An Exploratory Study of Functional Redundancy in Code Repositories

Marcelo Suzuki*, Adriano Carvalho de Paula†, Eduardo Guerra†,
Cristina V. Lopes‡, and Otávio Augusto Lazzarini Lemos*
*Science and Technology Department – Federal University of São Paulo at S. J. dos Campos – Brazil
{marcelo.suzuki, otavio.lemos}@unifesp.br
†Instituto Nacional de Pesquisas Espaciais, São José dos Campos, Brazil
{adriano.carvalho, eduardo.guerra}@inpe.br
‡Donald Bren School of Information and Computer Sciences – University of California, Irvine – USA
lopes@ics.uci.edu

***Abstract*—In large code repositories, the probability of functions to repeat across projects is high. This type of functional redundancy (FR) is desirable for recent code reuse and repair approaches. Yet, FR is hard to measure because it is closely related to program equivalence, which is an undecidable problem. This is one of the reasons most studies that investigate redundancy focus on syntactic rather than semantic replication (*e.g.*, cloning). In this paper we evaluate the extent of FR in a code repository with 68 Java projects taken randomly from SourceForge. Our technique approximates function similarity by first searching for methods that possess similar interfaces (return type, name, and parameter types). We then execute these methods to verify which candidate pairs have matching outputs for a given sample of inputs. Some recent studies have also focused on this type of semantic replication, but our detection approach is generally cheaper and more precise, because it focuses on methods and uses interfaces to reduce the search space. Although our scope is restricted to static methods, which makes our results conservative, our findings are promising. In particular, we found 984 pairs of redundant methods, and 28 out of the 68 (41.17%) projects in the repository presented redundancy. Moreover, the majority of redundant methods for which we had access to the source code did not refer to textual clones (only one redundant method pair referred to replicated code). Our study also indicates that the proposed redundancy detection approach has high precision and is generally inexpensive (only four executions were required per method to attain 100% precision).**

I. Introduction

In large code repositories it is reasonable to expect functions to reappear in multiple projects, with similar or different structure[1]. Recently, researchers have been trying to quantify this amount of redundancy [1, 2]. This property is important because it reflects the quantity of code rewritten across projects, either intentionally or unintentionally. Yet, most studies to date focus on syntactic replication and only recently researchers have started looking into actual functional redundancy [3–5].

One of the most popular types of code repetition investigated in code repositories is *cloning* [1]. Cloning happens when similar or identical code fragments repeat across software projects [6]. Recently, researchers have been investigating other types of syntactic replication, such as vocabulary, temporal, or interface redundancy. The idea is that sometimes two code fragments can be syntactically similar with respect to other aspects besides text. Vocabulary redundancy appears when different pieces of code share similar words [7], while temporal redundancy is concerned with the amount of code commits that are composed of previous commits [8]. In a recent study we have looked into the degree of repetition of whole method interfaces – return type, method name, and parameter types – across software projects, what we called Interface Redundancy (IR) [2].

While IR is important, we still cannot guarantee that two functions that share compatible interfaces always implement similar behavior. In this paper we go a step forward and try to measure the degree of repetition of functional behavior in a software repository. The idea is to gather candidate method pairs that possess similar interfaces – IR – and check whether they also possess similar semantics (what we call here Functional Redundancy – FR). We focus on *functions* implemented as methods because, as commented by Robert Martin, these can be considered the first line of organization in any program [9].

A minimum level of FR in code repositories is desirable for recent code reuse and repair approaches. For instance, in test case driven code search (TDCS) [10–12] and interface-driven code search (IDCS) [13–15], developers can search for code by using test cases or the interface of the desired method, respectively. These approaches will only be successful if functions do repeat across projects, at least to a certain degree. Automated software repair based on semantic search [16], on the other hand, works under the assumption that if a project contains a buggy method, other software projects might have implemented a bug-free version of the same or similar method.

Our strategy to detect method behavior similarity is composed of four steps. At first, by using IDCS, we obtain a set of candidate method pairs that possess similar interfaces. Then, by using Sourcerer, an infrastructure to analyze and index code [17], and a slicing technique [18], we generate slices of code necessary to execute each candidate method. The third step is to run each method a few times with some

[1]Pre-print. This is the author's version of the work. Not for redistribution. Once copyright information is available, it will appear here.

pre-defined inputs based on parameter types and record the obtained outputs. At last, we approximate behavior similarity by detecting which candidate pairs of methods give coincident outputs for the same set of inputs. We call this approach *method profiling*. To the best of our knowledge, although there are some studies that apply a similar strategy to detect analogous code, none of them focus specifically on methods, require only a few executions to obtain high precision (in our case, four was enough to reach 100% precision), and use IDCS to reach an initial set of candidate method pairs.

For this exploratory study we focused on a restricted set of methods, in order to facilitate our investigation. In particular, we looked at `static` methods that only manipulate primitive Java types and strings. These methods are easier to execute, since they do not require object instantiation. Also, to focus on non-trivial code, we targeted methods that do not return `void`, contain at least one parameter, have some implementation (that is, are not `abstract`), and are `public`. Our results are thus conservative in the sense that what we report pertains only these methods.

Although our scope is reduced, our results based on a repository with 68 Java projects from SourceForge, indicates that FR is fairly high in the corpus. In particular, more than 40% of the projects that contained at least one method that matched our criteria presented redundancy. In total, we found almost 1000 redundant method pairs. Moreover, from the projects that contained redundancy, on average, almost 20% of the target methods reappeared somewhere else in the repository. Our study also points out that method profiling is feasible. Specifically, we were able to compute slices for more than 98% of the targeted methods, and for the methods that we were able to generate `jar` files, more than 93% were successfully executed. Another interesting result from our study is that most of the redundant method pairs did not refer to textual clones (only 1 out of 16 pairs of redundant methods developed within projects was found to be cloned code).

