Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

What makes test programs similar in microservices applications?[☆]

Emanuele De Angelis [a,*], Guglielmo De Angelis [a,*], Alessandro Pellegrini [b,a,*], Maurizio Proietti [a,*]

[a] IASI-CNR, Rome, Italy
[b] University of Rome Tor Vergata, Rome, Italy

A R T I C L E I N F O

Article history:
Received 27 July 2022
Received in revised form 5 February 2023
Accepted 7 March 2023
Available online 13 March 2023

Dataset link: https://github.com/IASI-SAKS/hyperion/releases/tag/journal

Keywords:
Software testing
Microservices architecture
Test program similarity
Symbolic execution
Program instrumentation
Automated reasoning

A B S T R A C T

The emergence of microservices architecture calls for novel methodologies and technological frameworks that support the design, development, and maintenance of applications structured according to this new architectural style. In this paper, we consider the issue of designing suitable strategies for the governance of testing activities within the microservices paradigm. We focus on the problem of discovering implicit relations between test programs that help to avoid re-running all the available test suites each time one of its constituents evolves. We propose a dynamic analysis technique and its supporting framework that collects information about the invocations of local and remote APIs. Information on test program execution is obtained in two ways: instrumenting the test program code or running a symbolic execution engine. The extracted information is processed by a rule-based automated reasoning engine, which infers implicit similarities among test programs. We show that our analysis technique can be used to support the reduction of test suites, and therefore has good application potential in the context of regression test optimisation. The proposed approach has been validated against two real-world microservices applications.

1. Introduction

The microservices architecture style consists in building a software application as a collection of distributed software units, each abiding by the single responsibility principle (Cerny et al., 2018). The functionalities offered by a microservice are supposed to be contained within clearly defined boundaries, encapsulating the implementation of atomic features in the considered domain (Lewis and Fowler, 2014). Also, the microservices architecture principles suggest a strong control of the coupling among software units, advocating the adoption of design solutions that mitigate the impact of the evolution of each microservice. In other words, going through the various life-cycle phases of each microservice (i.e., its design, development, deployment, or update) should require minimal (or even zero) coordination effort with the others, possibly limited to immediate dependencies.

Both the technical and the managerial independence of microservices should cope with a dynamic scenario for the development and maintenance of applications built within this paradigm: the evolution of one or more constituents could occur according to several governance schemata, opening to different degrees of challenges about the resulting system (Bertolino et al., 2019). In order to take full advantage of this architectural style, novel methodologies and new technological frameworks are needed for designing, developing, and maintaining microservices applications. Also, testing activities demand appropriate strategies and tools covering each test phase: from unit testing to integration, and contract testing, up to end-to-end testing (Clemson, 2014). In addition, the continuous evolution of any of the constituents suggests the establishment of procedures and resources ascertaining that changes have not caused novel and undesired issues. Across the different stages of testing activities, regression testing (Yoo and Harman, 2012) aims to guarantee that the changes introduced in a software module do not harm its behaviour or the one exposed by the whole software system.

In the case of governance of regression testing activities, several classes of approaches aim at preventing the *retest-all* strategy by: (i) skipping redundant test cases from the test suite (Vahabzadeh et al., 2018), or (ii) selecting some test cases (Kazmi et al., 2017), or (iii) prioritising those expected to yield earlier fault detection (Khatibsyarbini et al., 2018; Paterson et al., 2019). However, in most cases, these approaches require some knowledge about the considered set of microservices, their immediate dependencies, and their possible interactions. Unfortunately, the lack of detailed specifications for the considered microservices and, in some cases, the unavailability of the source code could

[☆] Editor: Jacopo Soldani.
[*] Corresponding authors.
E-mail addresses: emanuele.deangelis@iasi.cnr.it (E. De Angelis), guglielmo.deangelis@iasi.cnr.it (G. De Angelis), a.pellegrini@ing.uniroma2.it (A. Pellegrini), maurizio.proietti@iasi.cnr.it (M. Proietti).

https://doi.org/10.1016/j.jss.2023.111674

hamper the direct application of such testing techniques (Mariani et al., 2007).

In addition, regression testing activities have to cope with the maintenance and the evolution of the regression test suites (Harrold and Orso, 2008): augmenting their significance by deriving new test cases from existing ones or by inferring a better understanding of the considered software system by leveraging pieces of evidence from the test cases. In this respect, the observation of the actual interactions among microservices instances can be exploited as a means of contributing to the evolution of regression testing test suite (Gazzola et al., 2020). Also, test cases have been proposed as viable solutions for checking compliance of contracts across service releases (Bruno et al., 2005).

This work contributes to the governance of regression testing in the specific context of microservices applications. One relevant piece of information often useful when designing regression testing strategies is the similarity between test cases. For example, test cases could be considered similar if they include the same activities but focus on a different testing strategy; if they target the same testing goal and strategy but use different test data; or if parametric tests have significant overlaps for some values of the parameters. Inferring such relationships is a complex task in the general case, as they strongly depend on the specific nature of the considered software system (e.g., application domain, referred architectural style, adopted technologies). As detailed in the following, this work leverages the specificity of the microservices paradigm in order to structure similarity information retrieval procedures that enable reasoning about test program similarities. Then, the knowledge of test case similarities allows the design of flexible regression testing strategies and policies, which avoid rerunning all test programs in an order fixed in advance.

Specifically, this work assumes that a set of test programs for a given microservices application is available because they are shipped with the microservices, or some system integrator made them available (e.g., contract tests for microservices that are commonly used together), or the integrator of the overall application provides them. Then, we propose a dynamic analysis of the given test programs to discover suitable similarities among them. Our analysis relies on both instrumented and symbolic execution techniques (Cadar and Sen, 2013) to gather information about the behaviour of a test program and the interactions it establishes among the microservices in the application under test. While the instrumented execution allows us to collect the trace of one concrete execution quickly, the symbolic approach allows the exploration of sets of concrete executions and allows us to handle parametric tests naturally. The information extracted is processed using logic-based reasoning techniques (Wielemaker et al., 2012) in order to establish similarity relations.

In order to evaluate our approach, we have implemented our analysis and reasoning techniques on a tool, called Hyperion, which is publicly available as open-source software.[1] Then, we have worked out two real-world case studies and we have shown that Hyperion is indeed capable of discovering similarities between sets of test programs, according to the various criteria we have defined. Our results also show that the similarities discovered can be used profitably, for example, to reduce the test case suite, and thus our approach has good potential for use in regression test optimisation. However, providing full-blown strategies for regression test optimisation, e.g., test case prioritisation, minimisation or selection, is beyond the scope of this paper.

This paper builds on the results presented in previous papers (De Angelis et al., 2021; De Angelis et al., 2021) and extends them in several ways. In particular, a first extension concerns the definition of a new set of similarity metrics: the work in De Angelis et al. (2021) does not concern similarity metrics, while the work in De Angelis et al. (2021) only refers to a collection of set-based similarity criteria; this work also defines and implements several sequence-based similarity criteria. Furthermore, both previous works only refer to scenarios where the similarities are computed starting from the symbolic execution of the test programs; in this work, we have extended both the methodology and its supporting framework to evaluate similarities starting from the concrete execution of the test programs. Finally, this work also enhances the validation of the proposed approach by performing, as mentioned above, an empirical evaluation on two popular open-source applications designed according to the microservices architectural style.

The rest of the paper is organised as follows. Section 2 provides some background about the main techniques used in this work. In Section 3 we present our overall approach to extracting relations among test programs from their concrete or symbolic executions. In Section 4 we describe the technique used to extract information from test programs, while in Section 5 we introduce the rules to determine their similarity. Section 6 describes the validation methodology that we have applied in the empirical study reported in Section 7. In Section 8 we comment on the threats to the validity of the empirical evaluation of our technique. Section 9 discusses related work and, finally, Section 10 draws the conclusions of this work.

2. Background

This section recalls some background notions about symbolic execution, software instrumentation, and logic programming, which are the core assets for the proposed contribution.

2.1. Symbolic execution

Symbolic execution (Baldoni et al., 2018) is an established technique in automated software testing (Cadar and Sen, 2013) for exploring program executions in search for runs that lead to error states, that is, states where some specified conditions are violated. 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 *symbolic variables*, that is, variables that take on symbolic values, besides concrete values. The use of symbolic variables allows the simultaneous exploration of multiple paths a program can take under different inputs. Every time that some condition is checked against a symbolic variable, a *branch* is taken and alternative control flows are maintained simultaneously by the *symbolic execution engine*.

