# Detecting Duplicate Examples in Behaviour Driven Development Specifications

Leonard Peter Binamungu, Suzanne M. Embury, and Nikolaos Konstantinou
School of Computer Science, University of Manchester, Manchester, UK
{leonardpeter.binamungu,suzanne.m.embury,nikolaos.konstantinou}@manchester.ac.uk

***Abstract*—In Behaviour-Driven Development (BDD), the behaviour of the software to be built is specified as a set of example interactions with the system, expressed using a "Given-When-Then" structure. The examples are written using customer language, and are readable by end-users. They are also executable, and act as tests that determine whether the implementation matches the desired behaviour or not. This approach can be effective in building a common understanding of the requirements, but it can also face problems. When the suites of examples grow large, they can be difficult and expensive to change. Duplication can creep in, and can be challenging to detect manually. Current tools for detecting duplication in code are also not effective for BDD examples. Moreover, human concerns of readability and clarity can rise. We present an approach for detecting duplication in BDD suites that is based around dynamic tracing, and describe an evaluation based on three open source systems.**

***Index Terms*—behaviour-driven development, duplication detection, dynamic tracing**

## I. Introduction

In Behaviour-Driven Development (BDD) [1], the required software functionality is specified as a collection of example interactions with the system, expressed using natural language sentences organised using a "Given-When-Then" structure. This gives a specification that can be read and understood by customers. Due to the "glue code" that links the natural language sentences to the code under test, the examples are also executable. Thus, the examples both specify the requirements for the software and act as acceptance tests for the verification of the implementation against the specification.

Although this approach has many advantages, it also raises problems. When BDD specifications grow large, they can become costly to maintain and extend [2]. In the worst case, functionality can be effectively frozen because of the costs and risks of changing lengthy BDD suites. The intellectual effort required to understand hundreds of examples, and how they relate to one another and to the production code under construction, is substantial. There is therefore a high cost for teams when redundancy creeps into their BDD suites. Execution times will lengthen, increasing the delay between the creation of software defects and their detection by the BDD engine. More seriously, maintaining the quality and conceptual integrity of the specification becomes harder, increasing the risk that further redundancies, omissions and inelegances will be introduced. Despite these challenges, studies investigating the problem are few in number and limited in scope.

We present an approach to detecting duplication in BDD suites based on an analysis of dynamic traces. Our goal is to determine when two examples exercise the production code in the same way. This is challenging because we need to identify which differences between execution traces are significant, and which are not. We propose an answer to that question, based around which we have created a duplication detection tool for use with BDD specifications written in Gherkin[1], with glue code written using Cucumber-JVM conventions[2]. The evaluation of our approach on 3 systems detected more than 70.0% of the injected duplicates, for each of the 3 systems.

## II. The Duplicate Detection Problem in BDD

In BDD, the behaviour of a software system is described by a collection of *scenarios*, grouped into *features*. Each scenario describes an end-to-end interaction with the system, in terms of concrete data examples. Scenarios are written in a form of structured English following a "Given-When-Then" pattern. For example, the following scenario is part of a suite describing how student final results are calculated:

```
Given a student scoring 30/70 in the exam
  And the student scores 10/30 in the lab
When the final marks are calculated
Then the student's mark will be 40/100
 And the student has passed the unit
```

These scenarios are made executable by providing "glue code"—methods annotated with regular expressions matching the text of the scenario steps. The Java glue code for the first step in our scenario could be:

```
@Given("a student scoring (\\d+)/(\\d+)
        in the exam")
public void exam(int mark, int total) {
   this.student.setExamMark(mark);
}
```

These methods "glue" the high level scenarios to the production code. When a scenario is executed, the BDD engine takes each step and searches for a method with a matching regular expression. The method is then executed with parameter values extracted from the step text.

[1]cucumber.io/docs/reference
[2]github.com/cucumber/cucumber-jvm

BDD suites consists of tens, hundreds and (in some cases) thousands of these scenarios [2]. Adding or changing scenarios can be challenging. We want to reuse step text patterns as much as we can, to keep the intellectual cost of understanding the scenarios down. But we also want scenarios to be natural to read, using the customer's preferred domain terms. Tool support for managing the step collection used in BDD suites is currently poor, and under time pressure it is easy for teams to miss opportunities to reuse step text patterns. It can also be difficult to know whether an example is already present in the suite, leading to the same or overlapping examples being specified using different step text patterns. Over time, this leads to bloated and hard-to-change BDD suites.

