SABER: Identifying SimilAr BEhavioR for Program Comprehension

Aditya Sridhar
Columbia University
New York, New York
aditya@cs.columbia.edu

Guanming Qiao
Columbia University
New York, New York
gq2135@columbia.edu

Gail E. Kaiser
Columbia University
New York, New York
kaiser@cs.columbia.edu

ABSTRACT

Modern software engineering practices rely on program comprehension as the most basic underlying component for improving developer productivity and software reliability. Software developers are often tasked to work with unfamiliar code in order to remove security vulnerabilities, port and refactor legacy code, and enhance software with new features desired by users. Automatic identification of *behavioral clones*, or behaviorally-similar code, is one program comprehension technique that can provide developers with assistance. The idea is to identify other code that "does the same thing" and that may be more intuitive; better documented; or familiar to the developer, to help them understand the code at hand. Unlike the detection of syntactic or structural code clones, behavioral clone detection requires executing workloads or test cases to find code that executes similarly on the same inputs. However, a key problem in behavioral clone detection that has not received adequate attention is the "preponderance of the evidence" problem, which advocates for more convincing evidence from nontrivial test case executions to gain confidence in the behavioral similarities. In other words, similar outputs for some inputs matter more than for others. We present a novel system, SABER, to address the "preponderance of the evidence" problem, for which we adapt the legal metaphor of "more likely to be true than not true" burden of proof. We develop a novel test case generation methodology with three primary dynamic analysis techniques for identifying important behavioral clones. Further, we investigate filtering and weighting schemes to guide developers toward the most convincing behavioral similarities germane to specific software engineering tasks, such as code review, debugging, and introducing new features.

KEYWORDS

Program comprehension, behavioral clones, dynamic analysis, metamorphic testing

1 INTRODUCTION

Program comprehension, or program understanding, continues to pervade today's software practices as developers attempt to acquire knowledge about existing software systems. Software engineering activities such as debugging, code refactoring, and feature enhancement require sufficient understanding of application programs; these tasks are essential for maintaining and scaling large-scale software in today's world. Many studies [36, 38, 46, 61, 90, 95] have shown that program comprehension continues to be nontrivial. Software developers repeatedly seek to understand unfamiliar codebases [38, 46]; this process is often very time-consuming [52, 61], and existing documentation is sometimes outdated or insufficient to completely aid developers' understanding of the code's functionality [1, 27, 50, 51, 93]. In industrial settings, co-workers may be reliable sources of assistance, but they may not always be available, or they may have left the company [24, 35, 53].

Software developers rarely seek to understand any part of the system in its entirety and are content to understand even single methods or classes to make necessary changes [3, 12, 42, 73, 76]. It may be helpful to present the developer with code snippets similar to her own. Code clone detection [2, 20, 34, 40] is ubiquitously used as a mechanism for identifying similarities in small pieces of software, primarily based on static pattern searches. *Static code clones* [33, 64, 66], or textually-; syntactically-; and structurally-similar code, are fairly well-understood, and identification techniques are reasonably mature. While most program analysis techniques have focused on detecting static code clones, we are interested in *behavioral clones*, or code fragments that exhibit similar behaviors and that may not look visibly similar (henceforth, we refer to behaviorally-similar or dynamically-similar code as "behavioral clones" and similar code based on syntax or structure as "static code clones"). Behavioral clone identification techniques [9, 16, 21, 29, 32, 48, 67, 78, 81] can aid developers' understanding of the functionality of the complex code fragments at hand by matching them to code from elsewhere in the same codebase or from other applications, particularly those the developer has previously worked with. The intuition is that developers may not recognize or remember where to locate behaviorally-similar code without automated assistance [65].

Prior research has studied behavioral clones by applying dynamic analysis techniques, including functional I/O similarity [21, 32, 48], similarity of execution traces [81], and similarity of concolic and dynamic symbolic executions [25, 39, 41, 45, 71, 86]. However, some works [21, 81] are overly optimistic and sometimes classify similarity based on only a single test case [80]. In contrast, SLACC [48] constructs test cases based on multi-modal grey-box fuzzing to generate inputs. SLACC reflects improvements on the approach taken in previous work and attempts to provide a systematic approach to selecting relevant test cases. Further, recent concolic execution techniques for automated test case generation have improved on their predecessors, which often suffer from scalability challenges such as path explosion and constraint solving inefficiency. Despite the advancements, all of these techniques fail to provide a rationale as to why their chosen test cases for similarity are convincing; we refer to this as the "preponderance of the evidence" problem.

"Preponderance of the evidence" is a standard used for civil claims in legal proceedings, requiring that evidence be "likely enough" (with greater than 50% probability) to prove a charge or assertion. The "preponderance of the evidence" standard is concerned

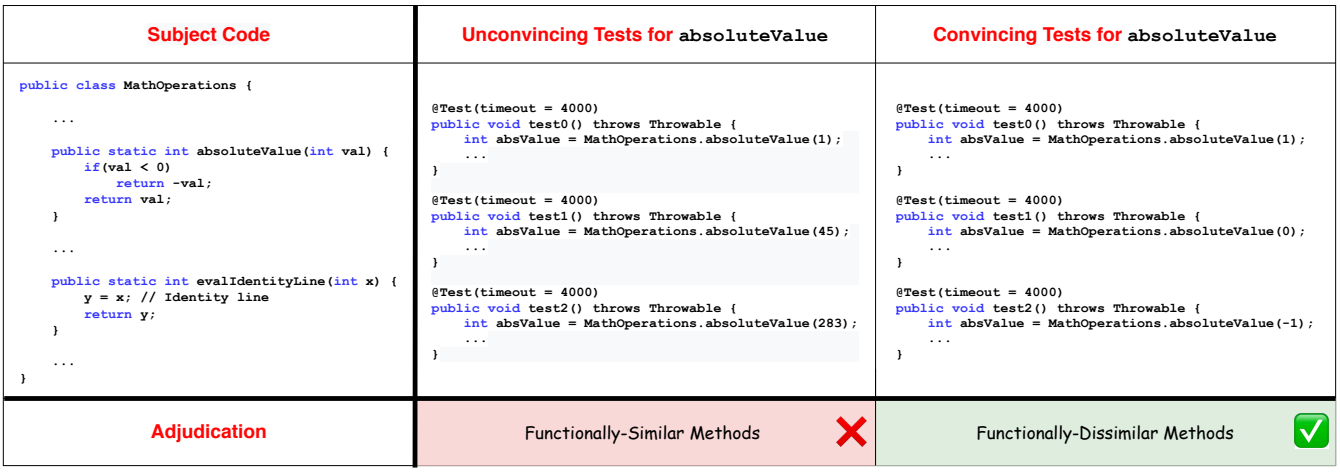

Figure 1: An example of unconvincing evidence vs. convincing evidence for method `absoluteValue`'s test executions using only functional I/O similarity to compare methods `absoluteValue` and `evalIdentityLine`.

with the quality, not the quantity, of convincing evidence. In the context of behavioral clone detection, more convincing evidence from nontrivial test case executions will lead to higher confidence on behavioral similarity classifications.

To demonstrate the difference between "convincing" and "unconvincing" evidence, Figure 1 presents a simple but nontrivial example for functional I/O similarity based on the rationale of test coverage. The set of unconvincing test cases for the method `absoluteValue` fails to address complete line and branch coverage by testing only positive integer inputs. As a result, methods `absoluteValue` and `evalIdentityLine` would be incorrectly adjudicated as functionally similar. Instead, a representative test suite considers all coverage criteria, thus including negative integer (i.e., "-1") and boundary-value (i.e., "0") inputs. The outputs differ on negative inputs, hence providing convincing evidence to rule the methods as functionally dissimilar.

The following considerations help expose and guide the search for convincing evidence that may not be captured by existing tools:

- Tests guided by code coverage criteria reach and execute all lines and branches of subject code, thus providing an elaborate and complete examination of the code prior to adjudicating similarity (as in Figure 1).
- The process of presenting evidence as convincing is expedited by actively supplying common similar-by-construction inputs to code subjects under comparison, thus constructing new "incentivized" invocations. It is intuitive that this approach is more effective than passively searching for "intrinsic" invocations containing common sets of inputs that happen to occur for some workload [21].
- Different software engineering use cases require different categories of evidence; therefore, they have different interpretations of convincing evidence (e.g., similar exception handling may be convincing for debugging but not for feature enhancement).
- The developer may favor certain test executions or specific inputs of interest for a subject method. She should have the ability to provide seed tests as "testimony" contributing to convincing evidence.
- A test suite that is convincing for the assessment of a specific definition of behavioral similarity may not be convincing for other definitions.

In comparison to the "beyond a reasonable doubt" standard, "preponderance of the evidence" is more relaxed; however, we opt for "preponderance of the evidence" over the other standard, which is much stricter. The "beyond a reasonable doubt" standard would strive for comparisons involving functional or behavioral *equality*, which is often unachievable or undecidable. "Preponderance of the evidence," on the other hand, would require only *sufficient similarity*, based on some notion of sufficiency, per engineering task.

The literature is severely limited in addressing this "preponderance of the evidence" problem. While LASSO [32] developed a coverage-based technique for test case generation in an attempt to provide convincing evidence, it only identifies functional I/O clones with the same method name and signature, thus significantly limiting the scope for finding such clones. LASSO only supports method comparisons with arguments of array, string, and primitive types. Further, LASSO fails to consider other sources of convincing evidence, including developer testimony (i.e., existing seed test workloads) and properties of methods' states. Finally, it only supports functional I/O similarity and does not profess an approach that generalizes across multiple categories of behavioral code similarity, which should be considered. These limitations undermine the argument to classify LASSO's evidence as convincing for the "preponderance of the evidence."

Selective test cases and definitions of behavioral similarity (e.g., functional I/O or execution traces) are sometimes more favorable and applicable than others, and a developer may or may not know how to characterize the behavioral similarities most germane to a specific software engineering task for improving developer productivity and software quality. For example, suppose the developer, given a test suite with known valid and faulty test cases, is

tasked with applying differential testing to find security vulnerabilities. While the developer might initially choose to find other functionally-equivalent code to compare against, it would be more prudent to find code functionally similar on just the valid test cases and functionally different on the faulty test cases. Such code pairs could better expose differences that would help indicate sources of vulnerabilities.

In addition to functional I/O, execution traces, and symbolic execution, we introduce a fourth definition of similarity known as *metamorphic similarity*. The premise of metamorphic similarity lies in a testing technique called *metamorphic testing*. Metamorphic testing was first developed by T.Y. Chen et al. [11] to solve the *test oracle problem*, which is the problem of determining the correct program outputs (or verifying that the actual outputs concur with expected outputs) for a defined set of inputs. Today, metamorphic testing is also used for program repair [30], program understanding [95], and software quality assessment [96]. Metamorphic testing involves identifying the "properties" of methods' program states; these properties are called *metamorphic properties* or *metamorphic relations*. Metamorphic similarity is distinct from functional I/O similarity in that the former is interested in the similarity among the *relationships between I/O pairs*, or metamorphic properties, instead of the similarity among the *I/O* themselves. In the previous example, presenting the developer with additional metamorphically-similar, but not metamorphically-equivalent, code may assist the developer in identifying violated method properties and localizing the vulnerabilities.

In contrast to previous clone detection approaches, we instead seek a more configurable approach that identifies behavioral clones based on a representative set of test cases, targeted for desired engineering objectives. We create a system, SABER, that significantly advances state-of-the-art techniques by identifying these "targeted" behavioral clones for four well-defined software engineering tasks involving program comprehension: *baseline program comprehension*, *debugging*, *code refactoring*, and *feature enhancement*. The system is implemented in Java. Contrary to previous techniques, SABER couples guided test case generation with three distinct dynamic analysis techniques to identify behavioral clones: functional I/O analysis, execution trace analysis, and metamorphic analysis. Automated test case generation is performed using dynamic symbolic execution and search-based software testing via the EvoSuite tool. Our system further employs filtering and weighting schemes for the "preponderance of the evidence" problem in order to choose the most convincing similarities across the engineering tasks.

A comprehensive qualitative and quantitative study conducted by Maalej et al. [46] considers different program comprehension strategies in engineering practices by software developers. Half of them directly involve searching for behaviorally-similar code, either from the same codebase or from open-source repositories. While [46] does not directly observe any program comprehension tools, it concludes that future research agendas should address context-aware tool support, a problem we propose to solve in SABER.

Our approach does not require that existing codebases include test cases to supply workloads, and no formal specifications are required. SABER is aimed at JVM-based languages such as Java; however, our methodology can be scaled and applied to most high-level statically-typed languages, as well as dynamically-typed languages.

Table 1: Software engineering use cases presented in this paper.