For clarity, let us consider the example code snippet in Fig. 1. Symbolic execution can effectively determine which inputs make the final assertion fail without having to enumerate the whole set of possible input values. Indeed, by relying on symbolic variables, one could reason on *classes of inputs*.

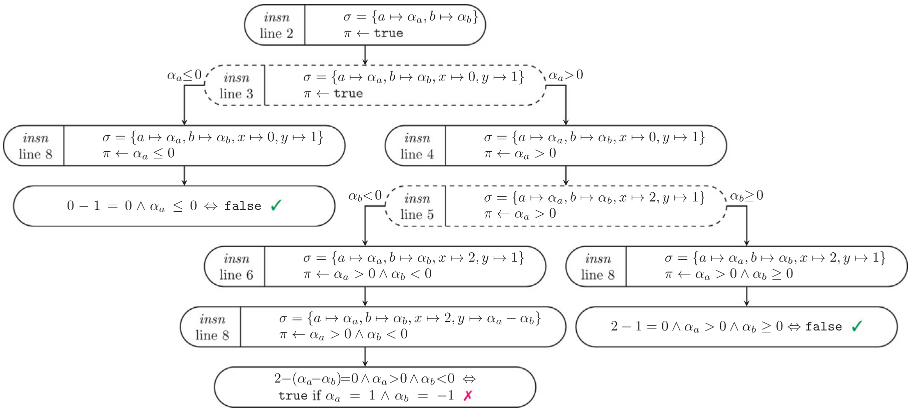
Every time a conditional branch instruction is symbolically executed, the symbolic execution engine creates a "snapshot" of the execution context up to that point. This snapshot can be used to backtrack to a previous execution state and restart the execution to explore alternative execution possibilities. Therefore, the overall symbolic execution of the program can be represented as a tree, where each conditional branch instruction generates two additional sub-trees describing the possible outcomes of the comparison.

The symbolic execution engine that supports the symbolic execution can be regarded as an abstract machine, which maintains a state represented by the triple $\langle insn, \sigma, \pi \rangle$, where:

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

```
1   public void foo(int a, int b) {
2       int x = 0, y = 1;
3       if(a > 0) {
4           x = 2 * y;
5           if(b < 0)
6               y = a - b;
7       }
8       assert(x - y != 0);
9   }
```

Fig. 1. Which values of `a` and `b` make the `assert()` fail?



insn line 2 | $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b\}$ $\pi \leftarrow$ true
$\alpha_a \leq 0$
insn line 3 | $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 0, y \mapsto 1\}$ $\pi \leftarrow$ true
$\alpha_a > 0$
insn line 8 | $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 0, y \mapsto 1\}$ $\pi \leftarrow \alpha_a \leq 0$
insn line 4 | $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 0, y \mapsto 1\}$ $\pi \leftarrow \alpha_a > 0$
$0 - 1 = 0 \wedge \alpha_a \leq 0 \Leftrightarrow$ false ✓
$\alpha_b < 0$
insn line 5 | $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 2, y \mapsto 1\}$ $\pi \leftarrow \alpha_a > 0$
$\alpha_b \geq 0$
insn line 6 | $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 2, y \mapsto 1\}$ $\pi \leftarrow \alpha_a > 0 \wedge \alpha_b < 0$
insn line 8 | $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 2, y \mapsto 1\}$ $\pi \leftarrow \alpha_a > 0 \wedge \alpha_b \geq 0$
insn line 8 | $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 2, y \mapsto \alpha_a - \alpha_b\}$ $\pi \leftarrow \alpha_a > 0 \wedge \alpha_b < 0$
$2 - 1 = 0 \wedge \alpha_a > 0 \wedge \alpha_b \geq 0 \Leftrightarrow$ false ✓
$2 - (\alpha_a - \alpha_b) = 0 \wedge \alpha_a > 0 \wedge \alpha_b < 0 \Leftrightarrow$ true if $\alpha_a = 1 \wedge \alpha_b = -1$ ✗

Fig. 2. Symbolic execution tree of the example program in Fig. 1. Dashed boxes correspond to states in which a branch is taken. The leaf node marked with a ✗ is associated with a terminal state which violates the `assert()` in the example program.

insn is the program point that has been reached during the symbolic execution of the program;

σ is a symbolic memory store, associating variables with either expressions over concrete values or symbolic values α_i;

π is a first-order logic formula—the so-called *path condition*–i.e., a formula that expresses a set of constraints on the symbols α_i built during the execution of the branches observed up to *insn*.

Any branch instruction executed on a symbolic variable updates the path condition π, while assignments update the symbolic store σ. A Satisfiability Modulo Theories (SMT) solver checks whether there are any violations of the constraints along each explored path and if the path itself is feasible. The tree associated with the code example in Fig. 1 is shown in Fig. 2. In this figure, we can observe that multiple symbolic states are traversed to reach leaf states. In the leaves, the symbolic store is used to check whether or not the `assert` condition fails.

The fact that symbolic execution traverses *all* states $\langle insn, \sigma, \pi \rangle$ up to a certain program point opens up for further exploitation of this technique, which is the basis of this work. Indeed, while the original goal of symbolic execution was to explore all the possible execution paths to determine what inputs to a test program might generate an error condition, since we can *observe* all the states during the symbolic execution, we can extract information about *all* possible activities carried out by a *parametric* test program. This can also be done in terms of the parts of the system under test (i.e., SUT), which are exercised by the test program itself, independently of the concrete values passed as input to the test by the developers.

2.2. Software instrumentation

Another technique typically employed to support code coverage analysis is software instrumentation (Yang et al., 2009). In Java, code instrumentation is typically carried out either at the source code level or the bytecode level. For our purposes, we focus on the latter.

Bytecode instrumentation techniques can be broadly classified as *static*, or *dynamic* approaches. In static bytecode instrumentation, all instrumentation code (e.g., software probes to inspect the application's behaviour) is inserted in the application before the program starts execution. Typically, this approach causes less runtime overhead, as all the bytecode mangling is performed before the process is launched. The major drawback is that dynamically generated or loaded code cannot be instrumented.

Conversely, dynamic bytecode instrumentation is interleaved with the program's execution under instrumentation. Typically, this approach relies on an *instrumentation agent* that is invoked every time a new class is loaded. The agent can analyse the bytecode of the loaded class and augment the loaded bytecode with instrumentation code. The major benefit of this approach is that multiple agents can coexist, and typical support offered by the Java Virtual Machine allows all the available agents to chain the bytecode modification. Usually, this induces higher overhead (mainly at program startup) and may affect measurements due to the runtime instrumentation process. Dynamic instrumentation also has the additional benefit that only those classes being actually loaded are instrumented, while static instrumentation requires processing all the classes, even though some may not be executed in a given scenario.

Typically, both static and dynamic instrumentation rely on *bytecode engineering libraries*, such as ASM (Bruneton et al., 2002), or Javassist (Chiba, 2000).

2.3. Logic programming

We recall the basic concepts of logic programming that we will use to reason about the similarity of test programs.

The logic programming syntax builds upon *terms* and *statements*. A term is either a *variable*, a *constant*, or a *compound term*. A statement is either a *fact*, a *rule*, or a *query*.

A fact is used to state a relation among objects, and is represented as an atomic formula, that is, a predicate symbol of arity

$n \geq 0$ applied to n terms. A rule is an implication of the form `head :- body`, where: (i) `head` is an atomic formula representing the conclusion of the implication, (ii) '`:-`' denotes the (reverse) implication symbol $\leftarrow$, and (iii) `body` is a conjunction of atomic formulas representing the premise of the implication. A logic program consists of a set of facts and rules. A query is used to ask whether a relation among objects holds. Syntactically, queries are atomic formulas, like facts, but the usage context can distinguish them. Any answer to a query with free variables provides values for the variables that make the query a logical consequence of the logic program.

We use the Prolog programming language as a concrete realisation of the logic programming paradigm and the SWI-Prolog system (Wielemaker et al., 2012) as the reference implementation of Prolog. In presenting logic programs, we will follow the usual conventions of Prolog: variables begin with an uppercase letter, while constant, function, and predicate names begin with a lowercase letter.

3. Overall approach

Often Quality Assurance (QA) teams agree on policies and strategies for regression testing within a shared governance framework (Bertolino et al., 2019). Such a framework supports the decision process during the testing campaigns, for example, by guiding the activities that could support the root-cause analysis of issues that have been spotted.

