average Known Test Length (AKTL): The average length in tokens of the known tests for the query functions.
- Median Average Edit Distance (MAED): The median of the average token-based edit distances between each recommendation and the known tests for the query function.
- Average Relative Distance (ARD): The median average token-based edit distance (MAED) divided by the average length of the known tests (AKTL) for the query functions.
- Average Edit Distance Standard Deviation (AEDSD): Standard dev. of the average token-based edit distances.

As the Median Average Edit Distance (MAED) and the Average Relative Distance (ARD) measure the token-based difference between the recommendations and the known desired tests, they can be seen as an approximation for the effort

required by the developer to adapt the recommendations to the desired tests versus writing the tests from scratch. This is discussed further in Section 4.3.8.

Findings The results show that, firstly, the recommendations at rank 1 in the lists were consistently the best performing in terms of average relative distance, with a maximum of a 17% improvement over the rank 2 recommendations, and that the quality (average relative distance) of the recommendations consistently decreases as the rank increases. As every list has a rank 1 recommendation, we can say that, in the average case, if recommendations are found, the developer will only have to expend 34%[4] of the effort required to adapt the rank 1 recommendation for use, as opposed to writing a test from scratch. This result is computed by taking the average of all the average relative distances (ARD) for the rank 1 recommendations for all projects, which is the proxy that we use for developer effort. From looking at the number of rank 1 recommendations generated and the total number of query functions, we can see that 75% of the query functions received recommendations. Further discussion of the results and their implications is presented in Section 4.3.8.

4.3.2 RQ2 (Inter-project Recommendations):

What is the effect on the performance of the tool when incorporating other projects into the corpus?

For RQ2 we perform the evaluation in the same fashion as for RQ1 except that, as we are now testing inter-project recommendations as well, the corpus used by RELATEST for making recommendations contains all subject projects. Table 4.3 presents the results for the same measures as in RQ1.

Findings The results show very little difference from the intra-project recommendations in RQ1 and where differences do occur the results tend to be slightly worse more often than they are slightly better. The reasons for this are discussed in Section 4.3.8.

[4] the average ARD using rank 1 recommendations

Table 4.2: RQ1 – intra-project recommendations results.

	Rank	NR	ARTL	AKTL	MAED	ARD	AEDSD
Commons Collections	1	752	326	326	150	46%	267
	2	622	302	319	167	52%	257
	3	514	294	314	178	57%	280
	4	420	303	300	172	58%	272
	5	344	265	296	157	53%	255
	All	2652	302	314	164	52%	256
Commons IO	1	487	266	231	81	35%	245
	2	403	243	219	114	52%	261
	3	343	240	210	110	52%	348
	4	294	220	209	109	52%	306
	5	253	196	200	113	56%	170
	All	1780	238	216	105	49%	347
Commons Lang	1	1592	264	245	59	24%	313
	2	1442	251	240	67	28%	265
	3	1340	247	235	70	30%	265
	4	1208	249	226	71	31%	272
	5	1133	246	223	76	34%	309
	All	6715	252	235	68	29%	293
JFreeChart	1	1669	250	261	85	33%	344
	2	1537	241	262	92	35%	447
	3	1417	216	263	98	37%	301
	4	1285	212	263	101	38%	355
	5	1196	238	268	125	47%	327
	All	7104	232	263	99	38%	325

4.3.3 RQ3 (Recommendation Evaluation):

To what extent does edit distance to known tests predict the usefulness of a recommendation to a developer?

We perform an investigation to determine the relationship between edit distances and the perceived usefulness of the recommendations to a developer. We first establish a range of edit distances from 0 to 1000 and split this range into ten bands. We then organised all recommendations into their respective bands and uniformly sampled 50 recommendations from each band for manual evaluation. The recommendations were classified by a judge as either true positive (a useful recommendation), or false positive (not a useful recommendation). The classified

Table 4.3: RQ2 – inter-project recommendations results.

	Rank	NR	ARTL	AKTL	MAED	ARD	AEDSD
Commons Collections	1	754	327	326	151	46%	268
	2	625	301	319	168	52%	257
	3	519	293	314	178	57%	279
	4	425	305	300	174	58%	275
	5	350	263	296	158	53%	254
	All	2673	302	313	164	52%	256
Commons IO	1	488	266	230	81	35%	246
	2	405	242	218	114	52%	260
	3	344	240	209	111	53%	348
	4	294	220	209	109	52%	306
	5	253	195	200	113	56%	170
	All	1784	238	215	105	49%	347
Commons Lang	1	1598	245	245	60	25%	313
	2	1452	250	239	68	28%	264
	3	1347	246	235	71	30%	266
	4	1213	249	226	72	32%	272
	5	1139	246	223	77	34%	309
	All	6749	252	235	68	29%	293
JFreeChart	1	1669	261	261	85	33%	344
	2	1537	241	262	92	35%	447
	3	1417	216	263	98	37%	301
	4	1285	212	263	101	38%	355
	5	1196	238	268	125	47%	327
	All	7104	232	263	99	38%	325

recommendations were then used to compute the precision (true positives / number recommendation samples) for each band. As usefulness is inherently subjective, the judge had to establish some criteria by which they would determine if a recommendation would reduce the time/effort of writing a test or improve the quality of the resulting test. The criteria included looking for elements in the recommendations that exactly matched or were close to the elements that would be required in a final test. These elements included asserts, function calls, control flow statements, and object declarations/initialisations. Where an element of the recommendation did not match exactly a required element, if only a minor change was required, e.g., simply changing an identifier, type name, or value, the element was still considered

Table 4.4: RQ3 – manual recommendation validation.

Avg. Edit Distance	Samples	TP	FP	Precision
1–99	50	45	5	90%
100–199	50	41	9	82%
200–299	50	32	18	64%
300–399	50	38	12	76%
400–499	50	36	14	72%
500–599	50	37	13	74%
600–699	50	39	11	78%
700–799	50	33	17	66%
800–899	50	33	17	66%
900–1000	50	28	22	56%

useful. To cross-validate the judgements, a second judge evaluated a sample of 100 of the same recommendations and the inter-rater agreement between the judges was computed using Fleiss' kappa [Fleiss, 1971].

Findings The results, as reported in Table 4.4, show firstly that the large majority of recommendations which achieve an average edit distance of less than 200 are judged to be useful, with an average precision of 86%. In contrast, the recommendations which have an average edit distance of over 900 are less likely to be judged as useful, with only 56% precision. Between these two extremes, the precision of the recommendations hovers between the mid-sixties and mid-seventies. The consequences of these results in the context of the RQ1 and RQ2 results is discussed in Section 4.3.8. The inter-rater agreement between the judges was $\kappa = 0.43$, which is interpreted as "moderate agreement".

4.3.4 RQ4 (Benefit of Using Relatest):

What benefit does using Relatest *provide in real-world test creation tasks?*

For RQ4 we conducted a user study in which we presented four unit test creation tasks to a set of developers. For each task, we asked the developers to create a JUnit test for a target function that was selected randomly from our subject projects (trivial functions such as getters and setters were ignored). The participants were provided with a dedicated interface in which, for each task, the participants were given the fully qualified name and source of the target function and anywhere

between zero and five RELATEST recommendations for the target function in the ranked order produced by RELATEST. The source code for the project of the target function was also given to the participants to simulate a real development scenario and the existing tests for the target functions of all tasks were removed to stop the participants simply copying the existing tests. The participants were asked to write a test for the target function and rate any recommendations that had been provided as either useful or not useful. The ability to run the written tests and receive the output was also provided and the participants were encouraged to not submit tests that did not pass. A link to the JUnit documentation was also provided to the participants and no time limit was imposed on the tasks.

To evaluate the effect of using RELATEST, we split the participants into two groups and gave group one RELATEST recommendations for the first and third tasks, while group two was given RELATEST recommendations for tasks two and four. The other two tasks had to be done without a provided recommendation (tasks two and four for the first group and tasks one and three for the second group). This allowed us to compare the tests created by the participants when they had recommendations versus when they had no recommendations.

The participants consisted of 17 computer science students, 16 at masters level and one at undergraduate level, with 9 participants assigned to group one and 8 assigned to group two. We collected information on the level of experience of the participants (Table 4.5). For the evaluation, we measured the median time taken to complete each task both with and without RELATEST recommendations. We also asked the participants to rate each recommendation as either useful or not useful.

We conducted the study over multiple sessions, using the first session as a pilot to determine if we needed to change the design. We did not deem it necessary to change the design as the results on four tasks revealed a noticeable difference. The desired outcomes and the practicalities of conducting a user study in a lab setting with supervision informed the study design, e.g., we limited the number of tasks to four to keep the participant's time commitment to reasonable 2–3 hours.

Table 4.5: Experience levels of user study participants.

Programming Experience (years)	None	< 1	1–3	> 3
Total Programming Exp.	0%	7%	40%	53%
Java Programming Exp.	0%	27%	47%	27%
Industrial Programming Exp.	20%	47%	27%	7%

Table 4.6: RQ4 – Median time taken in seconds for the task completion.

Task	1	2	3	4
With Recommendations	1766	549	795	708
Without Recommendations	2769	808	1324	1082

Findings The results for the task timings in Table 4.6 show that there is a clear difference in the median time taken to complete the task when recommendations are provided. On average, there is a 541 second difference in the median time taken to complete a task, representing a saving of almost 10 minutes to create a test when recommendations are provided. We performed a Wilcoxon signed-rank test ($\alpha < 0.05$) to assess if the time-to-complete the user study tasks with RELATEST recommendations is significantly lower than the time-to-complete them without recommendations; the difference is significant with a p-value of 0.034. The results for the participants ratings of recommendations, as reported in Table 4.7, show that at rank 1, the recommendation that RELATEST judges to be best, we achieve a 91% useful rating. At lower ranks we see that percentage drop to between 50% and 62%, demonstrating that the majority of recommendations are still seen as useful at lower ranks.

4.3.5 RQ5 (Developer Opinions of RELATEST):

What are the opinions of developers that use RELATEST*?*

Table 4.7: RQ4 – Developer ratings of RELATEST recommendations.

Rank	1	2	3	4	5
Total Rec.	34	34	26	26	18
Useful Rec.	31	21	13	16	11
Percent Useful	91%	62%	50%	62%	61%

Table 4.8: Questionnaire – answers to test development practice question: "While developing without a test recommendation tool, do you typically write tests from scratch or use existing tests as a template?".

I usually write tests from scratch	27%
I usually use an existing test as a template	47%
I usually write tests from scratch, but I look at other tests before	27%

Table 4.9: Questionnaire – RELATEST usage question.

Overall, did the recommendations help you complete the tasks?	Yes	100%
	No	0%
If freely available, how often would you use RELATEST or a similar tool in normal development?	Always	13%
	Most of the time	34%
	Some of the time	27%
	Rarely	13%
	Never	13%

For RQ5, we conducted a questionnaire with fifteen of the seventeen developers that participated in the RQ4 user study. As part of this questionnaire, we asked the participants to state if they believed that the recommendations helped them complete the task and if they would use RELATEST in their typical development workflow were it to be freely available (Table 4.9). We also asked if the developers were already using existing tests as templates when creating a new test to determine how easily RELATEST could be adopted by developers (Table 4.8). The intuition for this question is that if developers are already searching for existing tests to use as templates, it would not be a big change for them to use RELATEST and they would be more likely to adopt it. The questionnaire was administered immediately after the participants completed the user study tasks so that they were fresh in their mind. Two of the participants did not complete the questionnaire and thus were not included in the results.

Findings The results for the questionnaire (shown in Tables 4.8 and 4.9) show that all participants believed that the recommendations were useful and that 74% would use RELATEST in normal development at least some of the time, with 47% saying they would use it most or all of the time. As for their current development practices, 74% used existing tests in some capacity when creating new tests.

4.3.6 Real-world Applicability Example

To help demonstrate applicability, we used RELATEST to create a set of new tests for a large open source system. We measured the reduction in effort achieved by using RELATEST versus writing the tests from scratch using the token-based edit distance, in the same way as the research questions. To select a suitable system to generate tests for, we used SonarCloud [SonarCloud, 2022] to create a set of projects which fulfilled the criteria of being currently active (last analysed within the previous week), having the appropriate level of test coverage (30% to 70%), and being of adequate size (over 100k lines of Java code). We then randomly selected and inspected the projects from this set to find an appropriate untested class as a target to generate new tests for. To do this we used the "Explore" section of SonarCloud to browse the set of open source projects that use SonarCloud and set the filters to match our selection criteria. This allowed us to extract a list of projects that satisfied the criteria and then perform a random selection by assigning a number to each project and using a random number generator to pick the numbers. Our criteria for target class selection included looking for a class that contained approximately ten uncovered methods that were neither trivial, such as getters/setters, nor overly complex, such as graphics rendering methods. Using this process we selected the *SmsUtils* class from the District Health Information Software 2 (DHIS2)[5] project, which contains 12 such methods. RELATEST was used to generate the recommendation lists for each of these methods and the best recommendation was selected and transformed into a test for that method. These new tests were submitted to the project in a pull request, which has been merged. We computed the absolute and relative token-based edit distance, in the same manner as the research questions, between the recommendations and the final tests. This showed an average absolute edit-distance of 12 tokens and an average relative edit-distance of 0.55. This means that, on average, 12 tokens were changed in a recommendation to produce a final test, an average of 55% of the tokens in a recommendation.

[5] `https://github.com/dhis2`

4.3.7 Edge Prediction Technique Comparison

The RASHID framework is compatible with any bipartite edge prediction technique for generating the predicted edge set; therefore we must select which of these techniques to use for RELATEST. As discussed in Section 2.5 there are many different techniques for performing bipartite edge prediction including neighbourhood methods, path methods, and machine learning. The triangle method, which was ultimately selected for use in our approach, was also our initial approach as it reflects our intuition on how the recommendations should be informed by the relationships between the artefacts. We validated this choice by comparing the results obtained using the triangle method with several other bipartite edge prediction techniques, specifically spectral clustering, matrix factors learning, and cross-graph learning.

In order to compare these techniques to the triangle method we present a subset of measures from the full evaluation that can efficiently illustrate the important differences in performance between the techniques, specifically, the number of recommendations produced and how much effort these recommendations save. We calculate these measures both for recommendations at rank 1 only, and for recommendations at all ranks:

- The number of Recommendations (NR @ R1) – The number of test recommendations made at rank 1 in the list.
- Average Relative Distance (ARD @ R1) – The average relative effort, as described in Section 4.3, when considering the rank 1 recommendations.
- The number of Recommendations (NR @ R-All) – The number of test recommendations made at all ranks.
- Average Relative Distance (ARD @ R-All) – The average relative effort, as described in Section 4.3, when considering the recommendations from all ranks.

The baseline results provided here differ from the results provided in section 4.3 as these experiments were conducted prior with different parameter values, such

as the similarity threshold. However, as these values were consistent throughout these experiments the results still serve to provide a fair comparison to the other techniques. The approach and findings for each technique is presented below, with a general discussion of the results provided in Section 4.3.8.

4.3.7.1 Matrix Factors Learning

The matrix factors learning technique was adapted from one typically used in the field of collaborative filtering [Koren et al., 2009] where the goal is to predict a users rating for a given product based on their ratings for other products. In our formulation the products are replaced with tests, the users are replaced with functions, and the ratings signify the relevance of a test to a function.

The goal is to learn a factor vector for each function and test such that $\hat{r}_{ui} = q_i^T p_u$ where q_i is the vector representing test i, p_u is the vector representing function u, and $\hat{r}_{ui}$ is the predicted relevance of test i to function u. These vectors are learnt by minimising the loss function in Equation 4.4 by stochastic gradient decent. The training data is the known bipartite edges in the E_B edge set.

$$\min_{q,p} \sum_{u,i \in \kappa} (r_{ui} - q_i^T p_u) + (\|q_i\|_2 + \|p_u\|_2) \tag{4.4}$$

The results for this technique compared to the baseline of the triangle method are presented in Table 4.10 which shows the performance of both techniques over the evaluation data used in RQ1.

Findings The triangle method outperforms matrix factors learning in all measures as the triangle method consistently achieves better average relative effort while finding a similar number of recommendations.

4.3.7.2 Spectral Clustering

We applied spectral clustering to bipartite edge prediction by constructing a product graph over the two unipartite graphs: the function similarity graph $\mathcal{G}$ and test similarity graph $\mathcal{H}$, so that each possible bipartite edge between the vertex sets in $\mathcal{G}$ and $\mathcal{H}$ is encoded as a vertex in the product graph $\mathcal{P}$. We then cluster the vertices in $\mathcal{P}$ into two clusters and mark the cluster that contains the most vertices representing

Table 4.10: Comparison of Matrix Factors Learning to Triangle Method over Commons Collections.

Triangle Method Baseline				
Project	NR @ R1	ARD @ R1	NR @ R-All	ARD @ R-All
Commons Collections	228	28%	594	39%
Commons Lang	1007	27%	3962	30%
JFreeChart	945	35%	3893	41%
Matrix Factors Learning				
Project	NR @ R1	ARD @ R1	NR @ R-All	ARD @ R-All
Commons Collections	288	41%	1440	56%
Commons Lang	986	65%	4725	62%
JFreeChart	889	109%	4413	104%

bipartite edges in E_B as the predicted true edge set, while the other is marked as the predicted false edge set. The product graph is constructed using the Kronecker (Tensor) product, as shown in Equation 4.5, where $\otimes$ is the kronecker product, $\mathbf{G}$ is the weight matrix of $\mathcal{G}$, $\mathbf{H}$ is the weight matrix of $\mathcal{H}$, and $\mathbf{P}$ is the weight matrix of $\mathcal{P}$, which is used to represent it. This approach to constructing the product graph was informed by Liu and Yang [2015].

$$\mathbf{P} = \mathbf{G} \otimes \mathbf{H} \tag{4.5}$$

However, these product graphs have the problem that they are extremely large for any non-trivial software system as the number of vertices in the product graph is the product of the number of vertices in each of the unipartite graphs $|V_P| = |V_G| * |V_H|$. This problem is compounded by needing to use the weight matrix of the product graph, thus needing a matrix of size $|V_P| * |V_P|$. For example, when applying this to the Commons Collections project: $|V_P| = 1947114$ and $\mathbf{P} \in \mathbb{R}^{1947114 * 1947114}$ which is so large as to make the approach intractable. In order to combat this we perform size reduction on both $\mathcal{G}$ and $\mathcal{H}$ by first constructing a reduced graph $\mathcal{G}'$ by culling all vertices from $\mathcal{G}$ that are not 1-hop connected to the query function. We then construct another reduced graph $\mathcal{H}'$ by culling all the vertices from $\mathcal{H}$ that are not connected to a vertex in $\mathcal{G}'$ via the bipartite graph. We then construct a

Table 4.11: Comparison of spectral clustering to triangle method over RQ1 data set.

Triangle Method Baseline				
Project	NR @ R1	ARD @ R1	NR @ R-All	ARD @ R-All
Commons Collections	228	28%	594	39%
Commons Lang	1007	27%	3962	30%
JFreeChart	945	35%	3893	41%
Spectral Clustering				
Project	NR @ R1	ARD @ R1	NR @ R-All	ARD @ R-All
Commons Collections	137	25%	299	31%
Commons Lang	758	35%	2775	34%
JFreeChart	–	–	–	–

product graph $\mathcal{P}'$ over $\mathcal{G}'$ and $\mathcal{H}'$. This results in a very large reduction in the size of the product graph, making the Commons Collections and Commons Lang projects tractable, however, JFreeChart is still intractable.

To perform the clustering we use an unnormalised spectral clustering over the vertices in $\mathcal{P}'$. This is done by computing the unnormalised graph laplacian $\mathcal{L} = \mathbf{D}_{\mathcal{P}'} - \mathbf{P}'$ where $\mathbf{D}_{\mathbf{P}'}$ is the degree matrix of the product graph, then the first two eigenvectors of $\mathcal{L}$, u_1 and u_2, and then construct a matrix $\mathbf{U}$ with u_1 and u_2 as the columns, so that each row of $\mathbf{U}$ is a 2 dimensional vector representing a vertex in the product graph. These points are then clustered using k-means.

The results for this technique compared to the baseline of the triangle method are presented in Table 4.11 which shows the performance of both techniques over the evaluation data used in RQ1.

Findings While spectral clustering produces a better average relative effort when making recommendations for Commons Collections, the number of recommendations made is significantly lower than the triangle method. This represents a trade-off between the usefulness of the recommendations and the number of query functions that will generate no recommendations. For Commons Lang the triangle method is better for all measures and for JFreeChart, the size of the project makes the spectral clustering approach intractable due to the size of the graphs.