Use Case	Description
Baseline	Preliminary understanding of the codebase using isolated code subjects
Debugging	Detection, localization, and correction of defects in the codebase
Legacy Code Refactoring	Codebase modifications to improve its maintainability and readability
Feature Enhancement	Software upgrades and modifications

Our paper presents the following contributions:

- A novel approach to identifying behaviorally-similar code at method-level granularity
- The first exploration of the "preponderance of the evidence" problem for software engineering tasks that mandate program comprehension
- Filtering and weighting schemes to choose the most convincing behavioral code similarities
- The application of metamorphic properties for test case generation and similarity detection as part of dynamic analysis

The rest of the paper is organized as follows. §2 explains the motivation of our work across different software engineering use cases, as well as details about the use cases and other key definitions and notation used in the paper. §3 introduces test case generation techniques and definitions of behavioral code similarity supported by our system for the aforementioned software engineering tasks. §4 presents SABER, our behavioral clone detection system, in further detail based on a well-defined input generation methodology. §5 provides an analysis of the filtering and weighting schemes for targeted behavioral clones, and §6 applies the general solution framework to the specific engineering use cases introduced and motivated in §2. §7 provides the related work in behavioral clone detection, concolic execution, program comprehension, and metamorphic testing. Finally, §8 concludes the paper by summarizing our work.

2 BACKGROUND

2.1 Motivation

To emphasize the need for finding targeted behavioral clones for specific software engineering tasks involving program understanding, we present the following scenario.

Alice is a former software developer at a large technology company, BigTech. Carol is a current employee and has just recently been instructed by her boss, halfway through the development cycle, to take over a large Java project from Alice, who has left the company. Carol must enhance a set of software features for an upcoming deliverable. Although Carol is familiar with the programming language, the IDE, the compiler, and the version repository, she is unfamiliar with the project's codebase. The code is rather poorly documented and poorly written, making the task even more

cumbersome. Carol has reached out to her boss, who is also unfamiliar with the codebase. She has received little assistance from Alice and previous development teams at BigTech.

Fortunately, Carol has found some existing unit tests, which may help her to understand the relationships among different classes and methods. However, she still cannot understand the functionality of some methods in specific classes, and certain unit tests fail due to existing bugs in the codebase. It appears that Carol will have a painful experience taking over the new (for her) software project.

Since Carol is knowledgeable in proper software engineering practices, she understands the process for handling legacy code. She must begin by formulating a baseline understanding of the codebase. Next, she must fix existing bugs and then refactor the code to improve its long-run maintainability and extensibility. Finally, she must learn how large volumes of code function together in order to add a new feature. Given the unpleasant nature of the codebase documentation and presentation, Carol's task will be very time-consuming. She would definitely benefit from a tool that could aid her in her endeavor.

We present SABER as such a tool that streamlines the programming understanding process of code review, debugging, and feature extension. SABER uses dynamic analysis to find behaviorally-similar code that may be useful to developers like Carol. SABER assists with four software engineering tasks: baseline program comprehension, debugging, legacy code refactoring, and feature enhancement. These use cases are briefly described in Table 1.

2.2 Use Cases

This section provides commentary on each of the four use cases presented in SABER. Note that presenting similar code from the same codebase would be counterintuitive for all use cases, as similar perplexing code written by the same author(s) would not improve comprehension. To circumvent this issue, SABER instead searches for individual matches strewn across multiple unrelated codebases. Moreover, SABER aggregates the similarity across code obtained from different codebases. While [44] presents a code recommendation system inspired by clone detection, they search for *structurally*- or *syntactically*-similar code from other codebases. Instead, we are interested in *behaviorally*-similar code.

2.2.1 Baseline Program Comprehension. The baseline use case refers to a preliminary understanding of the codebase and is a fundamental first step in working with legacy code. Performing a line-by-line analysis of whole classes or methods with high *LOC* and numbers of invocations is cumbersome, especially if the code has poor readability. Further, new developers are often interested in comprehending only particular code fragments and methods in the context of their application rather than the whole application itself.

Hence, this use case focuses on analyzing subject methods and/or classes in isolation and presenting similar code to developers without requiring additional effort from developers. In SABER, code subjects are compared against other code from the same application, and possibly also code from other available applications. In both cases, neither finding all similar code nor obtaining the most closely-similar code is necessary for baseline comprehension.

2.2.2 Debugging. Before new developers can cater to feature requests, the code must be fully tested to ensure its correctness. However, the code may have existing bugs, and new developers may be unsure about the approach to fixing them. Developers may take advantage of *faulty test cases*, which are test cases that either pass but are incorrect or unexpectedly fail on one or more assertions. For the latter category of faulty tests, failing unit tests in Java can be exposed by executing test suites using JUnit [55], which reveals tests that contain one or more failing assertions. The former category of faulty tests are tests that themselves contain defects (e.g., incorrect assertions or outputs). SABER assumes these defects are known beforehand, are flagged by testers or previous developers, and are presented separately to the system as inputs for debugging.

Although faulty test cases provide information about the classes and methods under test and about the exceptions thrown to aid in localization of bugs, code correction requires knowledge about the intended functionality of the code, which may be unclear to the developers.

By leveraging similar code, SABER guides developers by presenting them with a "handbook" of similar code to the subject code under test based on variegated subsets of valid and faulty test case executions. Design decisions for constructing this handbook and choosing these subsets are described in §6. This approach may also enable SABER to detect security vulnerabilities in large-scale applications.

2.2.3 Legacy Code Refactoring. Legacy code is code that is outdated or difficult to maintain and extend. Although legacy code is often functional and contains unit tests for verification, software may still have poor readability and high complexity.

Improving legacy code can be time-consuming, attributed primarily to a lack of knowledge of the codebase, and has been shown to divert developers' attention away from primary workday responsibilities [50]. In particular, legacy code is often difficult to debug, quickly ages, and is brittle. Nevertheless, refactoring is paramount to improving the design, readability, extensibility, and performance of software. Thus, code refactoring conventionally succeeds debugging in the pipeline to ensure new bugs are not introduced in the working code.

For legacy code refactoring, SABER empowers developers with the ability to replace existing application code with less complex but functionally-equivalent and comprehensible code that can be stitched into the application. Because the notion of "complexity" is often subjective, we adapt our own definitions of complexity in §5.2.

2.2.4 Feature Enhancement. Extending the legacy code with new features or enhancing feature functionality are primary objectives in software development for businesses to satisfy user demand. Once legacy code for unfamiliar software applications has been repaired and refactored, new software developers must have a sufficient understanding of the code to add new features without injecting new bugs.

In contrast to previous use cases, software developers often require an understanding of multiple methods from different classes (as opposed to a single method in isolation) to add and extend features.

2.3 Definitions and Notation

For inputs and outputs, our system supports both primitive and reference data types, which are compatible with the Java Virtual Machine (JVM) [43].

SABER defines *inputs* as any data whose scope begins prior to method execution (i.e., they are defined outside of the method) and are used in the computation of any of the method's outputs. Inputs are considered live prior to entering the method body. Therefore, inputs are comprised from a subset of both the method's parameters and the method's state. *Outputs* are defined as any data computed in the method body and whose scope extends beyond the method (e.g., the data is stored in memory and accessed outside of the method). These definitions are adapted from [21]. *Test cases* are templates for executing methods and contain specifications for execution based on a set of test conditions. Test cases also house inputs that drive method invocations, and their setup construction enables simple extraction of these inputs and potential outputs for similarity analysis.

Additionally, SABER references the following notation used later in the paper. θ_{sim} is the similarity threshold for determining whether two code subjects are considered behavioral clones. $\mathbb{A}$ is the developer's application code, ρ is the set of code subjects on which the developer performs a specific software engineering task, and $\mathbb{U}$ is the list of developer specifications (including testimony) for her task at hand, if any exist. $\mathbb{C}$ is the list of targeted behavioral clones for the presented code subjects; they are derived by executing test suites Γ, generating test executions Π, and analyzing I/O profiles η (§4.1, §4.3) and execution traces $\mathbb{E}$ (§4.4). Finally, with regard to metamorphic testing (§3.1.3), $\mathbb{M}$ is the list of constructed metamorphic tests, and $\mathbb{P}$ is a map of valid metamorphic properties that exist for each code subject.

3 TEST GENERATION AND CODE SIMILARITY CLASSIFICATIONS

3.1 Test Generation Techniques

Test generation is pervasive in software development today to improve and enhance the quality of software [72]. Unfortunately, creating, understanding, and assessing meaningful test suites are time-consuming. To address this problem, we first investigate test generation techniques as a preliminary step to constructing input sequences well-suited for driving test executions. The generated tests will be instrumental for analyzing similar behaviors in the context of each software engineering use case. Previous work has employed random testing [16, 29], search-based software testing [8, 23, 26], guided testing [28, 58, 92], and symbolic/concolic execution [4, 7, 25, 39, 45, 62, 71, 86] as avenues for dynamic test input selection and generation. We present three different sources of test generation used in this paper: *pre-existing tests*, *automated tests*, and *metamorphic tests*. SABER supports unit testing with the JUnit [55] framework.

3.1.1 Pre-Existing Tests. Codebases often provide well-defined test suites to supplement their applications. Pre-existing test cases are designed by developers to provide inputs that are considered relevant to testing their code's functionality; these tests amount to developer *testimony*. Suppose a developer is tasked to work with a subject method or class in the code and corresponding pre-existing unit tests. It is natural that these unit tests should be exercised, and they can be leveraged to help guide input generation for candidate similar code. Pre-existing tests have been incorporated into workloads executed by SABER.

3.1.2 Automated Tests. SABER does not require that codebases have pre-existing tests or pre-existing workloads. There are several automated test generation tools or techniques designed for unit testing in Java.

While random testing, which generates test cases based on a random distribution, and fuzzing are often useful for detecting reliability and security problems [15, 57, 58], they usually achieve low coverage and offer limited guidance in complex object-oriented codebases. Instead we chose *coverage*-based automated test generation. This approach provides the developer with a representative set of test cases for verifying the functional correctness of her subject code. [32] has demonstrated that coverage-based test generation tools result in better precision and quality of inputs than random test generation tools.

SABER uses EvoSuite [8, 22, 23], which is a mature and readily-available tool for Java. EvoSuite combines genetic algorithms, mutation testing, and search-based mechanisms to generate test suites. Moreover, EvoSuite supports tuning of parameters for search budgets, assertions, and coverage criteria.

3.1.3 Metamorphic Tests. Metamorphic testing was first developed by T.Y. Chen et al. [11] to solve the *test oracle problem*. The test oracle problem is the problem of determining the correct outputs of a program given a defined set of inputs. Often, the assumption that a valid test oracle is available for a software system may not always hold in practice.

Metamorphic testing was introduced to alleviate this issue by leveraging *metamorphic properties*—properties of methods' program states generated by analyzing transformations to those methods' inputs—to augment the original test suite. An advantage of metamorphic properties is that for a given test case Γ_1 belonging to a method M, additional test cases can be generated from Γ_1 based on Γ_1 and M alone; metamorphic properties do not require comparisons to other methods or test cases.

Metamorphic properties can be summarized as follows:

(1) Given a program f and an input set x, its output is $f(x)$.
(2) Apply a transformation function T, derived from a defined metamorphic property for the method, on subsets of x to obtain the resultant transformed input set $T(x)$. The output with input set $T(x)$ is $f(T(x))$.
(3) Apply a check function C on the original output $f(x)$ to obtain $C(f(x))$, the predicted output. A metamorphic property holds if $f(T(x))$ can be predicted from $f(x)$, i.e., $f(T(x))$ is consistent with (but not necessarily equal to) $C(f(x))$, and the metamorphic property exists on $\{T, C\}$.

One major challenge with metamorphic testing is that metamorphic relationships among pairs of executions are not usually known *a priori* [11, 68]. To determine possible metamorphic properties, we adopt the approach taken in [79]. Using a series of well-defined metamorphic transformations on the inputs and checks on the outputs, we can verify the existence of specific metamorphic properties.

Once metamorphic properties have been identified, new test cases can be selected and constructed based on the transformed inputs used in the verification of those properties.

3.2 Code Similarity

3.2.1 Functional I/O Similarity. Functional I/O similarity is predicated on the execution of code subjects' test cases and the successive extraction and comparison of inputs and outputs. We refer to functionally-similar code as *functional I/O clones*. To be considered functional I/O clones, methods or applications must output similar values when driven with similar inputs. Unlike previous techniques that studied functional I/O similarity, SABER considers the conditions for functional I/O similarity to have been met if the similarity across executions satisfies the "preponderance of the evidence."

SABER reports functional I/O similarity even when methods have different signatures or different output vehicles, such as return values or values in static fields. Our system does not ignore situations in which outputs have different data types (including object type) or the number of outputs is different; instead, it imposes similarity penalties where applicable. SABER enforces a relaxed approach to analyzing I/O comparisons; note that we do not seek functional equality, but functional similarity, as it is often infeasible to find functional I/O clones that have completely identical I/O profiles in practice.

While such policies are relatively similar to those in [21], SABER also actively supplies common similar-by-construction inputs to code subjects under comparison instead of passively searching for "intrinsic" invocations that contain common sets of inputs. Finally, the system weighs certain executions more highly than others, hence presenting a stronger characterization of similarity.