Detecting such duplication automatically is challenging. Well written BDD suites will exhibit high degrees of textual similarity, since step text patterns should be widely reused across scenarios. Each feature will be described by multiple similar scenarios, giving different parameter values to indicate different happy and sad path cases. Whether these scenario "clones" are redundant or not depends on whether they describe test cases that are part of the same equivalence class for the code under test, or whether they come from different equivalence classes.

## III. Related Work

Duplicate detection for software artefacts is a well-trodden area of research. Here, we briefly survey the state of the art, in order to understand whether duplicate detection in BDD is supported by existing approaches. Current approaches can be grouped into those using static analysis of the artefacts under study [3], [4], those using dynamic analysis [5]–[7] and those using a combination of the two [8].

Duplication detection techniques make use of a variety of different representations of the artefacts under study, including token sequences [9], syntax trees [10], [11], program analysis graphs [12], metrics [13], [14] and strings of characters [15], [16]. Novakovic [17] and Bellon *et al.* [18] summarise existing duplication detection techniques, including comparison as text artefacts, comparison of sequences of lexical tokens, comparison of vectors of code metrics and comparison of semantic representations of the artefacts (such as program dependence graphs [12]).

We focus on detection of duplicates in expressions expressed using a domain specific language (DSL) that represent tests. There have been prior attempts to detect duplication in DSLs. Tairas *et al.* [19], [20] performed duplication detection on artefacts developed using the Object Constraint Language[3], by comparing abstract syntax trees (ASTs). This allows syntactic duplicates to be detected, but not semantic duplicates, when the meaning of the two artefacts is the same even though the specific syntactic elements they use to express the functionality are quite different.

Test Suite Reduction (TSR) techniques [21] also relate to our problem. But, to the best of our knowledge, existing TSR techniques focus on unit tests expressed in conventional programming languages, not acceptance tests expressed in a DSL, as in BDD. Also, contrary to what most TSR techniques aim to achieve, we do not want the reduced suite only; we also want duplicate BDD scenarios to be flagged explicitly.

As regards duplication in BDD suites, we are aware of only the work of Suan, who used textual and AST similarity measures to detect syntactic duplicates in Gherkin features [22]. This approach however can only locate duplicates that are textually similar. To the best of our knowledge, ours is the first attempt to detect semantically equivalent BDD scenarios.

## IV. Existing Duplication Detection Tools

To understand how far existing clone detection techniques could detect duplicates in BDD suites, we experimented with three mature duplicate detection tools: PMD/CPD[4], CloneDR[5], and DECKARD[6]. These were selected based on support given by developers on Stack Exchange, and because they are available for download and go beyond text-based similarity, as needed for our context.

Working with a benchmark that we had prepared for evaluation of our own work (described in section VI-A), we manually identified 13 pairs of duplicate BDD scenarios from 3 systems. Refer to Table II for more information about the 3 systems. Since the tools we selected operate on code artefacts and not on BDD scenarios, we manually unfolded the glue code that would be executed when each scenario was run, to extract a code level description of each scenario. We created one method per scenario (called the *scenario-representing method*), and put pairs of methods for duplicate scenarios into classes.

We then ran these classes through the selected tools. DECKARD and CloneDR were run under default configurations as the alternatives we tried did not show important differences in terms of the produced results, and we did not find any study suggesting optimal values. We used 10 as the minimum token size for PMD/CPD, after some trial and error, since that was the smallest value that returned any matches of whole scenarios, as opposed to returning only matches of partial unfolded glue code. Other token sizes either didn't give any result or returned too many false positives while missing the duplicates that were relevant for our context.

Table I shows the results. We considered a tool to have detected the duplicate scenarios if it reported their scenario-representing methods as duplicates of each other. Where only part of the scenario-representing methods were reported as duplicates, we considered that the duplication in scenarios was *not* detected.