4.3.7.3 Cross-graph Transductive Learning

Given the inituition that the relatively poor results from the matrix factors learning and spectral clustering techniques are partially a result of training data sparsity, we decided to try a semi-supervised transductive learning technique. Transductive learning is an attractive approach to dealing with label sparsity in label propagation problems as it does not need to learn a general model which is then used to infer predictions, and instead directly learns labels for the unlabelled points in the specfic problem instance. To test this technique we implemented a version of the approach detailed in Liu and Yang [2016] that is simplified to only support 2 unipartite graphs as opposed to n unipartite graphs in the original approach. This approach formulates the cross graph learning problem as a label propagation problem over a product graph constructed from the unipartite graphs, similar to that used for the spectral clustering. However, this technique does not need the product graph size reduction techniques that we used for the spectral clustering approach as it does not need to actually construct the product graph or its weight matrix. The construction of these elements are avoided by using a tucker decomposition [Tucker, 1966] of the objective tensor, parameterised by spectral approximations of each unipartite graph. Despite this we found performing this technique on the JFreeChart project to still be intractable on a Ryzen 2700x CPU due to the size of the graphs.

The results for this technique compared to the baseline of the triangle method are presented in Table 4.12 which shows the performance of both techniques over the evaluation data used in RQ1.

Findings The triangle method outperforms this technique in all measures as the triangle method consistently achieves better average relative effort while finding a similar number of recommendations.

4.3.8 Discussion

We use the average relative distance (ARD) as a proxy for relative effort as there is no established method for accurately measuring required developer effort [Shihab et al., 2013]. While there are instances where token-based edit distance may not translate into effort, we believe that token-based edit distance is an adequate proxy

Table 4.12: Comparison of transductive learning to triangle method over RQ1 data set.

Triangle Method Baseline				
Project	NR @ R1	ARD @ R1	NR @ R-All	ARD @ R-All
Commons Collections	228	28%	594	39%
Commons Lang	1007	27%	3962	30%
JFreeChart	945	35%	3893	41%
Transductive Learning				
Project	NR @ R1	ARD @ R1	NR @ R-All	ARD @ R-All
Commons Collections	369	45%	1840	61%
Commons Lang	1332	68%	6660	68%
JFreeChart	–	–	–	–

(not requiring correlation). Our user study showed no results that would invalidate our assumption.

When comparing the results from RQ1 and RQ2 to examine the impact of inter-project recommendations, the minimal difference and slight worsening of results from RQ1 to RQ2 can be accredited to the fact that the shared lexicon between functions from the same project is much larger than between functions from different projects, where the purpose and formatting of the code often differ greatly. This means that firstly, RELATEST is far more likely to make recommendations from the same system as the query function since it is using the similarity between functions to find recommendations. Secondly, this factor means that in the instances where RELATEST does make inter-project recommendations, the average edit distances are likely to be larger than for intra-project recommendations.

Another useful observation provided by the RQ1 and RQ2 results is that the performance of the rank 1 recommendations is always the best rank and the performance almost always decreases as the rank increases. This is a positive result as rank 1 always has the most recommendations and demonstrates that our approach is correct in assuming that the more similar two functions are, the better the recommendations that they generate for each other will be.

A further notable observation emerges from the average relative distance in RQ1 and RQ2: Commons Lang has only a small reduction in performance as

the rank of the recommendations increases, compared to the other projects. This indicates the number of high-quality recommendations is higher in Commons Lang, which may be indicative of a high number of similar functions in the project.

The RQ3 results give important context to the RQ1 and RQ2 results as they provide an indication as to how average edit distances translate into the usefulness of the recommendations to the developer and can, therefore, give us a more complete picture of the value of using RELATEST as opposed to writing tests from scratch. For example, when considering the rank 1 results from RQ1 we can see that all of the median average edit distances for recommendations at rank 1 fall into the first and second bands (0–200), which have an average precision of 86% in the RQ3 results. Given the 75% rate of at least one recommendation being made, and the 34% average relative distance for rank 1, we can state that in 86% of the 75% of cases where at least one recommendation was generated, the developer saved 66% of development effort on average. This gives us a final figure of a 43% reduction in effort to create tests overall, even when only considering rank 1 recommendations. This represents a large amount of effort, especially over the full development cycle of a large project.

The RQ4 results reveal that the average task completion time was 36% less when recommendations were given, indicating a reduced amount of effort required to create the tests. This result is complemented by the developer ratings of the recommendations which show that 91% of the rank 1 recommendations were seen as useful and, on average, 59% of the lower-ranked recommendations were also seen as useful. The RQ5 results further demonstrate that the developers believed the recommendations, and RELATEST in general, to be useful.

The real-world applicability example demonstrated that it was possible to use the recommendations to create useful tests even when the test author has no previous expertise in the system. This is not the usual scenario in which the recommendations would be used, however, the recommendations will still be useful when used by developers for their own systems.

One important consideration to bear in mind when considering the results is the

that the traceability technique used to build the test-to-function links and the used similarity measure have a direct impact on the results and if the same experiments were to be conducted with a better traceability technique or a better similarity measure, we would expect the results to improve as there will be less noise in the recommendations.

4.3.9 Threats To Validity

As the evaluation was done on a small set of projects, the results may not generalise to other projects. Moreover, only Java projects have been analysed. As the implementation relies on a third-party traceability technique, the evaluation results depend on the accuracy of the links generated by TCTracer because developers rarely annotate their tests with the methods it is supposed to test [Bouillon et al., 2007].

The main threat comes from the reliance on a manual investigation for the RQ3 results. Firstly, the manual evaluation is a very time-consuming process which limited the size of the sample that we could evaluate. This is an external threat to validity as we have no clear evidence as to the representativeness of this sample. However, the fact that the subjects are large and diverse projects helps to ameliorate this threat. The use of manual investigation for evaluation also poses an internal threat to validity as there is some amount of subjectivity for what constitutes a useful recommendation. However, this risk is mitigated by our approach of firstly, establishing criteria for how recommendations should be judged and secondly, computing an inter-rater agreement between two judges using Fleiss' kappa, which shows statistically significant agreement.

There is also an external threat to validity created by the sample of participants for the user study who are all students and may not be representative of the wider developer community. However, given that all but one of the participants were graduate students at the masters level, there is a reasonably diverse range of experience levels and the majority (80%) had at least one year of industry experience, as shown in Table 4.5.

The user study underwent Ethics Review and was approved by the UCL

Research Ethics Committee. The participation was anonymous.

4.4 Related Work

The related work breaks down into two distinct sections: recommendation systems and similarity of functions.

4.4.1 Code Recommender Systems

Code reuse is a common practice in software development as developers look to solve problems and save time by using pre-existing code from other projects and the web. Most of this reuse comes from libraries, APIs, and code snippets [Gallardo-Valencia and Sim, 2014]. Due to the prevalence of reuse, a number of tools have been developed to discover and reuse existing code. Early examples of these tools include SCRUPLE [Paul and Prakash, 1994], which locates code features using a simple pattern-based query processor.

One recent approach is Test-Driven Code Search (TDCS) which uses test suites to define the desired behaviour and ensure that the code fragments returned by a code search engine can be tested in the context of the target system. CodeGenie [Lemos et al., 2011] utilises this approach by taking the designated test as input and performing a Sourcerer [Bajracharya et al., 2014] code search using keywords, identifiers, and interface definitions. Discovered candidate functionality is extracted via slicing and presented in the plugin for testing. However, CodeGenie and TDCS, in general, cannot be used directly with RELATEST as they are geared towards a Test Driven Development (TDD) scenario where a new test is used to find an existing function.

Other approaches use textual IR methods to search code and make suggestions. One example of this is Prompter, an IDE plug-in that searches Stack Overflow discussions and recommends code fragments for developers [Ponzanelli et al., 2014]. Prompter uses the IDE context, such as the currently displayed code, to formulate a query for a Stack Overflow search which is presented to the developer. The weakness of this system in comparison to RELATEST is that the user must manually extract the code snippets and there is a high probability the snippets will

not be executable without extensive modification. In comparison, RELATEST returns whole functions from existing projects.

Erfani et al. [2013] is related to RELATEST as their system also recommends unit tests based on the similarity between tested code. This can be shown to be an instantiation of RASHID as it is also based on the principle of utilising the relationships between artefacts in one domain (functions) to make recommendations of artefacts from another domain (tests).

However, the results presented by Erfani et al. show that they were only able to make a recommendation for 8.6% of untested functions and the quality of these recommendations was not evaluated. In contrast, RELATEST makes at least one recommendation 75% of the time, saving at least 43% of developer effort in total.

Landhäußer and Tichy [2012] utilises the idea of using existing tests for similar functions, with the difference that they attempt to programmatically transform the tests instead of providing them as recommendations. However, given the limited evaluation and the acknowledged issues with the approach to test transformation, we believe that RELATEST has greater applicability.

Makady and Walker [2013] explore the concept of aiding the reuse of existing tests when the corresponding tested code is reused. This provides an alternative scenario in which RELATEST could be used where the reused tested function is provided as the query function instead of a new function.

Recently, machine learning techniques have been utilised to generate code recommendations by learning from existing code. TESTNMT [White and Krinke, 2018] and REASSERT [White and Krinke, 2020b] use recurrent and transformer neural networks respectively to generate code recommendations, specifically unit tests, by learning from existing tests.

4.4.2 Similarity Measurements

Like traceability, code similarity measures have received a great deal of interest from the community. The tasks for which code similarity is useful include refactoring, fixing a bug, or performing plagiarism detection [Ragkhitwetsagul et al., 2017].

Natural language processing (NLP) can be used for measuring the similarity between code, such as in recent work by Zilberstein and Yahav [Zilberstein and Yahav, 2016]. Their tool, SIMON, uses NLP to establish similarity between snippets of code. Finding code snippets that are similar to a query snippet involves searching a database of code snippet to natural language description pairs to find a snippet that is semantically similar to the query snippet. The similarity of the natural language descriptions is then used to establish similarity between their respective code snippets.

The approach utilised by SIMON can also be shown to be an instantiation of RASHID defined in Section 4.1.2 as it utilises inter-domain relationships between artefacts to discover other relationships between artefacts, specifically using the two domains of code snippets and natural language descriptions to find similar code snippets. The way that SIMON instantiates RASHID is slightly different from RELATEST and Erfani et al. [2013] in that the relationships that are being searched for are intra-domain (code-snippet-to-code-snippet). In this instance E_B is the first set of edges constructed, by using a relation defined by a database of code snippet to natural language description mappings. This database is used for the R_B relation such that vR_Bv' $(v, v' \in (V_1 \cup V_2))$ if the code snippet represented by v and the natural language description represented v' are linked in the database. The intra-domain edges in the natural language description domain E_2 are constructed using textual similarity as the relation, such that vR_2v' $(v, v' \in V_2)$ if the descriptions represented by v and v' have a textual similarity above a certain threshold. The predicted edges are not the new inter-domain edges E_P, as is the case for RELATEST, but are the intra-domain edges in the code snippet domain E_1. The prediction method uses squares, specifically that an edge (v, v'), $v, v' \in V_1$, is added to E_1 if there exists a path (v, n_1, n_2, v') of length three from v to v', where $n_1 \in E_2$ and $n_2 \in E_2$.

The advantage that RELATEST has over this system is that RELATEST does not require a database of code snippet to natural language description pairs, which is difficult to build and maintain.

4.5 Conclusion

We have presented RASHID, an abstract framework for assisting the reuse of existing artefacts, and RELATEST, an instantiation of RASHID that discovers existing tests which can be recommended to developers to test new functions. We have also presented an empirical evaluation of RELATEST with four real-world open source projects in two possible usage scenarios (intra and inter-project) and through a user study. The results show that overall RELATEST can make reductions of 43% of the total amount of developer effort required to test new functions. The user study shows that, on average, developers needed 10 minutes less to develop a test when given RELATEST recommendations and all developers reported that the recommendations were useful. The results demonstrate the power of discovering and exploiting connections between artefacts to improve the software development process.

Chapter 5

Test Transplantation

This chapter presents the work completed for an investigation into the transplantation of tests between projects. The concept for this work follows on naturally from the work presented in chapter 4, which is concerned with the discovery and recommendation of existing test cases for reuse. The logical next step from this is to attempt to automatically transplant those recommended tests into the target project, rather than leaving it up to the developer to manually adapt them. We, therefore, conducted an investigation into the feasibility of performing automated test transplantation, the approach and results of which are described in this chapter.

5.1 Motivation

The ability to transplant tests between projects has the potential to provide great utility in the software engineering process by opening up a whole corpus of existing tests to use, such as those contained in open source repositories. A reliable approach to test transplantation could improve the software engineering process in the same ways as RELATEST: by reducing the cost of development and improving the quality of the output, however, with the added automation these benefits would be even greater.

5.2 Approach

To perform the transplantation we use a staged process to identify the test for reuse (*donor test*), retarget the test for the new function (*target function*), transplant it into

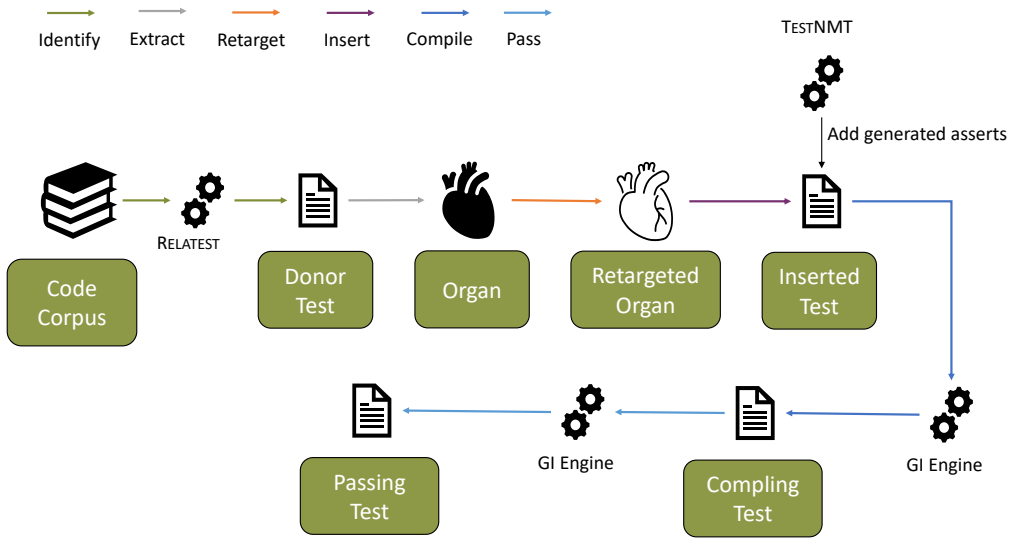

Figure 5.1: Transplantation Approach.

the new project (*target project*), and transform it to work in its new environment, creating the desired final test (*target test*). This process is shown in Figure 5.1. The specific stages used are: *Identify*, *Extract*, *Retarget*, *Insert*, *Compile*, and *Pass*. The extraction stage extracts the donor test from its project. The retarget stage transforms the extracted donor test to target the new function to be tested. The insert stage inserts the donor test into the target project. The compile stage further transforms the inserted test so that it can compile. The pass stage makes more transformations to the inserted test so that it can pass. Once the inserted test passes, the transplantation is complete.

5.2.1 Identify

We start the process by identifying appropriate existing tests to use as the donor tests that we will attempt to transplant. The goal of this stage is to identify a test or set of tests that we consider to be sufficiently close to the target test to use as donor test(s). This is important as the closer a donor test is to the target test, the fewer transformations we have to make to convert the donor test into the target test. In the case where we identify multiple appropriate donor tests, we execute the rest of the stages for each donor test until we achieve a successful transplantation or run out of

donor tests.

There are multiple approaches for identifying appropriate donor tests and we, therefore, investigated several approaches. The first approach was to use RELATEST from Chapter 4, the second approach was to use an interface-based filter, and the third approach was to use unit test generation tools, such as EvoSuite [Fraser and Arcuri, 2013].

To use the RELATEST approach we simply provide the target function to RELATEST, generate the recommendations, and use those as the donor tests. The filter-based approach uses the interface of the target function, specifically the types of the parameters and the return type to find tests that test other functions with the same interface. We utilised this approach as it has two primary benefits. Firstly, if the interface of the function tested by the donor test is the same as the interface of the target function, the task of transforming the donor test to test the new function is easier as we can replace calls to the old function with calls to the target function and not cause a compilation issue. Secondly, if the interfaces of the previously tested function and target function match, there is a higher chance of the donor test being a good candidate to test the target function. The issue with this approach is that it can create a very large number of donor tests for functions that have common interfaces, usually where only built-in types are used and the number of parameters is small. For example, a function that takes in a string and returns a string will produce a large number of donor tests. Functions with void return types and no parameters are extreme examples of this issue. To avoid this issue, we add another condition to the filter to include only tests from the same package as the target function. This is a trade-off as it may reduce the number of donor tests discovered to a point where no suitable donor test is found. However, if there are too many donor tests that are not suitable, we will waste a lot of computational time attempting to transplant unsuitable donor tests.

The third approach utilises generated unit tests from test generation tools, such as EvoSuite [Fraser and Arcuri, 2013], as donor tests. This approach was used due to our intuition that our transplantation process could improve upon what the test

generation tools can produce and generate target tests that are more likely to reveal a fault or provide higher coverage.

5.2.2 Extract

Once we have identified our donor test for transplantation, the first step is to extract it from its current project in a manner that allows us to then insert it into another project after we have targeted it. As tests are often not entirely self-contained and can rely on other code elements, it is not sufficient to simply extract the code of the test itself. Instead, we build an object, called an *organ* which contains all the elements required for the insertion of the donor test into a new project. The number of elements included in the organ is determined by the number of dependencies the test has. These dependencies are usually contained with the test class that the donor test is a member of and can include import statements, class fields, setup and teardown functions, and inner classes. All of these elements are required to be transplanted with the donor test to be able to execute it in the target project and, therefore, must be saved in the organ. A further complication is added if the test relies on helper functions as we must then also take a backward slice from the call site to the helper function. This may not make the organ significantly more complicated if the helper function is self-contained or if the slice is small but a large slice will make it very difficult to insert the organ without causing compilation failures. This complication is mostly avoided if the donor test is from the same project as the target function as any public helper functions will be available to the inserted test as well and therefore they do not need to be added to the organ. If helper functions that are defined outside the test class need to be included in the organ, the file structure is preserved in the organ.

5.2.3 Retarget

Once we have constructed the organ we can begin the process of transforming it and inserting it. The first stage of this is a transformation performed before the organ is inserted called retargeting which performs the initial switch from testing the previous function to testing the target function. To do this we must first replace

any references to the class of the previous test with references to the class of the target test and any calls to the previous function with calls to the target function. If the name of the donor test contains the name of the previous function we also update the name to replace these instances with the name of the target function.

The complexity of this transformation is situational as it depends on the interfaces of the previously tested function and the target function. If the interfaces of the functions match, the transformation is simpler but if the interfaces do not match, and especially if the return types do not match, the process is more complicated and we have to also update the types of objects that have been returned from the new calls to the target function. Another complicating factor is that it is not uncommon for a test to test multiple functions. In this case, we have to update references to the containing classes and the call sites, as described above, for all of the previously tested functions. We also have to handle the special cases where one or more of the previously tested functions or the target function are constructors, as calls to constructors do not have a call scope. Additionally, if the organ contains helper functions from the original project, the retargeting may have to also be applied to those functions if they called the previously tested function.

5.2.4 Insert

Once the retargeting is complete we insert the retargeted organ into the target project. To do this we must first identify a test class to insert the test into or create a new test class if there is no existing appropriate test class. The identification of an existing test class is done by searching the project for a test class that matches the name of the class containing the target function by naming conventions, as described in Section 3.3.1.1 If this does not find any existing test class to use, we create a new test class that does satisfy the naming convention. Now we have established our target test class, we insert the organ by adding the code elements that were present in the original test class into the new test class. These include the retargeted test itself, the import statements, the class fields, any setup and teardown functions, and any helper methods that were defined within the original test class and copied into the organ. If the organ contains helper functions that were not defined in the

original test class and are, therefore, contained in other files, the files containing those classes are inserted into the target project using the same file structure with the directory of the target test class as the root.

As the final step in the insertion stage, we utilise our assert generation tool, TestNMT, described in Chapter 6 to generate new asserts for the test and insert them at the end of the test. After the insertion stage is complete, all of the elements required for the new test have now been inserted into the target project in the appropriate locations.