3.2.2 Behavioral Similarity with Execution Traces. Instead of comparing source inputs and sink outputs, this definition of similarity is interested in the trace program executions across different methods. Dynamic dependency graphs are constructed in real time for each method invocation during the execution of code subjects' test cases. SABER compares graphs for a specific method invocation to those of other method invocations and their respective subgraphs as part of a subgraph isomorphism problem. As with functional I/O similarity, we are concerned with similarity and not equality; hence, exact matches are not mandatory. Execution traces are considered similar if the "distance" between the traces is below some tolerance threshold, based on some distance metric. To measure this distance, the instruction sequences from dynamic dependency graphs are first linearized into vector representations using the PageRank [59] algorithm. The Jaro-Winkler distance [13] metric is then applied on these vectors. Pieces of code that are behaviorally similar on execution traces are referred to as *executional clones*.

3.2.3 Metamorphic Similarity. Recall with functional I/O similarity that two code subjects are considered functional I/O clones if for *similar* inputs, their outputs are *similar*. Additionally, we seek to investigate the similarity among the *relationships between I/O pairs*, transcending input type differences between the code subjects.

We devise a new definition of behavioral similarity based on "known properties" of code subjects. In §3.1.3, we introduced metamorphic testing as an approach to test case generation based on

```java
// Insertion Sort
public List<Integer> insertionSort(List<Integer> list) {
   for (int i = 1; i < list.size(); i++) {
      int value = list.get(i);
      int j;
      for (j = i; j > 0 && list.get(j - 1) > value; j--)
         list.set(j, list.get(j - 1));
      list.set(j, value);
   }
   return list;
}

// Selection Sort with Value Shifts
public static Double[] selectionSortAndShift(Double[]
     values, double shiftValue) {
   for (int i = 0; i < values.length - 1; i++) {
      int index = i;
      for (int j = i + 1; j < values.length; j++)
         if (values[j] < values[index])
            index = j;
      double smallerNumber = values[index];
      values[index] = values[i];
      values[i] = smallerNumber;
   }
   for (int i = 0; i < values.length; i++)
       values[i] += shiftValue;
   return values;
}
```

Figure 2: An example of metamorphic clones detected by SABER.

derived metamorphic properties. We leverage these same properties as part of our definition of behavioral similarity by considering two code subjects as similar if they have similar sets of metamorphic properties and different otherwise; we refer to such behavioral clones as *metamorphic clones*.

Consider Figure 2, which shows two Java methods determined to be metamorphic clones by SABER. One method shows an insertion sort algorithm that accepts an `Integer` list as an input and outputs a sorted `Integer` list. The second method, on the other hand, takes in a `Double` array as input and executes a selection sort algorithm, concluding with each element in the array being shifted by the amount given by the input `shiftValue`. Note that due to the differences in data types, number of inputs, and the value shifts in the second method, these methods are unlikely to be considered similar by functional I/O analysis. Further, the sorting algorithms have different instruction sequences and execution traces; thus, executional analysis will also fail to detect the methods as executional clones. However, these methods are metamorphically similar, as the sets of metamorphic properties are similar despite differences in execution traces, syntax, or structure.

To summarize, the implications of metamorphic similarity are twofold. It provides an approach to measuring the similarity between two code subjects without requiring matching input types or output types. Metamorphic similarity may also reveal existing bugs

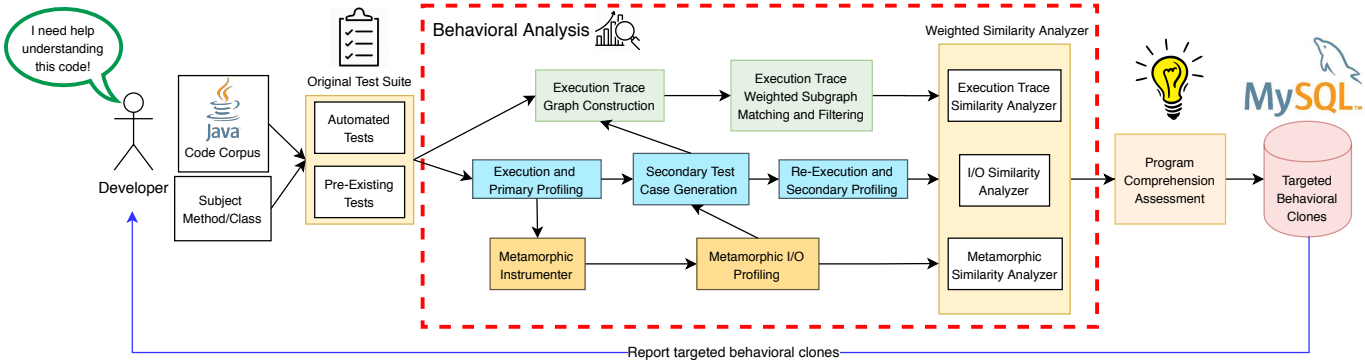

Figure 3: High-level architecture of SABER.

in the codebase and guide the developer toward a solution through the analysis of common or dissimilar metamorphic property sets.

4 SABER: BEHAVIORAL CLONE DETECTION

We create SABER as a behavioral clone detection system designed to identify "targeted" behavioral clones for different software engineering tasks based on a potpourri of test generation techniques and definitions of behavioral code similarity. Figure 3 provides a high-level overview of the different functioning components in our work, further illustrated in Algorithm 1.

SABER first constructs the original test suite from any pre-existing test cases (*line 5*) (along with user specifications, if applicable) and automated tests (*line 11*); these initial tests are referred to as *primary* tests. Next, the system performs functional I/O analysis (*lines 14-17, 22-31*) on the test suite. I/O profiles or records created (*line 16*) during functional I/O analysis are examined and exploited to generate additional metamorphic tests (*lines 18-19*), which reveal patterns in a code subject's metamorphic properties. Using the primary test cases, data type conversion techniques, and the generated metamorphic tests, the original test suite is augmented with handcrafted tests cases, labeled as *secondary* tests (*line 20*). Executional analysis (*line 22*) is performed on both the primary tests and the newly-created secondary tests. We then conduct offline similarity analysis coupled with well-guided filters and weighting schemes (detailed in §5) of the code subject's execution traces, I/O, and metamorphic properties (*lines 32-33*). Finally, we tailor these results to the specific software engineering task at hand in order to identify the most valuable behavioral clones for the developer (*lines 34-37*). Algorithm 1 will be used as a reference in §4-6.

4.1 Primary I/O Profiling

The original test suite is first constructed based on test generation techniques highlighted in §3.1 (*lines 4-12*). Using the approach previously developed in HitoshiIO [21] to leverage the definitions of inputs and outputs from §2.3, we perform static analysis on the complete application code and insert bytecode instrumentation with Java's ASM bytecode manipulation library [6] to identify and record method inputs and outputs at runtime (*line 13*).

Our system then executes the modified test suites (*line 14*) to generate I/O profiles (*lines 16-17*) from the executions. The profiles are serialized to XML using the XStream library [14] for Java. These I/O profiles are classified as *primary* I/O profiles, distinct from the handcrafted I/O profiles generated later in the methodology.

4.2 Metamorphic Profiling

In this stage of SABER, primary I/O profiles serve as templates for constructing new tests based on individual methods' metamorphic properties (Alg 1, *lines 18-19*). Algorithm 2 describes our approach to generating metamorphic profiles, extracting likely metamorphic properties, and constructing reliable metamorphic tests. Metamorphic transformers (*line 2*) and checkers (*line 3*) are defined as in [79] and utilized as per the approach highlighted in §3.1.3.

- **Transformers**: SABER uses 11 well-defined input transformers for inputs of both primitive and reference data types. Primary transformer categories include adder, multiplier, reverser, shuffler, negator, inclusion, and exclusion.
- **Property Verifiers/Checkers**: similar to the case of input transformers, SABER provides 17 well-defined checkers to be applied on the original and metamorphic profiles in verifying metamorphic properties. The main classifications of checkers include additive, multiplicative, identity, shufflable, negatable, invertible, inclusive, exclusive, and correlative.

Once inputs $\mathbb{I}$ have been extracted (*line 5*) from the primary I/O profiles η, transformations are applied on all subsets of $\mathbb{I}$ to form the transformed set $\mathbb{I}^T$ (*line 7*). The primary I/O profiles are transformed into profiles η^T by replacing the existing inputs with the respective transformed inputs (*line 8*). Test cases Γ^T are constructed from these transformed profiles via the technique explained in §4.3 (*line 9*); these tests only represent "possible" metamorphic tests. Next, the Γ^T tests are executed as in §4.1 (*line 10*), and metamorphic profiles η^T_{out} (*line 11*) are constructed from the executed tests. Thereafter, for each metamorphic I/O profile (*line 12*) and corresponding primary I/O profile, checkers verify whether metamorphic properties hold for that transformer-checker pair (*line 13*). If a metamorphic property exists and the profile belongs to a developer-specified subject method found in ρ (*line 14*), the property's existence is reported (*line 15*). Moreover, the metamorphic test from which the metamorphic profile was derived is flagged as a valid metamorphic test for the subject method (*line 17*). This process continues until all profiles have been analyzed.

Algorithm 1 Solution Overview of SABER

Input: Developer application code $\mathbb{A}$, a set of code subjects ρ, and any developer specifications $\mathbb{U}$ (including use case type)

Output: A list $\mathbb{C}$ of targeted behavioral clones for each code subject

1: $\Omega_{orig} \leftarrow$ SetUpRepos($\mathbb{A}$);
2: $TYPE \leftarrow$ GetUseCaseType($\mathbb{U}$);
3: $\{\Gamma_{orig}, \eta_{orig}, \mathbb{M}, \eta_{new}, FT\} \leftarrow \varnothing$;
4: **for** $\Omega_i \in \Omega_{orig}\ \forall i \in \{1, \ldots, |\Omega_{orig}|\}$ **do**
5: $\quad\Gamma_{E,i} \leftarrow$ ExtractExistingTests(Ω_i);
6: $\quad$**if** $TYPE$ = "debug" **then**
7: $\quad\quad FT \leftarrow$ ExtractFaultyTests($\Gamma_{E,i}$);
8: $\quad\quad VT \leftarrow$ ExtractValidTests($\Gamma_{E,i}$);
9: $\quad\quad\Gamma_{orig} \leftarrow \Gamma_{orig} \cup VT$;
10: $\quad$**else**
11: $\quad\quad\Gamma_{A,i} \leftarrow$ GenerateAutomatedTests(Ω_i);
12: $\quad\quad\Gamma_{orig} \leftarrow \Gamma_{orig} \cup (\Gamma_{E,i} \cup \Gamma_{A,i})$;
13: $\Gamma^{mod}_{orig} \leftarrow$ PreAnalyzeAndCompile(Γ_{orig});
14: $\Pi_{orig} \leftarrow$ ExecuteTests(Γ^{mod}_{orig});
15: **for** $\Pi_i \in \Pi_{orig}\ \forall i \in \{1, \ldots, |\Pi_{orig}|\}$ **do**
16: $\quad\eta_{orig,i} \leftarrow$ GenerateTypeAIOProfile($\Gamma^{mod}_{orig,i}$, Π_i);
17: $\quad\eta_{orig} \leftarrow \eta_{orig} \cup \eta_{orig,i}$;
18: $\quad\mathbb{M}_i \leftarrow$ GenerateMetamorphicTests($\eta_{orig,i}$);
19: $\quad\mathbb{M} \leftarrow \mathbb{M} \cup \mathbb{M}_i$;
20: $\Gamma_{new} \leftarrow$ ConstructTypeBTests(Γ_{orig}, η_{orig}, $\mathbb{M}$, ρ);
21: $\mathbb{E} \leftarrow$ GenerateExecutionTraces(Γ_{orig}, Γ_{new});
22: $\Pi_{new} \leftarrow$ ExecuteTests(Γ_{new});
23: $\mathbb{O} \leftarrow$ ConstructProfileOrganizers(Π_{new});
24: **for** $\Pi_i \in \Pi_{new}\ \forall i \in \{1, \ldots, |\Pi_{new}|\}$ **do**
25: $\quad\eta_{new,i} \leftarrow$ GenerateTypeBIOProfile($\Gamma_{new,i}$, Π_i);
26: $\quad O \leftarrow$ GetProfileOrganizer($\mathbb{O}$, $\eta_{new,i}$);
27: $\quad$**if** IsAlphaProfile($\eta_{new,i}$, ρ) **then**
28: $\quad\quad$AddToAlphaProfilePool(O, $\eta_{new,i}$, ρ);
29: $\quad$**else if** IsBetaProfile($\eta_{new,i}$, ρ) and $TYPE \neq$ "debug" **then**
30: $\quad\quad$AddToBetaProfilePool(O, $\eta_{new,i}$, ρ);
31: $\quad\eta_{new} \leftarrow \eta_{new} \cup \eta_{new,i}$;
32: $SIM \leftarrow$ ComputeAllWeightedSimilarities($\mathbb{O}$, $\mathbb{E}$, $\mathbb{M}$, ρ);
33: $COM \leftarrow$ ComputeAllComplexities(ρ, ρ');
34: $\mathbb{C} \leftarrow$ FindTargetedBehavioralClones($TYPE$, ρ, SIM, COM);
35: **if** $TYPE$ = "debug" **then**
36: $\quad$ApplyClonesOnFaultyTests($\mathbb{C}$, FT);
37: **return** $\mathbb{C}$;

As the number of classes, methods, and primary I/O profiles increases, the number of metamorphic tests becomes increasingly large. To reduce the number of metamorphic tests, we implement a top-k heuristic that searches for the "strongest" metamorphic tests for each code subject. Metamorphic tests are grouped by code subject (*line 18*) and ranked in decreasing order of the number of metamorphic properties revealed by those tests. The top k tests are selected per code subject (*line 20*) as representatives in the final set of metamorphic tests (*line 21*), with k chosen from a fixed range of values.