While PMD/CPD and CloneDR were not able to identify almost any of the duplicates, DECKARD correctly identified all but 3 of the duplicate scenarios. However, it also identified a great many false positive duplicates. In our context, each false positive means that a customer or other member of the development team will have to examine the pair of scenarios

[3]`http://lcm.csa.iisc.ernet.in/soft_arch/OCL.htm`

[4]`pmd.github.io`

[5]`http://www.semdesigns.com/Products/Clone/`

[6]`https://github.com/skyhover/Deckard`

TABLE I
RESULTS FOR SELECTED CLONE DETECTION TOOLS ON BDD SCENARIOS

| System | Tool | Known Duplicates | Detected Known Duplicates | Candidates | Duplicates Between Whole BDD Scenarios | Detection for Known (%) | Precision (%) |
|---|---|---|---|---|---|---|---|
| System 1 | PMD | 9 | 0 | 12 | 0 | 0.00 | 0.00 |
| | CloneDR | 9 | 1 | 6 | 1 | 11.11 | 16.67 |
| | DECKARD | 9 | 9 | 589 | 112 | 100.00 | 19.02 |
| System 2 | PMD | 1 | 0 | 7 | 0 | 0.00 | 0.00 |
| | CloneDR | 1 | 0 | 1 | 0 | 0.00 | 0.00 |
| | DECKARD | 1 | 0 | 43 | 0 | 0.00 | 0.00 |
| System 3 | PMD | 3 | 0 | 8 | 0 | 0.00 | 0.00 |
| | CloneDR | 3 | 0 | 1 | 0 | 0.00 | 0.00 |
| | DECKARD | 3 | 1 | 951 | 48 | 33.33 | 5.05 |

**Algorithm 1: Detecting Semantic Duplication in BDD Specifications**

```
Input: A BDD suite $\Sigma$
       A set $T$ of public API traces of the scenarios in $\Sigma$
Output: Duplication report
foreach scenario $s$ in $\Sigma$ do
    foreach scenario $s'$ in $\Sigma$ (where $s' \neq s$) do
        $t \leftarrow$ public API trace for $s$ in $T$
        $t' \leftarrow$ public API trace for $s'$ in $T$
        $Comp \leftarrow$ A set of already compared scenario pairs
        if (($s$, $s'$) $\notin Comp$ and ($s'$, $s$) $\notin Comp$) and ($t = t'$ or subseq($t$, $t'$) or subseq($t'$, $t$)) then
            report $s$ as a semantic duplicate of $s'$
        $Comp \leftarrow$ ($s$, $s'$) and ($s'$, $s$)
```

implicated, and decide whether duplication exists or not. This is not feasible (or worthwhile) with so many candidates to inspect. In the case of this investigation, many of the false positives were reported because of partial duplication between the scenario-representing methods. The tools had correctly identified *code duplication*, but not forms of code duplication that were useful in identifying BDD scenario duplication.

## V. BDD DUPLICATE DETECTION

### A. Overall Approach

It seems likely that an optimal solution to this problem will require a combination of dynamic information from both glue code and production code, plus analysis of the specific values used in each scenario (to identify redundant test cases) and some analysis of the free text elements of the scenarios. But, at present, the precise kinds of information needed is unknown. As part of our attempt to discover this, we are exploring various hypotheses relating to duplicate detection in BDD suites. This paper presents the results of assessing one such hypothesis.

Since BDD scenarios are primarily examples that specify the behaviour needed of production code, one view is to regard two scenarios as duplicates if they exercise the production code in exactly the same way, and make the same assertions on the production code state that results. What does it mean to say that scenarios *exercise production code in the same way*? We can compare the dynamic traces for the scenarios, but we would expect to see some differences and some similarities. Both traces might use the same values for some parameters, where they are hard-coded into glue code/test harness code. Both might vary in some unimportant ways, such as the creation of objects within the production code that are not visible to the glue code. These essential and accidental differences need to be distinguished for successful duplication detection.

We set out to test a simple operation definition based on the hypothesis that: calls to the public API of the production code should be highly similar in duplicated scenarios, while calls to private internal methods within the production code may vary widely, even in duplicated scenarios. We work with filtered version of the call traces of the executed scenarios in which calls to methods which are not part of the public API of the system are removed. We call this filtered trace the *public API trace* (PAT) of the scenario. Formally, we use *pat(s)* to denote the PAT for scenario *s*.