The main contributions of our study are the following:

1) *An approach to detect functional redundancy in large code repositories which is generally cheap and precise*. In our study only four executions per method were required to attain high precision. The redundancy detection approach can also be used as a code search technique where the user submits a method and a tool that implements the approach can indicate similar methods in a given repository. Such a tool can be very useful for automated software reuse and repair;
2) *An investigation of FR in a code corpus containing 68 projects*. Results indicate that FR is fairly prevalent in the repository. These results are important for recent code reuse and automated repair approaches as they indicate the feasibility of these techniques;
3) *An investigation of the relation between FR and traditional code cloning*. Results suggest that FR significantly deviates from code cloning since most of the redundant method pairs we analyzed did not refer to cloned code. This is important as we show that analyzing syntactic clones is not enough to detect FR; and
4) *An investigation of IDCS as a technique to narrow down results for FR*. Results show that IDCS is effective in indicating candidate method pairs that might also be semantically redundant.

The remainder of this paper is structured as follows. Section II presents background information about the topics necessary to understand our study, such as IDCS and Sourcerer (the infrastructure used to run our experiment). Section III lays down details about our study, such as the target research questions and used repository. Section IV presents our results and discusses them in the light of our research questions, and Section V examines the limitations of our study. Section VI discusses related work and, finally, Section VII concludes the paper with our main findings and suggestions for future work.

II. Background

Software redundancy in large code corpora has become a popular topic among software engineering researchers [1, 19–21]. This property, if proven to be high, is an indication of how much rework can be going on among software developers. In fact, most studies indicate that the redundancy level tends to be significantly high in software *corpora* [1, 19–21].

As commented earlier, recent code search and repair approaches rely on software redundancy to be able to perform well. Next we discuss Sourcerer, an infrastructure to analyze and index code, and interface-driven code search (IDCS), an approach to code search that uses interface descriptions to search and reuse source code. Sourcerer was the basic groundwork required to run our study. IDCS, on the other hand, was used to narrow down candidate pairs of methods that might implement similar functionality. Moreover, as a secondary evaluation we want to check whether IDCS is in fact a good technique to reduce the number of method pairs required to check for functional redundancy.

A. Sourcerer and Interface-Driven Code Search

With the goal to support sophisticated code search, recently there has been many attempts to implement advanced infrastructure to analyze and index code. Sourcerer [17] is one of these efforts. Given a set of Java projects with available source code, Sourcerer can do the following: (1) automatically collect and store code in a repository with a standardized directory structure; (2) find and perform an in-depth structural analysis of the available source code (*e.g.*, dependence analyses); and (3) store the code structure information in a centralized database. With this information it is then possible to perform several types of mining tasks within the target repository, including code search.

Sourcerer stores essential structure information of the target source code in a relational database. For example, by running queries in the database, we can tell which packages contain which classes, which classes contain which fields or methods, which methods are called by which methods, and what are the return and parameter types of a given method. With this information in hands we are able to perform a number of different types of queries about the source code. In particular, it is possible to slice out a given method with everything it requires to be run. This is key to our study, as we need to execute arbitrary methods of the repository [11]. The *slicing service* we use extracts all classes, interfaces, methods and fields necessary to ensure correct compilation and execution of

the target method. The slicer computes the transitive closure of certain relations (such as method and constructor calls and field accesses) based on the code dependency graph stored in the database, so no further code analysis is needed. Further analysis is done to ensure that the class hierarchy remains intact in cases where it is relevant, and that implemented interfaces are pulled in when appropriate. For example, if two classes are extracted and one of them is a descendant of the other, then the extends relation will be extracted, as well as all classes between them in the hierarchy. A conservative approach is taken with respect to relations terminating outside of the project (*i.e.*, library calls), so the explicit types found in these relations are fully extracted.

With Sourcerer, it is also possible to perform searches in which information such as the return and parameter types of the desired function are specified in the queries. For instance, a user interested in the implementation of a function to *merge* a list of strings into a single string can search for a method that, given a *list of strings* (in Java: `List<String>`) returns the merged `String` and contains the term *merge* in its identifier [14]. We have been calling this type of search interface-driven code search (IDCS) [2, 13, 14].

B. Query Refinement

The relative ineffectiveness of information retrieval systems is mainly caused by the inaccuracy with which a query formed by few keywords can model the actual user information need [22]. Another known issue is that, in most collections, the same concept may be referred to with different words. This issue, known as *synonymy* (or vocabulary mismatch problem), impacts the recall of most retrieval systems [23]. This is also true in the context of code search, because keywords are still required.

Query refinement (QR) methods have been proposed to deal with this problem. In general, QR methods are classified as *global* or *local*. Global methods expand query terms independently of the results returned from the original query, so that changes in its wording will cause the new query to match other terms with similar semantics. Local methods, on the other hand, adjust a query relative to the documents that originally appear to match the initial query [23].

Automatic query expansion (AQE) is a well known global method to overcome synonymy [22]. It augments the user's original query with new features with similar semantics. In order to perform the queries in our experiment to measure FR, we also included a query refinement approach to IDCS that was used before. The approach used in the experiment described in this paper [13] is a global approach that does not require running the query prior to expanding it: the query is first expanded with similar terms, and then executed to find relevant functions. This strategy generally improves performance, as queries are run only once. Another advantage is not requiring additional user input [23]. We use this AQE approach in our experiment to improve recall, as high precision is generally obtained by using method profiling (as will be explained below).

The AQE approach used in our experiment can be configured with different thesauri [13]. In this paper, we configured it to be used with WordNet [24], a lexical database for the English language, and SimplType, a type thesaurus that expands numerical and Java collection types. In the above example where the user is looking for a function to merge a list of strings into a single string, the expanded IDCS query with the two selected thesauri would look like the following [14]:

return_type: *(String)*
keywords: *(merge^10 ∨ unite ∨ ...) ∧*
!(divide ∨ disunify)
parameter_types: *(List<String> ∨ List ∨ ...)*

Note that the query field where keywords are used is expanded with synonyms. More weight is given to the original terms selected by the user with the Lucene[2] boost factor, so that candidates that possess such terms are given a higher rank (in the example, *merge^10*). When there are multiple keywords, a conjunction is used among them, and a disjunction is used among the synonyms (*e.g.*, *merge* OR *unite*). In this way, a candidate is matched when it contains at least one term related to each of the concepts used in the query. The type thesaurus allows for a relaxed matching of the types. In the example, `List<String>` is expanded with other types of lists contained within the Java API (including the alternative where the type parameter `<String>` is removed). In queries, antonyms are also used to filter out unwanted results that might implement the opposite of the desired function (*e.g.*, *unzip* when *zip* is being searched)[3] [14].