Algorithm 2 Metamorphic Profiling

Input: Primary I/O profiles η and a set of code subjects ρ

Output: Metamorphic tests $\mathbb{M}$ and properties $\mathbb{P}$

1: $\{\mathbb{M}, \mathbb{M}_{init}\} \leftarrow \varnothing$;
2: $T \leftarrow$ SetUpTransformers();
3: $V \leftarrow$ SetUpCheckers();
4: **for** $\eta_i \in \eta\ \forall i \in \{1, \ldots, |\eta|\}$ **do**
5: $\quad\mathbb{I} \leftarrow$ GetInputs(η_i);
6: $\quad$**for** $T_j \in T\ \forall j \in \{1, \ldots, |T|\}$ **do**
7: $\quad\quad\mathbb{I}^T_j \leftarrow$ TransformInputs(T_j, $\mathbb{I}$);
8: $\quad\quad\eta^T_{i,j} \leftarrow$ TransformProfiles(η_i, $\mathbb{I}^T_j$);
9: $\quad\quad\Gamma^T_{i,j} \leftarrow$ SetUpPossiblyValidMetTests(η_i, $\eta^T_{i,j}$);
10: $\quad\quad\Pi_{i,j} \leftarrow$ ExecuteTests($\Gamma^T_{i,j}$);
11: $\quad\quad\eta^T_{i,j,out} \leftarrow$ CreateMetamorphicProfiles($\Gamma^T_{i,j}$, $\Pi_{i,j}$);
12: $\quad\quad$**for** $V_k \in V\ \forall k \in \{1, \ldots, |V|\}$ **do**
13: $\quad\quad\quad propExists \leftarrow$ VerifyProperty(V_k, η_i, $\eta^T_{i,j,out}$);
14: $\quad\quad\quad$**if** $propExists$ and $\eta_i.member \in \rho$ **then**
15: $\quad\quad\quad\quad\mathbb{P}[\eta^T_{i,j,out}, V_k]$ = true;
16: $\quad\quad\quad\quad$**if** $\eta^T_{i,j,out} \notin \mathbb{M}_{init}$ **then**
17: $\quad\quad\quad\quad\quad\mathbb{M}_{init} \leftarrow \mathbb{M}_{init} \cup \eta^T_{i,j,out}$;
18: $G \leftarrow$ GroupByCodeSubjects($\mathbb{M}_{init}$, ρ);
19: **for** $G_i \in G\ \forall i \in \{1, \ldots, |G|\}$ **do**
20: $\quad G_{i,k} \leftarrow$ SelectTopKMetTests(G_i);
21: $\quad\mathbb{M} \leftarrow \mathbb{M} \cup G_{i,k}$;
22: **return** $\{\mathbb{M}, \mathbb{P}\}$;

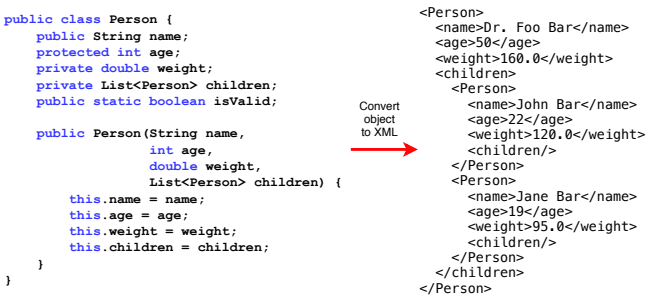

Figure 4: XML serialization of an object instantiated from a sample Java class.

4.3 Secondary Test Case Generation and I/O Profiling

To overcome the challenge of candidate methods never being invoked with similar inputs during available workloads, such as pre-existing test cases, SABER also directly supplies common inputs to both the method of interest and other methods that potentially behave similarly. In this section, we describe how inputs are created and used in test case generation.

4.3.1 Input Type Conversion. For a method M_1 to be compared to a method M_2, we must supply M_1's inputs to M_2 and M_2's inputs to M_1. However, methods often have variegated signatures, which

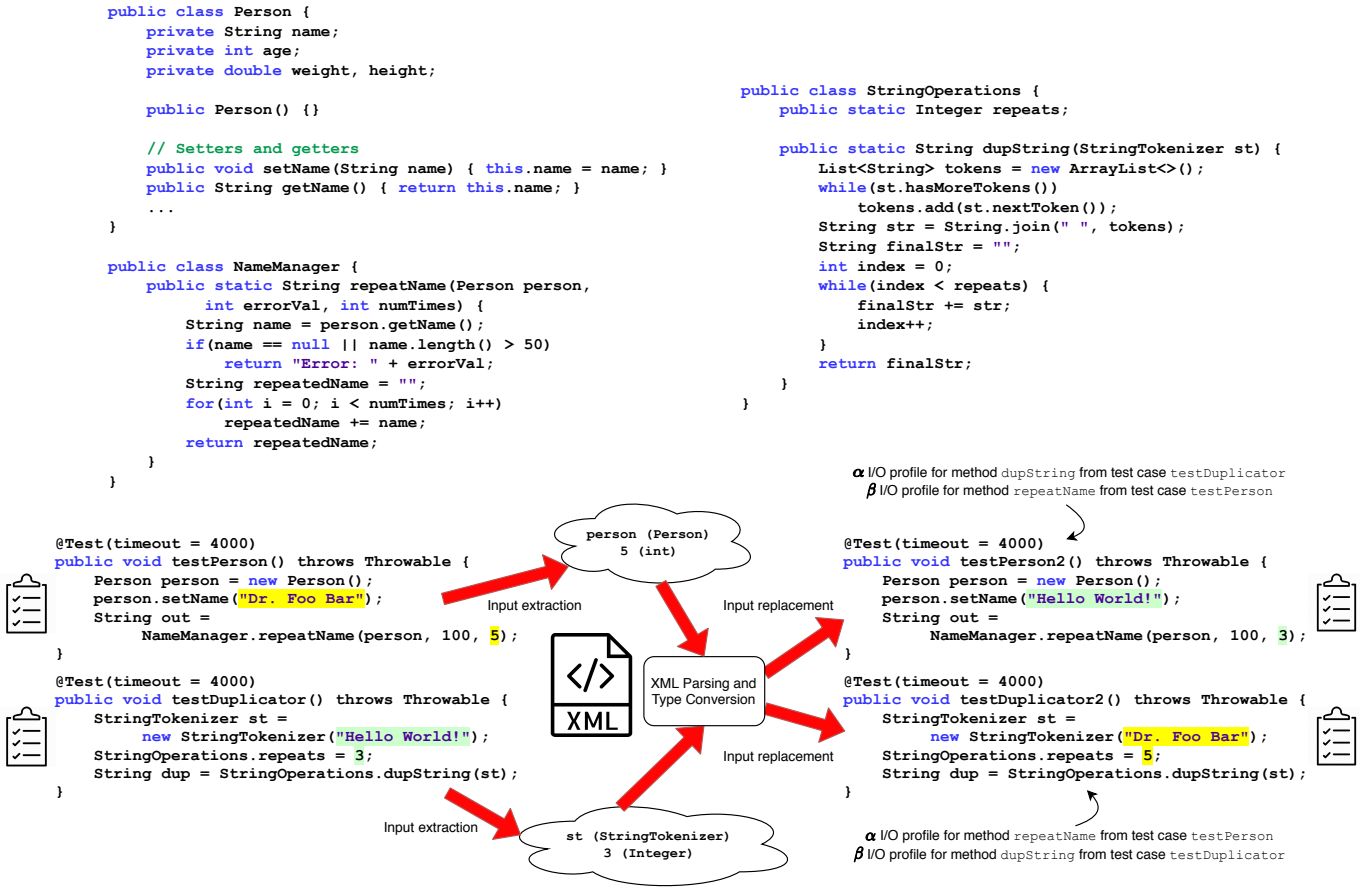

Figure 5: An example of secondary test case generation performed by SABER.

result in different argument types and counts. This observation can be extended to *input state*, which is not limited to method arguments. Unlike most of the previous techniques, SABER supports conversion of data types across methods with not only different signatures, but also different input types and counts. Further, SABER supports all JVM primitive and reference data types.

To convert data types across methods, we implement a type conversion adapter that supports conversion across primitive data types, strings, objects (which are enclosed in wrappers), multi-dimensional arrays, collections, and file types based on a comprehensive set of existing converters. Comparing two methods requires comparing their primary I/O profiles from §4.1 and selected metamorphic profiles from §4.2, which are serialized using XML. All permutations of input orders are considered. For each permutation, XML input elements are recursively dissected, individually manipulated, and stitched together to form new I/O profiles. During input type conversion for a specific input order, if a type conversion for any individual input in the order cannot be supported, then SABER drops the input order and proceeds to the next one. This policy helps to limit the number of test cases generated in §4.3.2. The type conversion process continues until all input orders have been considered for a method pair.

One advantage of our approach to serialization is our ability to manipulate both private and non-private members of classes, as input state variables may be private or non-private. An example of the flexibility of XML serialization is shown in Figure 4 (object creation is omitted for brevity). Public, private, and protected access modifiers can be serialized in XML; hence, SABER supports the manipulation of all members with these modifiers. Since static variables (e.g., the `isValid` field) belong to the class in Java, they cannot be serialized by XStream.

Similar to SLACC [48], SABER can handle objects whose members are initialized via constructors. However, SABER also supports conversion of objects that may not be initialized via their constructors; this is due to the flexible structure of test cases and accessibility to input values from serialized profiles. To compare object types across objects with the same number of fields, objects are internally manipulated by pairwise comparison of those fields. Since individual fields may have object types themselves, objects are recursively examined until all serialized entities have been converted.

4.3.2 Test Case Generation. To generate new test cases (Alg. 1, *line 20*), we implement a searching technique to replace inputs from the test suite of both the original Γ_{orig} and metamorphic $\mathbb{M}$ test cases with the new type-converted inputs. Pre-instrumentation monitors

are inserted in the test suite to track runtime values around variable declarations and literal expressions. Based on the runtime values, variables and literals from the primary test cases are mutated and then loaded into new *secondary* test cases. Since complex inputs such as objects often require setup code, the values are loaded from a MySQL database and extracted from the result set.

Figure 5 demonstrates a simple example of secondary test case generation. Note that SABER is able to handle object inputs (e.g., `person`, which is of type `Person`) and file type inputs (e.g., `st`, which is of type `StringTokenizer`). Objects can be initialized through means other than their constructors (e.g., the `Person` class only has a default constructor, but the `name` field can be initialized via a setter method, as in `testPerson`). Method arguments may not necessarily be inputs, as in the case of `testPerson`, where `errorVal = 100` is unused in the specific method invocation to `repeatName`. Further, the example underscores the flexibility of state variables as inputs by not requiring inputs to be passed in via method arguments (e.g., "3" in `testDuplicator`). Finally, for demonstration purposes, note that the setup code for the `person` object in `testPerson` conveniently contains the value of the field (i.e., `name`) to be modified during input replacement. However, SABER can cleverly modify or replace the values of the "implicit" object fields—`age`, `weight`, and `height`—via serialization if their values happen to influence the outputs of any method, even if their original values are not initialized in the test case.

4.3.3 I/O Profiling. The secondary tests are executed (Alg. 1, *line 22*), and secondary I/O profiles are generated (Alg. 1, *line 25*) for analysis. For developer-specified subject methods of interest, these secondary I/O profiles are organized (Alg. 1, *line 23*) into two categories: α and β. These categories may be influenced by test cases the developer herself elects to be "more important" or "less important" for her software engineering task at hand; however, as noted in §1, SABER does not require such test cases and does not assume the developer knows how to characterize the importance of test cases. α I/O profiles (Alg. 1, *lines 27-28*) for a method M_1 are profiles from *highly-relevant* tests cases that represent the bodies of other methods M_2 but with M_1's inputs driven to M_2. Conversely, β I/O profiles (Alg. 1, *lines 29-30*) may be either: 1) profiles containing M_1's method body and inputs from other methods M_2, or 2) profiles from *less-relevant* tests cases containing the bodies of other methods M_2 and inputs from M_1. Priority is given to α I/O profiles, since we anticipate that they are involved in comparisons more useful to the developer. In §5, this bifurcation of secondary I/O profiles will be instrumental for characterizing similarity.