The enforcement of specific decisions can be either planned in advance of the regression testing activities (e.g., test suite reduction, test case selection/prioritisation) or online through a test case orchestrator (García et al., 2018) that dynamically makes decisions on how the regression testing process proceeds by taking into consideration the actual outcomes resulting from the test cases executions. In both cases, the role of test suites dependencies/similarities can foster the definition of parallel, sequential, or alternative flows of test cases to be executed (Bertolino et al., 2019).

Different factors can lead to establishing dependencies across regression test cases. On the one hand, members of the QA team or even software developers could declare them either in the software specifications or in the configurations of the referred build automation frameworks. On the other hand, implicit similarities can also be drawn from available software artefacts (e.g., test programs) using some (semi-)automatic mining procedures. In this article, we focus on this latter scenario.

The microservices architectural style suggests the design of highly modular applications, where the responsibilities of each microservice, its boundaries, and its interconnections are clearly identifiable (Lewis and Fowler, 2014). Given a collection of integration test cases, their elements could be considered similar if they concern the same set of microservices. In addition, all the unitary tests for a given microservice ms_i could also be considered related to integration tests that involve ms_i: a failing integration test should also prompt to check whether any unit regression has occurred in the microservices it refers to. Criteria of this kind have been initially introduced in Bertolino et al. (2019) where the discussion also covered dependencies that could be established at all the test levels (e.g., contracts, end-to-end). In the following, test programs for microservices applications are considered "similar" if they:

1. involve the same microservice instance, or they connect to the same remote API;
2. locally activate overlapping APIs (i.e., they refer similar local modules/libraries).

Microservices are distributed components whose interactions occur across some abstraction of the network interface and, in most cases, abide by the REST architectural style (Cerny et al., 2018). Test programs opening connections against the same remote APIs act as test drivers for the same type of microservices or, in some cases, among the same instances. Such connections to remote APIs can be directly coded in the test program employing basic frameworks that provide functionalities for accessing resources via HTTP (e.g., the HTTP Clients in the JDK or Apache libraries). Also, their implementation could be mediated by means of some structured application framework (e.g., Spring[2] or Postman[3]), or even mediated by means of some local libraries automatically generated starting from the remote APIs specifications (e.g., the client SDKs generated by means of Swagger Codegen[4]). This last technological solution opens the possibility of looking for similarities among those test programs that locally activate overlapping APIs. In addition, item 2 is also considered useful when looking for test programs that converge onto a cohesive set of activities: for testing purposes, but also for the configuration of the test environment or their referred assertions.

Our overall approach is depicted in Fig. 3. Identifying similarities among test programs is guided by an automatic analysis of their implementations and executions, which does not rely on any specification of the tests. The analysis procedure assumes that test programs are clearly identifiable from the rest of the source code or compiled classes, for example, through explicit JUnit annotations. Also, it is based on two different modes of test program execution: concrete or symbolic.

When the former mode is enabled, the analysis procedure runs each available test program and, through program instrumentation, automatically records the API that the test program locally activates. This mode aims to extract implicit dependencies across test programs that are actually coded in their implementations or due to specific values associated with the test program's arguments (see Fig. 3(a)). By enabling the latter, the analysis procedure still runs the available test programs, but it can also be configured to automatically switch their execution to symbolic processing (see Fig. 3(b)). The aim of this mode is twofold: (i) to carve test data by exploring admissible but not explicitly coded executions that are subsumed by the test program, and (ii) to exercise (parametric) test programs independently of their arguments.

The outcome of the execution phase is a knowledge base consisting of facts representing the configurations reached by either concrete or symbolic execution. Then, in a subsequent phase, the knowledge base is analysed to reveal existing similarities among test programs (see Fig. 3(c)). Specifically, this second phase builds upon a given set of inference rules that define similarity criteria among test programs, and whose evaluation is performed by querying the knowledge base generated during the execution phase.

An initial set of inference rules has been investigated in De Angelis et al. (2021) and then extended in De Angelis et al. (2021). Even though the proposed approaches are modular enough to cover additional dependency criteria, in this work, we have further improved the existing inference rules and validated them by means of broader experimental activities that cover outcomes of both the concrete and symbolic modes of the analysis procedure.

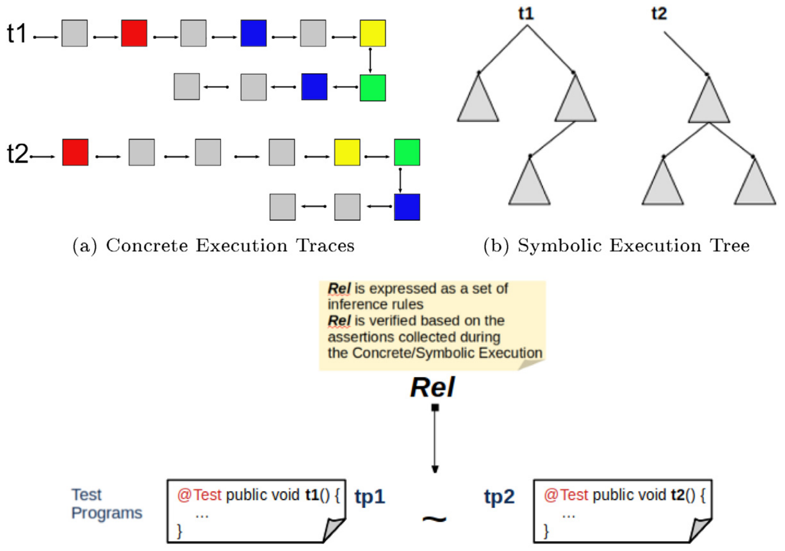
4. Carving behavioural features from test programs

This section details the methodology used to represent the information carved from test programs and the techniques we have explored to generate such data, namely symbolic and concrete execution. We also discuss our reference implementations.

[2] See: https://spring.io/microservices.

[3] See: https://www.postman.com.

[4] See: https://swagger.io/tools/swagger-codegen/.


t1
t2
t1
t2
Rel is expressed as a set of inference rules
Rel is verified based on the assertions collected during the Concrete/Symbolic Execution
Rel
Test Programs
@Test public void t1() {
...
}
tp1
~
tp2
@Test public void t2() {
...
}

(a) Concrete Execution Traces (b) Symbolic Execution Tree

(c) Analysis of Test Programs

Fig. 3. Overall approach.

4.1. Program traces representation

In order to identify the behavioural features of test programs for subsequent similarity characterisation, we must first be able to represent the execution trace of these programs.

Our representation is based on a set of Prolog facts that represent an invocation of a particular method within a specific class from a certain caller, then enabling suitable rule-based reasoning techniques. The simplest format of these facts is the following:

```
invokes(TestProgram, Caller, Callee)
```

where `invokes` is the predicate name, `TestProgram` is a unique identifier of the test program in the currently-analysed test suite, `Caller` is the invoking method, and `Callee` is the invoked method.

With this format, we are not explicitly considering the twofold nature of carving methods. In particular, when generating facts from a symbolic execution, we must keep track of the point where one specific invocation was observed in the symbolic execution tree. Conversely, we only refer to a single execution trace in a concrete execution. In other words, the facts must maintain the information that, in symbolic execution, they describe the *possibility* that, for specific concrete inputs to the test program, a particular method invocation could be materialised in a concrete execution.

To this end, we introduce a list of *branching points*, which are a linear representation of a path in the symbolic execution tree. These facts thus become:

```
invokes(TestProgram, BranchingPointList, Caller, Callee)
```

Of course, in the case of a concrete execution, `BranchingPointList` will be set to a placeholder value identifying that only a single execution trace has been generated, meaning that no branching was observed nor relevant.

To understand whether this format is sufficient to represent an execution trace to carry out a behavioural analysis, let us now consider the example code snippet in Listing 1. Here, we find repeated invocations to `g()`, which will in turn generate multiple `invokes` facts, as shown in Listing 2—`[1]` is used as `BranchingPointList` to indicate that the example in the listing refers to a single execution. The `invokes` in the figure are exactly the same, but one could argue that every single `invokes` fact generated by a call to `g()` is a different incarnation and should be therefore considered different. To enforce this difference, we extend the form of the `invokes` facts as follows: where `SeqNum` is a monotonic counter which is incremented every time that an `invokes` fact is generated. Therefore, every invocation of `g()` in the example in Listing 1 will bear a different value for `SeqNum`, thus allowing us to disambiguate invocations within iterations.

```
public void f() {
   for(int i = 0; i < 5; i++)
      g();
}
```

Listing 1: A function with calls within a **for** loop.

```
invokes(f, [1], f, g)
invokes(f, [1], f, g)
invokes(f, [1], f, g)
invokes(f, [1], f, g)
invokes(f, [1], f, g)
```

Listing 2: Sequence of `invokes` generated from Listing 1.

```
public void a(int count) {
    b();
    if(count > 0)
       a(0);
    c();
}
```

Listing 3: A recursive function.