C. Approximating function equivalence with method profiling

As it is well-known, program equivalence is generally undecidable [4]. Therefore, we need to apply some approach to approximate equivalence in order to measure redundancy. Moreover, for our purposes, we are more interested in implementations that are similar and not necessarily *completely* equivalent. This is because when we want to reuse some function, it does not need to be implemented exactly as desired, but some adaptation is somehow expected. Also, when looking for candidates that might possibly *repair* some target code – as it is the case for automated program repair approaches, it is in fact desirable that these candidates present some differences. After all, we are looking for bug-free versions of defective code, so these will be different (at least in the sense that they should not contain the bug).

Therefore, in this paper, we adopt an approach we call *method profiling* to approximate function equivalence. The idea is to first obtain a set of methods that contain similar interfaces (by applying IDCS), and then running these methods with a sample of inputs to check if the outputs match. If all outputs match for the given set of inputs, we say that the methods are *functionally redundant*. We call it *method profiling* because we generate a *profile* for each method which is composed by two characteristics: (1) its interface profile – return type, name terms, and parameter types; and (2) its execution profile – outputs given for a sample of inputs. In Section III we give specific details on how we generate inputs and how many inputs we adopted for this study.

[2]http://lucene.apache.org - 06/18/2017.

[3]In this paper we do not use Lucene to run the queries, but a simplified version of IDCS ran inside the relational database. The main differences are that we do not use antonyms to filter out results, but only synonyms to broaden our result set, and the boost factor for original terms is not applied.

The idea is very closely related to behavior sampling [25] and test-driven code search [11], whereas a set of test cases is used as input to search for some given function. If the candidate ever passes the tests, it is very likely to implement the desired function. Some recent studies also have applied similar metrics do approximate program equivalence [3, 5]. The difference from these approaches to ours is that we focus specifically on methods and use interfaces to narrow down method pairs that might implement similar functions. Moreover, some implementations use random testing to execute methods or take advantage of test cases available in the repository to generate inputs. Our approach is much simpler: we establish a set of pre-defined inputs based on types (see Section III for details on our implementation).

III. Study Setup

Our goal is to investigate the level of functional redundancy of methods inside source code repositories. In particular, we are interested in *reusable methods*, that is, non-trivial methods that have a more concrete interface (return a type other than `void`, have at least one parameter, are not *abstract*, and are `public`). We did this because trivial methods such as *getters* and *setters* would hardly be interesting from a reuse or automated repair perspective.

We also want to understand the feasibility of using method profiling to approximate redundancy and to verify its relation to classical code cloning. The feasibility of method profiling also entails the process of slicing out methods from the repository and executing them. Is this an applicable approach? Also, do the method pairs that match outputs for the same given inputs in fact have similar behavior? The idea of relating functional redundancy with cloning is the following: if most redundant methods are in fact clones, cloning detection should be sufficient to determine functional redundancy (we would not require a more expensive approach such as method profiling).

As a secondary investigation we also want to verify the usefulness of using IDCS with the AQE approach we adopted as a means to narrow down candidates that might be redundant. We do that by comparing the use of IDCS with verifying all method pairs in the repository that have type-compatible interfaces, regardless of their names (similar return and parameter types only).

Therefore, for our study we are interested in the following research questions. **In the context of large-scale Java source code repositories**:

RQ1. **Feasibility of our redundancy detection approach.** What is the feasibility of method profiling to detect functional method clones? In particular, what is the success rate for the executions of methods targeted in our experiment? Also, what is the *precision* given by our approach?

RQ2. **Extent of functional redundancy.** Given that our method is feasible, what is the extent of functional redundancy observed in the target repository according to our adopted redundancy criterion? More specifically, what percentage of projects contain functional redundancy? How many of the target methods in a project are redundant, on average? We also want to investigate the distribution of the different types of functional redundancy in the target repository (in particular, project-project vs project-library redundancy).

RQ3. **Relation of functional redundancy to cloning.** How does functional redundancy relates to classical code clone detection? More specifically, how much of functional redundancy is due to textual clones?

RQ4. **Effectiveness of interface-driven code search.** How effective is interface-driven code search to narrow down results that are more likely to be functionally redundant?

Since this is an exploratory study, in this paper we do not formally define hypotheses for our experiment. Rather, we decided to discuss our results descriptively, leaving more formal comparisons for future work.

A. Repository

Fraser and Arcuri [26] argue that several novel test data generation techniques are proposed without proper empirical evidence to support their usefulness. Case studies on such a topic tend to be either small or biased toward a specific kind of software. To cope with this problem, they have randomly selected 100 Java projects from SourceForge (SF[4]) in what was called a first attempt to run a statistically sound experiment in test data generation. The resulting benchmark was called SF100, a statistically representative sample of Java projects from SF. The representative sample issue also affects the evaluation of code search techniques: many times the target corpora are not a random sample of software projects, but a specific set of projects, which can introduce bias to the study. Concerned with this problem, and to construct a manageable and unbiased code base, we used Sourcerer to index Fraser and Arcuri's repository. However, instead of the original corpus, we use SF110, a version of SF100 the authors have recently extended with the 10 most popular SF projects[5]. SF110 thus forms the target repository for the experiment described in this paper.

As commented before, Sourcerer stores all code metadata in a relational database. The central element of the Sourcerer's model is the `entities` table, which contains information about all source code elements of a given code base (*e.g.*, classes, methods, fields). Once we had a Sourcerer database containing the SF110 metadata, we constructed the target repository to form the basis of our experiment. Since in our study we have to execute methods on sample inputs to record their outputs, as commented earlier, we decided to restrict our scope to `static` methods that manipulate only primitive types and strings. These are much easier to run since they do not need any object instantiation. Table I presents the set of filters we applied to the `entities` table to reach the search space set of reusable static methods contained in SF110. The last line of the table is the result of applying all filters containing the desired interfaces for our experiment, that is, all non-abstract, public, `static` methods that contain at least one parameter and return a non-`void` type and only handle primitive types and strings.

Project `075 openhre` was removed from our target repository because it contained a disproportional number of source

[4]http://sourceforge.net/ - 06/19/2017.

[5]We have used SF100 in other recent code repository experiments [2, 13, 14].