To increase the clarity of the "secondary" division, Figure 5 shows the α and β I/O profiles that would be generated from the secondary test cases for the previous example. The yellow and green highlights demonstrate the swapping of input values between (`testPerson`, `testDuplicator`) and (`testPerson2`, `testDuplicator2`), respectively.

4.4 Execution Tracing

SABER adopts the approach in DyCLINK [81] to generate traces of program executions using ASM (*line 22*). Dynamic dependency graphs are constructed from bytecode instructions, and link analysis matches dependency subgraphs based on approximations of isomorphism between subgraphs. Filtering of the "best" subgraph matches is implemented using the PageRank algorithm [59] to identify centroid instructions. Despite this filtering mechanism, DyCLINK is inefficient in its code similarity detection for "targeted" code subjects.

We apply subgraph isomorphism on the specific code subjects using method and class separation techniques. Method invocations are collected from both the primary tests and the secondary tests, thus providing a comprehensive workload for SABER's executional analysis. Within individual code subjects, invocations of other methods are also examined as part of subgraph comparison. Additionally, we modify the PageRank algorithm to filter important instructions, i.e., the graph centroids, targeted at Java API calls as opposed to arbitrary `load` and `store` bytecode instructions as in DyCLINK. Capturing interesting behavior is accomplished by prioritizing bytecode instructions such as `invokevirtual` and `invokestatic`. Overall, our approach minimizes the subgraph comparison runtime when limiting the scope to developer-specified code subjects.

4.5 Behavioral Clone Extraction

To find targeted behavioral clones (Alg. 1, *line 34*), filtering and weighting schemes are introduced in §5, and customizations of the software engineering use cases are described in §6.

We first describe the notation (Def. 4.1) and properties (Def. 4.2) of behavioral clones.

Definition 4.1. Let VT represent the set of valid test cases and FT the set of faulty test cases in the test suite. $\mathbb{C}^{\lambda_S}_{\lambda_E}$ is the set of behavioral clones whose similarity depends on inputs from tests designated by λ_S, based on executing only tests designated by λ_E (where $\lambda_E \in \{VT, FT, VT \vee FT\}$). The possible values of λ_S are defined as follows:

- VT**:** clones are similar on valid tests' inputs and may be similar or dissimilar on faulty tests' inputs
- FT**:** clones are similar on faulty tests' inputs and may be similar or dissimilar on valid tests' inputs
- $VT \wedge FT$**:** clones are similar on both valid tests' and faulty tests' inputs
- $\neg VT \wedge FT$**:** clones are similar on faulty tests' inputs and dissimilar on valid tests' inputs
- $VT \wedge \neg FT$**:** clones are similar on valid tests' inputs and dissimilar on faulty tests' inputs
- $\neg VT \wedge \neg FT$**:** clones are dissimilar on both valid tests' and faulty tests' inputs

Def. 4.2 provides a list of important behavioral clone properties and algebra useful in the analysis of engineering use cases investigated in §6.

Definition 4.2 (Properties of Behavioral Clones). Given $\mathbb{C}^{\#}_{\lambda_E}$ is the complete behavioral clone space obtained by executing tests λ_E, $\vee$ is the logical OR operator, and $\wedge$ is the logical AND operator, the following are properties of behavioral clones:

(1) $\mathbb{C}^{\lambda_S}_{\lambda_E} \wedge \mathbb{C}^{\lambda_S}_{\lambda_E} = \mathbb{C}^{\lambda_S}_{\lambda_E}$ and $\mathbb{C}^{\lambda_S}_{\lambda_E} \vee \mathbb{C}^{\lambda_S}_{\lambda_E} = \mathbb{C}^{\lambda_S}_{\lambda_E}$ (Idempotent Rule): the intersection or union of a set of behavioral clones with itself is the same set of clones

(2) $\neg\mathbb{C}_{\lambda_E}^{\lambda_S} = \mathbb{C}_{\lambda_E}^{\neg\lambda_S}$ (Negation Rule): the complement of a set of behavioral clones similar on inputs from tests designated by λ_S, or all other behavioral clones in the behavioral clone space, is the set of behavioral clones similar on inputs from tests not designated by λ_S

(3) $\mathbb{C}_{\lambda_E}^{VT} \vee \mathbb{C}_{\lambda_E}^{FT} = \mathbb{C}_{\lambda_E}^{\#}$ (Clone Sum Rule): the union of the set of behavioral clones similar on valid tests' inputs and the set of behavioral clones similar on faulty tests' inputs is the complete behavioral clone space

(4) $\mathbb{C}_{\lambda_E}^{\lambda_{S_1}} \wedge \mathbb{C}_{\lambda_E}^{\lambda_{S_2}} = \mathbb{C}_{\lambda_E}^{\lambda_{S_1}\wedge\lambda_{S_2}}$ (Similarity Intersection Rule): the intersection of the set of behavioral clones similar on inputs from tests designated by λ_{S_1} and the set of behavioral clones similar on inputs from tests designated by λ_{S_2} is the set of behavioral clones similar on inputs from the intersection of tests designated by λ_{S_1} and tests designated by λ_{S_2}

(5) $\neg\mathbb{C}_{\lambda_E}^{\lambda_{S_1}} \wedge \mathbb{C}_{\lambda_E}^{\lambda_{S_2}} = \mathbb{C}_{\lambda_{\lambda_E}}^{\neg\lambda_{S_1}\wedge\lambda_{S_2}}$ and $\mathbb{C}_{\lambda_E}^{\lambda_{S_1}} \wedge \neg\mathbb{C}_{\lambda_E}^{\lambda_{S_2}} = \mathbb{C}_{\lambda_E}^{\lambda_{S_1}\wedge\neg\lambda_{S_2}}$ (Composite Rule #1): this property follows from the negation and similarity intersection rules

(6) $\mathbb{C}_{\lambda_E}^{\neg VT\wedge FT} = \mathbb{C}_{\lambda_E}^{\neg VT}$ and $\mathbb{C}_{\lambda_E}^{VT\wedge\neg FT} = \mathbb{C}_{\lambda_E}^{\neg FT}$ (Composite Rule #2): this property follows from the negation and clone sum rules

5 WEIGHTED SIMILARITY COMPUTATION AND RANKING

5.1 Similarity Analysis

Recall that our objective is to find behavioral clones targeted to a given software engineering task. In §3.2, we presented different definitions of behavioral similarity. We now apply these definitions in our work to identify *sufficient* similarity for the task at hand. It is difficult to classify similarity as a binary concept, as either "yes" or "no." We adapt the preponderance of the evidence metaphor to find convincing code matches by investigating filters and weighting schemes for assessing similarity.

Previously, we established the two classifications of secondary I/O profiles that we construct for characterizing similarity: α and β. We propose that α I/O profiles for a particular subject method of interest are more important for similarity than β I/O profiles because the driving inputs in the α I/O profiles are the subject method's own highly-relevant test cases. Consequently, β I/O profiles may be similar to the α I/O profiles but on less-relevant test cases, or they may instead represent inputs of "foreign" methods' invocations applied to our subject method. The latter type of β I/O profiles may be understood as more germane to the respective foreign methods than to the subject method. Thus, we seek to construct a weighting scheme that places more emphasis on the former comparisons than on the latter.

We introduce the following definitions to facilitate the construction of the weighting scheme. Our goal is to aggregate the similarity among comparisons between method invocations. Previous dynamic analysis techniques [21, 48, 81] did not differentiate among convincing and unconvincing comparisons, which is crucial for preponderance of the evidence.

5.1.1 I/O Similarity. We first consider I/O similarity separately.

Definition 5.1. Let Φ represent a function that measures the I/O similarity between two methods. Suppose we have two methods M_1 (with I/O profiles of the form η^{M_1}) and M_2 (with I/O profiles of the form η^{M_2}). Then, $\Phi(\eta^{M_1}, \alpha^{M_1,M_2})$ is the similarity for an M_1 α I/O profile comparison to M_2, and $\Phi(\eta^{M_2}, \beta^{M_1,M_2})$ is the similarity for an M_1 β I/O profile comparison to M_2.

We compute Φ for two methods M_1 and M_2 based on the exponential model leveraged in [21]. The exponential model applies penalties on invocation pairs when either or both of their respective input sets and their respective output sets are dissimilar. Higher values correspond to greater I/O similarity between the methods. We set the locally-optimized value of the model's parameter setting to be 3, as was experimentally determined in [21]. Input set or output set similarity is calculated using a Jaccard coefficient. For object types, we apply the following approaches:

- *DeepHash*: we adopt the approach used in HitoshiIO [21] to recursively aggregate Java hashcodes of object fields. Two objects are considered similar if they have matching hash values.
- Ancestral Walk: as a deviation of disjoint sets, we measure the distance between two objects as follows. We represent the root *anc* of an inheritance hierarchy tree T as the closest common ancestor of the objects' classes C_1 and C_2, and all subclasses are represented as children nodes of their respective superclasses. Let class C_{max} represent the class whose node has greater depth in T. Further, let $d_{anc,o}$ represent the distance from *anc* to Java's `Object` class and $d_{c_{max},o}$ represent the distance from C_{max} to `Object`. The ancestral walk factor (AWF) is computed as the ratio between $d_{anc,o}$ and $d_{c_{max},o}$. Larger values of AWF imply closer relation between the objects' classes.

Using the similarity function Φ from Definition 5.1, we measure the "sufficiency" of similarity based on the relevance of certain comparisons, given by specific weight factors, or *relevance factors*.

Definition 5.2. Suppose M_1 is a subject method and M_2 is a candidate match. Let $\alpha_i^{M_1,M_2}$ and $\beta_j^{M_1,M_2}$ represent the i-th α I/O profile and the j-th β I/O profile, respectively, for method M_1 in the comparison between methods M_1 and M_2. Finally, let ζ_1 and ζ_2 be the relevance factors applied to the α and β I/O profile comparisons, respectively. The *total I/O relevance similarity* $\Psi_{I/O,(M_1,M_2)}$ between M_1 and M_2 is computed as:

$$\Psi_{I/O,(M_1,M_2)} = \frac{\sum_i \zeta_1 \Phi(\eta_i^{M_1}, \alpha_i^{M_1,M_2}) + \sum_j \zeta_2 \Phi(\eta_j^{M_2}, \beta_j^{M_1,M_2})}{\sum_i \zeta_1 + \sum_j \zeta_2} \quad (1)$$

It follows from Definition 5.2 that the total I/O relevance similarity between two methods will range between 0 and 1.

The default values of ζ_1 and ζ_2 are 2 and 1, respectively. Note that multiplying both relevance factors by a constant does not change $\Psi_{I/O}$, so we can simply normalize them. Instead, we analyze the *relevance ratio*.

Definition 5.3. The *relevance ratio*, ζ_r, is calculated as:

$$\zeta_r = \begin{cases} \frac{\zeta_1 - \zeta_2}{\zeta_1} & \zeta_1 \neq 0, \zeta_1 \geq \zeta_2 \\ 0 & \zeta_1 = 0 \end{cases} \quad (2)$$

As a consequence, larger values of the relevance ratio translate to greater importance placed on comparisons involving the subject method's own inputs applied to other methods.

5.1.2 Weighted Similarity. In addition to I/O similarity, we compute the similarity functions for metamorphic similarity and executional similarity. Metamorphic similarity between two methods is calculated as the Jaccard coefficient between the sets of metamorphic properties for both methods. Similarly, the similarity of execution traces between two methods is calculated by applying the Jaro-Winkler distance between the dynamic dependency graph PageRank vectors as in DyCLINK [81].

Given our definitions of behavioral similarity, we compute the *total relevance similarity* Ψ, which is the weighted aggregate of the different definitions of behavioral similarity (Alg. 1, *line 32*).

Definition 5.4. Let μ_1 be the weight factor for functional I/O similarity, μ_2 be the weight factor for executional similarity, and μ_3 be the weight factor for metamorphic similarity. Then, $\Psi_{(M_1,M_2)}$ is computed as:

$$\Psi_{(M_1,M_2)} = \begin{bmatrix} \mu_1 & \mu_2 & \mu_3 \end{bmatrix} \begin{bmatrix} \Psi_{I/O,(M_1,M_2)} \\ \Psi_{extr,(M_1,M_2)} \\ \Psi_{met,(M_1,M_2)} \end{bmatrix} \ni \sum_{i=1}^{3} \mu_i = 1 \quad (3)$$

Methods M_1 and M_2 are considered behavioral clones if their total relevance similarity $\Psi_{(M_1,M_2)}$ meets or exceeds the similarity threshold θ_{sim}.

For class similarity between two classes, we consider the individual methods between those classes. The similarity between two classes can be summarized as the ratio of the number of "similar" method pairs to the "total" number of method pairs. Formally, class similarity can be computed using graph theory, as follows.

Definition 5.5. Suppose G is a bipartite graph between two classes C_1 and C_2 such that all methods of C_1 are in one set U and all methods of C_2 are in another set V. Let an edge connect a node u in U for method M_u with a node v in V for method M_v if M_u and M_v are considered behavioral clones, i.e., their total relevance similarity meets or exceeds the similarity threshold, θ_{sim}. Further, suppose G' is a complete bipartite graph for the same nodes in G. Then, the total class similarity is the ratio of the number of edges in G to the number of edges in G'.

