Automatic Extraction of Behavioral Features for Test Program Similarity Analysis

Emanuele De Angelis
IASI-CNR
Rome Italy
emanuele.deangelis@iasi.cnr.it

Alessandro Pellegrini
IASI-CNR
Rome Italy
alessandro.pellegrini@iasi.cnr.it

Maurizio Proietti
IASI-CNR
Rome Italy
maurizio.proietti@iasi.cnr.it

***Abstract*—We present a methodology for performing automatic extraction of behavioral features from test programs, that is, for collecting pieces of information about the test programs execution. These features are then exploited to carry out analysis and reasoning about test program similarity. The similarity information can be used to drive the execution of test campaigns, in the attempt to either reduce the time-to-test, or to increase the testing capabilities of a given test suite. Our methodology is embedded in the Hyperion analysis framework, which can be configured to define a wide range of test program similarity criteria.**

***Index Terms*—Test Program, Similarity Analysis, Symbolic Execution, Automated Reasoning**

I. Introduction

The ability to determine a suitable degree of similarity among test programs creates several opportunities to be taken when these test programs are run. Indeed, depending on the particular phase of the software lifecycle in which the test suite is exercised, different strategies can be designed to either reduce the time to completion of the test programs, or exploit the available test programs for stress testing a given software system in a comprehensive way. As an example, during regression testing it might be advisable to avoid a retest-all strategy, skipping specific tests associated with portions of the system under test for which a test program has already detected a failure [1], [2]. Conversely, right before the deployment of a new release, the available test programs might be composed, e.g., by launching in parallel tests which access the same (shared) data structures, in the attempt to discover some concurrency bug [3], [4].

Deeming two test programs similar requires a deep understanding of both the test suite and the system under test. When the source code is (partially) unavailable, or when the application of interest is complex and/or large, identifying similarities can be difficult and time-consuming [5]. This can be particularly the case of microservice-based applications [6], where the (large) number and diverse nature of microservices and test suites can hinder human analysis: the burden placed on the QA Engineering team could be high to the extent that other development activities might be slowed down, if proper classification of the similarity between test programs is carried out manually. This is especially true if testing activities at different development stages are pursued.

This paper has been supported by the Italian MIUR PRIN 2017 Project: SISMA (Contract 201752ENYB) and by the POR FESR Project of the LAZIO Region: OPENNESS (Project code POR: A0375E0069).

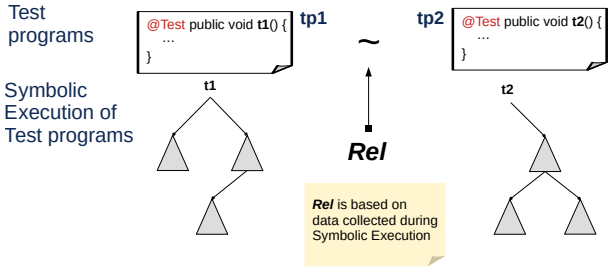


Fig. 1: Constructing similarity relations by analyzing the symbolic execution of test programs.

In order to define similarity relations between test programs it is particularly relevant the identification of suitable pieces of information about their execution, which we collectively call *behavioral features* in this paper. Some form of automated extraction of behavioral features is desirable, and the set of features which should be considered must be configurable to support different similarity discovery activities, exactly associated with the different phases at which this information could be leveraged.

In this paper we present Hyperion[1], a test suite analysis framework, which allows us to carry out automatic extraction of behavioral features from test programs, targeting Java applications and JUnit [7] test suites. The overall approach that we follow is depicted in Figure 1. Our framework relies on symbolic execution [8] to exercise (parametric) test programs and to collect information related to: i) what methods are invoked during a (possible) execution of test programs; ii) what endpoints are contacted—this latter piece of information is particularly useful in the context of microservice-based applications.

Our framework embodies a set of Prolog rules [9] that allow us to conduct, based on the collected features, different reasoning activities. In this way, it is possible to study test program similarity under different conditions, by relying on different similarity criteria. The set of Prolog rules can be

[1]Source code available at http://saks.iasi.cnr.it/tools/hyperion.

configured easily, thus allowing for a custom usage when interacting with our framework.

The current version of Hyperion implements and extends the technique presented in previous work [10], which focuses on the context of microservice applications. The present contribution stresses the point of configurability of the framework by presenting various general purpose similarity criteria. We also provide a new case study, consisting of a medium-sized application that implements a system for railway ticketing, including 682 JUnit test programs. The system under test is a microservice-based application and, besides the similarities between its test programs that refer specifically to the microservice architecture (i.e., to the microservice endpoints), we also infer other types of similarities based on the extraction of sequences of method calls.

The remainder of this paper is organized as follows. In Section II we discuss related work. The methodology to extract similarity features is discussed in Section III. Prolog rules to carry our similarity analysis are discussed in Section IV. Section V reports the results of our case study. Section VI presents some concluding remarks.

II. Related Work

The problem of identifying similarities among test programs is related to the broader problem of identifying similarities among generic programs [11], which has been studied for multiple purposes (and with different techniques), such as duplicate code detection [12], plagiarism detection or copyright infringement [13], and code compression [14].