*Definition 1:* Given two scenarios $s_1$ and $s_2$ in a BDD suite $\Sigma$, $s_1$ is a semantic duplicate of $s_2$ iff $pat(s_1) = pat(s_2)$, where two traces are considered equal if they describe the same sequence and number of method calls, and where calls to inherited methods are considered equivalent to calls to the methods they override.

### B. Algorithm and Implementation

Our implementation works with BDD feature files written in the Gherkin language, currently among the most widely used BDD languages [2], [23], with glue code written in Java, using Cucumber-JVM conventions[7]. The tool executes each scenario individually, using AspectJ[8] to generate trace information for the public methods of the code under analysis. Specifically, we capture: the qualified name of the class on which the method is defined, the name of the method, the parameter types, and the return type. The trace is produced as XML and we use XQuery to search for duplicates. Since BDD suites typically number in the tens to hundreds [2], we can compare all pairs of traces in acceptable time scales. The approach is given in Algorithm 1. For the function *subseq(x,y)* in Algorithm 1, we use the XQuery *contains(String, String)* function[9] to establish whether one sequence of API calls is subsumed into another.

## VI. EVALUATION

### A. Experiment Design

To evaluate our approach, we needed access to a BDD specification with known duplicates. Since no benchmarks exist, we recruited volunteers to inject duplicates into three open source software systems. We selected systems with sizeable BDD suites written in Gherkin, with Java glue code. We ruled out

[7]`cucumber.io/docs/reference/jvm`
[8]`www.eclipse.org/aspectj`
[9]`http://www.xqueryfunctions.com/xq/fn_contains.html`

TABLE II
CHARACTERISTICS OF SELECTED EVALUATION SOFTWARE

| | | Project | | |
|---|---|---|---|---|
| **S/n** | **Item** | **System 1** | **System 2** | **System 3** |
| 1 | Features | 23 | 8 | 14 |
| 2 | Scenarios | 142 | 41 | 4 |
| 3 | Scenario Outlines | 0 | 0 | 23 |
| 4 | Background steps | 21 | 0 | 8 |
| 5 | Packages | 76 | 2 | 19 |
| 6 | Production classes | 188 | 8 | 65 |
| 7 | Glue Code classes | 12 | 8 | 14 |

systems which were trivial demonstration projects or homework assignments, and focussed on systems in domains which can be understood based on general knowledge, rather than needing specialised domain expert input. A further criterion was that we should be able to execute all scenarios in full, to allow us to produce traces.

Based on these criteria, we searched the major popular open source project repositories for potential project. 7 projects were found with suitable BDD suites, but we could only execute the suites for 3 of these: jcshs[10], Facad Services[11], and ATest[12]. The *jcshs* system describes functions to facilitate interactions between customers of telecoms companies. *Facade Services* manages school information about teachers, pupils and associated services. *ATest*, amongst other things, facilitates online purchase of products. We hereafter use *System 1*, *System 2* and *System 3* to refer to jcshs, Facad Services and ATest, respectively. Table II characterises the three systems before injection of duplicates.

We recruited 13 volunteers for our study, all with at least 3 years experience in development, testing, or analysis and design or a general understanding of analysis, design and test-driven approaches. Some were working in industry, while some were PhD and MSc students with industry experience. Volunteers were approached physically and through mobile phone since all were known in person to the authors. They injected duplicates in two successive phases. In phase one, we worked with 2 volunteers to pretest the duplication injection instructions on System 1. Phase two involved the remaining 11 volunteers who were assigned evenly across the 3 systems to inject duplicates. To make the duplicates as realistic as possible from this somewhat artificial process, we used a variety of approaches to acquiring the duplicates. In one, we presented volunteers with the original scenario titles but without the scenario itself, and asked them to come up with steps matching the title. We then showed them the original scenarios but with the titles removed, and asked them to map their new scenarios to the originals. In another, we showed volunteers a feature description and three sample scenarios for it from the original system. We asked them to add more scenarios based on these but covering aspects of the feature not covered by the sample scenarios. Again, volunteers completed the task by mapping their new scenarios against the original. Yet, in the other, we gave complete scenarios and requested volunteers to write their duplicates by expressing them differently.