5.2.5 Compile

The retargeting and insertion stages have left us with inserted code in multiple places that have been transformed to target the target function but is likely not yet fully integrated into its new environment so the purpose of this stage is to transform the test so that it is well-integrated enough to be able to compile in its new environment. The likelihood of successfully completing this stage depends heavily on a range of factors, including how dissimilar the donor test is to the target test, i.e, how much further modification needs to be performed, and how many other elements the organ contained, such as the number of setup and teardown functions, class fields, and helper functions, as each of these things can cause compilation problems. For example, a setup, teardown, or helper function may contain calls to other functions or reference types that are not present in the target context.

There are multiple potential approaches to this stage, which may include the use of static analysis techniques such as inter-procedural analysis, data flow analysis, and other static analysis techniques. However, the approach that we selected to use was genetic improvement (GI) [Petke et al., 2018] as it has the benefit of being a generic technique that can be applied to any donor test. This avoids the complexity of having to define a large set of rule-based transformations. We experimented with two different GI algorithms, described in Section 5.3. To utilise our GI algorithms for this transplantation stage we defined the fitness function as the total number of compilation errors. This allows our GI algorithms to search for solutions that compile.

5.2.6 Pass

If we succeed on the compilation stage, we now have a transplanted test that compiles but we still do not know if it is testing the function in the way that it is supposed to. However, the evaluation of the quality and usefulness of tests is a complex issue that is outside the scope of this work. Therefore, we use the result of the test as a proxy for the correctness of the test and aim for getting the test to pass. To achieve this we apply our GI algorithms again but this time we use the number of failing asserts as the fitness function to search for solutions that pass.

5.3 Genetic Improvement

To properly assess the performance of GI for transplantation, we implemented two different GI algorithms: *Elite Breeding Group* and *Two Elite Parents* to explore if the choice of algorithm has a significant impact on the results. For both algorithms, a two-point crossover was used as the crossover operator and the mutation operators used were delete, move, insert, and replace.

Elite Breeding Group For each generation, we select the fittest $x\%$ of the population to breed by crossover. The created offspring are mutated and added to the population. The fitness of the population is computed and the fittest $y\%$ of the population is selected as the elite and this elite becomes the new generation.

Two Elite Parents For each generation, the fittest $x\%$ of the population is selected as the elite and the top two elite individuals breed by crossover. the offspring is mutated and evaluated for fitness. If the best offspring fitness is better than the best parents fitness, the offspring are added to the elite, otherwise, the parents are added back into the elite. The elite then becomes the new generation.

5.4 Evaluation

We evaluate our transplantation approach in two primary ways: how often we successfully complete our transplantation stages, and how effective our transplanted tests are at successfully revealing real faults.

Table 5.1: Subject statistics.

Project	Ver.	Num. Functions	Num. Tests	Coverage
Commons Collections	4.1	4132	2819	84%
Commons Lang	3.7	3169	3556	95%
JFreeChart	1.0.19	9061	2502	52%

Table 5.2: Genetic improvement algorithm parameters.

Parameter	Value
Search budget	300s
Initial Population Size	5
Minimum Population Size	5
Max Population Size	10
Reproduction Proportion	60%
Elite Proportion	40%
Crossover Probability	0.75
Insert Probability	0.25
Replace Probability	0.25
Delete Probability	0.25
Move Probability	0.25

5.4.1 Experimental Setup

For our evaluation, we used several mature real-world open source projects commonly used as research subjects, specifically Commons Collections, Commons Lang, and JFreeChart. The subject statistics are provided in Table 5.1. To assess the benefit of applying GI to the compile and pass stages, we also run our experiments with GI disabled to provide a baseline. Additionally, as we define a GI search budget of five minutes for each of the compile and pass stages, running experiments that require a large number of transplantation attempts have a high time cost. To reduce this burden, we ran some preliminary experiments on a small sample size to compare the performance of the two GI algorithms. This revealed that the two algorithms consistently perform very similarly to each other, with the Elite Breeding Group variant coming out slightly ahead being one to two percent more successful at completing the compiling and pass stages. Given this, we simply selected the Elite Breeding Group to be the GI algorithm that we used in the main experiments with the full sample. The parameters used for our GI algorithm

are given in Table 5.2. As each generation took approximately 20 seconds of computation time, we achieved approximately 15 generations within the 300 second time budget. For the identification of donor tests we use RELATEST and the interface-based filter approaches, as described in Section 5.2.1. While GI is stochastic, it became apparent when executing multiple runs that the results of the analysis were largely consistent from run to run so the results presented are each the result of a single representative run and not an average across a sample of runs.

5.4.2 Research Question 1 (Success of Transplantation Process)

How often are the transplantation attempts successful?

To evaluate how successfully we can complete our transplantation process, we used all three of our subject projects, took a random sample of functions, and executed our donor test identification and transplantation approach for these functions. We recorded the number of attempted transplantations, the number of transplantation attempts that resulted in a compiling test and the number of attempts that resulted in a passing test. We included the attempts that resulted in a compiling test but not a passing test as the compiling test may only need a minor manual change to become a passing test and therefore may still be valuable. There is also the possibility that the test may be revealing a real fault.

Findings The results, shown in Table 5.3 compare the results we achieve when using GI versus not using GI where "With GI" means we utililsed the full approach including the application of GI in the compile and pass stages and "Without GI" means we simply tried to compile and run the test after the insert stage, without performing any further transformation. The results reveal that using GI produces approximately double the number of compiling transplanted tests (from 127 to 248) and a greater than doubling of the number of passing transplanted tests (from 75 to 168).

Table 5.3: Transplantation success results.

GI Type	Num. Functions	Attempted Transplantations	Num. Compiling	Num. Passing
Commons Collections				
Without GI	1500	231	32 (14%)	15 (6%)
With GI	1500	231	62 (27%)	46 (20%)
Commons Lang				
Without GI	150	280	60 (21%)	28 (10%)
With GI	150	280	106 (38%)	52 (19%)
JFreeChart				
Without GI	300	282	35 (12%)	32 (11%)
With GI	300	282	80 (28%)	70 (25%)
Totals				
Without GI	1950	793	127 (16%)	75 (9%)
With GI	1950	793	248 (31%)	168 (21%)

5.4.3 Research Question 2 (Effectiveness of Transplanted Tests)

How successful are the transplanted tests at revealing real-world faults?

To assess the effectiveness of our transplanted tests, we utilise the Defects4J dataset [Just et al., 2014]. Defects4J is a dataset of faults from multiple projects, including the 64 faults from Commons Lang that we used for this evaluation. For each fault, Defects4J provides a "faulty" and a "fixed" version and we use these faults to assess the effectiveness of our approach by using the functions containing the faults as the target functions, performing the donor test identification and transplantations, and record which faults are revealed by the transplanted tests. We considered a fault revealed if the test fails on the buggy version but passes on the fixed version. To provide a comparison to existing unit test generation techniques, we also performed the evaluation using tests generated by EvoSuite.

Findings The results, provided in Table 5.4, show that our transplantation approach reveals faults 12 and 30 without GI and that using GI allows us to also reveal fault 27. EvoSuite on the other hand reveals more faults than the transplantation approaches, however, we were able to reveal fault 30, which EvoSuite was not able to reveal.

Table 5.4: Defects4J fault discovery results.

System	Faults Revealed
EvoSuite	5, 7, 9, 11, 12, 19, 27, 32, 33, 41, 43, 47
RELATEST Transplantation - Without GI	12, 30
RELATEST Transplantation - With GI	12, 27, 30

5.5 Discussion

Throughout our investigation of test transplantation, we discovered multiple points for discussion. Firstly, we discovered that the chance of success of transplantation is heavily dependent on both the complexity of the donor test and the size of the organ. This is because the biggest challenge during transplantation is how to handle references to classes, functions, and fields that are only present in the source context and not the target context as these references become compilation failures that need to be resolved. Resolving these compilation issues is difficult and, therefore, the less of them we have to resolve, the more likely successful transplantation is. This means that transplantations are much more likely to succeed if the source project and target project are the same as it reduces the number of references that will no longer be valid. These references are maximally reduced when transplanting within the same package. As we are using RELATEST and our interface-based filter for identifying donor tests, the majority of our donor tests are from the same project as the target function and often from the same package. This is because RELATEST is far more likely to recommend tests from the same project than from other projects, as discussed in Chapter 4 and our interface-based filter only selects from the same package, for the reasons discussed in Section 5.2.1.

When investigating the usefulness of applying GI to our transplantation approach we see that it approximately doubles how often we achieve a compiling test and more than double how often we get a passing test. GI also increases the number of Defects4J JFreeChart faults reveals from 2 to 3. These results show that utilising GI has a large positive impact on the effectiveness of the approach, however, it does increase the time taken to transplant as we allow a maximum GI search budget of five minutes for each of the compile and pass stages. In the future, the benefit of

the GI could potentially be improved by investigating the effect of utilising other algorithms and operators, such as those compared in Blot and Petke [2021].

The RQ2 results also show that overall EvoSuite reveals more faults in the Defects4J JFreeChart data set than our transplantation approach. This is expected as EvoSuite is a mature, well-developed tool that has been at the cutting edge of unit test generation for multiple years. However, our approach revealed a fault, fault number 30, which was not revealed by EvoSuite. This fault was from the project Commons Lang in a function that finds a given character sequence in a character array and returns its index. We were able to discover this fault while EvoSuite was not able to as our transplantation process injected a new assert that discovered the fault while the asserts generated by EvoSuite were less specific and could not detect it. This shows that our approach has the potential to be useful when used in addition to EvoSuite as it may reveal faults that EvoSuite cannot reveal.

5.6 Conclusion

In this chapter, we have presented an approach for the transplantation of test cases utilising a multi-stage process including assert generation and genetic improvement. We performed an evaluation of this approach which assesses how often the transplantation process completes successfully by sampling target functions from three large mature open source systems and how effective the transplanted tests are at revealing real faults by utilising the Defects4J data set. The results show that, when using genetic improvement, we achieve a compiling transplanted test 31% of the time and achieve a passing transplanted test 21% of the time. Our transplanted tests were able to reveal 3 of the JFreeChart faults in Defects4J, including 1 fault that was not revealed by EvoSuite.

Chapter 6

Automated Generation of Test Code

Chapter 4 and Chapter 5 present approaches to the identification, recommendation, and transplantation of existing tests to test new functions. However, while effective at reducing developer effort overall, there are some functions for which no good recommendations of existing tests can be found. To deal with these cases and to broaden the set of techniques available to developers looking for new tests, this chapter presents TestNMT, an approach to test code generation that uses neural networks adapted from the domain of natural language processing (NLP) to generate approximate JUnit tests for a given function. However, as these are approximate tests, developers still have to expend manual effort to transform them into the desired executable tests.

TestNMT is an exploratory approach to unit test generation which aims to generate tests by learning to translate from the function domain to the test domain. However, as the model only sees the output as a sequence of tokens and has no concept of the syntax of the target programming language or test framework, there is no guarantee that the output will be a syntactically correct test. We, therefore, refer to the output as approximate tests. Given this, the goal is to allow a user to provide a function as input and receive an approximate test as output, which the developer can then manually adapt to a working test for that function.

This document provides an overview of the preliminary investigation into the potential of this approach, including the network design, data collection, and a preliminary quantitative and qualitative evaluation of TestNMT for two usage

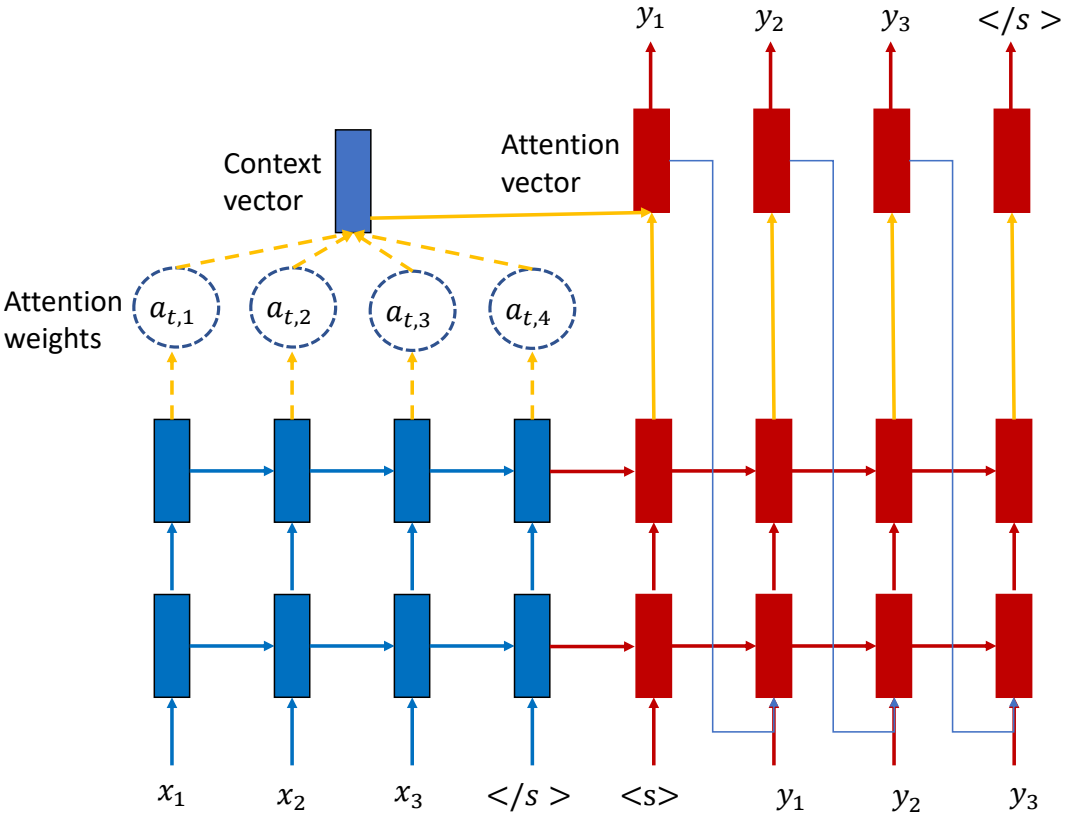

Figure 6.1: Overview of the TestNMT architecture.

scenarios: cross-project and within-project.

The evaluation shows that, while TestNMT is most likely not useful in the cross-project scenario, it does have the potential to be of use within-project. The quantitative evaluation demonstrates this with a maximum BLEU score of 21.2, a maximum ROUGE-L score of 38.67, and edit distances between existing linked tests and generated approximate tests which show that on average approximately half of the content of the tests can be generated by TestNMT. The qualitative evaluation also shows that TestNMT has a strong potential for usefulness by demonstrating that it can produce approximate tests that are easy to adapt to the desired tests.

6.1 Approach

TestNMT uses the techniques of neural machine translation for natural languages and applies them to the programming language domain, specifically to translate from functions to tests. TestNMT uses an encoder-decoder with attention archi-

tecture for sequence to sequence translation, such as that shown in Figure 5 of the Tensorflow neural machine translation tutorial [Luong et al., 2017]. In this architecture, the encoder encodes the source sequence into a vector representation which is then decoded into a translated target sequence by the decoder.

The encoder builds the vector representation of the source sequence by traversing the sequence one token at a time converting each token into a real-valued vector embedding via an embedding layer which is then provided as input to the encoder recurrent neural network (RNN) for that time step. After the source sequence has been fully processed, the final hidden state of the encoder RNN is used to initialise the hidden state of the decoder RNN. Then, at each time step, the decoder RNN uses the current hidden state, the previously generated target sequence token, and the attention mechanism, to generate a new target sequence token. This continues until the end-of-sequence token is generated. The TestNMT architecture is shown in Figure 6.1.

To create the training data we apply a function-to-test traceability technique over a corpus of software, generating a set of example function-to-test links. These links are pre-processed into source and target token sequences which are then used to train the embedding layer and RNN units by backpropagating the sequence-to-sequence cross-entropy loss. The implementation used for these experiments was derived from the implementation provided by Luong et al. [2017].

6.2 Experimental Setup

The experiments are split into two scenarios: cross-project, and within-project. The cross-project scenario tests the possibility of training a single model using a large, general corpus taken from many projects, and using it to generate an approximate test for any arbitrary function. The within-project scenario tests the possibility of training a model for an individual project and using it to generate an approximate test for a function from the same project.

Experiment configuration one tests the cross-project scenario using 156 open source projects from GitHub. Configurations two, three, and four test the within-

project scenario by individually using the OpenJDK 9, Netbeans, and OpenLiberty projects respectively. A natural language translation baseline for the chosen translation quality measures was obtained by also training the model for Vietnamese to English translation on a small corpus of TED talks [Luong and Manning, 2015].

6.2.1 Data

To train the model we need a dataset consisting of test-to-function traceability links – pairs of functions and tests where the test tests the function. As the performance of the model is dependent on the size and accuracy of the training data, we need to gather as much high-accuracy training data as possible. Ideally, the dataset would be constructed from manually labelled test-to-code traceability links but given the amount of data required this is infeasible and, therefore, we must use an automated traceability establishment technique.

When selecting an automated traceability establishment technique, we must balance precision with recall; low precision techniques will result in very noisy training data, whereas low recall techniques will result in a dataset that is too small. Both of these factors can limit the performance of the model.

6.2.1.1 Generation through Naming Conventions (NC)

For this set of initial experiments, we have limited the data to Java projects with JUnit test cases and are generating the training data using variants of the Naming Convention (NC) [Rompaey and Demeyer, 2009] technique. NC was selected for these initial experiments as it is performed statically and should have relatively high precision on projects that use the naming convention, however, in future work a more extensive set of traceability techniques could be used to increase both the size and precision of the dataset.

The two variants of NC that we tested are Strict NC and Relaxed NC. Strict NC matches both the class names and the function names, for example, the *toString()* function in the class *Point* will only be matched to a test called *testToString()* or *toStringTest()* in a test class called *TestPoint* or *PointTest*. Relaxed NC however, matches only on the function name, so *toString()* in any class will match to

Table 6.1: Number of test-to-code trace links generated by naming convention techniques.

Project	JUnit Tests	Strict NC	Relaxed NC
Apache Poi	1,582	0	10,546
JFreeChart	2,482	1,016	1,016
Closure Compiler	153	56	101
Commons Lang	3,061	2,385	10,647
Commons Math	4,461	770	10,033
Commons Collections	2,661	455	12,167
Eclipse CDT	856	42	1,759
Android Platform	4,021	1,128	34,711
Chromium	6,334	435	7,347
Netbeans	1,582	399	57,156
Total	28,500	5,688	145,828

testToString() or *toStringTest()* in any test class. Therefore, Strict NC is better for precision, while Relaxed NC is better for recall.

NC Generation Experiments Table 6.1 shows the results of an experiment comparing the sizes of the datasets generated by Strict NC and Relaxed NC on a set of popular open source Java projects.

We can see from these results that while projects which closely follow the naming convention, such as JFreeChart and Commons Lang, may produce a good amount of links for Strict NC, most projects do not follow the convention closely and produce little to no links for Strict NC. Overall the number of links found using Strict NC is not enough for training a model.

When we switch to Relaxed NC the number of links increases to levels that make training a model feasible. While this increase in links has a concomitant increase in noise due to false positives (functions and tests that are incorrectly matched), some of the false positives may provide useful information for the model in cases where the test is related to the function even though it does not directly test it. One example of this is typically overridden methods (toString, hashCode, equals), which should all share a similar structure and relationship to their tests. Therefore these links can provide useful information for the network to learn even if they are false positives. Given this, we selected Relaxed NC to generate the data for our experiments.

6.2.1.2 Preparation

The data for the experiments was obtained by first applying the Relaxed NC technique over the subject(s) to establish the traceability links. The source of the functions and tests was then pre-processed to remove camel-casing, convert all strings to lower case, remove all numbers, and add spaces between all words and symbols. Programming language keywords and syntax were retained. Camel-cased tokens were split into individual tokens to keep the vocabulary sizes manageable as the individual tokens are more likely to appear in multiple sequences than the complete camel-cased tokens, the same applies to adding spaces between words and symbols. Numbers were removed as some exploratory experiments showed that their inclusion degraded the results. The pre-processed sequence pairs are then split between the training and test sets and all training set pairs that contain a source or target sequence that also appears in the test set are removed, this helps to avoid overfitting.

6.2.2 Network Configuration

The network configuration used for the experiments set the dimensions of the embeddings at 128, the number of RNN layers at 2, dropout at 0.2, and uses an attention mechanism named Scaled Luong, a scaled variant of the attention mechanism described in Luong et al. [2015a]. This network configuration was selected simply as a default reference; the optimisation of the network configuration has not yet been explored. The maximum sequence length for training was set to 50 tokens for the baseline natural language translation and 200 tokens for the test translation experiments, with any longer sequences truncated to this limit. This difference is due to the fact that the functions and tests are significantly longer than the natural language sentences. This sequence length limitation hinders the learning of any relationships that are predominantly found in long sequences, after the 200 token limit. Therefore, increasing the maximum sequence length could improve the results, however, it also increases the training time which made using a sequence length longer than 200 infeasible for these initial experiments.