TABLE I: Filters applied to the `entities` table to reach the desired search space for our experiment. (Total = total # of methods after applying the filter)

Filter	Total
none	9,816,961
`entity_type = METHOD`	3,008,539
`modifiers like %PUBLIC%`	2,358,256
`modifiers not like %ABSTRACT%`	2,041,066
`params <> ()`	1,220,354
`return <> void`	598,453
`project <> openhre`	598,244
`manipulate only primitive types or strings`	45,781

methods, which could bias our results. From the set of 45,781 methods, we conducted a further filtering to reach the reusable methods that were actually developed inside the projects contained in the repository (that is, excluding the ones from libraries and the Java API). With this further filtering we reached the number of 1,488 *source methods*, which were then used to form the base queries to search the whole search space in order to measure FR[6]. It is important to note that the set of 1,488 methods is used to form the queries, but the search space also includes the remaining reusable methods in the repository. We did this to also target the situation where a developer implements a method that could be reused from a library or the Java API, instead of being written from scratch. Some projects did not have any methods that matched our criteria. Finally, the number of projects that contained at least one source method was 68.

B. Redundancy detection method

As commented before, to approximate redundancy we use *method profiling* which refers to running pairs of methods with compatible interfaces and comparing the outputs for a given sample of inputs. The process has four steps: (1) obtaining a first set of candidate method pairs that possess similar interfaces by running IDCS for every source method in the repository; (2) constructing slices of all methods located at developer projects and generating `jar` files to be able to run them (for the library methods we skip the first part of this step and gather the `jar` files only); (3) running each method on a set of inputs and recording the related outputs on the database; (4) verifying which candidate method pairs give matching outputs for the same inputs.

One of the parameters of method profiling is the size of the input sample (*i.e.*, number of executions per method) that will be used to compare methods. In our case, following a result reached by Podgurski and Pierce [25], we initially set the sample size to four. These authors have found that a sample size as small as four was enough to obtain a reasonable precision while performing behavior sampling. Since method profiling is very similar to that approach, we believe this result can also be used here. To be sure we are in fact obtaining high precision, we manually verify a set of method pairs that our approach identifies as similar by running four sample inputs.

[6]It is clear that library methods are much more likely to repeat across projects, that is why we decided not to include them in the set of *source methods*. If we did so, we would probably introduce bias to the experiment. Since we are more interested in code that was actually developed inside projects by the developers, and not inside libraries, the natural choice was to include only those for the searches.

For the input generation we decided to follow a simple approach. In this exploratory study we focus on primitive types and strings, for which it is easier to generate inputs. The main idea was to generate *default* values for each Java primitive type and string as to be able to construct four inputs for the methods. Note that we do not want to put methods to the test, we only want to obtain outputs for some given *normal* executions to be able to compare to outputs given by other possible similar methods. Therefore, we do not focus on boundary or exceptional values. Also, inputs for methods with multiple parameters are obtained by circularly iterating through the list of sample inputs. Table II shows a sample of template inputs we defined for each Java primitive type and string considered in our experiment, together with the guidelines used to derive each input.

TABLE II: Sample of pre-defined inputs and guidelines used for each Java primitive type and string in our study.

Types	**Guidelines used**	**Sample inputs**
`boolean`	All possible inputs.	`true`, `false`
`char`	Lowercase, uppercase, special and empty.	'K', 'm', ''
`byte`	Numerical, positive, special.	'9', '\n', 9
`int, short, long`	Positive, negative, zero.	-13, 25, 0
`float, double`	Positive, negative, zero.	25.2, -13.8, 0
`String`	Numerical, normal filename.	"235", "long", "filename.txt"

For our study, a given pair of candidate methods m_x and m_y is deemed similar ($S(m_x, m_y) = 1$) when the following is true: (1) return types are similar (either same or compatible according to our type thesaurus [14]), (2) method name is similar (either same or synonymous according to WordNet); (3) number and types of parameters are the same or similar (regardless of the order in which they were defined); (4) when m_x and m_y are run on a sample of coincident inputs, all outputs match. (1), (2), and (3) are given by IDCS, while (4) is given by matching the outputs for a given sample of inputs. Also, for a given pair of methods (m_x, m_y), we say that m_x is the *source method* and m_y is the *target method*.

m_x is redundant if the number of other methods similar to it in the repository is larger than one according to the similarity function used. The redundancy of a given method m_x ($R(m_x)$) is then defined as follows – for all other m_y target methods in the search space:

$$\mathbf{R(m_x)} = \begin{cases} 0 & \text{if } S(m_x, m_y) = 0 \text{ for any } m_y; \\ 1 & \text{if } S(m_x, m_y) = 1 \text{ for at least one } m_y. \end{cases}$$

C. Classifying redundant method pairs

To address the second part of research question 2, we classified pairs of redundant methods with respect to whether the source method behavior reappeared in code developed inside a project or inside a library (project-project vs project-library redundancy). Such classification is important as it shows whether possible reuse happened from code implemented by the developers of the projects themselves or from libraries or the Java API. For a given pair of methods m_x and m_y:

- (m_x, m_y) are called *project-project redundant* if $S(m_x, m_y) = 1$ and m_x and m_y are located in code inside projects and not in libraries; and
- (m_x, m_y) are called *project-library redundant* if $S(m_x, m_y) = 1$ and m_x is located in code inside a project and m_y is located in a library.

library-library redundant pairs could also be defined, but since we are more interested in pairs of methods for which at least one of the methods was implemented inside some project in the repository, we did not measure such a type of redundancy.

IV. Results and Analysis

To evaluate the extent of functional redundancy in a large open source Java repository, we applied the redundancy detection method described before to our target repository. First we analyze the successfulness of our approach. Then, we evaluate the amount of redundancy contained in the repository for static methods that manipulate primitive types and strings using method profiling. Thirdly, we compare functional redundancy to classical clone detection, to verify whether or not most of redundant functions are due to textual clones. A final evaluation pertains IDCS and its ability to narrow down method pairs that are more likely to be functionally redundant. For this endeavor we compare the use of IDCS with the use of pairs of methods that contain type-compatible interfaces (that is, without considering their names).

A. Feasibility of our redundancy detection approach

Executing arbitrary methods from code repositories is a complex task. In particular, there are a few things that could go wrong. For instance, the slicer might not pull out every dependency required to run the method; the method's code itself might contain faults; the method might require unanticipated resources in order to run (*e.g.*, a file or a System property); among other difficulties. For these and other reasons our method profiling approach is failure-prone. In this section we analyze the successful rate we obtained from trying to run static methods from our target repository using our approach, to check its feasibility.