[10]`https://bitbucket.org/manv6/jcshs`

[11]`https://bitbucket.org/mohamrah/facad_services_cucumber_automated_test`

[12]`https://gitlab.com/alexandermiro/atest_bdd`

TABLE III
REPORTED DUPLICATES, AND CANDIDATE SETS' MEMBERS DISTRIBUTION

| | Known Duplicates | | | Candidates and Distribution | | | |
|---|---|---|---|---|---|---|---|
| **Project** | **Injected** | **Detected** | **%** | **CS** | **OO** | **II** | **IO** |
| System 1 | 35 | 25 | 71.4 | 55 | 5 | 23 | 27 |
| System 2 | 20 | 20 | 100.0 | 24 | 1 | 11 | 12 |
| System 3 | 6 | 5 | 83.3 | 16 | 2 | 7 | 7 |

Through this process, we obtained 61 known duplicate scenarios (strictly speaking, 61 pairs of duplicating scenarios) across the 3 systems. While keeping the functionality of the original scenarios, the duplicates we obtained through this process were the result of either rewording of individual steps in the original scenarios; or rewording or rearranging individual steps in the original scenarios, or both; or completely rewriting the original scenarios. There was a good mix of these, reflecting some of the ways in which duplicate scenarios can manifest themselves in real projects.

At this stage, the scenarios created by the volunteers could not be executed since the steps they contained were not matched by the existing glue code. It was necessary for us to add step definitions for all the new steps, to make an executable BDD suite. We were able to create these naturally, without needing to extend the production code (since we had guided the volunteers to write scenarios that duplicated the existing functionality, rather than inventing new requirements that the existing production code did not support). This gave us the versions of the 3 host systems with 61 known duplicate pairs, which could be executed.

## B. Results and Discussion

We ran our tool on the suites with injected duplicates. An interesting feature of the results was caused by the presence of *Background* steps in the Gherkin scenarios. Backgrounds contain the same kind of steps as scenarios (typically *Given* steps) but are not executed separately. Instead, they are added to the beginning of each scenario in the file. They provide a convenient way to set up common fixture elements for all scenarios for a feature. Because of this, the traces for all scenarios with backgrounds contain the trace for the background steps. This is not an interesting form of duplication, since it is embedded in the specification by the writer of the feature file. We therefore filtered out the traces for the background sections before searching for duplication.

Table III presents the detected known duplicates, and the distribution of members of the candidate sets, across the 3 systems. Column **CS** gives the candidate set reported by our tool.

Column **OO** gives the number of the candidate set members reporting duplication between original scenarios (i.e., scenarios that existed before volunteers injected duplicates). Column **II** gives the number of reported duplicates between scenarios injected by volunteers (i.e., artefacts of the verification process, and reported for completeness only). Column **IO** represents the number of duplicates between injected and original scenarios. We manually inspected each candidate pair in the OO and IO categories, to accept or reject them as duplicates and identify the known false negatives. In this way, we also identified injected scenarios that unintentionally duplicated multiple scenarios in the original suite. Our analysis of each pair in the OO and IO categories, in all 3 systems, confirmed that it was reasonable for our tool to flag them as duplicates.

Our approach detected more than 70% of the injected duplicates across the 3 systems. Most of the detected duplicates had the same sequences of API method calls (notably in System 2). Injected duplicates whose sequences of API calls were different, because of how the overall scenario functionality was distributed into steps, and thus affecting the order of execution of glue and production code, were missed by our algorithm. (These were mainly observed in Systems 1 and 3).

## VII. Conclusion

BDD is now used by many software teams to allow them to capture the requirements for software systems in customer readable and yet executable form. The resulting sets of concrete scenarios describing units of required behaviour provides a form of *living documentation* for the system under construction. Unfortunately, management of BDD suites over the long term can be challenging, particularly when they grow beyond a handful of features. Redundancy can creep in, leading to bloated BDD specifications that are costly to maintain and use.