5.2 Complexity Analysis

For certain use cases, we further perform ranking of behavioral clones. The intuition is that the developer will be best served by behavioral clones less "complex" than the current code, based on some defined complexity metric (Alg. 1, *line 33*).

Hence, we define method complexity using the *cyclomatic complexity* metric [88], developed by T.J. McCabe. Using control flow graphs (CFGs), cyclomatic complexity quantifies the decision logic of a single software module, thus serving as a measure of a program's or method's code complexity. We present the formal definition of cyclomatic complexity in Definition 5.6.

Definition 5.6. Let E and V be the number of edges and vertices, respectively, in the CFG representation of a method M. Then, the cyclomatic complexity δ_M of method M is computed as:

$$\delta_M = E - V + 2 \quad (4)$$

Similarly, we define complexity for a full class based on the *Lack of Cohesion in Methods* (LCOM) [60] metric. LCOM measures the correlation between methods and class instance variables. Lower values of LCOM indicate higher class cohesion and lower complexity. We implement a definition of LCOM known as *Pairwise Field Irrelation* (PFI).

Definition 5.7. For a class C with n fields, let $R_C(F_i)$ represent the set of methods that access the field F_i. Further, let $J_D(R_C(F_i), R_C(F_j))$ be the Jaccard distance between two distinct sets of methods accessing fields F_i and F_j, respectively. Then, the PFI complexity δ_C of class C is the mean Jaccard distance across all pairs of fields and is computed as:

$$\delta_C = 2 \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{J_D(R_C(F_i), R_C(F_j))}{n(n-1)} \quad (5)$$

6 SOFTWARE ENGINEERING USE CASES

In this section, we investigate targeted behavioral clones for our four software engineering use cases (Alg. 1, *lines 34-36*).

6.1 Baseline Program Comprehension

Baseline understanding of programs adopts a straightforward model of the implementation presented in §4. Since the developer is only interested in understanding basic methods in isolation during this phase, arbitrary values of ζ_r, μ_1, μ_2, and μ_3 may be chosen (by default, $\zeta_r = 0.5$ and $\mu_1 = \mu_2 = \mu_3 = 0.\overline{3}$). It is preferable that both α and β I/O profiles be analyzed to decrease the likelihood of false positives. Similarly, metamorphic clones may aid the developer in understanding the properties of her own code subjects in the context of other application code.

6.2 Debugging

For debugging, developers need to identify flaws in the code that break intended functionality. An example of such a scenario is shown in Figure 6. `computeJaccardSimilarity` computes the Jaccard similarity between two lists of objects. In the example, three test cases—`test0`, `test1`, and `test2` (`test1` and `test2` are not shown)—are created for testing the method. In the ideal version of the method, a special check would be performed when the two lists have no elements. All three test cases would then pass with the original version. However, after injecting a mutant that removes the check, the first test case fails because the Jaccard similarity between two empty lists is incorrect in the mutated (buggy) version.

In order to aid the developer in fixing this bug, SABER divides the test suite into faulty test cases (Alg. 1, *line 7*)—`test0`—and valid

```
/* Jaccard Similarity Without Mutant */
public double computeJaccardSimilarity(List<Object> list1,
     List<Object> list2) {
   Set<Object> unionSet = new HashSet<>(list1);
   unionSet.addAll(list2);
   int union = unionSet.size();
   int intersection = list1.size() + list2.size() - union;
   double jaccardSimilarity = (union == 0) ? 0 :
        intersection * 1.0 / union;
   return jaccardSimilarity;
}

/* Jaccard Similarity With Mutant */
public double computeJaccardSimilarity(List<Object> list1,
     List<Object> list2) {
   ...
   // Mutant injected here
   double jaccardSimilarity = intersection * 1.0 / union;
   return jaccardSimilarity;
}

/* Test Cases */
@Test(timeout = 4000)
public void test0() throws Throwable {
    Jaccard jaccard0 = new Jaccard();
    List<Object> list0 = new ArrayList<>();
    List<Object> list1 = new ArrayList<>();
    double double0 =
         jaccard0.computeJaccardSimilarity(list0, list1);
    // Note: this assertion fails for the second method!
    assertEquals(0.0, double0, 0.01);
}
...
```

Figure 6: An example of the debugging use case. The presented test case passes for the original method, but it fails after the mutant is injected.

test cases (Alg. 1, *line 8*)—`test1` and `test2`. We proceed to perform at least one of following sets of executions on our test cases:

(1) Executing only valid test cases: behavioral clones are identified using only the valid tests as the test suite.
(2) Executing both valid and faulty test cases: just as with the baseline understanding use case, behavioral clones are identified by including all tests from our original test suite.

If the developer has a valid test oracle for inputs from the faulty test cases (e.g., via assertion statements), she may choose only the former set of executions to obtain candidate behavioral clones $\mathbb{C}^{VT}_{VT}$. She can then apply the faulty test cases' inputs on the candidate clones $\mathbb{C}^{VT}_{VT}$ determined from executing only valid test cases. This approach provides her with some level of verification on the true outputs of her subject code and identification of correct code. In the example from Figure 6, `test0`'s inputs can be driven to the behavioral clones found from executing only `test1` and `test2`. This helps the developer determine whether there are any clones similar to `computeJaccardSimilarity` on `test1` and `test2` but that also output a Jaccard similarity of 0 (as per `test0`'s assertion statement) on `test0`'s inputs.

Alternatively, suppose that the developer has tests for which the assertion statements themselves are faulty or nonexistent. Some tests may also have been generated via automated testing tools, which have been applied on the codebase only after it has been infected by the mutant. The developer might be interested in running the latter set of executions, obtaining a more restricted set of behavioral clones $\mathbb{C}^{VT\wedge FT}_{VT\vee FT}$, which consists of clones similar to the subject code on both valid and faulty test cases' inputs. The set $\mathbb{C}^{VT\wedge FT}_{VT\vee FT}$ is separately presented to the developer, annotated with references to the faulty test cases, indicating these clones are similar to the subject code at hand on all test cases but do not possess the desired functionality. SABER also constructs the set of behavioral clones $\mathbb{C}^{VT\wedge\neg FT}_{VT\vee FT}$, the set of clones similar to the subject code on valid test cases' inputs, but not on faulty test cases' inputs. The set $\mathbb{C}^{VT\wedge\neg FT}_{VT\vee FT}$ will be more helpful to the developer than $\mathbb{C}^{VT}_{VT}$ because the outputs of the former set of clones are known to be distinct from the invalid outputs of the faulty test cases.

We propose that functional I/O similarity is the predominant definition of behavioral similarity for debugging, as valid test cases guarantee valid inputs for executing those branches of code that are independent of bug-exposing code. Hence, μ_1 may be large in comparison to μ_2 and μ_3. Since the developer is only interested in finding behavioral clones on the subject code's inputs, β I/O profiles are ignored in secondary profile analysis. Therefore, ζ_r is always 1, and $\Psi_{I/O,(M_1,M_2)}$ is simply the average similarity across the α I/O profiles.

6.3 Legacy Code Refactoring

As a preliminary step to code refactoring, it is preferable to perform dead code elimination (DCE) on the codebase for removal of unused code and code optimization. Since our primary objective from §2.2 for code refactoring is to identify simpler, more comprehensible code than the subject code (yet still functionally equivalent), static simplification of all application code via DCE is recommended, although not necessary. Moreover, DCE may improve program performance and does not eliminate any candidate functional I/O clones, given our definitions of inputs and outputs.

Code complexity analysis from §5.2 is highly applicable to the code refactoring use case, as simpler code that preserves the functionality of the subject code may be easier to maintain and understand. We are interested in finding exact behavioral clones; therefore, we set μ_1 equal to 1.0 and set the similarity threshold θ_{sim} to 1.0. To further aid with refactoring, functionally-equivalent behavioral clones with different execution traces, i.e., possessing executional dissimilarity, may be considered "behaviorally different enough" from the legacy code to be used as an effective replacement. Behavioral clones are then ranked by decreasing complexity. Note that by default, only behavioral clones with lower complexity than that of the subject code are presented to the developer.

6.4 Feature Enhancement

For feature enhancement, we prioritize the search for behavioral clones from other codebases as per the motivation in §2.2. SABER is designed to support an aggregation of similarity among multiple

methods and classes. The choices of μ and ζ_r, by design, are the same as for baseline understanding but can be toggled accordingly. The similarity metric for classes has been explained in §5.1. Similar to code refactoring, this stage leverages similarity ranking based on the LCOM PFI complexity metric introduced in §5.2 for classes.

7 RELATED WORK

Our primary focus with SABER is to identify similarly-behaving code targeted for software engineering tasks that mandate some level of program comprehension. We also leverage dynamic symbolic execution and metamorphic testing for test case generation and similarity detection. Thus, we appropriately split related work into *behavioral clone detection*, *dynamic symbolic execution*, *program comprehension*, and *metamorphic testing*.

Behavioral Clone Detection. Many studies have proposed techniques for behavioral clone detection, including work for functional I/O analysis [21, 29, 32, 48], executional analysis [56, 81], concolic execution [25, 39, 41, 45, 70, 86], and other analysis approaches. We highlight the studies for functional I/O and executional analyses, which were implemented in our work, and provide succinct commentary on symbolic/concolic execution. We are unaware of any systems that perform metamorphic analysis and its ilk.

HitoshiIO [21] is a proof-of-concept functional I/O clone detector by Su et al. that extracts inputs and outputs from instrumented bytecode. SABER adapts HitoshiIO's approach to capturing, recording, and extracting inputs from state variables and method arguments, as well as outputs from return variables. However, HitoshiIO is contingent on executing existing workloads, restricted to classes with "main" methods. Further, HitoshiIO classifies similarity based on only a trivial set of test cases. The system sidesteps the method-level test case generation problem by utilizing only the I/O driven in the context of executions of the full application. While this is convenient, a pairwise comparison of all method invocations is highly inefficient; methods that should be functionally similar may never be invoked with similar inputs in HitoshiIO. SABER mitigates these issues by selecting the most convincing similarities on the most representative but nontrivial test cases, with common inputs passed to each pair of methods under comparison.

LASSO [32] by Kessel and Atkinson presents an approach for functional I/O analysis that constructs coverage-based test cases using the EvoSuite tool. However, LASSO only identifies functional I/O clones with the same method name and signature, which significantly limits the scope for finding functional I/O clones. In extension, LASSO is only able to detect clones for methods with arguments of array, string, and primitive types, whereas SABER can also support more complicated data structures such as objects, collections, and file types. Unlike LASSO, which does not work with existing workloads, SABER not only generates automated test cases using EvoSuite, but also enables developers to supply pre-existing test cases and can produce novel test cases based on methods' metamorphic properties.

For functional I/O analysis, SABER is most related to SLACC [48], a state-of-the-art functional I/O clone detector for both dynamically-typed and statically-typed languages. SLACC, developed by Mathew et al., generates test cases per method using an approach inspired by grey-box testing. However, SLACC provides no compelling explanation for which subset of these test cases is sufficient for executing all method-specific source code, or achieving high *test coverage*. Recent studies have shown that improving test coverage is essential to gaining more confidence in the test cases under consideration and the overall software quality. To truly assess code similarity and verify that tests extensively identify all aspects of the application code (including potential defects), all test coverage criteria must be considered; SLACC does not aim to achieve complete or high code coverage and may overshoot the number of test case executions required for certain coverage criteria, thus exacerbating the runtime performance. SABER leverages EvoSuite to generate test cases that attempt to target all coverage criteria (including method, line, and branch coverage), regardless of EvoSuite's achieved coverage and mutation score for each test file.

Furthermore, while SLACC only selects inputs from method arguments via an analysis of the source code, SABER adapts the approach in HitoshiIO to find inputs from both state variables and method arguments at the bytecode level, via ASM bytecode instrumentation [6]. As a result, SLACC has limited capability in input replacement for constructing new test cases; during I/O profile input replacement in the secondary test case generation phase, SABER can modify the values of both used state variable and used method argument inputs, whereas SLACC can only modify method arguments (without regard for data flow). SLACC also fails to support comparison of methods with a different number of inputs or arguments, lacks data type conversion of inputs during input replacement, and performs minimal primitive type casting of method arguments for similarity analysis; these limitations, including the lack of object type conversion, may account for fewer method-level clones in SLACC and have been addressed in SABER.

In contrast to HitoshiIO, SLACC attempts to provide a systematic approach to selecting relevant test cases. However, both suffer from the "preponderance of the evidence" problem by failing to provide a rationale as to why the test cases chosen for similarity are convincing. SABER responds to this problem by choosing and justifying the most representative test cases based on coverage, developer specifications, and metamorphic properties. SABER builds on this approach by constructing a filtering scheme that applies weight factors to representative and non-representative test cases and factors differences in code complexity to choose the most convincing similarities. Critically, unlike all of these techniques, SABER customizes exploitation of behavioral clones to specific software engineering tasks mandating program comprehension across multiple categories of behavioral similarity.