The idea of using test case similarity for the design of effective testing strategies has been explored in several works [15]–[19]. Noor and Hemmati [15] propose an approach for prioritizing test cases on the basis of their similarity with the ones that failed on previous versions of the software system under consideration. The similarity between test cases is defined by comparing sequences of method calls, extracted from execution traces. That paper makes use of concrete execution, while we perform a logic-based similarity analysis on traces extracted via symbolic execution.

Another similarity-based approach for regression test case prioritization is presented by R. Wang *et al.* [16]. The execution order of the test cases is scheduled based on the distance between them, where the notion of distance is defined with respect to branch coverage. That paper evaluates six similarity measures and shows, through experiments on a few benchmark programs, that Euclidean distance gives the best result.

Test case similarity is defined by Ledru *et al.* [18] on the basis of the string distance between the test cases, and hence no notion of execution is considered. Also Miranda *et al.* [19] base their test case prioritization technique on similarity relations defined on test cases, and not on their execution. To enforce scalability, similarity is computed by algorithms usually applied in the context of big data processing.

Similarity has been exploited for fault-localization in the paper by Hao *et al.* [17], where similarity between test cases is defined by using a fuzzy set representation of a matrix relating test cases and program statements, and candidate faulty statements are selected on a probabilistic basis.

Some papers propose techniques for computing similarity of programs (not necessarily test programs) based on static analysis or fuzz testing, whereas we employ symbolic execution. In particular, Raman *et al.* [20] use the call-dependency relation among program APIs to generate a trace of the API calling sequence. S. Wang and Wu [21] present a method that makes use of fuzz testing for a similarity analysis of binary code. The similarity score of two behavior traces generated by fuzzing from two program functions is computed according to their longest common subsequence.

In the context of automated software testing, symbolic execution has been largely used as an effective technique for finding errors in software applications and for generating high-coverage test suites [22]–[31]. This technique, which was first introduced in the mid 1970's [32], [33], has been conceived to exercise a software system by searching for potential configurations/states violating a given set of assertions. The basic idea of symbolic execution is strongly related to techniques for *bounded model checking* of software, which use Satisfiability Modulo Theories (SMT) solvers for checking that a specified program property is not violated by any execution path, up to a given length bound [34].

Some techniques for *relational verification* make use of *constraint logic programming* (i.e., logic programming augmented with constraint solving) to verify relations between programs [35], [36]. However, the kind of properties targeted by relational verification are very strong (in general, undecidable) relations, such as full functional equivalence, while here we focus on test programs and we are interested in much weaker dependency and similarity relations based on suitable abstractions of the finite set of paths generated by symbolic execution. In this respect, our work bears some relationships with symbolic execution techniques for crosschecking optimised versions of data-parallel programs with respect to the optimised ones [37].

III. Extraction of Behavioral Features

Our technique for extracting behavioral features is based on three main execution phases: i) test program enumeration; ii) feature extraction; iii) Prolog facts generation. These three phases generate a knowledge base which is later used to carry out automated logic-based reasoning to determine test program similarity, according to some (user-specified) criteria. In the following, we detail the methodological/technical organization of these phases.

A. Test Program Enumeration

The identification of similarities among test programs is guided by an automatic analysis procedure scanning their implementations (e.g., available from source-code repositories). The analysis procedure assumes that test programs are clearly identifiable from the rest of the source-code—we explicitly rely on JUnit annotations.

```
{
  "sut": [
    "path to classes 1",
    "path to classes 2"
  ],
  "testPrograms": [
    "path to test classes 1",
    "path to test classes 1"
  ],
  "includeTest": [
    "list", "of", "@Test",
    "methods", "to", "analyze"
  ],
  "excludeTest": [
    "list", "of", "@Test",
    "methods", "to", "skip"
  ],
  "additionalClasspath": [
    "path", "to", "any",
    "other", "needed", "dependency"
  ],
  "excludeTracedPackages": [
    "java/",
    "sun/"
  ]
}
```

Fig. 2: Hyperion JSON Configuration File.

Hyperion is based on a JSON configuration file, the structure of which is reported in Figure 2. For test program discovery, we rely on the `testProgram` list, which allows the user to specify the paths to the folders where the compiled test classes can be found on disk. Hyperion scans these paths, and recursively enumerates all classes. In each class found, all methods annotated as `@Test` (but without the `@Ignore` annotation) are located, and stored into an in-memory dictionary.

The possibility to rely on a list of paths is particularly important for microservice-oriented applications. Indeed, in this scenario, the test programs could be related to different microservices, while integration, contract, or end-to-end test programs might be delegated into different sources. In this way, Hyperion is compliant with multiple testing strategies for microservice architectures [38].

If, for any reason, the QA Engineering team wants to exclude some test programs, this can be done by relying on the `excludeTest` list in the JSON configuration file. Conversely, if only a certain number of test programs should be analyzed, the `includeTest` list in the JSON file can be configured accordingly.