6.3 Evaluation

We evaluate the performance of TestNMT both quantitatively and qualitatively. The quantitative evaluation is carried out using the standard techniques for measuring the quality of translations BLEU [Papineni et al., 2002] and ROUGE-L [Lin, 2004]. BLEU is a precision-based measure that utilises a modified n-gram precision to calculate the co-occurrence of tokens in the candidate and reference sequences. Clipping is applied at the frequency of the n-gram in the reference sequence. ROUGE-L uses the Longest Common Subsequence (LCS) between the candidate and reference to calculate the precision and recall of the matching unigrams which are then used to calculate the F-Score. At each training step, these measures are calculated over all instances in the test set and averaged to get the scores for that step. The maximum scores for each experiment are reported in the results.

The quantitative evaluation also includes statistics for the average length of the pre-processed linked tests and the median edit distance from the generated approximate tests to these linked tests for the functions from the test set. This gives us a metric for how close the approximate tests generated by TestNMT are to the linked tests, therefore indicating the potential usefulness of TestNMT in a real-world scenario.

The qualitative evaluation is conducted by inspecting some example generated approximate tests and comparing them to the pre-processed linked tests for the input function to determine the type of edits that the developer would have to make to adapt the approximate tests for use.

6.3.1 Quantitative Results

The results from the quantitative evaluation are presented in Table 6.2. The Config 1 results indicate that the cross-project scenario may not be feasible as the maximum BLEU (1.3) and ROUGE-L (16.3) scores are very poor and the median edit distance is larger than the average length of the pre-processed linked tests. These numbers also saw no improvement during training showing that no significant translation is occurring and suggests that no amount of training will improve the scores at this dataset size of approximately 750,000 training examples. More training data may

Table 6.2: TestNMT experimental results.

	Baseline	Config 1	Config 2	Config 3	Config 4
			Data		
Project(s)	N/A	156 OSS	OpenJDK 9	Netbeans	OpenLiberty
Train Set Size	133k	744k	105k	11k	88k
Test Set Size	1500	300	100	100	100
Src. Vocab Size	17k	24k	9k	6k	7k
Tgt. Vocab Size	8k	13k	2k	2k	4k
			Results		
Max BLEU	20.24	1.3	19.51	21.20	19.60
Max ROUGE-L	50.40	16.3	38.3	33.73	38.67
Avg. Test Len.	N/A	375	522	543	625
Med. Edit Dist.	N/A	1431	246	344	269

bring this scenario into a reasonable range of performance but the dataset may have to be extremely large.

Configs 2, 3, and 4 indicate that TestNMT has significant potential for usefulness in the within-project scenario as the maximum BLEU scores are equivalent to that of the Vietnamese to English natural language translation baseline, and the ROUGE-L scores average 73% of the baseline score, which is also a strong result. The edit distance results also reinforce this conclusion by indicating that, on average, the edit distance between the generated approximate tests and the pre-processed linked tests is within ˜50% of the length of the pre-processed linked tests. This demonstrates that TestNMT is producing useful content and indicates that using it should save the developer time over creating tests from scratch.

6.3.2 Qualitative Results

Here we present some examples taken from the approximate tests generated by the trained models in each experiment. The “linked test” is the test from an example link drawn from the test set after the data pre-processing has been applied. The “approximate test” is the output of the model when provided with the function from the example link as input. The differences are highlighted with red to indicate deletions and green to indicate insertions in the diffs between the linked and generated approximate tests.

Examples from Config 1 are not included as the model did not train, however, configs 2 and 4 demonstrate the potential of TestNMT in the within-project scenario as we can see that there are only a few differences between the linked and approximate tests. For the first example of Config 2, the differences are only a change in the naming of the test case and the arguments for one function call. For the second example of Config 2, there is just a type change to a related exception type and a change to an assert condition. The only differences in the examples from Config 4 are two string literals and two changes to an assert.

6.3.2.1 Config 2 – Example 1

Pre-processed linked test:

```
@ test public void test _ minus _ %*\textcolor{deleted}{long
    period unit}*) _ zero () { local
date time t = test _ _ _ _ _ _ _ . minus (
    %*\textcolor{deleted}{,chrono unit . }*)days ) ;
assert same ( t , test _ _ _ _ _ _ _) ; }
```

Generated approximate test:

```
@ test public void test _ minus _
    %*\textcolor{inserted}{period}*) _ zero ( ) { local date
time t = test _ _ _ _ _ _ _ . minus ( %*\textcolor{inserted}{mock
    simple period . zero _ }*)
days ) ; assert same ( t , test _ _ _ _ _ _ _ ) ; }
```

6.3.2.2 Config 2 – Example 2

Pre-processed linked test:

```
@ test public void test ( ) { sql
    %*\textcolor{deleted}{transaction rollback}*) exception ex
= new sql %*\textcolor{deleted}{transaction rollback}*) exception
    ( ) ; assert true ( ex . get
message ( ) %*\textcolor{deleted}{= = null}*) & & ex . get sql
    state ( ) = = null & & ex .
get cause ( ) = = null & & ex . get error code ( ) = = ) ; }
```

Generated approximate test:

```
@ test public void test ( ) { sql %*\textcolor{inserted}{data}*)
    exception ex = new sql %*\textcolor{inserted}{data}*)
exception ( %*\textcolor{inserted}{reason}*) ) ; assert true ( ex
    . get message ( ) %*\textcolor{inserted}{. equals ( reason}*)
    %*\textcolor{inserted}{ )}*) & & ex . get sql state ( ) = =
    null & & ex . get cause ( )
= = null & & ex . get error code ( ) = = ) ; }
```

6.3.2.3 Config 4 – Example 1

Pre-processed linked test:

```
@ test public void get unique group id ( ) throws exception {
    string
group = " %*\textcolor{deleted}{vmmgroup}*) " ; string unique
    group id = "cn = %*\textcolor{deleted}{vmmgroup}*) ,
%*\textcolor{deleted}{ou = users , dc = rtp , dc = raleigh , dc =
    ibm , dc = com}*) " ; log . info (
 c , " get unique group id " , " checking with a valid group . "
     ) ;
 assert d ns equal ( %*\textcolor{deleted}{null , }*)unique group
     id , servlet . get unique
group id ( group ) ) ; }
```

Generated approximate test:

```
@ test public void get unique group id ( ) throws exception {
    string
group = " %*\textcolor{inserted}{vmmgrp}*) " ; string unique
    group id = "cn = %*\textcolor{inserted}{vmmgrp}*) ,
    %*\textcolor{inserted}{o = ibm , c = us}*) " ; log . info ( c
    , " get unique group id " , " checking with
 a valid group . " ) ; assert d ns equal ( unique group id ,
     servlet
  . get unique group id ( group ) ) ; }
```

6.3.2.4 Config 4 – Example 2

Pre-processed linked test:

```
@ test public void get groups _ empty list ( ) throws exception {
```

```
search result result = reg . get groups ( " * " , ) ; assert not
    null
 ( " search result must never be null " , result ) ; assert
     %*\textcolor{deleted}{true}*) (
result . get list ( ) . %*\textcolor{deleted}{is empty}*) ( ) ) ;
    assert false ( result .
 has more ( ) ) ; }
```

Generated approximate test:

```
@ test public void get groups _ empty list ( ) throws exception {
search result result = reg . get groups ( " * " , ) ; assert not
null ( " search result must never be null " , result ) ; assert
%*\textcolor{inserted}{equals}*) ( %*\textcolor{inserted}{,
    }*)result . get list ( ) . %*\textcolor{inserted}{size}*) ( )
    ) ; assert false ( result
 . has more ( ) ) ; }
```

6.3.3 Results Discussion

Overall these preliminary results are encouraging and show that although TestNMT may not be able to produce complete and finished tests that can be plugged directly into the system in question, it has the potential to produce recommendations that are very close to the required test. A developer may then only have to make small changes to the generated approximate test to adapt it for use in the system.

6.4 Future Directions

There are a few key areas in which this research can be expanded to produce an extensive and robust investigation. Firstly no attempt has yet been made to optimise the network architecture or hyperparameters, which could have a large impact on the results. Aspects to explore in this area include using different types of RNN units, such as GRU, DGRU, or peephole LSTM units, as well as greater numbers of layers and different attention mechanisms. Bi-directional units should also be tested as previous works have noted the effect that the order of the input sequence can have [Sutskever et al., 2014], and the effectiveness of bi-directional units in

exploiting this [Bahdanau et al., 2014]. The use of Beam Search could also be investigated.

The evaluation also needs to be extended to confirm the findings of the cross-project scenario with a larger dataset and confirm the findings of the within-project scenario with a wider range of projects of varying sizes. This will facilitate the application of statistical analysis to strengthen the results and empirically test the effect of project size on the results. This analysis could also provide insights into any other properties of a project that may affect its viability for use with TestNMT and more generally uncover any currently hidden problems with the approach.

Further, as TestNMT learns from human-written tests that contain elements that are difficult or impossible for typical unit test generation tools to produce, such as oracles and specific test inputs, TestNMT may be more useful than typical test generation tools for generating these types of elements. Testing this hypothesis should also be addressed in future work.

6.5 Conclusion

This preliminary investigation into the performance of TestNMT and, more generally, the viability of using neural machine translation as a test generation technique has shown that it has the potential to be of use, especially when applied to large individual projects. However, there is still much work to be done to optimise the implementation and carry out a full evaluation to determine the overall benefit and generality achievable by TestNMT.

Chapter 7

Automated Generation of Test Asserts

In Chapter 6 we utilised recurrent neural networks adapted from the natural language processing domain to generate unit tests using our tool TestNMT. However, as TestNMT produces approximate tests, developers still have to expend manual effort to transform them into the desired executable tests. ReAssert takes a different approach by focusing on the generation of assertions only, requiring less developer intervention and allowing the generation of more accurate code. Given both of these approaches, developers can individually decide which best suits their needs.

The process of creating and maintaining unit tests is time-consuming, error-prone, and often disliked by developers, frequently resulting in software that has a low level of test coverage. Previous work has shown that to maintain a high level of unit test coverage, the tests must be created at the same time as the tested code as retroactively creating unit tests is rarely done and only partially successful when attempted [Klammer and Kern, 2015]. Therefore, by automating parts of the unit test creation process we hope to improve the efficiency of the software engineering process and the robustness of the resulting software. To achieve this, we present ReAssert, an approach for generating JUnit assert statements using deep learning.

Test suite generation tools such as EvoSuite [Fraser and Arcuri, 2013], Randoop [Pacheco and Ernst, 2007], and AgitarOne [Agitar Technologies, 2020] employ techniques that primarily focus on generating high-coverage tests rather

than meaningful asserts and, therefore, the asserts they generate are often weak and lack specificity. This problem contributes to the deficiencies these tools show when attempting to revealing real-world faults [Shamshiri, 2015, Shamshiri et al., 2015] and to the low opinion that developers generally have of generated asserts. Almasi et al. [2017] reveal these developer frustrations, including quotes such as *"... poor assertions, sometimes there is an assertion and sometimes there is not? The assertions are mostly checking for simple stuff like list size and so on"*.

To overcome the issues that existing unit test generation techniques have with generating asserts, we turn to deep learning. Previous work [White and Krinke, 2018] investigated generating JUnit tests using a sequence to sequence recurrent neural network (RNN) trained on individual projects in an approach named TestNMT. After this, Watson et al. [Watson et al., 2020] used a similar RNN model in their Atlas approach, trained on a general corpus mined from GitHub for generating just the assert statements for JUnit tests. These previous works have demonstrated that this type of deep neural network is capable of generating useful test code, however, in the case of TestNMT, the generated candidate tests need some manual transformation before being usable and, in the case of Atlas, only 17% of the generated asserts were exact matches with the ground truth when using the raw dataset and the test (minus the asserts) was required to already been written. In addition, only a single assert could be generated for a given method and no analysis beyond lexical accuracy was performed to assess the usefulness of the asserts.

Our approach, called ReAssert, builds on the previous work by focusing on generating JUnit asserts, similar to Watson et al. [2020], but utilises a project-based approach that does not require the (assert-less) test to be written before asserts can be generated and allows for the generation of more than one assert per tested method. ReAssert can use three different models and includes the new Reformer model [Kitaev et al., 2020] in addition to the two RNN models used in TestNMT [White and Krinke, 2018] and Atlas [Watson et al., 2020]. Reformer utilises a state-of-the-art deep learning architecture and may push the accuracy and

usefulness of the generated asserts beyond that of the previous models. All three models are applied to both ReAssert and a re-implementation of Atlas, and we also expand the evaluation in two other ways to focus more on real-world usefulness and applicability. Firstly, we perform an extended lexical accuracy evaluation (how close is the text of the generated asserts to the ground truth from the test set) and an analysis of the uniqueness of the generated asserts, which gives further evidence as to their usefulness. Secondly, for ReAssert, we go beyond the static lexical accuracy analysis, to use a dynamic analysis that determines how many generated asserts compile and how many pass when inserted into existing tests. By evaluating all three models for both ReAssert and Atlas with the uniqueness and dynamic evaluation, along with the typical lexical evaluation, we demonstrate which approach and model combinations are the most useful in a real-world setting. The main contributions of this chapter are:

- ReAssert, a project-based deep learning approach for the generation of unit test asserts implemented for JUnit.
- An evaluation of ReAssert using lexical accuracy and dynamic analysis with Reformer, a new state-of-the-art transformer-based model and two RNN-based models from previous work.
- An extended comparative evaluation of all three models using lexical accuracy and uniqueness on a previous approach, Atlas.
- Takeaway messages for researchers and practitioners concerning the construction of data sets when applying sequence to sequence learning for code generation.

7.1 Background

Test assert generation has previously been the domain of test suite generation tools, such as EvoSuite [Fraser and Arcuri, 2013], Randoop [Pacheco and Ernst, 2007], and AgitarOne [Agitar Technologies, 2020]. However, these tools primarily generate tests through methods that optimise for coverage, such as genetic programming

and random testing. Therefore, these tools produce tests that aim primarily to achieve high coverage rather than include meaningful assert statements, resulting in a deficiency in the ability of these generated tests to detect real faults. This was quantified in a study [Shamshiri, 2015] which discovered that neither EvoSuite, Randoop, or AgitarOne were able to detect more than 40.6% of faults in the Defects4J [Just et al., 2014] database. As 63.3% of the undetected faults were covered, this indicates weaknesses in the asserts of the generated test cases.

With the advent of deep learning and the successful application of deep learning techniques to tasks that require the processing of sequential data, especially language-based tasks such as machine translation [Kalchbrenner and Blunsom, 2013, Sutskever et al., 2014, Cho et al., 2014], an opportunity to apply these methods to source code was created. These deep learning models have been applied to a wide range of software engineering problems such as code summarisation [Allamanis et al., 2016, Alon et al., 2018, Iyer et al., 2016], program comprehension [Henkel et al., 2018], clone detection [White et al., 2016], code similarity [Zhao and Huang, 2018], method name generation [Alon et al., 2019], comment generation [Hu et al., 2018], traceability [Guo et al., 2017], and type inference [Hellendoorn et al., 2018]. However, for the task of code generation, deep learning models initially were only applied to the generation of implementation code [Ling et al., 2016], not test code. TestNMT [White and Krinke, 2018] applied these techniques to unit test generation by utilising a sequence to sequence RNN-based neural network adapted from a model that had previously been used for neural machine translation and applied it to translate from Java methods to JUnit tests. TestNMT demonstrated that, when applied to large individual projects, this technique is capable of generating some tests that only require a small amount of manual effort on the part of the developers to turn into useful tests. However, many tests still required a large amount of effort to be converted and the approach was not effective when using a single multi-project data set to train a general model that works for any project. After TestNMT, Atlas [Watson et al., 2020] applied the same type of model to the problem of generating test code, however, instead

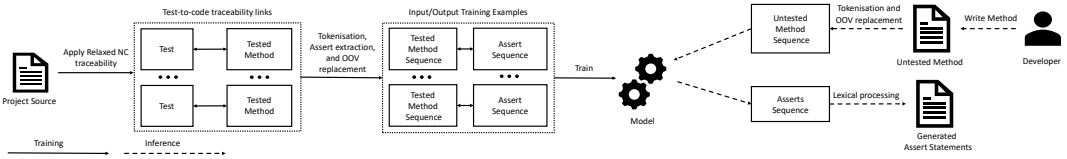

Figure 7.1: Overview of the ReAssert approach.

of attempting to generate whole tests, Atlas attempts only to generate the asserts for JUnit test cases. This removes the issue of developers having to expend a lot of effort to transform the output of the model into usable code and also allows the training of a single network on a corpus of general Java code to apply to any project. However, unlike TestNMT [White and Krinke, 2018], Atlas only uses tests with a single assert statement and the test code (minus the assert) is included in the source sequence, requiring that a developer writes a test before using Atlas, which can then only generate a single assert. The model used by TestNMT [White and Krinke, 2018] and the model used by Atlas [Watson et al., 2020] are utilised in this work for a comparative evaluation with ReAssert and the Reformer model.

7.2 Approach

The ReAssert approach, illustrated in Figure 7.1, facilitates the generation of assert statements for a given method by using deep neural network models trained on pairs of assert statements and tested methods, extracted from existing test-to-tested-method pairs. To train the model, we start by gathering the test-to-tested-method pairs from a target project via test-to-code traceability links [Rompaey and Demeyer, 2009]. Then, for each test-to-tested-method pair, we extract the assert statements from the test method and concatenate them to produce the string of assert statements associated with the tested method. The tested method and assert strings are then processed into input and output token sequences, known as method sequences and assert sequences respectively. These sequences are used to train the model. Once trained, the model can be used to generate an assert sequence, given a method sequence as input. The generated assert sequences are then processed into syntactically correct code that can be directly inserted into a test for that method. Figure 7.2 illustrates an example from Stanford CoreNLP

for how ReAssert generates asserts for a new method by processing the method into an input sequence, inferring over the trained model, and processing the output sequence into syntactically correct asserts. As the example shows, the generated assert statements can easily be expanded into a test.

We specifically target our work more toward applicability than previous work by ensuring that we do not apply any filtering or abstraction and we do not use any prediction techniques that generate multiple outputs, such as beam search. We also only use the training set to create the vocabulary that is used when generating the method sequences and assert sequences. This is to ensure, firstly, that we are evaluating in a scenario that is true to real-world development and, secondly, that we minimise the amount of work that a developer has to do to utilise the produced assert statements in their code. Also, in contrast to Atlas, ReAssert does not include the test code in the source sequence as this would require the developer to have already written the test case (except for the assert statements) before using ReAssert to generate asserts. As we believe that the generated asserts should help the developer to write the rest of the test, this is an important improvement over prior work. Our use of tests that have multiple assert statements is another improvement over Atlas which only uses tests that contain a single assert statement. We believe this further increases the applicability of ReAssert.