Table III shows the successful rate of method profiling applied to 1,488 static methods from our target repository. These methods are the ones for which there are at least another method in the repository with a type-compatible interface (that is, similar return and parameter types) with similar name.

TABLE III: Feasibility results of method profiling applied to our target repository (#m = number of methods; prev. = previous).

Set	#m	% of total	% from prev. phase
Total	1,488	100	N/A
Slice successfully created	1,472	98.92	N/A
`jar` file successfully created	944	63.44	64.13
Method executed without failures	885	59.48	93.75

Note that the successful rate is quite high. In particular, we were able to execute 93.75% of the methods for which we have successfully generated `jar` files. This indicates that our approach is feasible because it is able to run a fair amount of arbitrary methods from a code repository.

Another question regarding our method profiling approach pertains its precision. How often our approach classifies as redundant a pair of methods that are not at all related? In order to verify this property, we have manually analyzed a sample of 22 method pairs for which we have access to the source code and our approach have classified as redundant. This is the complete set of project-project pairs that were reported as redundant, that is, the ones for which there were four matching outputs for the input sample we used. We want to check whether these pairs indeed implement analogous functions. In this case we are measuring the *precision* of our approach. To make a comparison with the use of less matching outputs, we also include the pairs for which three outputs matched (and one did not) in our study. Table IV presents all these pairs. Consider (a, b) a method pair.

Note that all pairs with four matching outputs are in fact functionally redundant, which means that our approach classified all of them correctly. Hence the precision for functional redundancy for these method pairs was 100%. This is not the case when we consider only three matching outputs: there were two false-positives in that case. This indicates that our approach can achieve high precision while detecting functionally redundant methods.

It should be clear that the sample for which we are testing precision is small: 16 our of 984 pairs of methods detected as redundant in our experiment. However, these were the pairs for which we had access to the source code. The remaining majority of pairs refer to code that repeat from project to library (or the Java API). In any case we believe that results should not diverge significantly for these other pairs, because we are using the whole method interface to narrow down results, and on top of it executing the methods four times in order to verify redundancy. Of course there might be false-positives among the 968 remaining pairs, but we do not believe the numbers will be too high, as indicated by the verification of the project-project pairs.

> With respect to research question 1, our results indicate that method profiling is feasible in the task of detecting functional redundancy. In particular, we were able to compute slices for more than 98% of the targeted methods, and for the methods that we were able to generate `jar` files, more than 93% were successfully executed. The precision reached for a sample of 16 pairs of methods classified as redundant for which we had access to the source code was 100% (that is, all of them did in fact refer to analogous functions).

B. Extent of functional redundancy

In this section we evaluate the extent of functional redundancy for the non-trivial static methods that manipulate primitive and string types in the target repository. Table V shows our main results pertaining functional redundancy.

Note that FR is common in the repository: more than 40% of all projects that contain at least one non-trivial static method present redundancy. Moreover, we found 70 different methods whose functions reappear in other projects (almost 5% of all

TABLE IV: Project-project method pairs classified as redundant by our approach in the experiment ((a, b) is a method pair; #MO = number of matching outputs; FR = actual functionally redundant; C = Clone).

#	method a interface	method b interface	#MO	FR?	C?
1	`String MyString.replace(String, String, String)`	`String XStringSupport.replaceAll(String, String, String)`	4	Yes	No
2	`String RuntimeUtils.substitute(String, String, String)`	`String Util.Replace(String, String, String)`	4	Yes	No
3	`String StringUtils.substitute(String, String, String)`	`String Util.Replace(String, String, String)`	4	Yes	No
4	`String StringUtils.replace(String, String, String)`	`String Util.Replace(String, String, String)`	4	Yes	No
5	`String StringUtils.replace(String, String, String)`	`String RuntimeUtils.substitute(String, String, String)`	4	Yes	No
6	`String StringUtils.replace(String, String, String)`	`String StringUtils.substitute(String, String, String)`	4	Yes	No
7	`int JillionUtil.compare(long, long)`	`int ComparatorUtils.compare(long, long)`	4	Yes	No
8	`int JillionUtil.compare(int, int)`	`int ComparatorUtils.compare(int, int)`	4	Yes	No
9	`int JillionUtil.compare(short, short)`	`int ComparatorUtils.compare(short, short)`	4	Yes	No
10	`int JillionUtil.compare(byte, byte)`	`int ComparatorUtils.compare(byte, byte)`	4	Yes	No
11	`int JillionUtil.compare(int, int)`	`int ComparatorUtils.compare(byte, byte)`	4	Yes	No
12	`int JillionUtil.compare(short, short)`	`int ComparatorUtils.compare(int, int)`	4	Yes	No
13	`int JillionUtil.compare(byte, byte)`	`int ComparatorUtils.compare(int, int)`	4	Yes	No
14	`int JillionUtil.compare(byte, byte)`	`int ComparatorUtils.compare(short, short)`	4	Yes	No
15	`String Resources.getString(String)`	`String Messages.getString(String)`	4	Yes	Yes
16	`String ParseUtil.encode(String)`	`String HtmlEncoder.encode(String)`	4	Yes	No
17	`String GeneralUtils.replaceAll(String, String, String)`	`String XStringSupport.replaceAll(String, String, String)`	3	Yes	No
18	`String MyString.replace(String, String, String)`	`String GeneralUtils.replaceAll(String, String, String)`	3	Yes	No
19	`String UUIDGenerator.trim(String)`	`String StringUtils.trim(String)`	3	Yes	No
20	`boolean DBEHelper.isNumeric(String)`	`boolean Identifiers.isNumeric(String)`	3	No	No
21	`boolean Identifiers.isNumeric(String)`	`boolean Main.isNumeric(String)`	3	No	No
22	`String MacawMessages.getMessage(String)`	`String ResourceBundlePurchaser.getMessage(String)`	3	No	No

TABLE V: Functional redundancy figures found in our experiment (% = Percentage from total, Avg. = Average).