B. Feature Extraction

We base feature extraction on symbolic execution techniques [8]. Unlike concrete execution, where a program is run on a specific input and a single control flow path is explored, the basic idea of symbolic execution is to allow variables to take on *symbolic values*, as well as concrete values. A symbolic value could either be an elementary symbolic value or an expression in numbers, arithmetic operators, and other symbolic values. An elementary symbolic value is any text string which the programmer uses to stand for the value of a variable. Elementary symbolic values are often just variable names.

```
public static void main(String[] args)
        throws Exception {
    final Class<?> testClass = Class.forName(args[0]);

    junit = new JUnitCore();
    request = Request.method(testClass, args[1]);
    Result result = junit.run(request);

    if(!result.wasSuccessful()) {
        System.out.println("Test failed.");
    }
}
```

Fig. 3: Hyperion Test Launcher.

This characteristic of symbolic variables allows the simultaneous exploration of multiple paths that a program can take under different inputs. Every time that some condition is checked against a symbolic variable, a *branch* is taken, in the sense that multiple control flows are maintained at the same time by the *symbolic execution engine*, effectively building a *symbolic execution tree*. In Hyperion, we use the Java Bytecode Symbolic Executor (JBSE) [39] as the symbolic execution engine. JBSE is a symbolic Java Virtual Machine able to deal with complex heap data structures.

At startup, we load all classes associated with test programs (from the `testPrograms` classpath list in the JSON configuration file), as well as all classes associated with the system under test (the `sut` classpath list). If additional classes are needed to run the application, their classpath can be listed in the `additionalClasspath` list. These paths will be included in the JBSE classpath, which enables lazy loading of classes on demand. In this way, JBSE can symbolically run all test programs, as we describe below.

We also use a form of *concolic execution* [27], which is essentially a "mixed" concrete/symbolic execution, to handle methods in charge of setting up the environment for a test program execution (e.g., `@Before` or `@BeforeClass` in JUnit), as well as multiple mocking frameworks [40] (e.g., `Mockito`). In particular, we instruct JBSE to rely on a "guided" execution: we launch an additional concrete JVM, starting from an artificial `main()` program (its code is reported in Figure 3), which launches one test program enumerated as per Section III-A by instantiating a `JUnitCore` object. By relying on the Java Debug Interface (JDI), we set a breakpoint on the method associated with the test program. Once that breakpoint is hit, JBSE takes back control and runs the same code—up to the breakpoint—by making decisions using the actual branches which were taken in the concrete execution. In this way, we can quickly reach the entry point of our test program, without having to explore execution paths which are not relevant for the extraction of similarity information. This activation scheme is repeated for each enumerated test program.

As already mentioned, we are interested in extracting information as general as possible, to support multiple decision strategies when similarity measures are constructed at a later stage. To this end, we inspect all symbolic execution states explored by JBSE, and we focus only on the states associated

```
1  filter(
2    InvokesLst,
3    invokes(testProgram,
4            branchingPointList,
5            seqNum,
6            caller,
7            programPoint,
8            frameEpoch,
9            pathCondition,
10           callee,
11           parameters),
12   [ isHttpMethod(callee) ],
13   [ testProgram,
14     method(caller),
15     httpMethod(callee,parameters),
16     head(parameters)    ],
17   endpoint,
18   EndpointLst)
```

Fig. 4: Prolog query to generate `endpoint` facts

with the invocation of some (local) method. We keep track of all invoked methods, in all explored branches, in an in-memory data structure. The traversed symbolic execution states associated with the invocation of methods are then used to generate the knowledge base for similarity analysis, based on Prolog rules.

C. Prolog facts generation

As mentioned before, the pieces of information that we are interested in collecting are related to methods invoked, and endpoints contacted by test programs. The knowledge base that we construct from symbolic execution is composed of different Prolog facts, each one describing one of the above features.

When the symbolic execution is completed, we dump to a file on disk a set of Prolog facts named `invokes`, constructed as follows:

```
invokes(TestProgram, BranchingPointList,
    SeqNum, Caller, ProgramPoint, FrameEpoch,
    PathCondition, Callee, Parameters)
```

An `invokes` fact describes that, while running a certain test program (denoted by `TestProgram`), a certain method (`Callee`) has been invoked with a certain set of parameters (`Parameters`). Given the symbolic nature of the execution, an invocation is bound to a certain symbolic execution path in the symbolic execution tree, which is identified by a list of branching points (`BranchingPointList`). To discriminate invocations that take place in iterations within the same symbolic execution path, we stamp each `invokes` fact with the `ProgramPoint`, i.e., the identification of a certain statement within the program's body, and a monotonic counter (`SeqNum`) which is incremented every time that a new symbolic state is observed in the symbolic execution. Similarly, to discriminate between different recursive invocations, we rely on the `FrameEpoch` monotonic counter, which is associated with the creation of each stack frame.