Let us now consider the example shown in Listing 3. Here, a different set of methods is invoked depending on the (either concrete or symbolic) value of the method parameter `count`. If the example program is invoked as `a(1)`, a sequence of facts corresponding to the invocations of `b()`, `a(0)`, `b()`, `c()`, `c()` will be generated, all appearing as being called from `a()`. Here, the problem is that the same sequence of facts could also be

```
public void a(int count) {
    if (count == 0)
        return;
    b();
    a(0);
    b();
    c();
    c();
}
```

Listing 4: A fragment generating a sequence of `invokes` equivalent to that of Listing 3.

```
invokes(a, [1], 1, a, b)
invokes(a, [1], 2, a, a)
invokes(a, [1], 3, a, b)
invokes(a, [1], 3, a, c)
invokes(a, [1], 3, a, c)
```

Listing 5: A sequence of `invokes` that can be associated with both Listing 3 and Listing 4.

generated by the example program in Listing 4. In particular, both programs would generate the sequence of `invokes` shown in Listing 5. The two programs are inherently different though, and should not be described by the very same sequence of `invokes`. While the example deals with a recursive invocation, we note that the same problem might arise with repeated invocations of the same method from the same caller.

This anomaly stems from two different issues. First, the `invokes` fact as defined above cannot distinguish between different invocation contexts. Second, the invocations are different because they come from two different places in the source programs. To overcome this limitation, we enhance the form of the `invokes` facts as follows:

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

where `ProgramPoint` is a unique identifier of the location of the method call in the original program (for example its line number in the original source file), and `FrameEpoch` is an additional monotonic counter handled as follows. Every time a method invocation occurs in the symbolic execution, this counter is incremented. The new value is then pushed onto a stack. Every `invokes` fact is annotated with the value associated with the caller, i.e., the second-to-top element on the stack. Every time a return instruction is symbolically executed, we pop the top element from the stack. In this way, recursive or repeated invocations will bring a different frame epoch for every called method. This construction allows us to mimic the behaviour of stack frames employed by computer architectures for the same purpose. Additionally, if a method is invoked from a different location in the original source, it will have a different `ProgramPoint` value. The resulting (different) invokes for the snippets Listing 3 and Listing 4 in are reported in Listing 6.

Finally, we might consider two invocations to the same method as similar if they have the same set of parameters—in the case of symbolic execution, the parameters might be symbolic as well. We note that in the microservices scenario we target, we are not interested in argument values (except for strings) but rather in parameter types. Similarly, discriminating whether two invocations are the same might entail considering also the symbolic path condition. To this end, the final incarnation of the `invokes` becomes:

```
Invokes for Listing 3:
        invokes(a, [1], 1, a, 2, 1, b)
        invokes(a, [1], 2, a, 4, 1, a)
        invokes(a, [1], 3, a, 2, 2, b)
        invokes(a, [1], 4, a, 5, 2, c)
        invokes(a, [1], 5, a, 5, 1, c)

Invokes for Listing 4:
        invokes(a, [1], 1, a, 4, 1, b)
        invokes(a, [1], 2, a, 5, 1, a)
        invokes(a, [1], 3, a, 6, 1, b)
        invokes(a, [1], 4, a, 7, 1, c)
        invokes(a, [1], 5, a, 7, 1, c)
```

Listing 6: Invokes discriminating Listing 3 and Listing 4.

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

Similarly to the `BranchingPoint` case, `PathCondition` is set to a *don't care value* if the facts are generated from a concrete execution.

4.2. Information extraction via symbolic execution

Symbolic execution is one of the two primary techniques we have considered to extract information from (parametric) test programs. Indeed, being able to observe all execution states across which symbolic execution transits allows us to extract a large amount of information associated with what methods of the SUT are used or, more in general, what parts of the SUT are exercised.

Our solution for extracting behavioural features is based on three main execution phases: (i) test program enumeration; (ii) feature extraction; (iii) Prolog facts generation, according to the format described in Section 4.1. In the following, we detail the methodological/technical organisation of these phases.

Test program enumeration The analysis technique is based on JUnit 4/5 annotations and is controlled by a JSON configuration file. This file enables the declaration of those paths to be scanned to find the compiled test classes. The JSON file's structure and the presentation of the configuration it admits are reported in the appendix.[5]

Feature extraction As the symbolic execution engine, we use the Java Bytecode Symbolic Executor (JBSE) (Braione et al., 2016). JBSE is a symbolic Java Virtual Machine that deals with complex heap data structures.

At startup, we load all classes associated with test programs declared in the JSON configuration file, all classes associated with the SUT, and all those additional classes are needed to run the application. These paths will be included in the JBSE classpath, enabling the lazy loading of classes on demand. In this way, JBSE can symbolically run all test programs, as we describe below.

To reduce the time required to perform the symbolic execution and focus only on the test programs, we use a form of *concolic execution* (Sen, 2007). It is essentially a "mixed" concrete/symbolic execution which we use to quickly reach (in a concrete way) each test program's entry point, which is later executed symbolically. This way, we do not explore parts of the execution irrelevant for extracting similarity information, such as those in charge of setting up the environment for a test program execution (e.g., `@Before` or `@BeforeClass` in JUnit), as well as

[5] The appendix has been submitted in a separate file as "Supplementary material for on-line publication only".

related to multiple mocking frameworks (Spadini et al., 2018) (e.g., Mockito).

As already mentioned, we are interested in extracting general information 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 with the invocation of some (local) method. We keep track of all invoked methods, in all explored branches, in an in-memory data structure.

Prolog facts generation When the symbolic execution is completed, we dump a set of `invokes` Prolog facts to a file on disk. These facts are easily derived from the in-memory representation of the symbolic states of interest.

4.3. Information extraction via concrete execution

To collect behavioural information in a concrete execution, we rely on the *Java Agent* technology for the byte-code inspection and manipulation. Specifically, we developed a Java Agent in Javassist, directly attached to the JUnit run, relying on the Maven Surefire plugin. The very first time a class is loaded in memory, we check if it should be instrumented (i.e., it belongs to the test program or the SUT). Thus, the Java Agent injects tracing probes in some specific points of interest.

Specifically, the instrumented methods include a combination of activities performed just after its invocation and before it returns to the caller. These activities allow us to build an in-memory representation of the Prolog facts described in Section 4.1 and also consider the specific test programs that originated the call. Further operative information about the implementation is available in the appendix[5].

4.4. Prolog facts processing

The information extraction phase, obtained through either symbolic or concrete execution, generates a knowledge base consisting of `invokes` facts that is used to carry out automated logic-based reasoning to determine test program similarity, according to some (user-specified) criteria.

In order to analyse the sequence of methods executed by running a test program, that is, an *execution trace* of a test program, we provide the predicate `trace(TP,Trace)` (see Listing 7), which relates a test program TP to an execution trace `Trace` of the method annotated by `@Test` in TP, that is, the entry point of TP. The execution trace `Trace` is a list of `invokes`facts whose head Ep is the entry point of TP.

```
trace(TP,Trace) :-
  tp_entry_point(TP,Ep),
  trace_starting_with(Ep,Trace).
```

Listing 7: Prolog rule that defines `trace(TP,Trace)`.

Now, we can get the execution traces generated by executing a test program by collecting the answers to the query `trace(TP,Trace)`, where TP is bound to the test program name and `Trace` is an unbound variable, thereby getting values for `Trace` that can be further processed by using suitable helper predicates, and finally analysed to discover similarity relations between test programs. In particular, we provide the helper predicate `filter`, whose implementation details are reported in the appendix, which allows us to sieve through the `invokes` facts and reshape them into suitable data structures to be used within queries for reasoning about the test program similarity. Notably, in testing microservices applications, where we are interested only in analysing the similarity between test programs concerning their remote API invocations, the `filter` predicate allows us to perform the following operations: (1) select those `invokes` facts that represent invocations of methods belonging to remote APIs, (2) extract from the selected `invokes` facts specific information related to the remote API invocation, that is, the HTTP method used to perform the request (e.g., get and post) and its URI, and (3) generate new facts, called `endpoint`, with the following structure:

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

These facts showcase that the method `Caller` of the test program `TestProgram` invokes the remote API identified by `URI` using the HTTP method `HTTPMethod`.