	Number	%
Total # of projects considered	68	100
Total # of projects with redundant methods	28	41.17
Total # of redundant methods	70	4.7%
Avg. # of redundant methods per project with redundancy	2.5	19.28
Avg. # of matching methods per redundant method	35.14	–

methods that match our criteria, and more than 39% of the methods that we were able to execute and for which there is least another method with similar interface in the repository). Another interesting results is that, on average, almost 20% of the methods considered in our experiment reappear at least once in the repository, and the average time it resurfaces is more than 35. Such results reinforce the applicability of approaches that rely on a certain level of redundancy, such as IDCS and automated software repair via code search. It shows that the chances of having another method that implements a given function at hand in a large repository is high. It is important to remember that our scope is restricted to static methods that manipulate only primitive types and strings. Our results are thus conservative: they establish a lower boundary for FR in a large code repository. It might be the case that when we consider instance methods and static methods that manipulate more specific types, these numbers will rise.

TABLE VI: Distribution of the different types of redundant method pairs found in our study (Diff. = Difference in percentage; Avg. = Average per project).

	project-project	project-library	Diff.
Total	968	16	60.5
Avg.	14.23	0.23	98.38

Table VI shows the distribution of project-project and project-library redundant method pairs in the repository. Note that the vast majority of redundant pairs refer to methods that reappear in libraries or the Java API (they were more than 60 times more prevalent than project-project pairs). This is expected as code developed in libraries is more likely to be common functions that tend to be implemented by many developers. Our results show that there seems to be common the situation where a developer implements functions that are already present in some library. This suggests that programmers might be losing opportunities to reuse code from code collections.

From our initial results, one might think that IDCS alone should suffice to identify FR. However, this is not the case, since the initial number of candidate method pairs given by running IDCS in our experiment was 3,957 (from which 984 had four matching outputs – 24.86%). This indicates that both processes are important. Without IDCS we would have a very large number of false-positives when using only type-compatible interfaces, as shown in results for research question 4 below. On the other hand, without actually running the methods we would also have a large number of false-positives (more than 75% of the method pairs).

With respect to research question 2, our results indicate that FR is common in large code repositories. For instance, we noticed that for projects that contain redundancy, on average, almost 20% of the methods that match our criteria reappear somewhere else in the corpus with the average time that these methods resurface being more than 35. Moreover, more than 40% of all the projects that contained at least one method in the scope of our study presented redundancy. This indicates the applicability of code reuse and repair approaches that take advantage of large code corpora. We also found out that the vast majority of methods developed inside projects tend to reappear in libraries rather than in code developed inside other projects. This suggests that programmers might be losing opportunities to reuse code from these code collections.

C. Relation of functional redundancy to cloning

In research question 3, we investigate the relation between FR and classical clone detection; that is, method clone de-

tection based on textual analysis. The main question is how much of FR is due to textual clones, and how much is related to Type-4 clones (*i.e.*, methods that implement analogous functions but with diverse code). In this endeavor we focused on project-project redundant pairs, as these are the methods implemented by developers inside projects, and not in libraries.

To verify clones, we manually analyzed how many of the 16 project-project redundant method pairs referred to replicated code. In fact, for most of the pairs we observed that only the interfaces matched, while the corresponding code was significantly different. To verify our manual analysis we ran Simian[7], a tool that automatically identifies text duplication in Java and other programming languages. Table IV showed all the target method pairs. Note that only one – pair 15 – was classified as textual clone (6.25%), while the other 15 (93.75%) did not refer to replicated code. The other pairs only shared the interface, with the exception of one of them – pair 19 – which had only 14% of coincident lines of code according to Simian. This is an important result, as it shows that FR differs significantly from classical method clone detection.

> With respect to research question 3, our results show that only a very small percentage of FR is due to cloning – a single clone was found (6.25% of the project-project redundant method pairs for which we had access o the source code) –, while the majority did not refer to methods with similar code (93.75%). This is an evidence that FR diverges significantly from classical method clone detection.

D. Effectiveness of interface-driven code search

In research question 4 we are interested in the effectiveness of interface-driven code search to narrow down candidates of methods that might be functionally redundant. To evaluate this property we have ran our method redundancy approach to all project-project method pairs that possess *compatible* interfaces, regardless of the method name (*i.e.*, methods that have same or similar return and parameter types, and the same number o parameters). We will compare results for these pairs with the results for the 16 project-project pairs classified as redundant when IDCS was used (results presented in Table IV). Table VII shows numbers for the comparison. We decided to incorporate recall as a second metric to evaluate the approaches. For this endeavor we consider 100% recall in the case of the compatible interfaces approach, since these refer to *all* method pairs with compatible interfaces regardless of their names. It is clear that there might be analogous functions implemented with different interfaces but these are not being considered in our context.

TABLE VII: Comparison between using IDCS and searching for compatible interfaces only in the task of mining functionally redundant methods (#F-P = Number of false-positives, p = precision, r = recall).

Approach	# of pairs classified as redundant	#F-P	p	r
Compatible interfaces	1949	1900	0.025	1.00
IDCS	16	0	1.0	0.32

Note that precision is very low when using only compatible interfaces to narrow down candidates to verify functional redundancy: it is less than 3%. The number of false positives is very large: 1900. This indicates that method profiling works much better when combined with IDCS: running four times each method is not enough to differentiate most unrelated methods that only possess type-compatible interfaces. Our intuition is that this happens because many methods work alike when 'unusual' inputs are given to them. For instance, if a method that removes escaped characters from strings receives a string with no escaped characters, it will probably return the input string itself. Another compatible method that updates a string for other types of characters, which also receives a string and returns a string, might behave similarly for the same input. Nevertheless the methods are not redundant. However, when the method name is used in the search – such as in IDCS –, this type of problem will happen with much less probability.

With respect to recall, we notice that IDCS misses several redundant pairs of methods. This means that our results are even more conservative than initially expected: there are much more redundant pairs than reported for research question 1 (we found 49 project-project pairs, compared to the 16 found with IDCS). However, we decided to favor precision in our context, because we only want to classify a pair of methods as redundant when there is high confidence that the pair is in fact redundant. If we were to use only compatible interfaces it would not be possible to clear out all false-positives: if the number of project-project pairs was as high as 1949, imagine when we consider the project-library pairs (remember that the number of project-library pairs was more than 60 times larger than the number of project-project pairs for the IDCS approach – see Table VI). In any case, by manually looking at the pairs that were missed by IDCS we see lots of opportunities to improve this search approach. In the future we intend to use this information to be able to tune our method profiling approach.