To deal with the generation of facts associated with endpoints remotely invoked by test programs, we rely on the Prolog predicate `filter` that can be used to query the database of `invokes` facts for generating the `endpoint` facts. Figure 4 shows the query we have used to trigger the generation process. The first parameter `InvokesLst` (line 2) is the list of `invokes` facts, whose structure is specified as second parameter (lines 3–11). The third parameter is the utility predicate `isHttpMethod(callee)` (line 12), which selects all `invokes` facts whose callees make use of an HTTP method to invoke a remote API. The fourth parameter is the list of fields to be extracted from the selected `invokes` facts, specifically: (1) the name of the test program `testProgram`, (2) the caller method `method(caller)`), (3) the HTTP method `httpMethod(callee,parameter)` (which occurs either as part of the callee's name or as a callee's parameter), and (4) the first parameters of the HTTP method `head(parameters)`, that is, the URI of the remote API. These fields become the arguments of the newly generated fact, called `endpoint` (line 17), which is added to the output list `EndpointLst` (line 18).

Overall, this leads to the generation of `endpoint` facts which are in the form:

```
endpoint(TestProgram, Caller, HTTPMethod, URI)
```

IV. Similarity Analysis

The knowledge base generated by the behavioral feature extraction process, that is, the `invokes` and `endpoint` facts, represents the *domain* of the *similarity relations* between test programs. We now present the Prolog rules [9] defining such relations, and we show how to use them to query the knowledge base for analyzing similarity of test programs.

We first introduce two basic notions defining the similarity between elements of the domain, that is, similarity between `invokes` facts and between `endpoint` facts.

Similarity between elements of the domain is evaluated by using the predicate `matching(Dom,O1,O2)` shown in Figure 5, where `Dom` defines the domain of the elements `O1` and `O2` (either `invokes` or `endpoint`) compared according to the definitions introduced above.

Given two `invokes` facts `I1` and `I2`, we say that they are similar if and only if (c1) `I1` invokes the same method of `I2` (line 6).

Given two `endpoint` facts `E1` and `E2`, we say that they are similar if and only if: (c2) they make use of the same HTTP method to invoke a remote API (line 14), and (c3) their URIs match (line 15). Matching between URIs is performed by using a list of regular expressions specified using the Perl-Compatible Regular Expression (PCRE) format, and extracted from the test suite. The use of regular expressions enables us to compare URIs while ignoring all information related to the call site that any URI may include.

Now, building upon the `matching` predicate, we can define the `similar_tp` predicate, which evaluates the similarity between two test programs.

The predicate in Figure 6 states that the test program `TP1` is similar to `TP2` according to the similarity criterion `SimCr` based on the matching of elements, belonging to the

```
% SEMANTICS: I1 and i2 are two invokes facts, and
% their callee arguments are equivalent (c1).
matching(invokes,I1,I2) :-
  I1 = invokes(_,_,_,_,_,_,Callee1,_),
  I2 = invokes(_,_,_,_,_,_,Callee2,_),
  Callee1 == Callee2.                    % (c1)

% SEMANTICS: E1 and E2 are two endpoint facts, and
% (c2) E1 and E2 perform the same HTTP request, and
% (c3) E1 and E2 match a common pattern
matching(endpoint,E1,E2) :-
  E1 = endpoint(_,_,HTTPMethod1,URI1),
  E2 = endpoint(_,_,HTTPMethod2,URI2),
  HTTPMethod1 == HTTPMethod2,            % (c2)
  matching_URIs(URI1,URI2).              % (c3)
```

Fig. 5: Prolog rules that define `matching(Dom,O1,O2)`.

```
similar_tp(Dom,DomSrc,SimCr,TP1,TP2,WT1,WT2,Score)
```

Fig. 6: Prolog rule that defines `similar_tp`.

domain `Dom`, generated during the feature extraction phase from the source `DomSrc`. In particular, if we specif "`trace`" for `DomSrc`, the elements of `Dom` are generated from the `invokes` facts occurring in symbolic execution traces; while if we specify "`iseq`" the elements are generated by using only the `invokes` occurring in traces whose caller is the method annotated as `@Test`. `WT1` and `WT2` are lists of elements in `Dom` that witness the similarity of `TP1` and `TP2`, and `Score` is a numeric value that quantifies the *degree of similarity* of `TP1` and `TP2`. Since the symbolic execution of a test program may generate several execution traces, for a pair $\langle$`TP1`,`TP2`$\rangle$ of test programs there may be several pairs $\langle$`WT1`,`WT2`$\rangle$ of witnesses, and hence several score values.

We have defined the following set-based similarity criteria (values for the `SimCr` parameter):

- "`nonemptyEqSet`" stating that `WT1` and `WT2` are nonempty lists, and all the elements in setOf(`WT1`) match an element of setOf(`WT2`) and vice-versa;
- "`nonemptySubSet`" stating that `WT1` is nonempty and all the elements in setOf(`WT1`) match an element of setOf(`WT2`);
- "`nonemptyIntersection`", stating that there exist two elements `O1` in `setOf(WT1)` and `O2` in setOf(`WT2`) such that `O1` matches `O2`;

where setOf(`L`) is the set of distinct elements in the list `L`. The value of *Score* is computed as follows:

- If `SimCr` is "`nonemptyEqSet`", then:

$$\texttt{Score} = 1$$

- If `SimCr` is "`nonemptySubSet`", then:

$$\texttt{Score} = \frac{|\text{setOf}(\texttt{WT1})|}{|\text{setOf}(\texttt{WT2})|}$$

- If `SimCr` is "`nonemptyIntersection`", then:

$$\texttt{Score} = \frac{|\text{matchingSet}(\texttt{WT1},\texttt{WT2})|}{\min(|\text{setOf}(\texttt{WT1})|, |\text{setOf}(\texttt{WT2})|)}$$

where matchingSet(`L1`,`L2`) is the nonempty set of all elements `O1` in setOf(`L1`) such that there exists an element `O2` in setOf(`L2`) for which the predicate `matching(O1,O2)` holds.