In the appendix, we also report the query to perform operations (1)–(3).

5. Rules for similarity

We now present the Prolog rules defining *similarity relations* between test programs, and we show how to use them to query the knowledge base consisting of `invokes` and `endpoint` facts for inferring the similarity of test programs.

We start by introducing two basic notions defining the similarity between elements of the *domain*, that is, the similarity between `invokes` facts and between `endpoint` facts. The similarity between elements of the domain is evaluated by using the predicate `matching(Dom,O1,O2)` shown in Listing 8, where `Dom` defines the domain of the elements `O1` and `O2` (either `invokes` or `endpoint`) compared according to the definitions introduced as follows. 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 4). 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 8), and (c3) their URIs match (line 9).

Every occurrence of an anonymous variable '_' represents a distinct variable. It is used to denote any argument that is not taken into consideration for establishing the similarity between `invokes`(and between `endpoint`) facts.

```
1  matching(invokes,I1,I2) :-
2    I1 = invokes(_,_,_,_,_,_,_,Callee1,_),
3    I2 = invokes(_,_,_,_,_,_,_,Callee2,_),
4    Callee1 == Callee2.                         % (c1)
5  matching(endpoint,E1,E2) :-
6    E1 = endpoint(_,_,HTTPMethod1,URI1),
7    E2 = endpoint(_,_,HTTPMethod2,URI2),
8    HTTPMethod1 == HTTPMethod2,                 % (c2)
9    matching_URIs(URI1,URI2).                   % (c3)
```

Listing 8: Prolog rules that define `matching(Dom,O1,O2)`.

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

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

Listing 9: Prolog rule that defines `similar_tp`.

The predicate in Listing 9 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 domain `Dom`, generated during the feature extraction phase from the knowledge-base source `DomSrc`. In particular, if we specify the parameter `trace` for `DomSrc`, the elements of `Dom` are generated from the `invokes` facts occurring in execution traces. `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* between `TP1` and `TP2`.

Note that the execution of a test program based on symbolic execution may generate several execution traces, for a

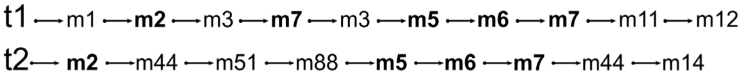

t1 → m1 → **m2** → m3 → **m7** → m3 → **m5** → **m6** → **m7** → m11 → m12
t2 → **m2** → m44 → m51 → m88 → **m5** → **m6** → **m7** → m44 → m14

Fig. 4. An example of concrete execution traces for the test programs t1 and t2.

$$\frac{|\{\mathrm{m2}, \mathrm{m7}, \mathrm{m5}, \mathrm{m6}\}|}{\min(|\{\mathrm{m1}, \mathrm{m2}, \mathrm{m3}, \mathrm{m7}, \mathrm{m5}, \mathrm{m6}, \mathrm{m11}, \mathrm{m12}\}|, |\{\mathrm{m2}, \mathrm{m44}, \mathrm{m51}, \mathrm{m88}, \mathrm{m5}, \mathrm{m6}, \mathrm{m7}, \mathrm{m14}\}|)} = 0.5$$

Box I.

Table 1
Values of Score for set-based SimCr similarity criteria. $\pi_{\mathtt{Dom}}$ is either (i) the function $\pi_{invokes}$ that, for any invokes fact i, returns the Callee argument of i, or (ii) the function $\pi_{endpoint}$ that, for any endpoint fact e, returns the pair $\langle m, re\rangle$ where m is the HTTPMethod argument of e and re is the regular expression accepting the URI argument of e.

SimCr	Score
nonemptyEqSet	1
nonemptySubSet	$\frac{\lvert\mathrm{setOf}(\mathtt{WT1}, \pi_{\mathtt{Dom}})\rvert}{\lvert\mathrm{setOf}(\mathtt{WT2}, \pi_{\mathtt{Dom}})\rvert}$
nonemptyIntersection	$\frac{\lvert\mathrm{setOf}(\mathtt{WT1}, \pi_{\mathtt{Dom}}) \cap \mathrm{setOf}(\mathtt{WT2}, \pi_{\mathtt{Dom}})\rvert}{\min(\lvert\mathrm{setOf}(\mathtt{WT1}, \pi_{\mathtt{Dom}})\rvert, \lvert\mathrm{setOf}(\mathtt{WT2}, \pi_{\mathtt{Dom}})\rvert)}$

Table 2
Values of Score for sequence-based SimCr similarity criteria. matchingSeq(L1,L2) is the longest non-empty list L3 such that the similarity criteria nonemptySubSeq holds between L1 and L3, and between L2 and L3.

SimCr	Score
nonemptyEqSeq	1
nonemptySubSeq	$\frac{\mathrm{length}(\mathtt{WT1})}{\mathrm{length}(\mathtt{WT2})}$
nonemptyCommonSeq	$\frac{\mathrm{length}(\mathrm{matchingSeq}(\mathtt{WT1}, \mathtt{WT2}))}{\min(\mathrm{length}(\mathtt{WT1}), \mathrm{length}(\mathtt{WT2}))}$

pair $\langle \mathtt{TP1}, \mathtt{TP2}\rangle$ of test programs there may be several pairs $\langle \mathtt{WT1}, \mathtt{WT2}\rangle$ of witnesses, and hence several score values.

We have defined several similarity criteria specified by means of the SimCr parameter of the similar_tp predicate. When defining the criterion SimCr, we will say that "the similarity criterion SimCr holds" as a shorthand for "the predicate similar_tp with similarity criterion SimCr holds".

First, we present the following set-based similarity criteria:

- nonemptyEqSet holds if WT1 and WT2 are nonempty lists and every element of WT1 matches an element of WT2 and vice-versa;
- nonemptySubSet holds if WT1 is a nonempty list and every element of WT1 matches an element of WT2;
- nonemptyIntersection holds if there exists an element of WT1 that matches an element of WT2.

The value of Score is 0 if the similarity criterion does not hold and, otherwise, it is a non-negative value computed as shown in Table 1, where setOf(L, $\pi_{\mathtt{Dom}}$) denotes the set $\{\pi_{\mathtt{Dom}}(o) \mid o$ is an element of L$\}$, for any function $\pi_{\mathtt{Dom}}$ on the domain Dom of the elements of list L.

As an example, let us consider the concrete execution traces of the two test programs t1 and t2 shown in Fig. 4. Specifically, both traces record (relevant) methods that have been invoked when executing the corresponding test programs and their relative invocation order. The methods occurring in the tails of lists starting with t1 and t2, respectively, represent the Callee arguments of the invokes facts. Given that among the methods invoked by t1 there is m1 that is not invoked by t2, and among the methods invoked by t2 there is m44 that is not invoked by t1, the similarity criteria nonemptyEqSet and nonemptySubSet between t1 and t2 do not hold. Conversely, the similarity criteria nonemptyIntersection holds because t1 and t2 have some method invocations in common, specifically they both invoke the methods m2, m7, m5, and m6. Therefore, the degree of similarity between t1 and t2 is as shown in Box I.

We have also defined the following sequence-based similarity criteria for a pair of nonempty lists WT1 and WT2 of the form $\langle a_1, \ldots, a_n\rangle$ and $\langle b_1, \ldots, b_m\rangle$, respectively:

- nonemptyEqSeq holds if $n = m$ and, for $i = 1, \ldots, n$, a_i matches b_i.
- nonemptySubSeq holds if $m \geq n$ and, by deleting zero or more elements from WT2, we get a list WT3 such that nonemptyEqSeq holds for WT1 and WT3;
- nonemptyCommonSeq holds if nonemptyIntersection holds.

Similarly to set-based criteria, the value of Score is 0 if the similarity criterion does not hold; otherwise, it is a non-negative value computed as shown in Table 2.