7.2.1 Test-to-code Traceability Establishment

Given the code for a project, we first need to extract the test-to-code traceability links to build our training and testing data sets. Establishing test-to-code traceability links is an open research problem in software engineering for which multiple different techniques have been developed. Each technique has its own strengths and weaknesses, resulting in different balances between precision and recall [Rompaey and Demeyer, 2009, White et al., 2020]. Finding the right balance of precision and recall is important for building a data set for machine learning as if the precision is too low, the data will have too much noise (incorrect links) but if the recall is too low the data set will be too small to effectively train from. In addition, the optimal precision versus recall trade-off differs depending on which data set we are

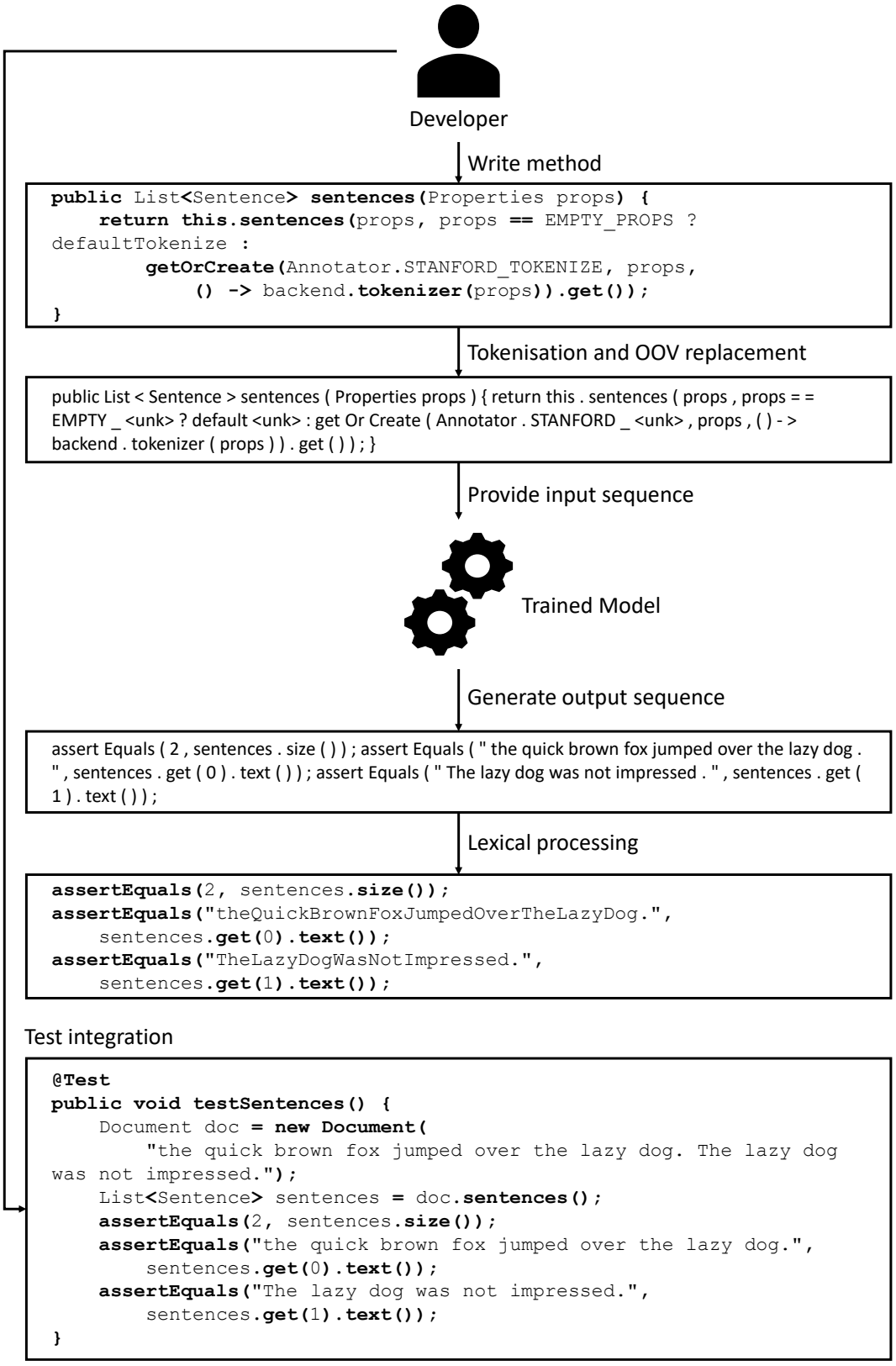

Figure 7.2: Example from the Stanford CoreNLP project demonstrating the ReAssert process to generate asserts for a method.

constructing. When constructing the training set, we prefer recall, however, for the validation set (used for configuring the parameters of the networks) and test set (used for the evaluation) we prefer precision. This is because when we are training the model we want to ensure we have as much data as possible, whereas, when we are evaluating the model using the validation or test set, we want to ensure that we are not evaluating the model with noisy data.

When training a model, we can tolerate some noise in the data as we want to maximise the amount of data and, even if a link is technically incorrect, we may still be able to learn some useful structure from it. An example of this can be seen when looking at tests for commonly overridden methods such as *equals* or *toString*. In these cases, even if a link is incorrect, a link between the test for the *equals* method of one class to the *equals* method of a different but similar class, the network may still learn some useful information about the general structure of *equals* tests because most tests for *equals* methods tend to be very similar. However, when we are evaluating the model, we want to ensure that there are as few incorrect links as possible as we don't want to be evaluating the model by asking it to generate something that is incorrect. Doing so will give an inaccurate view of how well the model performs, usually resulting in an underestimation of its accuracy.

Given the above concerns, we firstly want to find a high precision technique for building the validation and training sets. Using the recent work by White et al. which compares the precision and recall of multiple techniques [White et al., 2020], we selected the naming conventions (NC) technique for building these data sets as it has a precision of 100%. The naming convention technique establishes links by taking the fully-qualified names (FQNs) of both the tested method and the test method and comparing them after the word *test* has been removed from the test method name. If the names match exactly, the test method is linked to the tested method. However, given the very low recall of only 11%, this technique was not suitable for building the training set so we created a variant of this technique called "Relaxed NC" (we conversely call the standard NC "Strict NC"). Relaxed NC utilises the same concept as Strict NC but instead of performing the matching

on the FQN, the matching is performed only over the tested method and test method name. Therefore, a link will still be created even if the class name does not match the test class name. While this will create some incorrect links, we can often still learn some useful structure from these links (as described above) and it gives us much more data to train on, which is critical, especially for the smaller projects.

7.2.2 Data Set Construction

To build the data sets, we start by constructing all the Strict NC and Relaxed NC links and place all the Relaxed NC links into the training set. The Strict NC links are then split between the validation set (used for configuring the parameters of the networks) and test set (used for the evaluation), up to the maximum size of 100 links for each set. Any excess Strict NC links are placed in the training set. As we want to ensure that we are not unfairly biasing the model, we then filter out any links from the training set that appear in either the validation or the test sets. This filtering can result in a large reduction of the number of links in the training set, with the number of links in the validation and test set greatly influencing the magnitude of this reduction as the larger the validation and test sets are, the more links will have to be removed from the training set. Therefore, it is important to balance the sizes of the sets so that each set has an adequate amount of links to perform its function. This is why we limit the number of links in the test and validation sets to 100. Now that we have the links for each set we must process the links into pairs of source sequences (from the tested methods) to target sequences (from the tests). To do this we first tokenise the source for each artefact, build the vocabularies based on the tokenised sequences, then replace out-of-vocabulary (OOV) tokens with UNK. Our tokenisation process consists of stripping all non-printing and non-alphanumeric characters that are not used in Java, adding spaces around all programming language characters, de-camelcasing identifiers and adding spaces around the resulting tokens. This results in a sequence of individual tokens consisting of the split identifiers and programming language characters. At this stage the tested methods have been fully tokenised into source sequences, however, for the tests we want to keep only the assert statements so we add another stage of

processing for the tests where detect which tokens are part of assert statements by checking for the "assert" token, finding the next opening parenthesis and its partner closing parenthesis and treating all the tokens in-between as being part of the assert statement. Any tokens that are not determined to be part of an assert statement are deleted. This results in a target sequence that is just the tokenised assert statements for the test. It is important to note that, unlike Watson et al. [2020], we use tests that have multiple assert statements and we add all of the assert statements in the test to the target sequence. Once we have applied this process to all the code snippets for the tested methods and the tests we have our sets of input-output examples (source sequence to target sequence pairs). We then build the source and target vocabularies by collecting all the tokens in all the sequences and taking the top n most frequent tokens, where n is the desired size of the vocabulary. The vocabularies are then used to replace OOV tokens in the sequences by replacing any token which does not appear in the relevant vocabulary with UNK.

7.3 Models

To compare the performance of established and new models, we selected a set of networks consisting of two that use a traditional seq-to-seq architecture with recurrent neural network (RNN) units [White and Krinke, 2018, Watson et al., 2020], and a more efficient variant of the newer Transformer architecture called Reformer [Kitaev et al., 2020]. All of these models utilise the encoder-decoder with attention architecture type where the encoder encodes the source sequence into a vector representation which is then decoded into a target sequence by the decoder. The attention mechanism allows for modelling of out-of-sequence dependencies by attending over the whole source sequence. This is the typical architecture used for sequence to sequence learning tasks, such as neural machine translation.

7.3.1 RNN models

The two RNN type models we use are ATLAS [Watson et al., 2020] and TEST-NMT [White and Krinke, 2018]. For both of these models, the encoder builds the vector representation of the source sequence by traversing the sequence one token

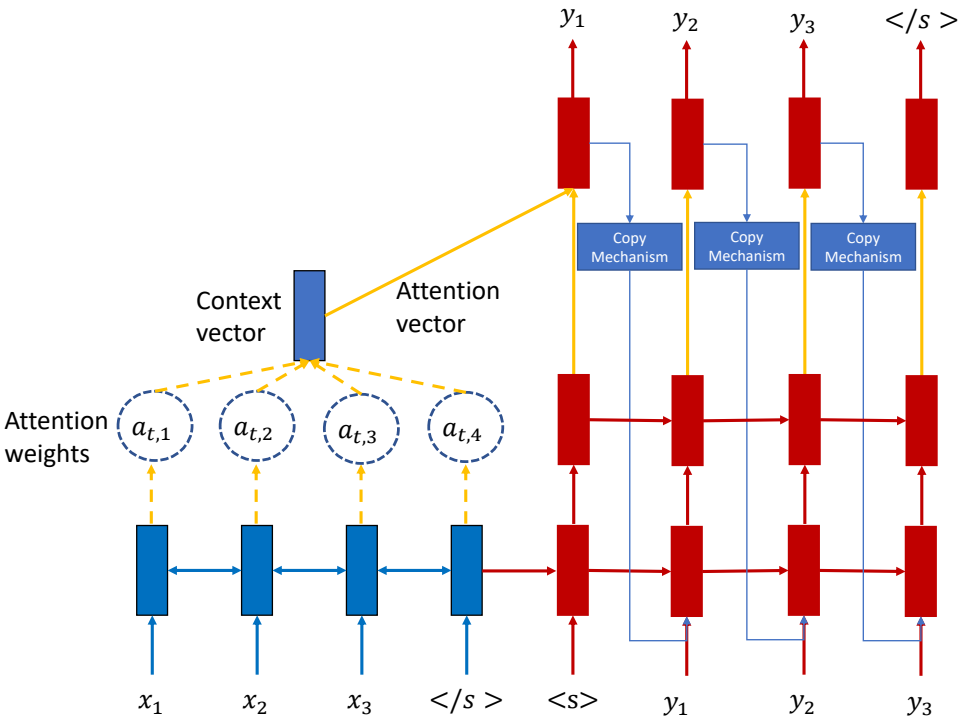

Figure 7.3: ATLAS/RNN

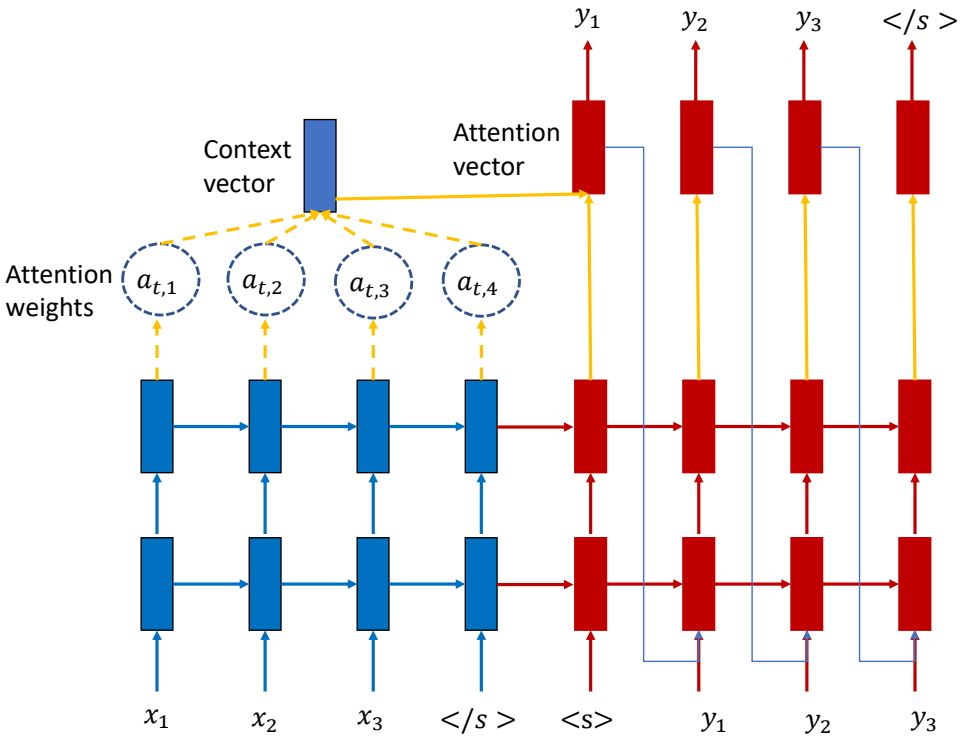

Figure 7.4: TESTNMT/RNN

at a time converting each token into a vector embedding via an embedding layer which is then provided as input to the encoder RNN unit for that time step. After the source sequence has been fully processed, the final hidden state of the encoder RNN is used to initialise the hidden state of the decoder RNN. Then, at each time step, the decoder RNN uses the current hidden state, the previously generated target sequence token, and the attention mechanism, to generate a new target sequence token. This continues until the end-of-sequence token is generated.

The attention mechanism assists in determining the next token by assigning attention weights to each of the tokens in the source sequence, computing a context vector representing the full attention, and combining them with the hidden state of the decoder to compute the attention vector. The attention weight α_{ts} for a given target token and source token is computed by performing a normalised comparison between the target hidden state $\mathbf{h}_t$ and the source hidden state $\overline{\mathbf{h}}_s$ using the score function:

$$\alpha_{ts} = \frac{\exp \text{score}(\mathbf{h}_t, \overline{\mathbf{h}}_s)}{\sum_{s'=1}^{S} \exp \text{score}(\mathbf{h}_t, \overline{\mathbf{h}}_{s'})}$$

These attention scores are used to compute the context vector $\mathbf{c}_t$ using a weighted sum $\mathbf{c}_t = \sum_{\mathbf{s}} \alpha_{ts} \overline{\mathbf{h}}_s$ and the attention vector is computed by combining the context vector with the current decoder hidden state $\mathbf{a}_t = \tanh(\mathbf{W}_c[\mathbf{c}_t; \mathbf{h}_t])$.

The attention vector is then passed to the softmax layer to generate the predicted target token. After decoding the target token for the current step, the attention vector is passed to the next step in the decoder to ensure that past attention information is carried forward. This helps to capture contextual and out-of-sequence dependencies by allowing the network to attend to the source tokens in differing amounts as the target sequence is generated. The attention mechanism can be global (attending over the whole source sequence) or local (attending over only a subsequence of the source sequence), as visualised in Figure 7.5.

While the Atlas and TestNMT networks both utilise this same basic architecture, as shown in Figure 7.3 and Figure 7.4, and both utilise LSTM cells with the tanh activation function, they do differ in several significant ways. One major difference is that the Atlas network includes a copy mechanism [Gu et al., 2016]

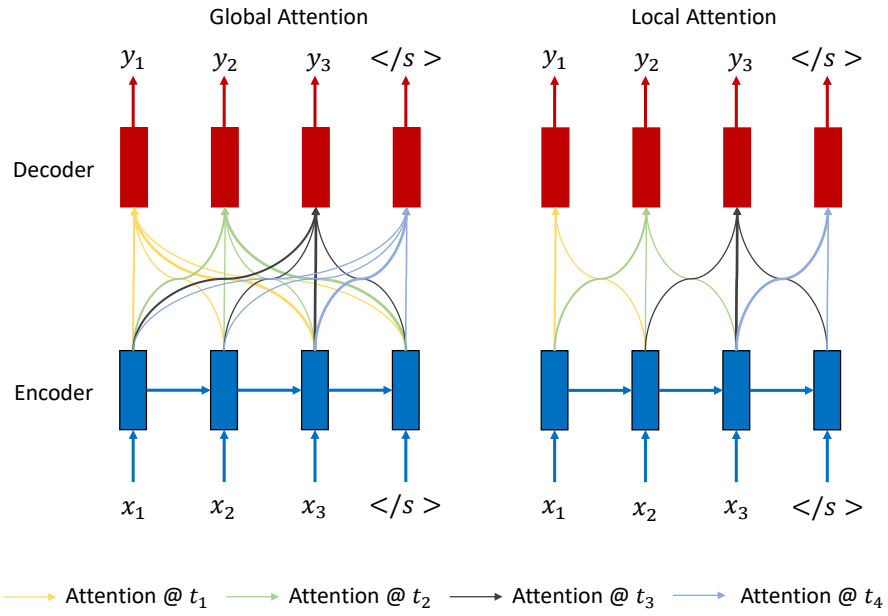

Figure 7.5: Overview of attention in RNN networks.

that replaces UNK token predictions with a token from the source sequence. In contrast, the TestNMT network does not use an UNK replacement mechanism. Another difference is that the TestNMT network uses two unidirectional layers in the encoder and two layers in the decoder, whereas the Atlas network uses a single bidirectional layer in the encoder and two layers in the decoder. This difference between the networks can provide some insight as to the relative effect of the directionality of layers vs the number of layers. The networks also differ in the way that attention is calculated. Atlas uses Bahdanau's additive technique [Bahdanau et al., 2014]:

$$\text{score}(\mathbf{h}_t, \overline{\mathbf{h}}_s) = \mathbf{v}_a^\top \tanh(\mathbf{W}_1 \mathbf{h}_t + \mathbf{W}_2 \overline{\mathbf{h}}_s)$$

TestNMT uses Luong's multiplicative style [Luong et al., 2015b]:

$$\text{score}(\mathbf{h}_t, \overline{\mathbf{h}}_s) = \mathbf{h}_t^\top \mathbf{W} \overline{\mathbf{h}}_s$$

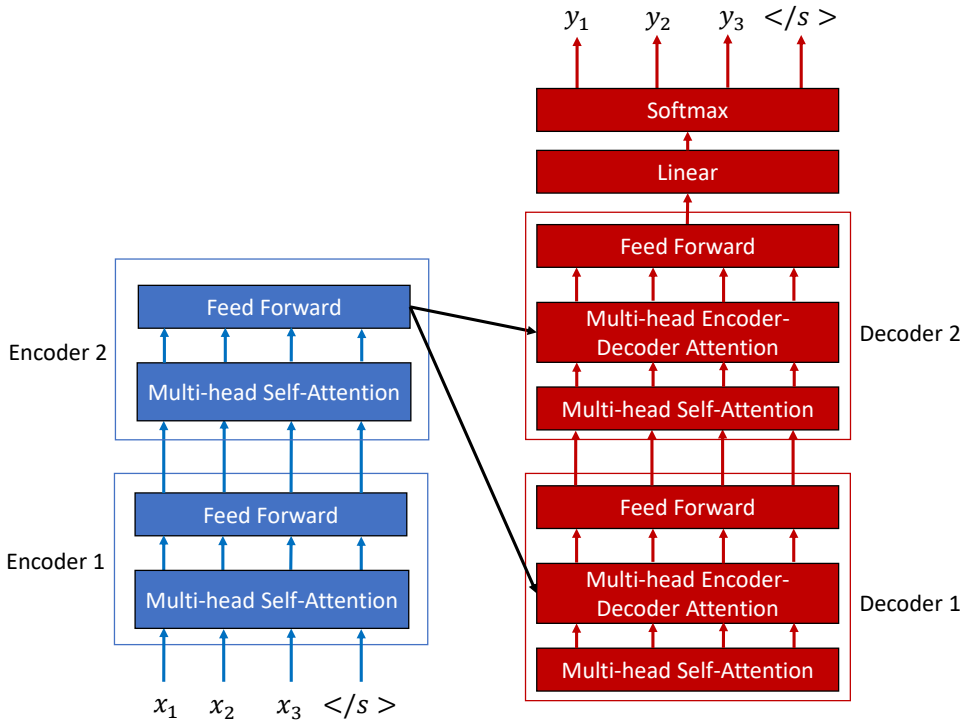

Figure 7.6: Two-layer Transformer (simplified)

7.3.2 Reformer model

The Reformer model [Kitaev et al., 2020] is a less resource-intensive iteration of the recently popularised Transformer model [Vaswani et al., 2017]. The Transformer model differs from the RNN based models in that it relies solely on attention and simple point-wise fully connected feedforward network layers. The Transformer still employs encoder-decode attention, however, it also utilises another form of attention called multi-headed self-attention. The Transformer architecture is comprised of a series of layers stacked on top of each other where each layer contains an encoder and a decoder. The source sequence is fed through each encoder sequentially and the result is given to each decoder along with the output from the previous decoder (if one exists). The encoders and decoders are themselves comprised of sub-layers with the encoders containing multi-head self-attention and feedforward sub-layers, while the decoders contain multi-head self-attention, multi-head encoder-decoder attention, and feedforward sub-layers. The output from the final decoder passes through a single linear layer and into the softmax to compute the output token predictions. Figure 7.6 shows a high-level example of a two-layer

Transformer architecture.