V. Experimental Case Study

We have evaluated the Hyperion framework by means of a case study based on a medium-size benchmark system. In particular, we have used a microservice-based application called *TrainTicket*[2] [41], which implements a system for railway ticketing. TrainTicket allows users to inquire about the train tickets between two cities on a certain day, to reserve tickets for a specific passenger on a specific class/seat, to pay for the reservations (and send the related confirmation email), and to manage ticket changes.

TrainTicket is currently composed of 43 total microservices, 38 of which are implemented in Java. These 38 microservices ship with a total of 682 test programs, implemented using JUnit 4. For the sake of readability of the results, we have performed experiments on a subset of the available test programs[3], excluding all the `invokes` of methods that belong to a base Java class (i.e., belonging to the `java` package), and to frameworks used to ease the development of the application (e.g., Spring). In this way, we only focus on the behavior of the target application. Using the Prolog facts generated by means of symbolic execution, we have computed our similarity scores for both the invoked methods and the contacted endpoints.

The similarity between test programs is based on the information extracted from their symbolic execution traces (see the definition of `similar_tp`). Hence, the evaluation of the similarity of each pair of test programs is based on the analysis of multiple execution traces generated from the symbolic execution of each test program. This means that, as already mentioned, for any similarity criterion discussed in Section IV, we may obtain more than one score value. Rather, we obtain a (possibly different) score value for each pair of symbolic execution traces. In the following, we report only the minimum score value for each pair of test programs deemed similar by a given criterion. Picking the minimum value allows us to compare the different criteria in a more stringent way, while comparing the similarity-detection capabilities of the presented criteria. Other choices would also be sensible, e.g., taking into account the maximum value, as discussed in previous work [10].

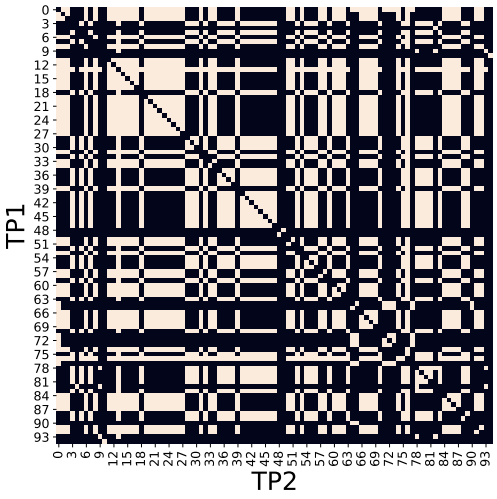
In Figure 7 we report similarity matrices (in the form of heatmaps) for all considered test programs[4]. The goal of this comparison is to show how the different criteria capture similarity with respect to the same test suite.

The first important aspect is the different cardinality of the sets of test programs which are deemed similar. In particular, conservative similarity criteria, such as `nonemptyEqSet`, consider as similar fewer test programs than more inclusive criteria such as `nonemptyIntersection` (as shown in

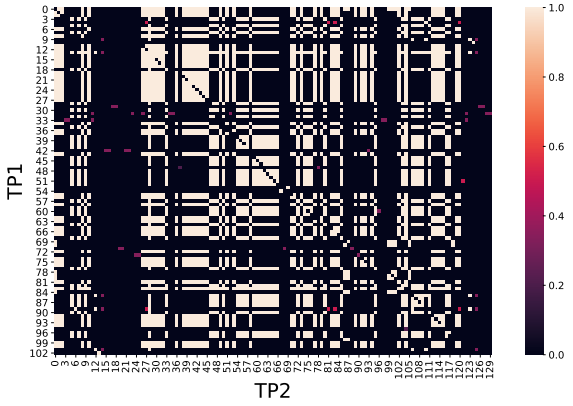
[2]https://github.com/FudanSELab/train-ticket.

[3]Details on the selected test programs are available online [42].

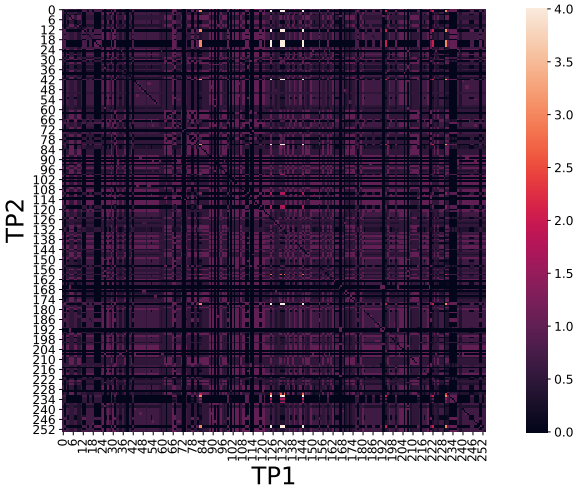
[4]To make the figures more readable, we have replaced test program names with numbers. The match with the test names in TrainTicket is available at [42].