Let us consider again the execution traces shown in Fig. 3(a). Similarly to nonemptyEqSet and nonemptySubSet, the similarity criteria nonemptyEqSeq and nonemptySubSeq do not hold due to the presence of methods invoked by t1 that are not invoked by t2, and vice versa. However, by considering the similarity criterion nonemptyCommonSeq, which holds whenever the criterion nonemptyIntersection holds, the degree of similarity between t1 and t2 is as shown in Box II. In this section we have introduced various criteria that aim to evaluate the similarity of test programs by taking advantage of the information about the local and remote API methods they invoke. As mentioned in Section 3, by taking advantage of such information collected during the dynamic analysis of test programs, these criteria can contribute to defining flexible policies within a governance framework for regression testing. Notably, they can be used to select test programs useful to exercise the modified microservices component, and therefore to avoid rerunning all available test programs when a small component changes. In the next sections, we present the validation methodology and the experimental evaluation we have performed to study the performance of the proposed criteria in inferring the similarity among test programs that belong to two test suites.

6. Validation methodology

In the rest of this section, we first describe the research questions (RQs) that guide our validation methodology (see Section 6.1); then, Section 6.2 presents the two case studies we referred to in our study.

$$\frac{\mathrm{length}(\langle \mathrm{m5, m6, m7} \rangle)}{\min(\mathrm{length}(\langle \mathrm{m1, m2, m3, m7, m3, m5, m6, m7, m11, m12} \rangle),\ \mathrm{length}(\langle \mathrm{m2, m44, m51, m88, m5, m6, m7, m14, m44} \rangle))} = 0.33$$

Box II.

6.1. RQs, strategies, and methods

The following presents the RQs we set out to answer in this work. For each RQ, we report the strategy we followed in order to provide an answer and the method we planned to conduct the experimental studies.

RQ1: *Can implicit similarities extracted from test programs support decisions in the context of a governance framework for regression testing?* The behavioural features extracted from each test program represent a valuable source of information that can be exploited while making decisions during the regression testing activities. In this work, we propose a technique to analyse the overlaps (if any) that execution traces of two test programs reveal either during either concrete or symbolic executions. In answering RQ1, we aim to show that our technique is indeed capable of inferring similarities and differences between test programs, according to the criteria defined in Section 5, thus making this information available for practical use. In particular, when answering RQ3 defined below, we will argue that the similarity information has a very good potential to support test case reduction in microservices applications. However, as already mentioned, the design and implementation of specific techniques for optimising regression testing are beyond the scope of this paper.

RQ2: *How stable are the similarity criteria?* Implicit similarities among test programs are identified using logic reasoning on the key features carved from their execution traces. Given a set of test programs (i.e., *TS*) for a SUT, a similarity criterion can be used to group test programs in clusters according to an agreed minimum degree of similarity (i.e., s_{min}). We consider that a similarity criterion is stable if the clusters it defines are composed of *homogeneous* test programs: any test program taken from a cluster should be a sample that is good enough to represent all the other test programs in that cluster. In other words, given a threshold s_{min}, all the possible subsets of *TS* built from an arbitrary selection of one element per cluster should always provide comparable outcomes.

To answer RQ2, we planned the following strategy for each similarity criterion. First, we build a subset *TS-small* from *TS*. Specifically, we randomly pick a test program *t* from *TS* and add it to *TS-small* only if its similarity score with all the current elements in *TS-small* is always lower than s_{min}(i.e., *t* is different enough from the elements in *TS-small*). All the test programs in *TS* are considered just once. When this first phase ends, the resulting *TS-small* is run against the referenced SUT, and we register the observed coverage. Both phases are repeated multiple times to experiment with different selections of *TS-small* for the same similarity criterion. The analysis of the coverage data and their variance across several repetitions gives arguments to answer RQ2.

RQ3: *Can the similarity criteria impact the decisions about test suite reduction?* Once the stability of the similarity criteria has been addressed, we are interested in investigating if and how much each similarity criterion can contribute to a test suite selection policy. In other words, we are trying to estimate the quality grain of the proposed selection criteria.

In this sense, we use two software coverage metrics as a consolidated and widely used means of estimating fault detection capabilities: higher levels of code coverage correspond to (but do not guarantee) higher confidence in the ability to detect the presence of bugs. Thus, given a software system and two different test suites for it, the difference in the coverage scored by the test suites estimates their relative potential for defect detection.

In general, any test-suite reduction strategy impacts the SUT coverage: fewer test programs to execute can only decrease the coverage metrics. Our notion of quality is concerned with estimating the coverage drop caused by the selection criteria. We want to exclude as many test programs as possible from the execution, but with limited impact on the coverage of the SUT. In the following, we refer to the quality of a similarity criterion as its capability to define proper subsets of *TS* whose cardinality is smaller than the one achieved by a random selection of test programs in *TS* but resulting in the same coverage drop.

Notably, information on code coverage is of limited use if the aim is to detect defects earlier. However, since this work intends to discover potential similarities between test programs in a microservices application, the answer to RQ3 is limited to an analysis of the relative defect detection capabilities of the considered test suites. Furthermore, in the context of regression testing, it can be assumed that the available test suites are quite stable, while the SUT (frequently) evolves over time. Thus, the calculation of similarities between test programs and their analysis can be done once. The resulting results are expected to be valid until some element in the regression test suites changes. For this reason, the answer to RQ3 does not include any study of the cost of computing the similarities between test programs.

The method we plan to support our validation strategy is similar to the approach described for RQ2. Specifically, for each similarity criterion, we select a subset of test programs *TS-small* as described above,[6] and for all the resulting *TS-small* we compute their coverage of the referenced SUT. We repeat these phases several times and then calculate the mean coverage value and the mean number of selected test programs per similarity criterion.

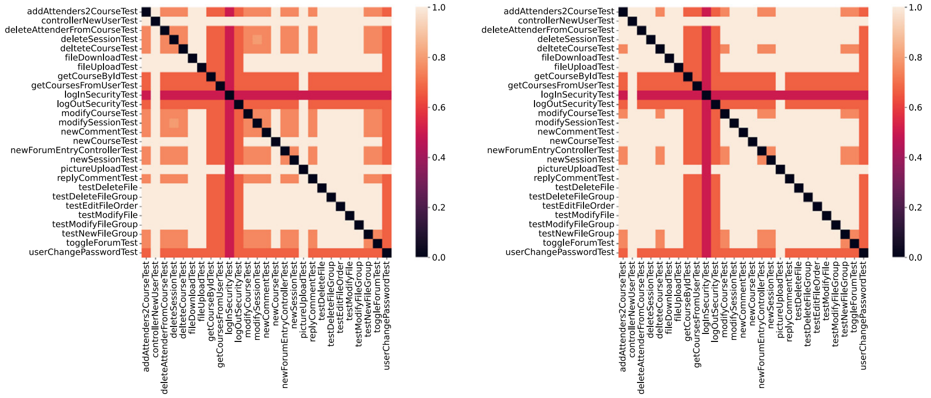
In addition, we also build random subsets of *TS*: we have precisely one random subset of *TS*(i.e., *TS-small-rnd*) for each cardinality between 1 (i.e., a selection with the maximum saving in terms of test programs to be executed) and the total number of test programs in *TS*(i.e., there is no selection and thus no saving). For all these *TS-small-rnd*s, we register their coverage of the SUT. We repeat this procedure several times to compute the mean coverage value expected by a defined-size random selection of test programs.

Thus, we finally answer RQ3 by analysing the mean coverage outcomes led by the similarity criteria and those resulting from the random selections. Specifically, having agreed on an acceptable coverage drop for running the whole *TS*, we compare the number of test programs in *TS-small* and those in *TS-small-rnd*.

6.2 Subjects

In this section, we introduce the subjects on which we conducted our study about test programs' similarity. The choice of benchmarks for our experimental study was guided by several factors. Firstly, we decided to use benchmarks from the research

[6] In the empirical evaluation, we have used the same subsets *TS-small* used for RQ2.



(a) Minimum Similarity Score.

(b) Maximum Similarity Score.

Fig. 5. *Subject*: Fullteaching; *Domain*: `endpoint`; *Criterion*: `nonemptyIntersection`.

community rather than building ad-hoc synthetic applications in order to avoid bias in the experiments. This made it possible to test our approach using applications built by third parties in a completely independent way of the testability purposes inherent in our proposal. Given the focus on microservices, we concentrated our selection on all applications implemented according to this paradigm. Furthermore, for technological reasons, we considered all and only those applications written in Java. Finally, the selection focused on applications that provided a test suite of non-minimal size, including integration and contract tests that exercised the microservices API. All these guiding factors led us to select two popular open-source applications, designed according to the microservice architectural style, against which we exercised the reference implementation of our proposal.