> With respect to research question 4, we found out that IDCS is very effective to narrow down candidates that are in fact redundant. The precision for that approach when considering project-project pairs was 100% against only 2.5% when using type-compatible interfaces only. However, we noticed that IDCS has a lot of room to improve with respect to recall. In fact, we note that our results with respect to the extent of FR are even more conservative than initially expected. Indeed, when using type-compatible interfaces we found 49 project-project redundant pairs, compared to the 16 found by method profiling using IDCS. This means that the degree of FR in the repository might be even higher than reported.

V. Study Limitations

We believe the main threat to the validity of our experiment is the size of our target repository, which is formed by 110 Java projects extracted from SourceForge (with 68 containing the methods targeted in our study). This might affect the generalization of our results to larger collections. In any case we believe our results are important as a first exploratory study into FR that focuses on methods, as it does involve a significant number of such modules in its search space (more than 45,000; *i.e.*, it can be representative of a company's local repository, for instance). On the other hand, it is unclear whether FR will change significantly when the size of the repository increases.

[7] http://www.harukizaemon.com/simian/index.html - 8/3/2017.

This is because it is not clear whether FR is relative to the number of methods contained in the repository; that is, even though there might be more modules to be redundant with respect to a given method, the number of queries will also increase with a larger number of modules. Nevertheless, we intend to replicate our experiment with larger repositories curated by our research group in the near future, to verify how FR can be affected by the repository size.

Another shortcoming related to our results is that we perceived that the recall of method profiling with IDCS to measure FR is generally low (0.32 considering the project-project pairs). However, this means that our results are conservative: with more recall we would probably conclude that FR is even higher than what we report. This is even more optimistic for reuse and repair approaches that make use of large repositories. We believe our study establishes a lower boundary for FR, since we focused on static methods and a high precision detection approach with generally lower recall.

VI. Related Work

The idea of looking at functional redundancy is closely related to many areas of software engineering research that have similar concepts.

Clone Detection. Over the years, several techniques have been proposed for clone detection. Roy et al. [27] presents a comprehensive survey highlighting the strength and limitation of various clone detection techniques. These techniques differ in many aspects, ranging from the type of detection algorithm they use to the code representation they operate on. Examples of representations used are tokens [28, 29], abstract syntax trees (AST) [30, 31], program dependence graphs [32, 33], suffix trees [28, 34], text representations [35, 36], and hash representations [37]. Each of these different approaches have their own merits and are useful for different use cases. For example, AST-based techniques have been shown to have high precision, and are useful for clone refactoring, but they may not scale. On the other hand, token based techniques have high recall but may yield clones which are not syntactically complete [38]. These are useful where high recall is important. In comparison to our study, we showed that the majority of the functionally redundant pairs of methods for which we had access to the source code (project-project pairs) where not syntactically similar: we found only a single textual clone. This indicates that our technique significantly diverges from classical clone detection.

Code Reuse. Gabel and Su [20] conducted a large-scale study on the uniqueness of source code in a repository containing 6,000 projects. They found high syntactical redundancy at various levels of granularity. Similarly, Mockus conducted a large-scale study of file-level reuse and found evidence of high code reuse. Jiang and Su identify functionally identical code fragments based on test executions [3]. Similar approaches were proposed by Elva & Leavens [39], Su et al. [5], Carzaniga et al. [4], Kim et al. [40], and Deissenboeck et al. [41]. While their goal of exploring functional similarity was similar to ours, our approach is essentially different. In particular, we use interface-driven code search and query expansion to narrow down candidates to be dynamically executed. Moreover, most related techniques proposed before require 10 or more executions to achieve high precision in identifying redundant code [39]. Our approach, on the other hand, was able to attain high precision with only four executions per method.

Ossher et al. [42] looked at reuse in open source Java systems using hash-based file level clone detection and classified the usage scenarios into good, bad, and ugly. These scenarios included good use cases like extension of Java classes and popular third-party libraries, both large and small. All the above studies noted that exploring code reuse at a large scale has many challenges. Our experience with this study suggests that method profiling to identify functional redundancy has a potential to be effectively used as a practical and inexpensive technique to identify potential reuse and repair opportunities at a large scale.

Code Search & Program Repair. Recently, researchers have started looking at the possibility of using code search for program repair [16, 43]. The idea is to identify a buggy chunk of code and then find relevant, probably correct code from large code repositories as the source for the patches. Since the success of such approach relies on how big the repository is, the search space of potential patches is thus greatly increased. We believe functional redundancy via method profiling can be used as a quick and cheap approach to narrow down to the potential candidates.

VII. Conclusions and Future Work

In this paper we have presented an exploratory study that looks into functional redundancy (FR) in a large code repository. In particular we propose a technique to detect redundancy that relies on interface-driven code search and output matching for sample inputs. Our results indicate that our approach is feasible to detect functionally redundant code (only four executions per method were required to attain high precision), and that large repositories seem to present significant levels of FR. In particular we found that 28 out of 68 (41.17%) projects that contained methods within our defined scope presented redundancy. Moreover, from the projects that presented at least one redundant method within our scope, on average, almost 20% of the methods reappeared somewhere else in the repository (35 times on average). This indicates that recent approaches of reuse and repair that rely on large repositories should be effective. Since we have focused on a limited set of methods (non-trivial static methods that manipulate primitive types and strings), our results are quite conservative. We believe they present a lower boundary for FR in large corpora.

Future work includes extending our study to incorporate other types of methods (*e.g.*, instance methods). We also plan on using our method profiling technique to implement a tool that suggests improvements on software projects based on the collective intelligence contained in large repositories. The idea is to submit all methods from a project and verify other similar methods in the repository that might improve the code of the original modules. Since FR seems to be quite prevalent in these collections, we believe such a tool has potential to be useful and help software developers improve their projects.

Acknowledgements

Otavio Lemos would like to thank FAPESP for financial support (grant 2015/12787-0). This work was partially supported by a grant from the DARPA MUSE program.

REFERENCES

[1] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 1157–1168.