(a) nonemptyEqSet criterion.

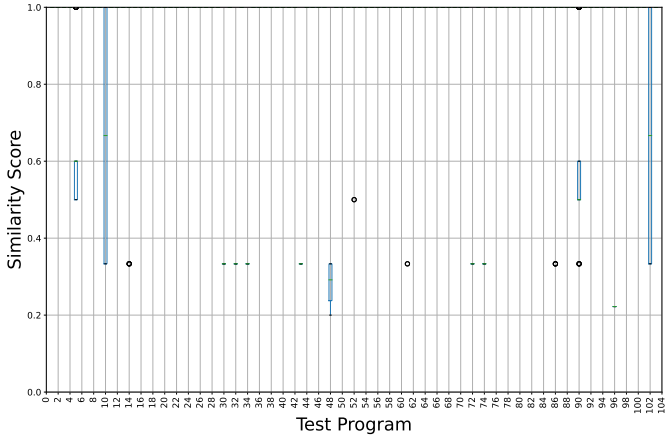


(b) nonemptySubSet criterion.

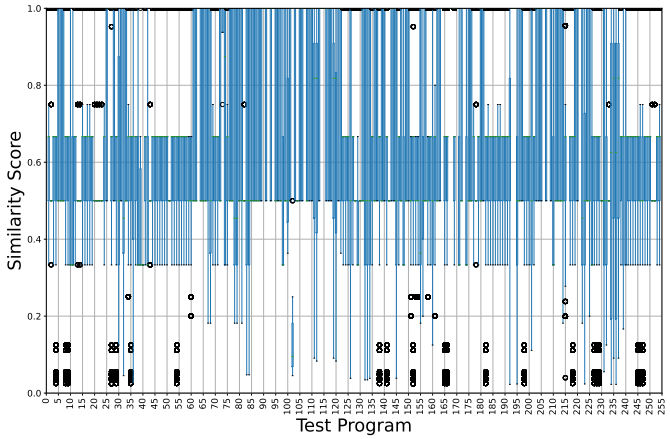


(c) nonemptyInterserction criterion.

Fig. 7: Similarity by the different criteria on invoked methods.



(a) nonemptySubSet criterion.



(b) nonemptyIntersection criterion.

Fig. 8: Range of similarity for test programs (on method invocation).

Figures 7a and 7c). Additionally, as expected from the definition of `nonemptySubSet`, we observe from the results that this criterion provides non-symmetric results, while the other two criteria's results are symmetric.

The values of the similarity scores obtained by the different criteria are also interesting to discuss. The `nonemptyEqSet` criterion (see Figure 7a) associates each pair of similar test programs with the value 1—in Figure 7a non-similar test programs are given the value 0. Therefore, this criterion behaves in a very selective way, deeming two test programs either as (fully) similar or not. In the test suite that we have used for our experimental case study, we anyhow find a non-marginal number of test programs that are considered similar by this criterion. This is related to the nature of TrainTicket's test programs, which are in a large number, and exercise many times the same portions of the system under test.

Conversely, the results shown in Figure 7b (associated with the `nonemptySubSet` criterion) provides a (small) number of intermediate similarity score values, while slightly increasing the number of test programs deemed similar, as mentioned before. At the same time, if we compare the results

presented in Figures 7a and 7b, we notice that many pairs have been evaluated as similar by the `nonemptyEqSet` similarity criterion. Indeed, this is expected by the definition of `nonemptyEqSet`, as every time that `nonemptyEqSet` assigns a score 1, so does `nonemptySubSet`. Nevertheless, `nonemptySubSet` is slightly more inclusive, and captures also the fact that some test programs are "not completely" similar, a notion that could be fruitfully exploited when prioritizing the execution of test suites.

The `nonemptyIntersection` similarity criterion relaxes this concept. From Figure 7c, we observe that this criterion is way more inclusive (i.e., the number of included test programs is significantly higher), and the assigned similarity scores are much finer, showing a richer range of values. When compared to, e.g., `nonemptyEqSet`, `nonemptyIntersection` tells that some pairs of test programs are also different, to some extent.

In Figure 8 we provide a different perspective on the presented similarity criteria, in the form of a box and whiskers plot. In particular, we show, for each test program, the variability in the similarity score with respect to the other programs in the test suite—we do not report the data for the `nonemptyEqSet` criterion because, as mentioned before, it cannot produce any variability in the score of similar test programs. By the results, we can confirm that `nonemptySubSet` provides more stable compared to `nonemptyIntersection`, meaning that `nonemptySubSet` is able to capture only stronger similarity between test programs.