The first benchmark that we have used is Fullteaching,[7] an educational platform based upon OpenVidu, an open-source video conferencing system employing the WebRTC API (Johnston and Burnett, 2012). It provides a test suite implemented using JUnit 4, including 88 test programs. Among them, 29 tests require contacting remote URIs for integration or end-to-end testing purposes. These are the test programs used for our case study, as they involve invoking URIs using get, post, put, and delete methods. All RESTful requests are managed through the `MockMvc` class by the Spring framework[].

The second benchmark used in our study is a medium-size microservices application called TrainTicket,[8] which implements a system for railway ticketing. TrainTicket allows users to inquire about the train tickets between two cities on a certain day, reserve tickets for a specific passenger on a specific class/seat, pay for the reservations (and send the related confirmation email) and manage ticket changes.

TrainTicket comprises 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. All the test programs have been used in our empirical evaluation.

7 Empirical evaluation

In this section, we present the result of our empirical evaluation based on the applications described in Section 6. The reference implementation of our methodology has been embedded in the Hyperion tool, which is released as open-source software (see the section titled: “Replication Package and Data Availability”). We use the results to answer the RQs listed in Section 6.1.

7.1 Capability to detect similarity

By the rules discussed in Section 4.4, we have generated the `endpoint` facts that describe the URI(s) invoked by the test programs for both benchmark applications. An excerpt for the Fullteaching application is provided in Fig. 6, where we show a subset of the `endpoint` facts generated from the symbolic execution of the two test programs `modifySessionTest` and `deleteSessionTest`. These facts allow us to answer multiple queries, such as: “*which test programs invoke the* `/api-users/new` *endpoint?*”, or “*which test programs use the* `/api-courses/new` *RESTful API after* `/api-users/new`*?*”. In general, these facts prominently capture that a given test program (the first argument) invokes a certain URI (the fourth argument) with a given method (the third argument), which is fundamental for detecting similarity in the context of microservices applications.

To answer RQ1, we now consider the result of the similarity analysis using various criteria. We consider the results related to both method invocation and endpoint activation. In the case of TrainTicket, we are considering the complete test suite. We only discuss some exemplary results related to symbolic execution at this point because they enable us to consider a broader range of similarity values and explore additional analysis possibilities. The reader can find the results associated with all the combinations of metrics, domains, and carving techniques in the appendix of this article.

We present similarity results in the form of matrices (heat maps). The value of the similarity score is represented by a coloured cell for each test program pair. Regarding the Fullteaching application, in Fig. 5 we present the results related to the `nonemptyIntersection` metric evaluated over the `endpoint` domain. Since symbolic execution can extract multiple traces from the execution of a single test program, no single similarity score value can be associated with a pair of test programs. Therefore, we report in Fig. 5(a) and Fig. 5(b) the minimum and the maximum score values, respectively—the diagonal is zero in all cells, as we do not compute the similarity between a test program and itself. By construction, the similarity matrix for the `nonemptyIntersection` metric is symmetric.

To understand whether this information can be used effectively to detect implicit similarities between test programs in the context of a governance framework for regression testing, let us discuss some values related to Figs. 5(a) and 5(b). If we consider the test program `logInSecurityTest`, which tests the login capabilities of Fullteaching users, we observe that it is associated with a minimum/maximum similarity score of 0.5 with respect to all other test programs. The test programs we have taken into account are all associated with authenticated APIs: all test

[7] https://github.com/OpenVidu/full-teaching

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

```
endpoint('SessionControllerTest:modifySessionTest', 'registerUserIfNotExists', 'post', '/api-users/new').
endpoint('SessionControllerTest:modifySessionTest', 'createCourseIfNotExist', 'post', '/api-courses/new').
endpoint('SessionControllerTest:modifySessionTest', 'newSession', 'post', '/api-sessions/course/1').
endpoint('SessionControllerTest:modifySessionTest', 'modifySessionTest', 'put', '/api-sessions/edit').
endpoint('SessionControllerTest:deleteSessionTest', 'registerUserIfNotExists', 'post', '/api-users/new').
endpoint('SessionControllerTest:deleteSessionTest', 'logIn', 'get', '/api-logIn').
endpoint('SessionControllerTest:deleteSessionTest', 'createCourseIfNotExist', 'post', '/api-courses/new').
endpoint('SessionControllerTest:deleteSessionTest', 'newSession', 'post', '/api-sessions/course/1').
```

Fig. 6. Example of generated endpoint facts. *Subject*: Fullteaching.

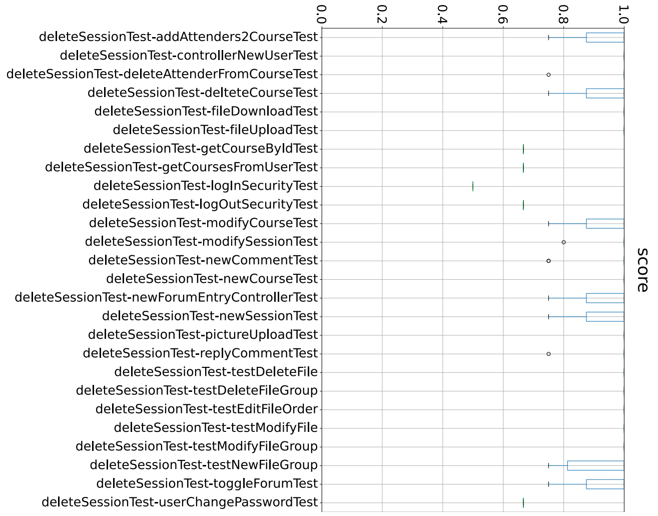


Fig. 7. Effect of multiple symbolic traces on pairs of test programs. *Subject*: Fullteaching; *Domain*: endpoint; *Criterion*: nonemptyIntersection.

programs try to create a user (if it does not exist), authenticate it, perform some action, and conclude the session. Therefore, logInSecurityTest's similarity score is stable compared to the other test programs, and it is set to a low value. In this sense, we cannot consider it significantly similar to other test programs.

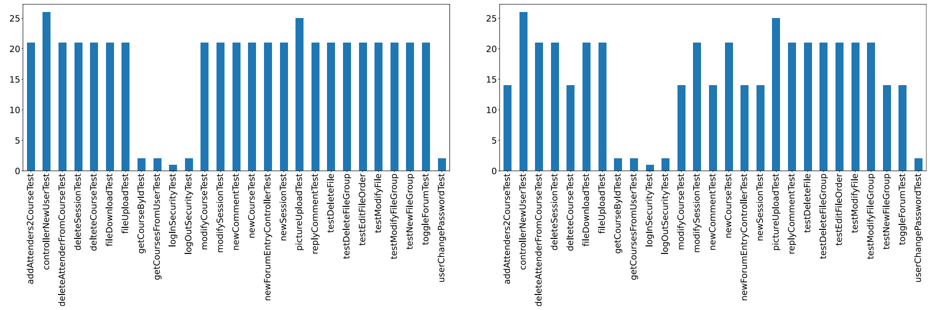
Let us now focus on the test program deleteSessionTest. If we compare the minimum and maximum scores against newCourseTest and getCourseByIdTest, we may try to answer the question: "*to which test is* deleteSessionTest *most similar?*". The pair deleteSessionTest-newCourseTest is associated with a minimum/maximum value of 1.0, while deleteSessionTest-getCourseByIdTest has a minimum/maximum value of 0.75. We might conclude that, as far as endpoint invocations are concerned, deleteSessionTest is more similar to newCourseTest than getCourseByIdTest. On the other hand, if we compare the values of the pairs deleteSessionTest-modifySessionTest and deleteSessionTest-getCourseByIdTest, the pair deleteSessionTest-modifySessionTest shows a minimum value of 0.75 and a maximum value of 1.0 (depending again on the multiple observed symbolic execution traces), while deleteSessionTest-getCourseByIdTest is stable at 0.67. In this case, we cannot conclude much on the similarity among deleteSessionTest, modifySessionTest, and getCourseByIdTest.

However, if we observe the results in Fig. 7, we can extract more information. In the figure, we have picked deleteSessionTest and displayed the dispersion of the similarity score compared to all the other test programs. By looking at these results, we might conclude that deleteSessionTest is more similar to modifySessionTest than getCourseByIdTest. Conversely, let us also consider newSessionTest. We might conclude that deleteSessionTest is more similar to newSessionTest than getCourseByIdTest, but not as much as we might imagine by looking at Fig. 5. It is also interesting to note that, for some pairs (e.g., deleteSessionTest-loginSecurityTest), there is no dispersion at all—this is also reflected in Fig. 5, where both the minimum and maximum values are the same. This phenomenon can be related to the fact that, in the symbolic execution tree, there is only one feasible path for the test program loginSecurityTest. In contrast, for other test programs, there are multiple execution traces to compare; therefore, different similarity scores are derived.