The multi-headed attention mechanism works to improve performance by allowing the model to attend to information from multiple different representation subspaces concurrently, enhancing the model's ability to focus on different positions. This is done by projecting the information from the input vectors h times, where h is the number of heads, performing the attention calculations over each head, and then combining the results from all heads. All heads are initialised randomly and trained with random dropout so that different heads learn to attend more appropriately over different positions, making the combination of multiple heads more effective than a single attention function.

However, although Transformers achieve state-of-the-art performance they can be very resource-intensive due to the extreme number of parameters and the size of the calculations required for multi-head attention. Given this limitation, the Reformer model was created to reduce the resource requirements of the model while still applying the concepts that make the Transformer effective. To do this, Reformer targets the three main sources of resource consumption in the Transformer, specifically the large self-attention computation, which is OL^2 for sequences of length L, the large numbers of layers, and that the feedforward layers are often much deeper than the attention activations. The Reformer deals with the size of the attention computation by employing Locality Sensitive Hashing (LSH) attention and deals with the large number and depth of layers by using a Reversible Residual Network (RevNet) [Gomez et al., 2017] with chunking. However, as the current implementation of the Reformer [Google, 2020] does not use LSH attention for encoder-decoder sequence to sequence tasks (only decoder-only language models), we omit discussion of LSH here and focus on RevNet and chunking.

RevNet improves the memory consumption of the model by replacing the Residual Network (ResNet) units of the standard Transformer with RevNet units. This reduces memory consumption as ResNet units need to store all of the activations for each layer in memory for the backpropagation calculations, requiring

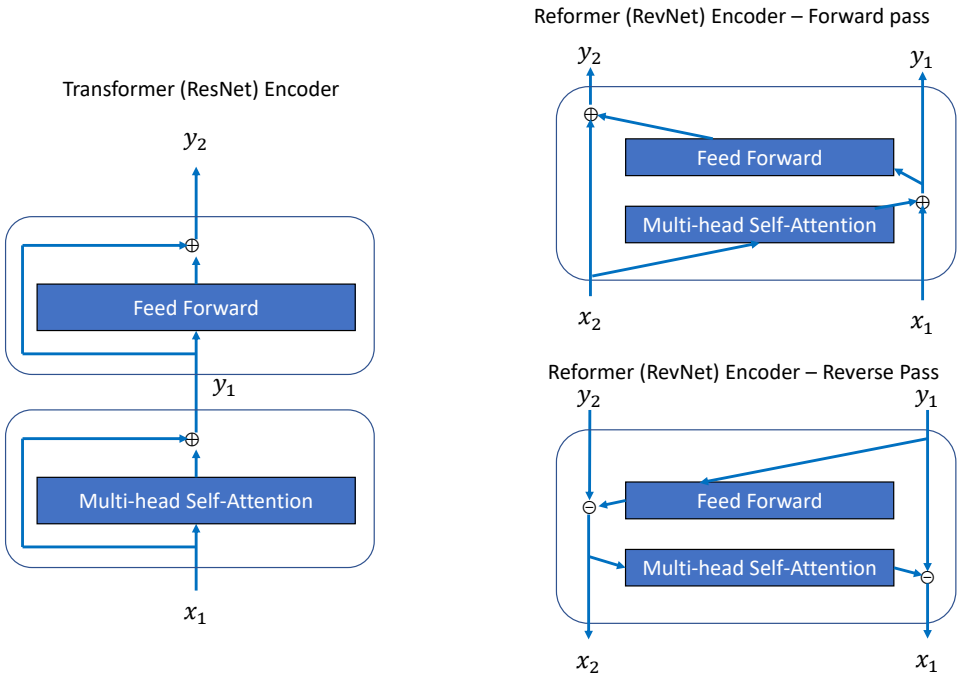

Figure 7.7: Residual Network (ResNet) units (used in Transformer) vs Reversible Network (RevNet) units (used in Reformer).

a lot of space when using deep networks. RevNet units, however, can recover the activations in the n^{th} layer from the activations in the $(n+1)^{th}$ layer, meaning only the activations from one layer need to be stored at any time. Figure 7.7 provides an example of this by comparing a Transformer encoder with ResNet units and a Reformer encoder with RevNet units. In traditional transformers, the self-attention and feedforward sub-layers are each contained in their own ResNet units and require that all the activations be stored. However, in the Reformer, the self-attention and feedforward layers are wrapped together in a single RevNet unit where the input (activations from the previous layer) can be recovered from the outputs (activations from the current layer). This is done by partitioning the units in each layer into two groups with inputs (x_1, x_2) and outputs (y_1, y_2) and arranging them so that the inputs can be recomputed from the outputs simply by reversing the direction of computation and changing the combination operators from addition to subtraction.

The final improvement is using chunking in the feedforward sub-layers. Improving the efficiency of these layers is important as the dimensionality of the vectors in these layers can reach 4K or higher. Chunking can be used because the computations are independent across the positions in a sequence, meaning that

Table 7.1: Subject project details.

Project	Version	Set Sizes		
		Training	Validation	Test
Apache OpenNLP	1.9.1	2027	66	66
DL4J	1.0.0	2122	67	68
EJML	0.38	26999	100	100
ND4J	1.0.0	5728	49	50
Stanford CoreNLP	3.9.2	3930	100	100

the computation can be split into n chunks, as shown in Equation (7.1), which are executed in series, reducing memory requirements.

$$\begin{aligned} &Y_2 = [Y_2^{(1)}; ...; Y_2^{(n)}] = \\ &[X_2^{(1)} + \text{FeedForward}(Y_1^{(1)}); ...; X_2^{(n)} + \text{FeedForward}(Y_1^{(n)})] \end{aligned} \tag{7.1}$$

By using this combination of efficiency improvements, the Reformer can achieve performance on par with that of traditional large Transformers while being much more memory-efficient and faster, especially on large sequences.

7.4 Evaluation – ReAssert

We evaluate the ReAssert approach using the two RNN-based models from TestNMT [White and Krinke, 2018] and Atlas [Watson et al., 2020] in addition to the new Reformer model [Kitaev et al., 2020]. The projects that we selected to perform the evaluation are Apache OpenNLP[1], Deep Learning for Java (DL4J)[2], Efficient Java Matrix Library (EJML)[3], ND4J[4], and Stanford CoreNLP[5]. These projects are well-tested, widely used, and include two natural language processing libraries (Apache OpenNLP and Stanford CoreNLP), two linear algebra libraries (EJML and ND4J), and one deep learning library (DL4J). The details of the data sets obtained from these projects are given in Table 7.1.

The evaluation of ReAssert is split into two research questions which col-

[1] https://opennlp.apache.org/
[2] https://deeplearning4j.org/
[3] http://ejml.org/
[4] https://github.com/deeplearning4j/nd4j
[5] https://stanfordnlp.github.io/CoreNLP/

lectively evaluate the usefulness of the generated asserts by performing a lexical accuracy and a dynamic analysis over individual asserts and the applicability of these asserts to the test suites of the projects.

7.4.1 Research Question 1 (Assert Accuracy)

How many of the generated asserts are exact matches, passing, and compiling? In RQ1 we examine the effectiveness of ReAssert at generating individual asserts when paired with each of the three models. We perform an analysis on the generated asserts that first establishes which are exact matches (the generated assert exactly matches an assert written by the developers), then which of the remaining asserts compile and which of those then pass when used to replace the developer written asserts in the existing test.

Experimental Setup To evaluate ReAssert, we first take each test-to-tested-method pair from the test set, provide the tested method as input to the model, get the output sequence, process the output sequence into syntactically correct assert statements, and compare those statements to those given in the test method. Where a generated assert exactly matches any assert in the test, we mark it as an exact match (and therefore also as passing and compiling). We can automatically categorise exact matches in this way as the test suites are fully green (have no failing tests) for all of the projects. For generated asserts that are not exact matches, we take the test from the pair, remove all existing asserts from the test method, insert the non-matching assert at the end of the test method, attempt to compile and, if compilation is successful, run the test to see if it passes. We repeat this process for all test-to-tested-method pairs in the test sets of all the projects.

Findings The results, presented in Table 7.2, show that, in general, the three models perform similarly. However, there are some noticeable trends, such as the TestNMT model being slightly higher for F1 score in most projects and the Reformer model being slightly lower in some projects (precision, recall, and F1 scores are for exact matches only). Note that in most cases, there are more asserts that pass than asserts that are exact matches and there are more asserts that compile than asserts that pass (i.e., there are some asserts generated that compile, but where the test fails). A

Table 7.2: RQ1 – Exact match, passing, and compiling asserts.

	Apache OpenNLP	DL4J	EJML	ND4J	Stanford CoreNLP
			TestNMT/RNN		
Gen. Asserts	173	191	179	118	236
Matches	45	39	51	10	103
Precision (%)	26	20	28	15	44
Recall (%)	21	13	27	5	30
F1	23	16	27	7	35
Passing	47	40	61	10	108
Passing (%)	27	21	34	15	46
Compiling	47	44	69	10	120
Compiling (%)	27	23	39	15	51
			Atlas/RNN		
Gen. Asserts	150	111	184	96	197
Matches	47	21	45	9	85
Precision (%)	31	19	24	9	43
Recall (%)	22	7	24	4	25
F1	26	10	24	6	31
Passing	47	21	53	14	87
Passing (%)	31	19	29	22	44
Compiling	47	28	63	14	100
Compiling (%)	31	25	34	22	51
			Reformer		
Gen. Asserts	192	217	212	132	271
Matches	30	17	38	8	88
Precision (%)	16	8	18	6	32
Recall (%)	15	6	20	4	25
F1	15	7	19	5	29
Passing	30	17	49	9	98
Passing (%)	16	8	23	7	36
Compiling	31	26	61	9	110
Compiling (%)	16	12	29	7	41

discussion exploring the implications of these results can be found in Section 7.6.1.

7.4.2 Research Question 2 (Test Applicability)

What percentage of tests contain at least one assert from the categories? In RQ2 we perform an analysis that uses the generated asserts from RQ1 where, for each assert

Table 7.3: RQ2 – Percentage of tests with at least one generated assert that is an exact match, passing, or compiling.

	Apache OpenNLP	DL4J	EJML	ND4J	Stanford CoreNLP
			TESTNMT$_{/RNN}$		
Matched	26%	27%	29%	20%	40%
Passing	27%	28%	32%	20%	44%
Compiling	27%	31%	37%	20%	49%
			ATLAS$_{/RNN}$		
Matched	26%	12%	32%	18%	40%
Passing	26%	12%	37%	22%	40%
Compiling	26%	18%	42%	22%	44%
			Reformer		
Matched	17%	13%	28%	16%	41%
Passing	17%	13%	33%	18%	45%
Compiling	18%	21%	38%	18%	48%

category (exact match, passing, compiling), we determine the percentage of tests that have at least one generated assert from that category. This is to give evidence as to how useful the generated asserts are across a whole test suite.

Experimental Setup To answer this research question, we use the asserts generated for RQ1 and, for each category, count the percentage of tests in each project that contains at least one assert from that category.

Findings The results, presented in Table 7.3, show that in the best case, using the TESTNMT model, nearly half of the tests in a project receive a generated assert that at least compiles (Stanford CoreNLP). On average, a third of tests receive at least one generated assert that compiles and 28% receive at least one exact match assert. When comparing models the performance is similar but, like RQ1, the Reformer model is slightly lower in some projects than the two RNN models TESTNMT and ATLAS.

7.5 Evaluation – ATLAS

We present our evaluation of the three models using ATLAS to determine if we can improve over previous results using the new Reformer model or the TESTNMT

model. ATLAS significantly differs from REASSERT in two aspects: (a) ATLAS uses the tested method and the test method for training and querying and (b) ATLAS uses a very simple and imprecise test-to-code traceability technique. However, it has been applied in a multi-project setting in which the corpus is created from a large number of projects. To construct the data set, Watson et al. [2020] mined 9,275 projects from GitHub and used the Spoon library [Pawlak et al., 2016] to extract the test methods by looking for the *@Test* annotation. However, any test that contained more than one assert statement or was longer than 1000 tokens was discarded, leaving 188,154 tests in total.

The traceability technique used by Watson et al. [2020] in ATLAS is a simplified version of Last Call Before Assert (LCBA) [Rompaey and Demeyer, 2009]. Instead of using a static or dynamic call graph, ATLAS simply extracts the name of the last called method before the assert and then searches the package for methods of the same name. If no match can be found, ATLAS extends the search to the whole project. While having the benefit of being able to be used on a large and diverse corpus, this method for establishing test-to-code traceability links can result in a lot of noise in the data. The noise can be especially bad if multiple classes define methods with the same names or if there are a lot of overloaded methods. After establishing the links, ATLAS processes them into input-output examples by extracting the asserts from the tests to use as the outputs with their respective tested methods as the inputs. Further filtering is then performed on the resulting data set to remove duplicate examples and any example where the assert contains a token that does not appear in the vocabulary. The data set provided by Watson et al. is already filtered, so our evaluation uses the data set directly without mining or extraction.

The evaluation of the three models using ATLAS is split into three research questions which collectively evaluate the usefulness of the generated asserts by looking at the accuracy (RQ3 and RQ4) and uniqueness (RQ5) of the generated asserts.

Table 7.4: RQ3 – Exact match asserts.

	TESTNMT/RNN	ATLAS/RNN	Reformer
Generated Asserts	18 817	18 817	18 817
Matched Asserts	1355	3323	**5262**
Accuracy	7%	18%	**28%**

7.5.1 Research Question 3 (Assert Accuracy)

How many of the generated asserts are exact matches for developer written asserts? For RQ3 we examine the effectiveness of our three models at generating exact match asserts, similar to RQ1, but in the ATLAS setting. The evaluation is limited to exact matches as performing a dynamic analysis to discover passing or compiling non-matched asserts is not possible with Watson et al.'s data set. We do not use beam search when applying the ATLAS model as it results in multiple tokens being predicted for the same position in the output sequence. Therefore, when it is utilised in the same way as Watson et al. and all of the predicted tokens are used to build a list of possible outputs, the output of the model is a set of candidate assert recommendations rather than a single assert.

Experimental Setup To answer RQ3, we use the model to generate an assert for each test-to-tested-method pair in the test set and compare the generated assert to the assert from the test as present in the data set. Where the generated assert and the test assert match, we count it as an exact match and use the number of exact matches divided by the total number of generated asserts to calculate the precision.

Findings The results, as shown in Table 7.4, reveal that TESTNMT is the worst-performing model with only 7% precision. While ATLAS fairs much better than TESTNMT with 17% precision, Reformer is the best by a wide margin at 28% precision. Note that our results of 3323 exact matches for our reimplementation of ATLAS is identical to the results reported by Watson et al. [2020], giving us confidence that our reimplementation is faithful to the original ATLAS. Discussion regarding these results can be found in Section 7.6.1.

7.5.2 Research Question 4 (Edit Distance Evaluation)

How far from exact matches are non-matched asserts? RQ4 investigates how much transformation, measured in absolute and relative token-based edit distance, is required to turn non-exact match asserts into exact matches. These measures give evidence as to how useful non-matched asserts are to developers as, intuitively, the easier it is to turn a non-exact match assert into an exact match, the more useful that assert would be for developers. We use the relative edit distance as we want to take the length of the asserts into account to avoid favouring models that are more likely to produce short asserts. Discussions relating to the length of generated asserts can be found in Section 7.6.2. We also report the count of asserts that are less than two token changes away from being a matched assert. This group, therefore, includes asserts that are either exact matches or only one token change away from an exact match. Given the ease of changing a single token, we consider these non-matched asserts to be in the group of asserts which should be of most use to developers.

Experimental Setup This evaluation is performed using the asserts generated for RQ5. First, we find the edit distance by computing the Levenshtein distance between the generated assert and each developer written assert, using tokens instead of characters as the atomic unit, and take the smallest distance. The distance is then used to compute the relative edit distance by dividing it by the number of tokens in the assert with the most tokens out of the generated assert and the developer written assert.

Findings The results, as shown in Table 7.6 reveal that the ATLAS and Reformer models perform essentially equivalently to each other in edit distance, with the TESTNMT model trailing behind them. However, when looking at asserts that are less than 2 token changes away from an exact match, the Reformer model has a clear advantage.

7.5.3 Research Question 5 (Uniqueness Evaluation)

What is the uniqueness of generated asserts? RQ5 investigates how unique the generated asserts are, which is important as the more unique an assert is, the more useful it is likely to be. This belief is driven by the fact that, in general, asserts that

Table 7.5: RQ4 – Exact match edit distance evaluation.

	TESTNMT/RNN	ATLAS/RNN	Reformer
Median Edit Dist.	5	**2**	**2**
Mean Edit Dist.	5.07	**3.85**	4.00
Median Rel. Edit Dist.	0.28	**0.15**	**0.15**
Mean Rel. Edit Dist.	0.26	**0.18**	0.19
Dist. < 2 Count	3375	6984	**8180**
Dist. < 2 (%)	18%	37%	**43%**

are more unique are more likely to encode specific information about the task. For example, an assert statement that simply checks the equality of two generically named variables contains less specific information than an assert statement that contains method calls. We use the asserts generated by each of the three models only with the Watson et al. data set because this data set is taken from a large number of projects and, therefore, demonstrating the ability to generate a diverse and unique range of asserts is important.

To evaluate uniqueness, we first look at the absolute number of unique asserts the models produce and what percentage of generated asserts were unique at generation time for all generated asserts and all matched asserts. This measures how frequently the models are generating unique asserts. However, we do not only want to look at unique asserts but also the distribution of non-unique asserts. We perform this analysis with a view that a more even distribution, in general, indicates a greater diversity of asserts and, therefore, greater useful informational content. This assumption is discussed in more detail in Section 7.6.2. To assess the distribution of non-unique asserts, we compute the absolute number and percentage of matched asserts that are among the top five and top ten most common asserts, essentially showing us how common the most common asserts are. To demonstrate a good ability to generate asserts with a high degree of uniqueness, we are looking for a model to maximise the unique assert percentages while minimising the most common assert percentages.

Experimental Setup To conduct RQ5, for each model, we first take the list of assert statements generated by the model and group identical asserts together. The sizes of

Table 7.6: RQ5 – Assert uniqueness analysis results.

	TestNMT/RNN	Atlas/RNN	Reformer
Unique Asserts	470	11 496	**13 331**
Unique Asserts (%)	2%	62%	**71%**
Unique matched	97	948	**2647**
Unique matched (%)	7%	29%	**50%**
Top 5 matched (%)	59%	25%	**16%**
Top 10 matched (%)	71%	37%	**20%**

these groups give us the count of how many times each assert appears. We take the number of groups as our count of distinct asserts and calculate this as a percentage of all the generated asserts. This is the percentage of asserts that were unique at the time of generation. The groups are then ordered by their cardinalities and we take the sum of the cardinalities of the top five and the top ten largest groups and use these to calculate the percentage of generated asserts that are members of these groups.

Findings The results, as shown in Table 7.6 reveal that Reformer is the best model for uniqueness as it has the highest percentages of unique asserts and the lowest percentages of asserts that are among the top 5 and top 10 most common asserts. These results show Reformer is better for uniqueness than the next best model, Atlas, by a sizeable margin in all measures. The TestNMT model performs poorly as it rarely generates unique asserts. Discussion regarding these results can be found in Section 7.6.2.

7.6 Discussion

We discuss the findings of the research questions and other subjects relating to our methodology and outcomes. The topics of assert accuracy (RQ1 – RQ4) and assert uniqueness (RQ5) are of particular interest as they constitute the primary ways in which we assess the usefulness of the generated asserts. In addition, there are important takeaway messages regarding the practicalities of applying this general approach to code generation tasks, both in research and in industrial practice.

7.6.1 Assert Accuracy

Assessing the accuracy of the generated asserts by comparing them to a ground truth test set is the primary method for evaluating assert generation techniques as it shows us how similar the generated asserts are to developer written asserts. Given the assumption that developers write useful asserts, this gives direct evidence for the usefulness of the generated asserts.

For RQ1 we use precision and recall as our measure for accuracy which shows that the accuracy achieved by ReAssert is heavily dependant on the project it is applied to. In the best case of our experiments, using the TestNMT model with Stanford CoreNLP, the accuracy is greater than what is achieved in the best case with Atlas, using the Reformer model. However, when using the ND4J project, the accuracy is lower. Despite this, when considering the RQ2 results, we see that even for the worst-performing project, ND4J, we still have 20% of tests receiving at least one exact match assert.