[2] A. de Paula, E. Guerra, C. V. Lopes, H. Sajnani, and O. A. L. Lemos, "An exploratory study of interface redundancy in code repositories," in *Proc. of the SCAM 2016*, 2016, pp. 107–116.

[3] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09. New York, NY, USA: ACM, 2009, pp. 81–92.

[4] A. Carzaniga, A. Mattavelli, and M. Pezzè, "Measuring software redundancy," in *Proceedings of the ICSE '15*. Piscataway, NJ, USA: IEEE Press, 2015, pp. 156–166.

[5] F.-H. Su, J. Bell, K. Harvey, S. Sethumadhavan, G. Kaiser, and T. Jebara, "Code relatives: Detecting similarly behaving software," in *Proc. of the FSE 2016*, ser. FSE 2016, New York, NY, USA, 2016, pp. 702–714.

[6] J. R. Pate, R. Tairas, and N. A. Kraft, "Clone evolution: A systematic review," *J. Softw. Evol. Process*, vol. 25, no. 3, pp. 261–283, Mar. 2013.

[7] Y. Higo and S. Kusumoto, "How should we measure functional sameness from program source code? an exploratory study on java methods," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 294–305.

[8] M. Martinez, W. Weimer, and M. Monperrus, "Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 492–495.

[9] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.

[10] S. P. Reiss, "Semantics-based code search," in *Proc. of the ICSE 2009*. IEEE Computer Society, 2009, pp. 243–253.

[11] 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," *Inf. Softw. Technol.*, vol. 53, pp. 294–306, April 2011.

[12] O. Hummel and W. Janjic, "Test-driven reuse: Key to improving precision of search engines for software reuse," in *Finding Source Code on the Web for Remix and Reuse*, S. E. Sim and R. E. Gallardo-Valencia, Eds. Springer New York, 2013, pp. 227–250.

[13] O. A. L. Lemos, A. C. de Paula, F. C. Zanichelli, and C. V. Lopes, "Thesaurus-based automatic query expansion for interface-driven code search," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 212–221.

[14] O. A. L. Lemos, A. C. de Paula, H. Sajnani, and C. V. Lopes, "Can the use of types and query expansion help improve large-scale code search?" in *Proc. of the IEEE 15th International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM 2015. IEEE, 2015, pp. 41–50.

[15] O. Hummel, W. Janjic, and C. Atkinson, "Evaluating the efficiency of retrieval methods for component repositories," in *Proc. of SEKE 2007*. KSI, 2007, pp. 404–409.

[16] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search," in *Proc. of the ASE '15*, Washington, DC, USA, 2015, pp. 295–306.

[17] S. Bajracharya, J. Ossher, and C. Lopes, "Sourcerer: An infrastructure for large-scale collection and analysis of open-source code," *Sci. Comput. Program.*, vol. 79, pp. 241–259, Jan. 2014.

[18] O. A. L. Lemos, S. K. Bajracharya, and J. Ossher, "Codegenie: a tool for test-driven source code search," in *Companion to the 22nd ACM SIGPLAN OOPSLA*. New York, NY, USA: ACM, 2007, pp. 917–918.

[19] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Commun. ACM*, vol. 59, no. 5, pp. 122–131, Apr. 2016.

[20] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10. New York, NY, USA: ACM, 2010, pp. 147–156.

[21] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 306–317.

[22] C. Carpineto and G. Romano, "A survey of automatic query expansion in information retrieval," *ACM Comput. Surv.*, vol. 44, no. 1, pp. 1:1–1:50, Jan. 2012.

[23] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[24] G. A. Miller, "WordNet: a lexical database for english," *Commun. ACM*, vol. 38, no. 11, pp. 39–41, Nov. 1995.

[25] A. Podgurski and L. Pierce, "Behavior sampling: A technique for automated retrieval of reusable components," in *Proceedings of the ICSE 1992*. New York, NY, USA: ACM, 1992, pp. 349–361.

[26] G. Fraser and A. Arcuri, "Sound empirical evidence in software testing," in *Proc. of the ICSE 2012*. IEEE Press, 2012, pp. 178–188.

[27] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program*, pp. 470–495, 2009.

[28] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, 2002.

[29] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proceedings of Working Conference on Reverse Engineering*, 1995.

[30] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *Working Conference on Reverse Engineering (WCRE'06)*. IEEE Computer Society, 2006.

[31] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of ICSE*, 2007.

[32] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," in *Proceedings of the 8th International Symposium on Static Analysis*. Springer-Verlag, 2001, pp. 40–56.

[33] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," in *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*. IEEE Computer Society, 2001, p. 301.

[34] R. Koschke, "Large-scale inter-system clone detection using suffix trees," in *Proceedings of CSMR*, 2012, pp. 309–318.

[35] A. Marcus and J. I. Maletic, "Identification of high-level concept clones in source code," in *Proceedings of International Conference on Software Engineering*, 2003.

[36] J. Cordy and C. Roy, "The nicad clone detector," in *Proceedings of ICPC*, 2011.

[37] S. Uddin, C. Roy, K. Schneider, and A. Hindle, "On the effectiveness of simhash for detecting near-miss clones in large scale software systems," in *Proceedings of Working Conference on Reverse Engineering*, 2011.

[38] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," *IEEE Trans. Softw. Eng.*, vol. 33, no. 9, pp. 577–591, 2007.

[39] R. Elva and G. T. Leavens, "Semantic clone detection using method ioe-behavior," in *Proceedings of the 6th International Workshop on Software Clones*. Piscataway, NJ, USA: IEEE Press, 2012, pp. 80–81.

[40] H. Kim, Y. Jung, S. Kim, and K. Yi, "Mecc: Memory comparison-based clone detector," in *Proceedings of the ICSE 2011*. New York, NY, USA: ACM, 2011, pp. 301–310.

[41] F. Deissenboeck, L. Heinemann, B. Hummel, and S. Wagner, "Challenges of the dynamic detection of functionally similar code fragments," in *Proceedings of the CSMR 2012*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 299–308.

[42] J. Ossher, H. Sajnani, and C. V. Lopes, "File cloning in open source Java projects: The good, the bad, and the ugly." in *ICSM*. IEEE, 2011.

[43] F. Long and M. Rinard, "Automatic patch generation by learning correct code," *SIGPLAN Not.*, vol. 51, no. 1, pp. 298–312, Jan. 2016.