Overall, the three criteria provide results which are comparably different. `nonemptyEqSet` is a stronger similarity criterion, which anyhow leaves out a large number of test programs from the suite. `nonemptyIntersection`, on the other hand, includes a larger number of test program, while being less "categorical" about the similarity between test programs. `nonemptySubSet` captures capabilities of both criteria. Overall, we advocate that all these three similarity criteria could be beneficial to testing campaigns, depending on the actual phase of the lifecycle of the application. As an example, when dealing with testing during feature development, the `nonemptyEqSet` criterion might help at determining what test programs to execute after a failure, to reduce the time to completion of the test suite—a test program similar to the failed one might be skipped. Conversely, the `nonemptyIntersection` criterion might help at determining what test programs could be run in parallel before releasing a new stable version of the application, e.g., in the possible attempt to detect reëntrance bugs—multiple test programs that invoke methods from the same package of the application might be run concurrently.

We finally discuss the results related to the invocation of endpoints. In our study, all three criteria have identified similarity in a negligible percentage out of the total number of test programs. This is not surprising, because the test suite in TrainTicket typically exercises different endpoints in different test programs. From this result, we conclude that the test suite of TrainTicket is well designed from a unit testing perspective, while it could be demanding from an integration/end-to-end testing point of view. We therefore believe that our approach could also be repurposed to study some quality metrics of test suites, although this specific goal might require the definition of additional and more comprehensive similarity criteria.

VI. Conclusions

In this paper we have presented an approach to the automatic extraction of behavioral features from test programs, with the goal of performing a similarity analysis. Our technique is based on a balanced mixture of symbolic execution and Prolog rules, which jointly allow the creation of a knowledge base of (possible) execution traces of the test programs and the evaluation of queries to determine similarity scores. The results presented in this paper support the claim that our approach is viable for multiple purposes related to both test suite prioritization, and assessment of the quality of test suites. As future work, we plan to substantiate this claim in practice by developing effective testing strategies that indeed exploit the knowledge about test program similarity.

Acknowledgments

We would like to thank the anonymous referees for useful comments.

References

[1] A. Vahabzadeh, A. Stocco, and A. Mesbah, "Fine-grained test minimization," in *Proc. of ICSE*, 2018, pp. 210–221.

[2] R. Kazmi, D. N. A. Jawawi, R. Mohamad, and I. Ghani, "Effective regression test case selection: A systematic literature review," *ACM Comput. Surv.*, vol. 50, no. 2, pp. 29:1–29:32, May 2017.

[3] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, and R. Tumeng, "Test case prioritization approaches in regression testing: A systematic literature review," *Information and Software Technology*, vol. 93, pp. 74 – 93, 2018.

[4] D. Paterson, J. Campos, R. Abreu, G. M. Kapfhammer, G. Fraser, and P. McMinn, "An empirical study on the use of defect prediction for test case prioritization," in *Proc. of ICST*. IEEE, 2019, pp. 346–357.

[5] L. Mariani, S. Papagiannakis, and M. Pezzè, "Compatibility and regression testing of cots-component-based software," in *Proc. of ICSE*. IEEE CS, 2007, pp. 85–95.

[6] T. Cerny, M. J. Donahoo, and M. Trnka, "Contextual understanding of microservice architecture: current and future directions," *ACM SIGAPP Applied Computing Review*, vol. 17, no. 4, pp. 29–45, 2018.

[7] D. Parsons, "Unit Testing with JUnit," in *Foundational Java*, ser. Texts in Computer Science. Springer Nature Switzerland AG 2020, 2020, ch. 10, pp. 251–278.

[8] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 50:1–50:39, 2018.

[9] L. S. Sterling and E. Shapiro, *The Art of Prolog*. The MIT Press, 1994, second Edition.

[10] E. De Angelis, G. De Angelis, A. Pellegrini, and M. Proietti, "Inferring relations among test programs in microservices applications," in *Proceedings of the 15th IEEE International Conference on Service Oriented Systems Engineering*. IEEE, Aug. 2021.

[11] A. Walenstein, M. El-Ramly, J. R. Cordy, W. S. Evans, K. Mahdavi, M. Pizka, G. Ramalingam, and J. W. v. Gudenberg, "Similarity in Programs," in *Duplication, Redundancy, and Similarity in Software*, ser. Dagstuhl Seminar Proceedings, R. Koschke, E. Merlo, and A. Walenstein, Eds. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[12] A. Sheneamer and J. Kalita, "A Survey of Software Clone Detection Techniques," *International Journal of Computer Applications*, vol. 137, no. 10, pp. 1–21, 3 2016.

[13] T. Lancaster and F. Culwin, "A Comparison of Source Code Plagiarism Detection Engines," *Computer Science Education*, vol. 14, no. 2, pp. 101–112, 6 2004.

[14] W. S. Evans and C. W. Fraser, "Grammar-based compression of interpreted code," *Communications of the ACM*, vol. 46, no. 8, pp. 61–66, 8 2003.

[15] T. B. Noor and H. Hemmati, "A similarity-based approach for test case prioritization using historical failure data," in *Proceedings of the 26th International Symposium on Software Reliability Engineering*, ser. ISSRE. IEEE, 11 2015, pp. 58–68.