DyCLINK [81] is the state-of-the-art system for executional clones, using subgraph isomorphism on dynamic dependency graphs to trace program executions at the bytecode level. SABER improves on DyCLINK's approach for executional analysis by applying subgraph isomorphism on targeted code subjects and prioritizing Java API calls to capture the most interesting behaviors. SABER's test case generation also helps to alleviate DyCLINK's problem of requiring a sufficient test suite to increase confidence in behavioral code similarities.

Dynamic Symbolic Execution. Concolic execution has remained an effective technique for behavioral analysis. It generates high-coverage test suites while considering multiple execution paths simultaneously. One representation of concolic execution is dynamic symbolic execution (DSE), a hybrid technique that combines classical symbolic execution and concrete execution while overcoming the limitations of both techniques.

Sen et al. [71] present a new DSE technique for JavaScript that performs incremental state merging of symbolic states without the introduction of auxiliary variables. Li et al. [39] introduce a paired-program DSE approach and associated behavioral similarity metrics tailored for online programming and software engineering education. JDart by Luckow et al. [45] is a Java-based DSE framework that executes and symbolically analyzes Java bytecode for specific program methods based on an "explorer-and-executor" architecture. Vartanov [86] constructs a DSE engine for Java programs targeting the Java Native Interface (JNI) framework, based on the connection of native code with Java bytecode.

However, these techniques may result in missed paths and frequent path divergences. In extension, they might fail to generate test inputs along certain execution paths. While these techniques alleviate the issues associated with scalability, they do not guarantee the elimination of path explosion for more complex programs or software systems. Such limitations reduce the soundness of these techniques and lead to greater false negatives.

Nevertheless, DSE has shown to be promising. SynFuzz by Han et al. [25] is a hybrid fuzzer designed to address scalability challenges. It performs concolic execution by leveraging dynamic taint analysis and program synthesis based on branch predicate synthesis and branch flipping. Liang et al. [41] has recently developed a practical concolic execution engine, Pracolic, that aims to alleviate the state explosion problem by focusing on program state generation strategies targeting different classifications of symbolic memory. As an adaptation of DSE for automated test generation, SABER uses EvoSuite, a scalable automated testing tool that integrates DSE with search-based software testing to maximize code coverage.

Program Comprehension. Prior work in program comprehension has been diverse in the evaluation of novel software tools and in bridging the gaps between such tools and developers' level of program understanding [1, 10, 27, 35–38, 46, 50–52, 54, 61, 63, 74, 75, 90, 93]. Some studies have found overlap even in metamorphic testing [17, 30, 31, 77, 85, 95, 96]. However, there are few studies that appear germane to behavioral code similarity.

Earlier studies [35, 36, 74] hinted at limitations in gauging the effectiveness of development tools to answer the questions that developers ask when they evolve their application codebases. Due to findings that showed developers spending more time understanding than modifying or enhancing the code, these works called for new, well-directed tools as a replacement for manual, misguided, and failed searches for relevant code. Robillard et al. [63] constructed a tool, ConcernDetector, that takes advantage of two Java systems' revision histories to make recommendations about relevant, high-level features, requirements, and design decisions to the developer.

Recent studies continue to be interested in and build tools for enhancing program comprehension. Meyer et al. in both [50] and [51] study developers' workday productivity and find that developers still continue to face problems with existing documentation, which is often outdated or insufficient, for improved understanding of their code's functionality. They elaborate that participants require sufficient tool support to form good work habits with identification and monitoring of well-defined goals. Chen et al. [10] develop a tool to measure changes in performance across software revisions to help developers better comprehend and resolve performance issues that prevail in practice. CROKAGE [75] by Silva et al. identifies and presents solutions to developers in response to queries about unfamiliar code by leveraging Lucene indexing and FastText modeling to filter candidate solutions.

Mooij et al. [54] explore an approach that combines code refactoring and model-based rejuvenation to reduce the complexity of the code and improve program understanding, extensibility, and maintainability. While SABER applies dynamic analysis, [54] uses a static analysis technique that is limited to specific cases. Latoza et al. [37] build a mixed-initiative strategy description language called Roboto and design a strategy tracker tool for programming strategies to bridge the gap between human decision-making and computer processing.

Metamorphic Testing. To the best of our knowledge, SABER is the first work to apply metamorphic testing as a technique for both test case generation and code similarity for dynamic analysis. Metamorphic testing was first introduced by T.Y. Chen et al. [11] and was developed to solve the test oracle problem. Recent literature has applied metamorphic testing to RESTful web services [47, 69, 82, 83], deep learning [18, 19, 49, 84, 87, 89, 94], software engineering and program comprehension [5, 17, 30, 31, 77, 79, 85, 91, 95, 96], and many other areas.

The application of metamorphic testing in SABER is inspired by [79]. Su et al. implement Kabu, an intensive heuristic designed to predict metamorphic properties based on a series of input transformations and output verifications. With regard to identifying and extracting metamorphic properties, SABER adopts a similar approach to Kabu. While Kabu is limited to transforming only inputs from method arguments, SABER is able to magnify the input transformation space by incorporating inputs from both method arguments and state variables. SABER leverages the reported metamorphic properties to construct new test cases.

8 CONCLUSION

We have presented SABER, the first system to explore the "preponderance of the evidence" problem for identifying behaviorally-similar code. The proposed methodology leverages functional I/O analysis, executional analysis, and metamorphic analysis to identify behavioral clones targeted to baseline program comprehension, debugging, legacy code refactoring, and feature enhancement software engineering use cases. Furthermore, we have demonstrated the application of filtering and weighting schemes for adapting the "preponderance of the evidence" to choosing the most convincing similarities for a given use case. SABER enables developers to leverage behavioral clones to better understand their codebase toward the goals of improving workplace productivity, software quality, and code reliability.

ACKNOWLEDGMENTS

The Programming Systems Laboratory is supported in part by NSF CNS-1563555, CCF-1815494, and CNS-1842456.

REFERENCES

[1] E. Aghajani, C. Nagy, O. L. Vega-Márquez, M. Linares-Vásquez, L. Moreno, G. Bavota, and M. Lanza. 2019. Software Documentation Issues Unveiled. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1199–1210.

[2] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool. 2019. A Systematic Review on Code Clone Detection. *IEEE Access* 7 (2019), 86121–86144. https://doi.org/10.1109/ACCESS.2019.2918202

[3] Shulamyt Ajami, Yonatan Woodbridge, and Dror G Feitelson. 2019. Syntax, predicates, idioms—what really affects code complexity? *Empirical Software Engineering* 24, 1 (2019), 287–328.

[4] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated Concolic Testing of Smartphone Apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article Article 59, 11 pages. https://doi.org/10.1145/2393596.2393666

[5] Joshua Brown, Zhi Quan Zhou, and Yang-Wai Chow. 2019. Metamorphic Testing of Mapping Software. In *Towards Integrated Web, Mobile, and IoT Technology*, Tim A. Majchrzak, Cristian Mateos, Francesco Poggi, and Tor-Morten Grønli (Eds.). Springer International Publishing, Cham, 1–20.

[6] Eric Bruneton. 2007. ASM 4.0 A Java bytecode engineering library. (2007).

[7] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, USA, 209–224.

[8] J. Campos, A. Panichella, and G. Fraser. 2019. EvoSuite at the SBST 2019 Tool Competition. In *2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)*. 29–32. https://doi.org/10.1109/SBST.2019.00017

[9] Patrick P. F. Chan, Lucas C. K. Hui, and S. M. Yiu. 2011. Dynamic Software Birthmark for Java Based on Heap Memory Analysis. In *Communications and Multimedia Security*, Bart De Decker, Jorn Lapon, Vincent Naessens, and Andreas Uhl (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 94–107.

[10] J. Chen, D. Yu, H. Hu, Z. Li, and H. Hu. 2019. Analyzing Performance-Aware Code Changes in Software Development Process. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 300–310.

[11] TY Chen, SC Cheung, and SM Yiu. 1998. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01. *Hong Kong Univ. of Science and Technology* (1998).

[12] F. Coelho, T. Massoni, and E. L.G. Alves. 2019. Refactoring-Aware Code Review: A Systematic Mapping Study. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*. 63–66.

[13] William W Cohen, Pradeep Ravikumar, Stephen E Fienberg, et al. 2003. A Comparison of String Distance Metrics for Name-Matching Tasks.. In *IIWeb*, Vol. 2003. 73–78.

[14] XStream Committers. 2018. XStream. https://x-stream.github.io/

[15] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience* 34, 11 (2004), 1025–1050. https://doi.org/10.1002/spe.602 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.602

[16] F. Deissenboeck, L. Heinemann, B. Hummel, and S. Wagner. 2012. Challenges of the Dynamic Detection of Functionally Similar Code Fragments. In *2012 16th European Conference on Software Maintenance and Reengineering*. 299–308. https://doi.org/10.1109/CSMR.2012.38

[17] Junhua Ding, Dongmei Zhang, and Xin-Hua Hu. 2016. An Application of Metamorphic Testing for Testing Scientific Software. In *Proceedings of the 1st International Workshop on Metamorphic Testing (MET '16)*. Association for Computing Machinery, New York, NY, USA, 37–43. https://doi.org/10.1145/2896971.2896981

[18] A. Dwarakanath, M. Ahuja, S. Podder, S. Vinu, A. Naskar, and M. Koushik. 2019. Metamorphic Testing of a Deep Learning Based Forecaster. In *2019 IEEE/ACM 4th International Workshop on Metamorphic Testing (MET)*. 40–47.

[19] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M. Rao, R. P. Jagadeesh Chandra Bose, Neville Dubash, and Sanjay Podder. 2018. Identifying Implementation Bugs in Machine Learning Based Image Classifiers Using Metamorphic Testing. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 118–128. https://doi.org/10.1145/3213846.3213858

[20] Abdulrahman Abu Elkhail, Jan Svacina, and Tomas Cerny. 2019. Intelligent Token-Based Code Clone Detection System for Large Scale Source Code. In *Proceedings of the Conference on Research in Adaptive and Convergent Systems (RACS '19)*. Association for Computing Machinery, New York, NY, USA, 256–260. https://doi.org/10.1145/3338840.3355654

[21] Fang-Hsiang Su, J. Bell, G. Kaiser, and S. Sethumadhavan. [n.d.]. Identifying Functionally Similar Code in Complex Codebases, year=2016. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. 1–10. https://doi.org/10.1109/ICPC.2016.7503720

[22] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 416–419. https://doi.org/10.1145/2025113.2025179

[23] G. Fraser and A. Zeller. 2012. Mutation-Driven Generation of Unit Tests and Oracles. *IEEE Transactions on Software Engineering* 38, 2 (March 2012), 278–292. https://doi.org/10.1109/TSE.2011.93

[24] Thomas Fritz, Jingwen Ou, Gail C. Murphy, and Emerson Murphy-Hill. 2010. A Degree-of-Knowledge Model to Capture Source Code Familiarity. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 385–394. https://doi.org/10.1145/1806799.1806856

[25] Wookhyun Han, Md Lutfor Rahman, Yuxuan Chen, Chengyu Song, Byoungyoung Lee, and Insik Shin. 2019. SynFuzz: Efficient Concolic Execution via Branch Condition Synthesis. *arXiv preprint arXiv:1905.09532* (2019).

[26] Zahir Hasheminasab, Zaniar Sharifi, Khabat Soltanian, and Mohsen Afsharchi. 2020. Using Augmented Genetic Algorithm for Search-Based Software Testing. In *Data Science: From Research to Application*, Mahdi Bohlouli, Bahram Sadeghi Bigham, Zahra Narimani, Mahdi Vasighi, and Ebrahim Ansari (Eds.). Springer International Publishing, Cham, 248–257.

[27] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep Code Comment Generation. In *Proceedings of the 26th Conference on Program Comprehension (ICPC '18)*. Association for Computing Machinery, New York, NY, USA, 200–210. https://doi.org/10.1145/3196321.3196334

[28] Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K. Chang. 2010. OCAT: Object Capture-Based Automated Testing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA '10)*. Association for Computing Machinery, New York, NY, USA, 159–170. https://doi.org/10.1145/1831708.1831729

[29] Lingxiao Jiang and Zhendong Su. 2009. Automatic Mining of Functionally Equivalent Code Fragments via Random Testing. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA '09)*. Association for Computing Machinery, New York, NY, USA, 81–92. https://doi.org/10.1145/1572272.1572283

[30] Mingyue Jiang, Tsong Yueh Chen, Fei-Ching Kuo, Dave Towey, and Zuohua Ding. 2017. A metamorphic testing approach for supporting program repair without the need for a test oracle. *Journal of Systems and Software* 126 (2017), 127 – 140. https://doi.org/10.1016/j.jss.2016.04.002

[31] U. Kanewala and T. Yueh Chen. 2019. Metamorphic Testing: A Simple Yet Effective Approach for Testing Scientific Software. *Computing in Science Engineering* 21, 1 (2019), 66–72.