In this paper, we have analysed the problem of duplicates in BDD suites, highlighted the limitations of existing tools on it, and described our initial dynamic tracing approach to detect BDD scenarios that exercise the production code API in the same way. Our evaluation results show that it detected the majority of the injected duplicates. In the future, we will investigate further hypotheses for detecting duplicates. We will explore what information needs to be added to our traces (for example, production method arguments and return values) and will consider approaches that deal with non-determinism in the scenario code, given the fact that production methods' runtime arguments may be decided in a non-deterministic way. We will also look at including static information from the scenarios, and combining this with the runtime trace information, to give more information about which of a pair of duplicates should be retained, in order to maximise the readability and usefulness of the BDD suite.

## References

[1] D. North, "Introducing BDD," Better Software Magazine, 2006.

[2] L. P. Binamungu, S. M. Embury, and N. Konstantinou, "Maintaining Behaviour-Driven Development Specifications: Challenges and Opportunities," in *Software Analysis, Evolution and Reengineering (SANER), 2018 IEEE 25th International Conference on*. IEEE, 2018.

[3] Z. F. Fang and P. Lam, "Identifying Test Refactoring Candidates with Assertion Fingerprints," in *International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages and Tools*, Melbourne, Florida, USA, 2015.

[4] D. Mathew and K. Foegen, "An Analysis of Information Needs to Detect Test Smells," Software Construction Research Group, Faculty of Mathematics, Computer Science, and Natural Sciences,Rwthaachen University, Germany, Tech. Rep. FsSE/CTRelEng 2016, February 2016.

[5] F.-H. Su, J. Bell, K. Harvey, S. Sethumadhavan, G. Kaiser, and T. Jebara, "Code relatives: detecting similarly behaving software," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 702–714.

[6] F.-H. Su, J. Bell, G. Kaiser, and S. Sethumadhavan, "Identifying functionally similar code in complex codebases," in *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. IEEE, 2016, pp. 1–10.

[7] M. Greiler, A. van Deursen, and A. Zaidman, "Measuring Test Case Similarity to Support Test Suite Understanding," *Springer*, vol. 7304, pp. 91–107, 2012, in C. A. Furia and S. Nanz, editors, TOOLS.

[8] R. Elva, "Detecting Semantic Method Clones in Java Code Using Method IOE Behavior," PhD Thesis, University of Central Florida, Orlando, Florida, USA, 2013.

[9] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[10] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and Accurate Tree-based Detection of Code Clones," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 96–105.

[11] R. Koschke, R. Falke, and P. Frenzel, "Clone Detection Using Abstract Syntax Suffix Trees," in *2006 13th Working Conference on Reverse Engineering*. IEEE, 2006, pp. 253–262.

[12] Y. Higo, Y. Ueda, M. Nishino, and S. Kusumoto, "Incremental Code Clone Detection: A PDG-based Approach," in *2011 18th Working Conference on Reverse Engineering*. IEEE, 2011, pp. 3–12.

[13] J. Mayrand, C. Leblanc, and E. M. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics," in *Software Maintenance 1996, Proceedings., International Conference on*. IEEE, 1996, pp. 244–253.

[14] K. Kontogiannis, R. DeMori, M. Bernstein, M. Galler, and E. Merlo, "Pattern Matching for Design Concept Localization," in *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*. IEEE, 1995, pp. 96–103.

[15] C. K. Roy and J. R. Cordy, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization," in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*. IEEE, 2008, pp. 172–181.

[16] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code," *IEEE Transactions on software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[17] M. Novakovic, "Language Evolution to Reduce Code Cloning," Masters Thesis, University of Waterloo, Waterloo, Ontario, Canada, 2013.

[18] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[19] R. Tairas and J. Cabot, "Cloning in DSLs: Experiments with OCL," in *International Conference on Software Language Engineering*. Springer, 2011, pp. 60–76.

[20] R. Tairas, S. H. Liu, F. Jouault, and J. Gray, "CoCloRep: A DSL for Code Clones," in *International Workshop on Software Language Engineering*, 2007, pp. 91–99.

[21] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[22] S. W. Suan, "An Automated Assistant for Reducing Duplication in Living Documentation," Masters Thesis, School of Computer Science, University of Manchester, Manchester, United Kingdom, 2015.

[23] A. Okolnychy and K. Foegen, "A Study of Tools for Behavior-Driven Development," Software Construction Research Group, Faculty of Mathematics, Computer Science, and Natural Sciences, Rwthaachen University, Germany, Tech. Rep. FsSE/CTRelEng 2016, February 2016.