When using the accuracy to compare models within the Atlas approach, in RQ3 we see that the Reformer model with 5262 matched asserts is 58% more accurate than the Atlas model with 3323 matched asserts, the next best performing model, while the TestNMT model is far behind with only 1355 matched asserts. The poor performance of TestNMT is due to its lack of an UNK replacement mechanism which results in an UNK token appearing in 80% of the asserts it generates. As any assert which contains an UNK token cannot be matched, the accuracy of the model is very poor. This is one of the primary ways in which the Atlas model differs from the TestNMT model in that it implements a copy mechanism that replaces UNK token predictions with a token from the source sequence, the effects of which are seen in these results. RQ1 paints a different picture in terms of the comparisons between models when using the ReAssert approach. This shows that all the models are close to each other in general but where there are larger differences, the ordering from RQ1 is typically reversed, with TestNMT coming in first and Reformer coming in last. The reasons for this are two-fold. Firstly, it seems that the accuracy of the models is ultimately

Table 7.7: Top 5 most common matched asserts across all models.

Assert	Count
assertEquals(expected, actual)	1288
assertEquals(expResult, result)	419
assertEquals(expected, result)	337
assertTrue(true)	253
assertNotNull(result)	181

being bottlenecked by the quantity and diversity of training data. Given that the amount of training data that can be extracted from a single project is limited, there are a proportion of asserts appearing in the test set that bear no resemblance to any assert in the training set and, therefore, will never be able to be replicated by any model. Given this, it seems that TestNMT may be the best model at learning and recreating a restricted set of asserts that appear frequently, while Reformer is better at generalising when given a more diverse data set. This would explain the differences between these models when comparing the RQ1 and RQ3 results. The subject of the effect of data sets on the performance of the models is discussed further in Section 7.6.3.

The takeaway message from these RQs is that the best performing model is dependent on the usage scenario. If generating asserts for a project that has a data set that is conducive to sequence to sequence learning, ReAssert with a TestNMT model is the best performing with up to 44% precision and the ability to generate at least one matching assert for up to 40% of tests with the projects we used in the evaluation. Otherwise, when using the Atlas approach, the Reformer model may be the best choice.

7.6.2 Assert Uniqueness

In RQ5, we performed a uniqueness evaluation on the generated asserts to provide more evidence for how useful the asserts are in practice. This was done as uniqueness is an indicator of specificity and the more specific information an assert contains, the more useful that assert is likely to be in practice. Therefore, we use uniqueness as a partial proxy for evaluating usefulness. The intuition behind this

Table 7.8: Top 5 most common matched asserts containing a method call.

Assert	Count
assertEquals(0, result.size())	172
assertTrue(getNoErrorMsg(), result)	59
assertEquals(0, meldingen.size())	57
assertEquals(200, response.getStatusCode())	38
assertEquals(“test”, echo.echo(“test”))	26

evaluation is clear when inspecting the least unique (most commonly generated) asserts, as shown in Table 7.7. This demonstrates how the most common asserts are extremely generic and provide almost no specific information to the developers as they simply compare values of generically named variables. In the extreme case, as exemplified by the fourth most common assert, *assertTrue(true)*, the assert is of no use at all and has been learnt from developers writing a placeholder assert into their tests (which is considered bad practice). As a comparison, if we look at the top five most common generated asserts that contain a method call, as shown in Table 7.8 we see that, while still somewhat generic, these asserts contain more specific information for how to test the tested method. This comparison highlights how uniqueness relates to specificity, which in turn relates to practical usefulness. We, therefore, favour models which generate the greatest diversity of asserts.

Given that Reformer produces more unique asserts and has a lower percentage of its asserts belonging to the top 5 and top 10 most common asserts as compared to Atlas in the evaluation for RQ5, Reformer is the most desirable model in this regard. TestNMT performs poorly in this evaluation for the same reason as its poor performance in accuracy, namely that the lack of an UNK replacement mechanism limits the range of matched asserts that it can produce. The takeaway message is that the use of a state of the art model like Reformer can improve the usefulness of the generated asserts due to the higher uniqueness of the asserts.

7.6.3 Data Set Size, Diversity, and Quality

As discussed in Section 7.6.1, we see a surprising result when we compare the accuracy between models when using ReAssert versus when using Atlas, in that

the models perform much more similarly with ReAssert. This indicates that some of the projects selected for the ReAssert evaluation produce data sets that do not allow all the models to generalise maximally, most likely due to insufficient size, low diversity, or too much noise. This is an important takeaway for those wishing to use these code generation techniques in the future as these properties are determined not just by the size of the projects from which the data is taken but also by the traceability technique used to establish the test-to-tested-method links, the filtering that is applied to the data sets, and the way the code is written. This diversity of concerns is evident when investigating the relationship between data set sizes and performance. In this regard, it's important to note that the project with the largest data set (EJML) is only the second-best performing project in terms of F1 score in RQ1 with the Reformer model (and third-best with the RNN models), while the project that performs best for F1 score with all models (Stanford CoreNLP) has only the third largest data set. This shows that the size, diversity, and quality of the data set has a large impact and, for projects of this size, ultimately limits the ability of the models to generalise. The takeaway message, therefore, is that it is crucial to select a corpus of software that is large and diverse and that appropriate techniques which balance data set size and accuracy must be selected.

7.7 Threats to Validity

The threats to validity are related to the data used for training and evaluating the models, both in terms of subject selection and the method of data set collection. An external threat to validity is the representativeness of the subjects chosen for the ReAssert evaluation, as we have no strong evidence that the subjects are representative of the general population of software. However, the subjects cover a range of project types, are widely used in research and industry, and are large enough to demonstrate applicability to complex software. The second threat comes from the method by which the data was collected as the traceability techniques used to build the test-to-tested-method links do not have complete precision and there is, therefore, some noise in the data. However, as discussed in Section 7.2.1, we

believe that having some noise in the data does not necessarily significantly hamper the training. For the individual project data sets, we ensure that the validation and test sets contain minimal noise by using a very high precision traceability technique for constructing those sets. However, when using multi-project data sets, such as in the ATLAS evaluation, the results may vary if using a data set with a significantly different amount of noise in the data sets.

7.8 Related Work

Prior to the application of the machine learning techniques that are the subject of this chapter, assert generation was done primarily by test suite generation tools. These tools can be split into several categories depending on the general approach used for the generation of their tests. Randoop [Pacheco and Ernst, 2007], Nighthawk [Andrews et al., 2007], JCrasher [Csallner and Smaragdakis, 2004], and CarFast [Park et al., 2012] are the primary examples of tools that use approaches based on random generation while EvoSuite [Fraser and Arcuri, 2013] and eToc [Tonella, 2004] are examples of meta-heuristic search-based tools and Symbolic Pathfinder [Păsăreanu and Rungta, 2010] and jCUTE [Sen and Agha] are examples of tools that use dynamic symbolic execution. However, despite the diversity of approaches to unit test generation employed by these tools, they all focus primarily on things other than the generation of meaningful asserts. The usual goal for these tools is achieving coverage or exposing faults in other ways, such as generating exceptions and crashes. Therefore, even the most well developed and studied examples of these tools which do have some form of assert generation, such as EvoSuite and Randoop, the asserts they generate are often trivial or not meaningful, contributing to the relatively high rate of missed faults in real-world projects [Shamshiri et al., 2015] and poor developer opinions of the quality of generated asserts [Almasi et al., 2017].

7.9 Conclusion

We have presented REASSERT, a project-based deep learning approach for the generation of JUnit test asserts. We also utilise the state-of-the-art Reformer

model and two RNN-based models from previous work to evaluate ReAssert and provide an extended evaluation of Atlas, allowing us to compare models and approaches for assert generation. ReAssert improves over previous work by generating asserts that are, in general, more accurate and does not require that a test be written before being able to generate asserts, in addition to being able to generate multiple asserts for a single function. Also, the Reformer model is shown to improve the results achievable by the Atlas approach [Watson et al., 2020] generating asserts that are more accurate and more unique. However, when the Reformer model is used with ReAssert, the difference in effectiveness between the models is greatly lessened. This indicates that some of the projects selected for the ReAssert evaluation produce data sets that do not allow all the models to generalise maximally, most likely due to insufficient size, low diversity, or too much noise. Therefore, researchers and practitioners must be aware of this limitation and select code corpora and traceability techniques that provide suitably large, diverse, and clean data sets.

Chapter 8

Discussion, Conclusion, and Future Work

This chapter concludes the thesis starting with a discussion section containing general observations and discussion points emerging from the work. This is followed by the conclusion which summarises the primary outcomes and achievements of the thesis.

8.1 Discussion

This section provides a recap and exploration of the general themes and discussion points raised by the work presented in this thesis. One recurring theme is the issues surrounding the construction and quality of datasets, especially for use with machine learning techniques or as a ground truth for evaluation. All three main research areas explored in this thesis utilise data sets of test-to-code traceability links: TCTracer uses a set of ground truth links for evaluation, Relatest uses links for finding test recommendations, and TestNMT and ReAssert utilise them for training neural networks.

To evaluate TCTracer we needed a set of ground truth links that we could use to evaluate our predictions. For this we manually created a new data set of ground truth links and also attempted to find existing data sets of ground truth links to use, however, we found only two existing sets of links that were appropriate to use out of the many sets that we investigated. This experience resulted in some

important takeaway lessons as we learned that there are a set of mistakes that are commonly made which introduce bias or generally result in a poor quality data set. Firstly, failure to uniformly sample tests from the project can lead to the data set containing mostly very similar links, resulting in the data set not being representative of the project as a whole or tests in general. Secondly, selecting projects that are too small, out of date, or have low test coverage also produces data sets that are poor in variation and present challenges in building and executing the project to test your approach. Additionally, a valid link needs to be defined in such a way that avoids links that are not useful being included. This includes not considering interface methods or overridden methods as tested methods as doing so can result in many tests being linked to a single interface or overridden method when they are actually testing different methods which implement or override it. Finally, determining which functions are tested by which tests includes some amount of judgement and is error-prone. Therefore, to ensure that judgements are consistent and reasonable, links should be validated by a second judge. The conclusion from this is that building high precision data sets of test-to-code links is time-consuming and difficult.

These time and effort constraints make it non-viable to manually establish data sets of links that are large enough to train machine learning models, where we ideally need at least thousands of training examples. Therefore, to build data sets appropriate for this task, automated techniques must be used. Other work that has attempted this, such as Atlas [Watson et al., 2020], has used techniques that produce data sets with high amounts of incorrect links, resulting in noisy data sets which may impact the maximum achievable accuracy of the model and the reliability of its evaluation. This issue served as one of the motivations for TCTracer: to develop an automated approach to test-to-code traceability establishment which can produce large data sets for training machine learning models with less noise than existing approaches. TCTracer was, therefore, used by us to create the data set for ReAssert.

The difficulties surrounding the construction of large high-quality datasets

leads to the next discussion point which is the effectiveness of machine learning techniques vs more traditional approaches. While machine learning is a very powerful tool that has many applications and proved effective for TestNMT and ReAssert, we also found instances in which traditional approaches were more effective. The two main areas in which we applied machine learning and discovered it did not perform better than our more traditional approaches were bipartite edge prediction in Relatest and technique combination in TCTracer. For bipartite edge prediction in Relatest, discussed in Section 4.3.7, we experimented with three different machine learning techniques: matrix factors learning, spectral clustering, and cross-graph learning. This represents a good sample of machine learning techniques as there is one supervised method, one unsupervised method, and one semi-supervised method. Despite this, our graph-based method called the *triangle method*, outperformed all of these machine learning techniques. There are two primary reasons for this: firstly, to utilise these methods we have to convert the artefact relationship graphs to matrices and secondly, the quantity and quality of training data available. Due to the nature of the task, predicting edges in a very sparse bipartite graph, the resulting matrices are also very sparse and when this is coupled with a lack of training examples, the models struggle to learn a good function. Additionally, as we are dealing with projects that have thousands of tests and functions, the sizes of the matrices very quickly become extremely large making computation intractable. Therefore, some form of dimensionality reduction has to be used which adds further complication and can affect accuracy. In comparison, our traditional graph-based approach is effective and has low time and space complexity, making it the best choice. The other example where machine learning was trumped by a traditional method was the technique combination method in TCTracer. Here we see that simply taking an average of the individual technique scores performed better than using a feedforward network to combine them.

Overall, the takeaway message is that there is no universal approach to solving all of the problems around the automation of discovery, reuse, and generation of tests that can be applied to all aspects of the problem. Only by combining different

types of approaches from software engineering, graph theory, and machine learning, along with a hybrid of manually and automatically generated data sets were we able to achieve a body of work that can identify relationships between tests and functions and utilise those relationships for automating reuse and generation of new test code, resulting in a saving of developer effort and ease of testing.

8.2 Conclusion

This thesis has presented a body of work addressing the general problem of how to establish and maintain, in a time-efficient manner, a collection of high-quality tests for software systems. This top-level problem was tackled by decomposing it into several sub-problems: establishing test-to-code traceability links, reusing existing tests, and generating new test code.

To establish test-to-code traceability links, Chapter 3 introduced TCTracer, the first multilevel approach to test-to-code traceability which establishes links at both the method level and class level. TCTracer utilises an ensemble of individual static and dynamic techniques and a cross-level flow of information to achieve state-of-art performance. Our evaluation of TCTracer reveals that only static information is required to achieve the best performance at the class level, however, at the method level dynamic information is also needed for the best performance. Our evaluation also investigates the use of machine learning for technique combination and the application of weightings for individual techniques. Additionally, we conducted a manual investigation into the causes of false positives and false negatives which revealed that the majority of these cases are due to developers breaking conventions or writing code in unusual ways. To perform our evaluation of TCTracer we also created a ground truth data set which has been made publicly available and a set of takeaway messages for researchers that are creating or using similar data sets.

For assisting the automation of reuse, Chapter 4 introduced the concept of the artefact relation graph and used it to formulate Rashid, a general framework for modelling the relationships between artefacts. We then created Relatest, an approach for the recommendation of existing tests for new functions by instantiating

RASHID using functions and tests as the artefact types and code similarity measures and test-to-code traceability to define the relations. Our evaluation demonstrated that using RELATEST can substantially reduce the amount of effort required to write tests and in the majority of cases only a single recommendation is needed to get the maximum benefit. However, given that some manual effort is still required to transform the recommendations made by RELATEST, we also investigated the automatic transplantation of tests between environments. This investigation showed that fully automated transplantation is not always feasible but can be achieved, especially if certain conditions are met. In addition, applying genetic improvement to the transplantation process greatly increases the likelihood of success. Additionally, while transplanted tests overall reveal fewer faults than EvoSuite, they also have the potential to reveal faults that are not revealed by EvoSuite.

As we are not able to find and transplant an existing test for all functions we also focus on the task of unit test generation as we can then generate a test for any function. This thesis contains two approaches to the generation of test code: TESTNMT and REASSERT. Chapter 6 presents TESTNMT, an approach that uses recurrent neural networks repurposed from the field of neural machine translation to generate full unit tests. This approach was demonstrated to be capable of generating approximate tests which are close to ground truth tests, however, as the generated tests are approximations, some manual work is still required by the developers to transform the generated approximate tests for use. Additionally, as generating entire tests often requires a large number of output tokens the accuracy can be low for complicated or long tests. To counter these issues, Chapter 7 presents REASSERT, an approach to test code generation which focuses on the generation of asserts as opposed to entire tests. This is motivated by the fact that asserts are the most important components of the tests and contain fewer tokens than the whole tests which are generated by TESTNMT. REASSERT defines a new approach for data collection, pre-processing, and post-processing and utilises the Reformer, a new transform-based model. This allows us to generate asserts with higher accuracy than TESTNMT generates approximate whole tests and removes the need for developers

to make manual syntax changes before integrating the generated code into existing project code. The evaluation of ReAssert shows that we generate asserts with greater accuracy and greater diversity than previous approaches. Additionally, unlike previous approaches, ReAssert does not require that the body of the test already be written to generate asserts for it and can generate multiple asserts for a single test.

In total, this thesis presents a set of novel approaches and their implementations in tools for tackling the key problems associated with assisting the automation of test suite development, specifically, establishing relationships between tests and tested code, automating reuse recommendations and test transplantation, and the generation of new test code. The evaluations of these approaches show state-of-the-art effectiveness in their respective tasks and demonstrate the potential to make large savings of manual time and effort when used in the software development process.

Bibliography

L. A. Adamic and E. Adar. Friends and neighbors on the web. *Social networks*, 25 (3):211–230, 2003.

Agitar Technologies. Automated JUnit Generation, 2020. URL `http://www.agitar.com/solutions/products/agitarone.html`.

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

M. Allamanis, H. Peng, and C. Sutton. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*, pages 2091–2100, 2016.

M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81, 2018.

M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 263–272. IEEE, may 2017. ISBN 978-1-5386-2717-4. doi: 10.1109/ICSE-SEIP.2017.27. URL `http://ieeexplore.ieee.org/document/7965450/`.

U. Alon, S. Brody, O. Levy, and E. Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3 (POPL):1–29, 2019.

J. H. Andrews, F. C. H. Li, and T. Menzies. Nighthawk: A two-level genetic-random unit test data generator. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE '07*, page 144, New York, New York, USA, 2007. ACM Press. ISBN 9781595938824. doi: 10.1145/1321631.1321654. URL http://portal.acm.org/citation.cfm?doid=1321631.1321654.

G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

H. U. Asuncion, A. U. Asuncion, and R. N. Taylor. Software traceability with topic modeling. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, 1:95, 2010. ISSN 0270-5257. doi: 10.1145/1806799.1806817.

S. Axelsson. Using normalized compression distance for classifying file fragments. In *2010 International Conference on Availability, Reliability and Security*, pages 641–646. IEEE, 2010.

D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *Annual Review of Neuroscience*, 26(1):105–131, sep 2014. ISSN 0147-006X. doi: 10.1146/annurev.neuro.26.041002.131047. URL http://arxiv.org/abs/1409.0473.

S. Bajracharya, J. Ossher, and C. Lopes. Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Science of Computer Programming*, 79:241–259, 2014. ISSN 01676423. doi: 10.1016/j.scico.2012.04.008. URL http://dx.doi.org/10.1016/j.scico.2012.04.008.

E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, pages 306–317, 2014. doi: 10.1145/2635868.2635898. URL http://dl.acm.org/citation.cfm?doid=2635868.2635898.

E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke. Automated software transplantation. pages 257–269, 2015. doi: 10.1145/2771783.2771796. URL http://dl.acm.org/citation.cfm?doid=2771783.2771796.

N. Benchettara, R. Kanawati, and C. Rouveirol. Supervised machine learning applied to link prediction in bipartite social networks. In *International Conference on Advances in Social Network Analysis and Mining, ASONAM 2010*, pages 326–330, 2010. ISBN 9780769541389.

A. Blot and J. Petke. Empirical comparison of search heuristics for genetic improvement of software. *IEEE Transactions on Evolutionary Computation*, pages 1–1, 2021. doi: 10.1109/TEVC.2021.3070271.

P. Bouillon, J. Krinke, N. Meyer, and F. Steimann. EZUNIT: A framework for associating failed unit tests with potential programming errors. In *Agile Processes in Software Engineering and Extreme Programming*. Springer Berlin / Heidelberg, 2007. doi: 10.1007/978-3-540-73101-6_14. URL http://www.springerlink.com/content/d474n74vw7084777/.

S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998.

K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

J. Cleland-Huang. Traceability in agile projects. In *Software and Systems Traceability*, pages 265–275. Springer, 2012.

J. Cleland-Huang, B. Berenbach, S. Clark, R. Settimi, and E. Romanova. Best practices for automated traceability. *Computer*, 40(6):27–35, June 2007. ISSN 0018-9162. doi: 10.1109/MC.2007.195.

J. Cleland-Huang, A. Czauderna, M. Gibiec, and J. Emenecker. A machine learning approach for tracing regulatory codes to product specific requirements. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, 1:155, 2010. ISSN 0270-5257. doi: 10.1145/1806799.1806825. URL http://portal.acm.org/citation.cfm?doid=1806799.1806825.

J. R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

F. Crestani. Application of spreading activation techniques in information retrieval. *Artificial Intelligence Review*, 11(6):453–482, 1997.

C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for java. *Software: Practice and Experience*, 34(11):1025–1050, sep 2004. ISSN 0038-0644. doi: 10.1002/spe.602. URL http://doi.wiley.com/10.1002/spe.602.

V. Csuvik, A. Kicsi, and L. Vidács. Source code level word embeddings in aiding semantic test-to-code traceability. In *10th International Workshop on Software and Systems Traceability*, pages 29—-36, 2019a. doi: 10.1109/SST.2019.00016. URL https://dl.acm.org/citation.cfm?id=3355319.

V. Csuvik, A. Kicsi, and L. Vidács. Evaluation of textual similarity techniques in code level traceability. In *International Conference on Computational Science and Its Applications*, pages 529–543. Springer, 2019b.

D. Davis, R. Lichtenwalter, and N. V. Chawla. Multi-relational link prediction in heterogeneous information networks. In *Advances in Social Networks Analysis and Mining (ASONAM), 2011 International Conference on*, pages 281–288. IEEE, 2011.