[32] M. Kessel and C. Atkinson. 2019. On the Efficacy of Dynamic Behavior Comparison for Judging Functional Equivalence. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 193–203. https://doi.org/10.1109/SCAM.2019.00030

[33] Kisub Kim, Dongsun Kim, Tegawendé F. Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. 2018. FaCoY: A Code-to-Code Search Engine. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 946–957. https://doi.org/10.1145/3180155.3180187

[34] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. 2005. An Empirical Study of Code Clone Genealogies. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. Association for Computing Machinery, New York, NY, USA, 187–196. https://doi.org/10.1145/1081706.1081737

[35] A. J. Ko, R. DeLine, and G. Venolia. 2007. Information Needs in Collocated Software Development Teams. In *29th International Conference on Software Engineering (ICSE'07)*. 344–353. https://doi.org/10.1109/ICSE.2007.45

[36] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering* 32, 12 (Dec 2006), 971–987. https://doi.org/10.1109/TSE.2006.116

[37] Thomas D LaToza, Maryam Arab, Dastyni Loksa, and Amy J Ko. 2020. Explicit programming strategies. *Empirical Software Engineering* (2020), 1–34.

[38] Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining Mental Models: A Study of Developer Work Habits. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. Association for Computing Machinery, New York, NY, USA, 492–501. https://doi.org/10.1145/1134285.1134355

[39] S. Li, X. Xiao, B. Bassett, T. Xie, and N. Tillmann. 2016. Measuring Code Behavioral Similarity for Programming and Software Engineering Education. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. 501–510.

[40] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. 2006. CP-Miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering* 32, 3 (March 2006), 176–192. https://doi.org/10.1109/TSE.2006.28

[41] Hongliang Liang, Wenqing Yu, Lu Ai, and Lin Jiang. 2020. A Practical Concolic Execution Technique for Large Scale Software Systems. In *Proceedings of the Evaluation and Assessment in Software Engineering (EASE '20)*. Association for Computing Machinery, New York, NY, USA, 312–317. https://doi.org/10.1145/3383219.3383254

[42] B. Lin, C. Nagy, G. Bavota, and M. Lanza. 2019. On the Impact of Refactoring Operations on Code Naturalness. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 594–598.

[43] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java virtual machine specification*. Pearson Education.

[44] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: Code Recommendation via Structural Code Search. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 152 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360578

[45] Kasper Luckow, Marko Dimjašević, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamarić, and Vishwanath Raman. 2016. JDart: A Dynamic Symbolic Analysis Framework. In *Tools and Algorithms for the Construction and Analysis of Systems*, Marsha Chechik and Jean-François Raskin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 442–459.

[46] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. 2014. On the Comprehension of Program Comprehension. *ACM Trans. Softw. Eng. Methodol.* 23, 4, Article Article 31 (Sept. 2014), 37 pages. https://doi.org/10.1145/2622669

[47] Phu X Mai, Fabrizio Pastore, Arda Goknil, and Lionel Briand. 2019. Metamorphic Security Testing for Web Systems. *arXiv preprint arXiv:1912.05278* (2019).

[48] George Mathew, Chris Parnin, and Kathryn T Stolee. 2020. SLACC: Simion-based Language Agnostic Code Clones. In *International Conference on Software Engineering*. arXiv preprint arXiv:2002.03039.

[49] R. R. Mekala, G. E. Magnusson, A. Porter, M. Lindvall, and M. Diep. 2019. Metamorphic Detection of Adversarial Examples in Deep Learning Models with Affine Transformations. In *2019 IEEE/ACM 4th International Workshop on Metamorphic Testing (MET)*. 55–62.

[50] A. Meyer, E. T. Barr, C. Bird, and T. Zimmermann. 2019. Today was a Good Day: The Daily Life of Software Developers. *IEEE Transactions on Software Engineering* (2019), 1–1.

[51] A. N. Meyer, G. C. Murphy, T. Zimmermann, and T. Fritz. 2019. Enabling Good Work Habits in Software Developers through Reflective Goal-Setting. *IEEE Transactions on Software Engineering* (2019), 1–1.

[52] R. Minelli, A. Mocci, and M. Lanza. 2015. I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time. In *2015 IEEE 23rd International Conference on Program Comprehension*. 25–35. https://doi.org/10.1109/ICPC.2015.12

[53] A. Mockus and J. D. Herbsleb. 2002. Expertise Browser: a quantitative approach to identifying expertise. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. 503–512.

[54] A. J. Mooij, J. Ketema, S. Klusener, and M. Schuts. 2020. Reducing Code Complexity through Code Refactoring and Model-Based Rejuvenation. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 617–621.

[55] Jose Luis Morales and Stefan Gwihs. 2020. The new major version of the programmer-friendly testing framework for Java. https://junit.org/junit5/

[56] Ginger Myles and Christian Collberg. 2004. Detecting Software Theft via Whole Program Path Birthmarks. In *Information Security*, Kan Zhang and Yuliang Zheng (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 404–415.

[57] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-Directed Random Testing for Java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion (OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 815–816. https://doi.org/10.1145/1297846.1297902

[58] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. 2007. Feedback-Directed Random Test Generation. In *29th International Conference on Software Engineering (ICSE'07)*. 75–84. https://doi.org/10.1109/ICSE.2007.37

[59] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report 1999-66. Stanford InfoLab. http://ilpubs.stanford.edu:8090/422/ Previous number = SIDL-WP-1999-0120.

[60] S Rao Polamuri, S Rama Sree, and M Rajababu. 2012. Fault Prediction OO Systems Using the Conceptual Cohesion of Classes. *International Journal of Computer Science and Information Technologies IJCSIT* 3, 4 (2012), 4684–4688.

[61] Akond Rahman. 2018. Comprehension Effort and Programming Activities: Related? Or Not Related?. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*. Association for Computing Machinery, New York, NY, USA, 66–69. https://doi.org/10.1145/3196398.3196470

[62] David A. Ramos and Dawson Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 49–64. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ramos

[63] M. P. Robillard and P. Manggala. 2008. Reusing Program Investigation Knowledge for Code Understanding. In *2008 16th IEEE International Conference on Program Comprehension*. 202–211. https://doi.org/10.1109/ICPC.2008.10

[64] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Sci. Comput. Program.* 74, 7 (May 2009), 470–495. https://doi.org/10.1016/j.scico.2009.02.007

[65] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. 2015. How Developers Search for Code: A Case Study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 191–201. https://doi.org/10.1145/2786805.2786855

[66] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-Code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 1157–1168. https://doi.org/10.1145/2884781.2884877

[67] David Schuler, Valentin Dallmeier, and Christian Lindig. 2007. A Dynamic Birthmark for Java. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*. Association for Computing Machinery, New York, NY, USA, 274–283. https://doi.org/10.1145/1321631.1321672

[68] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés. 2016. A Survey on Metamorphic Testing. *IEEE Transactions on Software Engineering* 42, 9 (2016), 805–824.

[69] S. Segura, J. A. Parejo, J. Troya, and A. Ruiz-Cortés. 2018. Metamorphic Testing of RESTful Web APIs. *IEEE Transactions on Software Engineering* 44, 11 (2018), 1083–1099.

[70] Koushik Sen. 2015. Automated Test Generation Using Concolic Testing. In *Proceedings of the 8th India Software Engineering Conference (ISEC '15)*. Association for Computing Machinery, New York, NY, USA, 9. https://doi.org/10.1145/2723742.2723768

[71] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: Multi-Path Symbolic Execution Using Value Summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 842–853. https://doi.org/10.1145/2786805.2786830

[72] D. Serra, G. Grano, F. Palomba, F. Ferrucci, H. C. Gall, and A. Bacchelli. 2019. On the Effectiveness of Manual and Automatic Unit Test Generation: Ten Years Later. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 121–125.

[73] Y. Shinyama, Y. Arahori, and K. Gondow. 2018. Analyzing Code Comments to Boost Program Comprehension. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. 325–334.

[74] J. Sillito, G. C. Murphy, and K. De Volder. 2008. Asking and Answering Questions during a Programming Change Task. *IEEE Transactions on Software Engineering* 34, 4 (2008), 434–451.

[75] R. F. G. Silva, C. K. Roy, M. M. Rahman, K. A. Schneider, K. Paixao, and M. de Almeida Maia. 2019. Recommending Comprehensive Solutions for Programming Tasks by Mining Crowd Knowledge. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 358–368.

[76] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. 2010. An Examination of Software Engineering Work Practices. In *CASCON First Decade High Impact Papers (CASCON '10)*. IBM Corp., USA, 174–188. https://doi.org/10.1145/1925805.1925815

[77] M. Srinivasan. 2018. Prioritization of Metamorphic Relations Based on Test Case Execution Properties. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 162–165.

[78] Kathryn T. Stolee, Sebastian Elbaum, and Matthew B. Dwyer. 2016. Code Search with Input/Output Queries. *J. Syst. Softw.* 116, C (June 2016), 35–48. https://doi.org/10.1016/j.jss.2015.04.081

[79] F. Su, J. Bell, C. Murphy, and G. Kaiser. 2015. Dynamic Inference of Likely Metamorphic Properties to Support Differential Testing. In *2015 IEEE/ACM 10th International Workshop on Automation of Software Test (AST)*. 55–59.

[80] Fang-Hsiang Su. 2018. *Uncovering Features in Behaviorally Similar Programs*. Ph.D. Dissertation. Columbia University.

[81] Fang-Hsiang Su, Jonathan Bell, Kenneth Harvey, Simha Sethumadhavan, Gail Kaiser, and Tony Jebara. 2016. Code Relatives: Detecting Similarly Behaving Software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 702–714. https://doi.org/10.1145/2950290.2950321

[82] C. Sun, Y. Liu, Z. Wang, and W. K. Chan. 2016. MT: A Data Mutation Directed Metamorphic Relation Acquisition Methodology. In *2016 IEEE/ACM 1st International Workshop on Metamorphic Testing (MET)*. 12–18.

[83] Chang-ai Sun, Guan Wang, Qing Wen, Dave Towey, and Tsong Yueh Chen. 2016. MT4WS: an automated metamorphic testing system for web services. *IJHPCN* 9, 1/2 (2016), 104–115.

[84] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 303–314. https://doi.org/10.1145/3180155.3180220

[85] Sandro Tolksdorf, Daniel Lehmann, and Michael Pradel. 2019. Interactive Metamorphic Testing of Debuggers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 273–283. https://doi.org/10.1145/3293882.3330567

[86] S. Vartanov. 2017. Dynamic symbolic execution of Java programs using JNI. In *2017 Computer Science and Information Technologies (CSIT)*. 83–86.

[87] Shuai Wang and Zhendong Su. 2019. Metamorphic Testing for Object Detection Systems. *arXiv preprint arXiv:1912.12162* (2019).

[88] Arthur Henry Watson, Dolores R Wallace, and Thomas J McCabe. 1996. *Structured testing: A testing methodology using the cyclomatic complexity metric*. Vol. 500. US Department of Commerce, Technology Administration, National Institute of ....

[89] Adrian Wildandyawan and Yasuharu Nishi. 2020. Object-based Metamorphic Testing through Image Structuring. *arXiv preprint arXiv:2002.07046* (2020).

[90] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li. 2018. Measuring Program Comprehension: A Large-Scale Field Study with Professionals. *IEEE Transactions on Software Engineering* 44, 10 (Oct 2018), 951–976. https://doi.org/10.1109/TSE.2017.2734091

[91] B. Zhang, H. Zhang, J. Chen, D. Hao, and P. Moscato. 2019. Automatic Discovery and Cleansing of Numerical Metamorphic Relations. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 235–245.

[92] Sai Zhang, David Saff, Yingyi Bu, and Michael D. Ernst. 2011. Combined Static and Dynamic Automated Test Generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. Association for Computing Machinery, New York, NY, USA, 353–363. https://doi.org/10.1145/2001420.2001463

[93] Junji Zhi, Vahid Garousi-Yusifoğlu, Bo Sun, Golara Garousi, Shawn Shahnewaz, and Guenther Ruhe. 2015. Cost, benefits and quality of software development documentation: A systematic mapping. *Journal of Systems and Software* 99 (2015), 175 – 198. https://doi.org/10.1016/j.jss.2014.09.042

[94] Zhi Quan Zhou and Liqun Sun. 2019. Metamorphic testing of driverless cars. *Commun. ACM* 62, 3 (2019), 61–67.

[95] Z. Q. Zhou, L. Sun, T. Y. Chen, and D. Towey. 2018. Metamorphic Relations for Enhancing System Understanding and Use. *IEEE Transactions on Software Engineering* (2018), 1–1. https://doi.org/10.1109/TSE.2018.2876433

[96] Z. Q. Zhou, S. Xiang, and T. Y. Chen. 2016. Metamorphic Testing for Software Quality Assessment: A Study of Search Engines. *IEEE Transactions on Software Engineering* 42, 3 (2016), 264–284.