[16] R. Wang, S. Jiang, and D. Chen, "Similarity-based regression test case prioritization," in *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*, ser. SEKE. KSI Research Inc., 2015.

[17] D. Hao, L. Zhang, Y. Pan, H. Mei, and J. Sun, "On similarity-awareness in testing-based fault localization," *Automated Software Engineering*, vol. 15, no. 2, pp. 207–249, 6 2008.

[18] Y. Ledru, A. Petrenko, S. Boroday, and N. Mandran, "Prioritizing test cases with string distances," *Autom. Softw. Eng.*, vol. 19, no. 1, pp. 65–95, 2012.

[19] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, "FAST approaches to scalable similarity-based test case prioritization," in *Proceedings of the 40th International Conference on Software Engineering*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 222–232.

[20] D. Raman, B. Bezawada, T. V. Rajinikanth, and S. Sathyanarayan, "Static Program Behavior Tracing for Program Similarity Quantification," in *Proceedings of the First International Conference on Computational Intelligence and Informatics*, ser. Advances in Intelligent Systems and Computing (AISC), S. C. Satapathy, V. K. Prasad, B. P. Rani, S. K. Udgata, and K. S. Raju, Eds. Springer, Singapore, 2017, vol. 507, pp. 321–330.

[21] S. Wang and D. Wu, "In-memory fuzzing for binary code similarity analysis," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE. IEEE, 10 2017, pp. 319–330.

[22] C. Meudec, "ATGen: automatic test data generation using constraint logic programming and symbolic execution†," *Software Testing, Verification and Reliability*, vol. 11, no. 2, pp. 81–96, 2001.

[23] W. Visser, C. S. Pasareanu, and S. Khurshid, "Test input generation with java pathfinder," in *Proc. of ISSTA*. ACM, 2004, pp. 97–107.

[24] N. Williams, B. Marre, P. Mouy, and M. Roger, "PathCrawler: Automatic generation of path tests by combining static and dynamic analysis," in *Proc. of EDCC*, ser. LNCS, vol. 3463. Springer, 2005, pp. 281–292.

[25] C. Cadar and D. R. Engler, "Execution generated test cases: How to make systems code crash itself," in *Proc. of Int. Workshop SPIN*, ser. LNCS, vol. 3639. Springer, 2005, pp. 2–23.

[26] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in *Proc. of PLDI*. ACM, 2005, pp. 213–223.

[27] K. Sen, "Concolic testing," in *Proc. of ASE*. ACM, 2007, p. 571–572.

[28] E. Albert, I. Cabanas, A. Flores-Montoya, M. Gomez-Zamalloa, and S. Gutierrez, "jPET: An automatic test-case generator for java," in *Working Conference on Reverse Engineering*, 2011, pp. 441–442.

[29] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[30] P. Braione, G. Denaro, A. Mattavelli, and M. Pezzè, "SUSHI," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. New York, NY, USA: ACM, 5 2018, pp. 21–24.

[31] K. Nivethithaa and V. Krishnapriya, "A Brief Survey on Symbolic Execution Test-Selection Techniques," *International Journal of Computer Sciences and Engineering*, vol. 06, no. 08, pp. 81–85, 10 2018.

[32] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT — a formal system for testing and debugging programs by symbolic execution," in *Proc. of the Int. Conf. on Reliable Software*. ACM, 1975, p. 234–245.

[33] J. C. King, "A new approach to program testing," *SIGPLAN Not.*, vol. 10, no. 6, p. 228–233, Apr. 1975.

[34] A. Armando, J. Mantovani, and L. Platania, "Bounded model checking of software using SMT solvers instead of SAT solvers," *Int. J. Softw. Tools Technol. Transf.*, vol. 11, no. 1, pp. 69–83, 2009.

[35] D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich, "Automating regression verification," in *Proc. of ASE*, 2014, pp. 349–360.

[36] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti, "Relational verification through horn clause transformation," in *Proc. of SAS*, ser. LNCS, vol. 9837. Springer, 2016, pp. 147–169.

[37] P. Collingbourne, C. Cadar, and P. H. J. Kelly, "Symbolic crosschecking of data-parallel floating-point code," *IEEE Trans. Software Eng.*, vol. 40, no. 7, pp. 710–737, 2014.

[38] T. Clemson, "Testing strategies in a microservice architecture," Nov. 2014.

[39] P. Braione, G. Denaro, and M. Pezzè, "JBSE: a symbolic executor for Java programs with complex heap inputs," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 11 2016, pp. 1018–1022.

[40] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli, "Mock objects for testing java systems: Why and how developers use them, and how they evolve," *Empirical Software Engineering*, vol. 24, no. 3, pp. 1461–1498, 6 2019.

[41] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao, "Benchmarking microservice systems for software engineering research," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 323–324.

[42] E. De Angelis, A. Pellegrini, and M. Proietti, "Automatic Extraction of Behavioral Features for Test Program Similarity Analysis: Results Name Matches. [Online]. Available: https://doi.org/10.5281/zenodo.5161585." Aug. 2021.