J. Davis and M. Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd International Conference on Machine Learning*, pages 233–240. ACM, 2006.

A. De Lucia, F. Fasano, and R. Oliveto. Traceability management for impact analysis. In *2008 Frontiers of Software Maintenance*, pages 21–30. IEEE, 2008.

P. T. Devanbu. GENOA: A customizable language- and front-end independent code analyzer. In *Proceedings of the 14th International Conference on Software Engineering*, ICSE '92, pages 307–317, New York, NY, USA, 1992. ACM. ISBN 0-89791-504-6. doi: 10.1145/143062.143148. URL http://doi.acm.org/10.1145/143062.143148.

D. Elsner, F. Hauer, A. Pretschner, and S. Reimer. 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*, pages 491–504, 2021.

M. Erfani, I. Keivanloo, and J. Rilling. Opportunities for clone detection in test case recommendation. In *37th Annual Computer Software and Applications Conference Workshops*, pages 65–70, July 2013. ISBN 978-1-4799-2159-1. doi: 10.1109/COMPSACW.2013.11.

J. L. Fleiss. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378, 1971.

G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, feb 2013. ISSN 0098-5589. doi: 10.1109/TSE.2012.14. URL http://ieeexplore.ieee.org/document/6152257/.

G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, mar 2012. ISSN 0098-5589. doi: 10.1109/TSE.2011.93. URL http://ieeexplore.ieee.org/document/6019060/.

R. E. Gallardo-Valencia and S. E. Sim. *Source Code Seeking on the Web: A Survey of Empirical Studies and Tools*. Lulu.com, 2014.

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

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

M. Ghafari, C. Ghezzi, and K. Rubinov. Automatically identifying focal methods under test in unit test cases. In *15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 61–70. IEEE, sep 2015. ISBN 978-1-4673-7529-0. doi: 10.1109/SCAM.2015.7335402. URL `http://ieeexplore.ieee.org/document/7335402/`.

N. E. Gold and J. Krinke. Ethical mining: A case study on MSR mining challenges. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 265–276, New York, NY, USA, jun 2020. ACM. URL `https://dl.acm.org/doi/10.1145/3379597.3387462`.

A. N. Gomez, M. Ren, R. Urtasun, and R. B. Grosse. The reversible residual network: Backpropagation without storing activations. *Advances in Neural Information Processing Systems*, 2017-Decem(Nips):2215–2225, jul 2017. ISSN 10495258. URL `http://arxiv.org/abs/1707.04585`.

Google. google/trax, 2020. URL `https://github.com/google/trax`.

J. Gu, Z. Lu, H. Li, and V. O. Li. Incorporating copying mechanism in sequence-to-sequence learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 3, pages 1631–1640, Stroudsburg, PA, USA, 2016. Association for Computational Linguistics.

ISBN 9781510827585. doi: 10.18653/v1/P16-1154. URL http://aclweb.org/anthology/P16-1154.

J. Guo, J. Cheng, and J. Cleland-Huang. Semantically enhanced software traceability using deep learning techniques. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 3–14. IEEE, may 2017. ISBN 978-1-5386-3868-2. doi: 10.1109/ICSE.2017.9. URL http://ieeexplore.ieee.org/document/7985645/.

M. Harman, W. B. Langdon, and W. Weimer. Genetic programming for reverse engineering. *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 1–10, 2013. ISSN 10951350. doi: 10.1109/WCRE.2013.6671274. URL http://ieeexplore.ieee.org/document/6671274/.

V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, volume 18, pages 152–162, New York, New York, USA, 2018. ACM Press. ISBN 9781450355735. doi: 10.1145/3236024.3236051. URL https://doi.org/10.1145/3236024.3236051http://dl.acm.org/citation.cfm?doid=3236024.3236051.

J. Henkel, S. K. Lahiri, B. Liblit, and T. Reps. Code vectors: understanding programs through embedded abstracted symbolic traces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, pages 163–174, New York, New York, USA, 2018. ACM Press. ISBN 9781450355735. doi: 10.1145/3236024.3236085. URL http://dl.acm.org/citation.cfm?doid=3236024.3236085.

X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010. IEEE, 2018.

V. Hurdugaci and A. Zaidman. Aiding software developers to maintain developer tests. In *16th European Conference on Software Maintenance and Reengineering*, pages 11–20. IEEE, 2012.

S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, 2016.

P. Jaccard. Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bull Soc Vaudoise Sci Nat*, 37:547–579, 1901.

L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering*, pages 96–105, 2007.

J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 467–477. IEEE, 2002.

R. Just, D. Jalali, and M. D. Ernst. Defects4J: a database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*, pages 437–440, New York, New York, USA, 2014. ACM Press. ISBN 9781450326452. doi: 10.1145/2610384.2628055. URL `http://dl.acm.org/citation.cfm?doid=2610384.2628055`.

N. Kalchbrenner and P. Blunsom. Recurrent continuous translation models. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1700–1709, 2013.

T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

A. Kicsi, L. Tóth, and L. Vidács. Exploring the benefits of utilizing conceptual information in test-to-code traceability. In *6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, pages 8–14, 2018. ISBN 9781450357234. doi: 10.1145/3194104.3194106.

A. Kicsi, L. Vidács, and T. Gyimóthy. TestRoutes. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 593–597, New York, NY, USA, jun 2020. ACM. ISBN 9781450375177. doi: 10.1145/3379597.3387488. URL https://dl.acm.org/doi/10.1145/3379597.3387488.

N. Kitaev, Ł. Kaiser, and A. Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

C. Klammer and A. Kern. Writing unit tests: It's now or never! In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–4. IEEE, apr 2015. ISBN 978-1-4799-1885-0. doi: 10.1109/ICSTW.2015.7107469. URL http://ieeexplore.ieee.org/document/7107469/.

Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, aug 2009. ISSN 0018-9162. doi: 10.1109/MC.2009.263. URL http://ieeexplore.ieee.org/document/5197422/.

M. Landhäußer and W. F. Tichy. Automated test-case generation by cloning. In *7th International Workshop on Automation of Software Test (AST)*, pages 83–88, June 2012. ISBN 978-1-4673-1822-8. doi: 10.1109/IWAST.2012.6228995.

O. A. L. Lemos, S. Bajracharya, J. Ossher, P. C. Masiero, and C. Lopes. A test-driven approach to code search and its application to the reuse of auxiliary functionality. *Information and Software Technology*, 53(4):294–306, 2011.

V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8):707–710, 1966.

M. Li, X. Chen, X. Li, B. Ma, and P. Vitanyi. The similarity metric. *IEEE Transactions on Information Theory*, 50(12):3250–3264, dec 2004. ISSN 0018-9448. doi: 10.1109/TIT.2004.838101. URL http://ieeexplore.ieee.org/document/1362909/.

D. Liben-Nowell and J. Kleinberg. The link-prediction problem for social networks. *Journal of the American Society for Information Science and Technology*, 58(7): 1019–1031, May 2007. ISSN 1532-2890.

R. N. Lichtenwalter, J. T. Lussier, and N. V. Chawla. New perspectives and methods in link prediction. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 243–252. ACM, 2010.

C. Y. Lin. Rouge: A package for automatic evaluation of summaries. *Proceedings of the workshop on text summarization branches out (WAS 2004)*, (1):25–26, 2004. ISSN 00036951.

W. Ling, E. Grefenstette, K. M. Hermann, T. Kočiskỳ, A. Senior, F. Wang, and P. Blunsom. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744*, 2016.

H. Liu and Y. Yang. Bipartite edge prediction via transductive learning over product graphs. *Icml*, 37:1880–1888, 2015. ISSN 1938-7228. URL http://jmlr.org/proceedings/papers/v37/liuc15.pdf.

H. Liu and Y. Yang. Cross-graph learning of multi-relational associations. *arXiv preprint arXiv:1605.01832*, 2016.

M. Luong, E. Brevdo, and R. Zhao. Neural machine translation (seq2seq) tutorial. *https://github.com/tensorflow/nmt*, 2017.

M.-T. Luong and C. D. Manning. Stanford neural machine translation systems for spoken language domain. In *International Workshop on Spoken Language Translation*, Da Nang, Vietnam, 2015.

T. Luong, H. Pham, and C. D. Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, Stroudsburg, PA, USA, 2015a. Association for Computational Linguistics. ISBN 9781941643327. doi: 10.18653/v1/D15-1166. URL http://arxiv.org/abs/1508.04025http://aclweb.org/anthology/D15-1166.

T. Luong, H. Pham, and C. D. Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, Stroudsburg, PA, USA, 2015b. Association for Computational Linguistics. ISBN 9781941643327. doi: 10.18653/v1/D15-1166. URL http://arxiv.org/abs/1508.04025http://aclweb.org/anthology/D15-1166.

M. Madeja and J. Porubän. 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, 2019. doi: 10.4230/OASIcs.SLATE.2019.3.

S. Makady and R. J. Walker. Validating pragmatic reuse tasks by leveraging existing test suites. *Software: Practice and Experience*, 43(9):1039–1070, 2013.

C. Manning, P. Raghavan, and H. Schütze. Introduction to information retrieval. *Natural Language Engineering*, 16(1):100–103, 2010.

A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. *Proceedings. 25th International Conference on Software Engineering*, pages 125–135, 2003.

C. McMillan, D. Poshyvanyk, and M. Revelle. Combining textual and structural analysis of software artifacts for traceability link recovery. In *2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, pages 41–48. IEEE, may 2009. ISBN 978-1-4244-3741-2. doi: 10.1109/TEFSE.2009.5069582. URL http://ieeexplore.ieee.org/document/5069582/.

C. Mcmillan, D. Poshyvanyk, M. Grechanik, Q. Xie, and C. Fu. Portfolio: Searching for relevant functions and their usages in millions of lines of code. *ACM Trans. Softw. Eng. Methodol.*, 22(4):37:1—-37:30, 2013. ISSN 1049-331X. doi: 10.1145/2522920.2522930. URL http://doi.acm.org/10.1145/2522920.2522930.

R. Meimandi Parizi, A. Kasem, and A. Abdullah. Towards gamification in software traceability: Between test and code artifacts. In *Proceedings of the 10th International Conference on Software Engineering and Applications*, pages 393–400. SCITEPRESS - Science and Technology Publications, 2015. ISBN 978-989-758-114-4. doi: 10.5220/0005555503930400. URL http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0005555503930400.

T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review*, 63(2):81, 1956.

C. Mills and S. Haiduc. A machine learning approach for determining the validity of traceability links. *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 2017*, pages 121–123, 2017. doi: 10.1109/ICSE-C.2017.86.

M. E. Newman. Clustering and preferential attachment in growing networks. *Physical review E*, 64(2):025102, 2001a.

M. E. Newman. The structure of scientific collaboration networks. *Proceedings of the national academy of sciences*, 98(2):404–409, 2001b.

A. Nigam and N. V. Chawla. Link prediction in a semi-bipartite network for recommendation. *Intelligent Information and Database Systems*, 9622:127–135, 2016. doi: 10.1007/978-3-662-49390-8.

H. Niu, I. Keivanloo, and Y. Zou. Learning to rank code examples for code search engines. *Empirical Software Engineering*, 22(1):259–291, 2017. ISSN 15737616. doi: 10.1007/s10664-015-9421-5. URL http://dx.doi.org/10.1007/s10664-015-9421-5.

G. Orellana, G. Laghari, A. Murgia, and S. Demeyer. On the differences between unit and integration testing in the TravisTorrent dataset. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 451–454. IEEE, may 2017. ISBN 978-1-5386-1544-7. doi: 10.1109/MSR.2017.25. URL http://ieeexplore.ieee.org/document/7962394/.

C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion - OOPSLA '07*, volume 2, page 815, New York, New York, USA, 2007. ACM Press. ISBN 9781595938657. doi: 10.1145/1297846.1297902. URL http://portal.acm.org/citation.cfm?doid=1297846.1297902.

K. Papineni, S. Roukos, T. Ward, and W. Zhu. BLEU: a method for automatic evaluation of machine translation. *ACL '02 Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, (July):311–318, 2002. ISSN 00134686. doi: 10.3115/1073083.1073135. URL http://dl.acm.org/citation.cfm?id=1073135.

R. M. Parizi. 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)*, pages 193–200. IEEE, jun 2016. ISBN 978-1-5090-0809-4. doi: 10.1109/SERA.2016.7516146. URL http://ieeexplore.ieee.org/document/7516146/.

R. M. Parizi, S. P. Lee, and M. Dabbagh. Achievements and challenges in state-of-the-art software traceability between test and code artifacts. *IEEE Transactions*

on Reliability, 63(4):913–926, dec 2014. ISSN 0018-9529. doi: 10.1109/TR.2014.2338254. URL http://ieeexplore.ieee.org/document/6862933/.

S. Park, B. M. M. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, and Q. Xie. CarFast: Achieving higher statement coverage faster sangmin. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*, page 1, New York, New York, USA, 2012. ACM Press. ISBN 9781450316149. doi: 10.1145/2393596.2393636. URL http://dl.acm.org/citation.cfm?doid=2393596.2393636.

S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, 1994.

R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience*, 46(9):1155–1179, 2016.

J. Petke, M. Harman, W. B. Langdon, and W. Weimer. Specialising software for different downstream applications using genetic improvement and code transplantation. *IEEE Transactions on Software Engineering*, 2017.

J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward. Genetic improvement of software: A comprehensive survey. *IEEE Transactions on Evolutionary Computation*, 22(3):415–432, jun 2018. ISSN 1089-778X. doi: 10.1109/TEVC.2017.2693219. URL https://ieeexplore.ieee.org/document/7911210/.

L. S. Pinto, S. Sinha, and A. Orso. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*, volume 1, page 1, New York, New York, USA, 2012. ACM Press. ISBN 9781450316149. doi: 10.1145/2393596.2393634. URL http://dl.acm.org/citation.cfm?doid=2393596.2393634.

L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza. Mining stackoverflow to turn the ide into a self-confident programming prompter. In *11th Working Conference on Mining Software Repositories*, pages 102–111, 2014.

C. S. Păsăreanu and N. Rungta. Symbolic PathFinder: Symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM international conference on Automated software engineering - ASE '10*, volume 2, page 179, New York, New York, USA, 2010. ACM Press. ISBN 9781450301169. doi: 10.1145/1858996.1859035. URL http://portal.acm.org/citation.cfm?doid=1858996.1859035.

A. Qusef, G. Bavota, R. Oliveto, A. D. Lucia, and D. Binkley. Evaluating test-to-code traceability recovery methods through controlled experiments. *Journal of Software: Evolution and Process*, 25(11):1167–1191, Nov 2013. ISSN 20477473. doi: 10.1002/smr.1573. URL http://doi.wiley.com/10.1002/smr.1573.

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

C. Ragkhitwetsagul, J. Krinke, and D. Clark. Similarity of source code in the presence of pervasive modifications. In *International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 117–126, 2016.

C. Ragkhitwetsagul, J. Krinke, and D. Clark. A comparison of code similarity analysers. *Empirical Software Engineering*, pages 1–56, 2017. ISSN 15737616. doi: 10.1007/s10664-017-9564-7.

B. V. Rompaey and S. Demeyer. Establishing traceability links between unit test cases and units under test. In *2009 13th European Conference on Software Maintenance and Reengineering*, number ii, pages 209–218. IEEE, 2009. ISBN 978-1-4244-3755-9. doi: 10.1109/CSMR.2009.39. URL http://ieeexplore.ieee.org/document/4812754/.

C. K. Roy and J. R. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *16th IEEE International Conference on Program Comprehension*, pages 172–181, 2008.

G. Salton and M. J. McGill. Introduction to modern information retrieval. 1986.

K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 419–423. ISBN 354037406X. doi: 10.1007/11817963_38.

M. Shahin, M. A. Babar, and L. Zhu. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017.

S. Shamshiri. Automated unit test generation for evolving software. *2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings*, pages 1038–1041, 2015. doi: 10.1145/2786805.2803196.

S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 201–211. IEEE, nov 2015. ISBN 978-1-5090-0025-8. doi: 10.1109/ASE.2015.86. URL `http://ieeexplore.ieee.org/document/7372009/`.

E. Shihab, Y. Kamei, B. Adams, and A. E. Hassan. Is lines of code a good measure of effort in effort-aware models? *Information and Software Technology*, 55(11): 1981–1993, Nov. 2013. ISSN 09505849. doi: 10.1016/j.infsof.2013.06.002.

Q. D. Soetens, S. Demeyer, A. Zaidman, and J. Pérez. Change-based test selection: an empirical evaluation. *Empirical Software Engineering*, 21(5):1990–2032, oct 2016. ISSN 1382-3256. doi: 10.1007/s10664-015-9405-5.

SonarCloud. Sonarcloud explore, 2022. URL https://sonarcloud.io/explore/projects.

D. Ståhl, K. Hallén, and J. Bosch. Achieving traceability in large scale continuous integration and delivery deployment, usage and validation of the Eiffel framework. *Empirical Software Engineering*, 22(3):967–995, 2017.

I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. *Mathematical Programming*, 155(1-2):105–145, sep 2014. ISSN 0025-5610. doi: 10.1007/s10107-014-0839-0. URL http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neuralhttp://link.springer.com/10.1007/s10107-014-0839-0http://arxiv.org/abs/1409.3215.

P. Tonella. Evolutionary testing of classes. *ACM SIGSOFT Software Engineering Notes*, 29(4):119, jul 2004. ISSN 01635948. doi: 10.1145/1013886.1007528. URL http://portal.acm.org/citation.cfm?doid=1013886.1007528.

F. Trautsch, S. Herbold, and J. Grabowski. Are unit and integration test definitions still valid for modern java projects? an empirical study on open-source projects. *Journal of Systems and Software*, 159:110421, 2020.

L. R. Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3):279–311, Sep 1966. ISSN 1860-0980. doi: 10.1007/BF02289464. URL https://doi.org/10.1007/BF02289464.

UCL Research Ethics Committee. *Research Ethics at UCL*, 2020. URL https://ethics.grad.ucl.ac.uk/.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *Behavioral and Brain Sciences*, 40(Nips):e253, jun 2017. ISSN 0140-525X. doi: 10.1017/S0140525X16001837. URL http://papers.nips.cc/paper/7181-attention-is-all-you-need.

J. Villmow, J. Depoix, and A. Ulges. CONTEST: A unit test completion benchmark featuring context. In *The First Workshop on Natural Language Processing for Programming*, pages 17–25, 2021.

C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk. On learning meaningful assert statements for unit test cases. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea*, 2020. ISBN 9781450371216. doi: 10.1145/3377811.3380429.

M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, pages 87–98, New York, New York, USA, 2016. ACM Press. ISBN 9781450338455. doi: 10.1145/2970276.2970326. URL http://dl.acm.org/citation.cfm?doid=2970276.2970326.

R. White and J. Krinke. TestNMT: Function-to-test neural machine translation. In *Proceedings of the 4th ACM SIGSOFT International Workshop on NLP for Software Engineering*, NL4SE 2018, page 30–33, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450360555. doi: 10.1145/3283812.3283823. URL https://doi.org/10.1145/3283812.3283823.

R. White and J. Krinke. TCTracer evaluation data – International Conference on Software Engineering 2020 [data set], 2020a.

R. White and J. Krinke. ReAssert: Deep learning for assert generation. *arXiv preprint arXiv:2011.09784*, 2020b.

R. White, J. Krinke, and R. Tan. Establishing multilevel test-to-code traceability links. In *42nd International Conference on Software Engineering (ICSE '20)*, Seoul, Republic of Korea, 2020. ACM. ISBN 9781450371216. doi: 10.1145/3377811.3380921. URL https://doi.org/10.1145/3377811.3380921.

S. Winkler and J. von Pilgrim. A survey of traceability in requirements engineering

and model-driven development. *Software & Systems Modeling*, 9(4):529–565, 2010.

A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3): 325–364, jun 2011. ISSN 1382-3256. doi: 10.1007/s10664-010-9143-7. URL http://link.springer.com/10.1007/s10664-010-9143-7.

T. Zhang and M. Kim. Automated transplantation and differential testing for clones. *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017*, pages 665–676, 2017. doi: 10.1109/ICSE.2017.67.

G. Zhao and J. Huang. DeepSim: deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, pages 141–151, New York, New York, USA, 2018. ACM Press. ISBN 9781450355735. doi: 10.1145/3236024.3236068. URL http://dl.acm.org/citation.cfm?doid=3236024.3236068.

M. Zilberstein and E. Yahav. Leveraging a corpus of natural language descriptions for program similarity. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward!*, pages 197–211, 2016. ISBN 9781450340762. doi: 10.1145/2986012.2986013.