In Fig. 8, we show the number of test programs that can be deemed similar by relying on our metric. In particular, for each test program, we report the number of other test programs that have a median similarity score among all symbolic execution traces above 0.75 (Fig. 8(a)) and exactly 1.00 (Fig. 8(b)). As expected, the number of test programs deemed similar decreases for higher median values. This result is an additional indication of the versatility of our approach. Indeed, higher similarity score values might help define narrower governance policies that can be enforced to reduce regression test suites by selecting some test cases, skipping redundant ones, or prioritising those expected to yield earlier fault detection.

In Fig. 9 we report the similarity matrices (again, distinguishing between the minimum and maximum score values) for the nonemptySubSet criterion. As expected from the definition of nonemptySubSet, we observe from the results that this criterion provides non-symmetric results. The first important difference compared to the results in Fig. 5 is the different cardinality of the sets of test programs deemed similar. In particular, more conservative similarity criteria, such as nonemptySubSet, consider as similar fewer test programs than more inclusive criteria such as the aforementioned nonemptyIntersection.

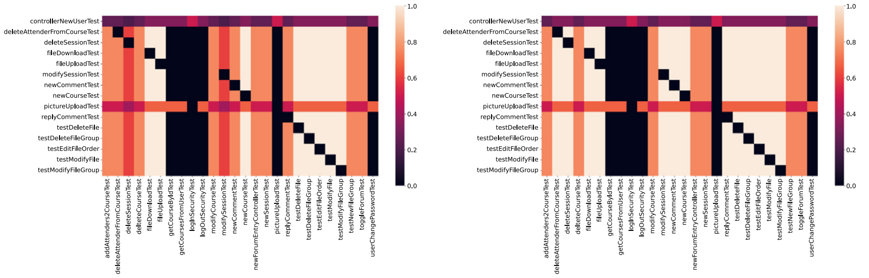
The values of the similarity scores obtained by the different criteria are also interesting to discuss. The nonemptyEqSet criterion (see Fig. 10) associates each pair of similar test programs with the value 1—non-similar test programs are not shown. Therefore, this criterion behaves very selectively, deeming two test programs either as (fully) similar or not. This criterion is even more selective than nonemptySubSet (fewer test programs are deemed similar). Yet, it is more difficult to discriminate the relative similarity between pairs of test programs due to the boolean nature of the similarity score. Conversely, the aforementioned nonemptySubSet criterion shows a (small) number of intermediate similarity score values, while slightly increasing the number of test programs deemed similar compared to nonemptyEqSet. If we compare the results in Figs. 9 and 10,



(a) Similarity Score ≥ 0.75.

(b) Similarity Score $= 1.00$.

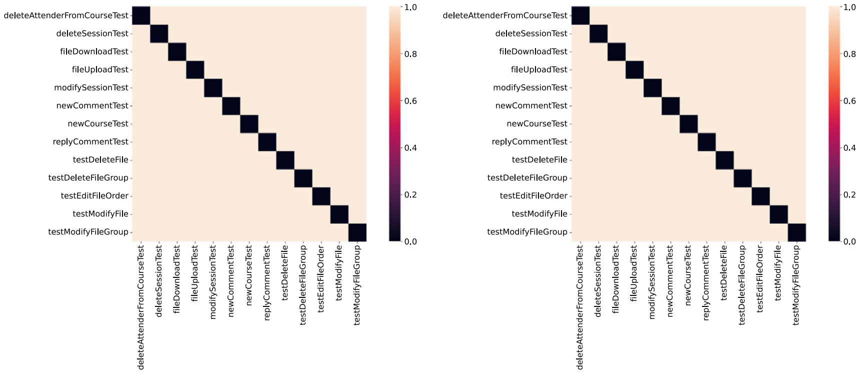
Fig. 8. Number of test programs deemed similar. *Subject*: Fullteaching; *Domain*: `endpoint`; *Criterion*: `nonemptyIntersection`.



(a) ft:endpoint-nonemptySubSet-min

(b) ft:endpoint-nonemptySubSet-max

Fig. 9. *Subject*: Fullteaching; *Domain*: `endpoint`; *Criterion*: `nonemptySubSet`.



(a) Minimum Similarity Score.

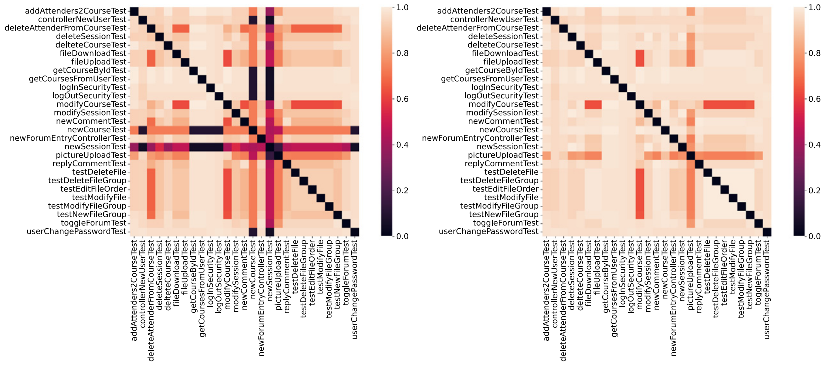
(b) ft:invokes-nonemptyEqSeq-max

Fig. 10. *Subject*: Fullteaching; *Domain*: `invokes`; *Criterion*: `nonemptyEqSeq`.

we notice that many pairs have been evaluated as similar also by the `nonemptyEqSet` similarity criterion with the same score. 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 prioritising the execution of test programs.

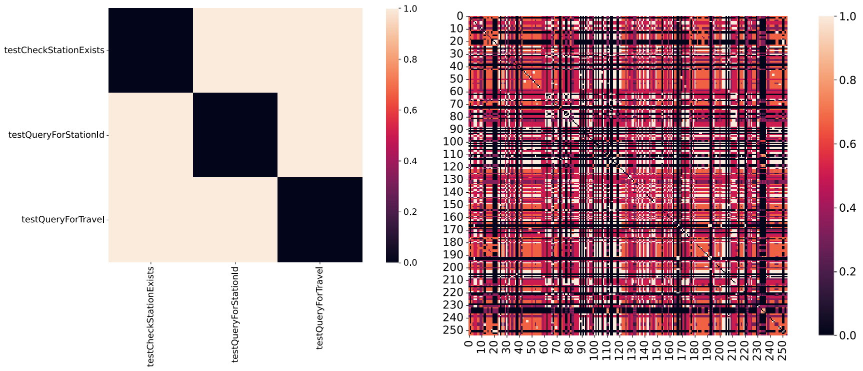
Concerning the similarity comparison based on `invokes`, we only present here the results related to the `nonemptyIntersection` and `nonemptyCommonSeq`—the complete experimental data are again located in the appendix. An interesting result can be observed by comparing Figs. 5 and 11. Indeed, the results are mostly comparable. This result is related to the nature of the test suite in Fullteaching. Indeed, the test programs that contact some remote endpoints also directly exercise non-minimal parts of the SUT as if they were compounded unit tests. If a test program contacts the same endpoint, it will likely exercise the same parts of the SUT. This behaviour is not common for all test suites. Indeed, in Fig. 12 we report the results for both invokes and endpoints in the case of TrainTicket, for the same `nonemptyCommonSeq` criterion—we only report the minimum scores. As can be seen, the results are highly different. The TrainTicket test suite is such that few test programs exercise the same endpoints. Conversely, the SUT is directly exercised more at large. This characteristic is clearly emerging from the results, considering the relevantly different number of test programs deemed similar by the same metric using the two different domains and the more diversified similarity scores observed in the `invokes` case.

To conclude the analysis, in Fig. 13 we present the results related to Fullteaching when using the `nonemptyCommonSeq` criterion. When compared to `nonemptyIntersection` (Fig. 11), we observe that the number of test programs considered similar is the same. Nevertheless, the score values are more scattered in the range. By the definition of the criteria, this is an expected result. Indeed, considering sequences rather than sets allows us



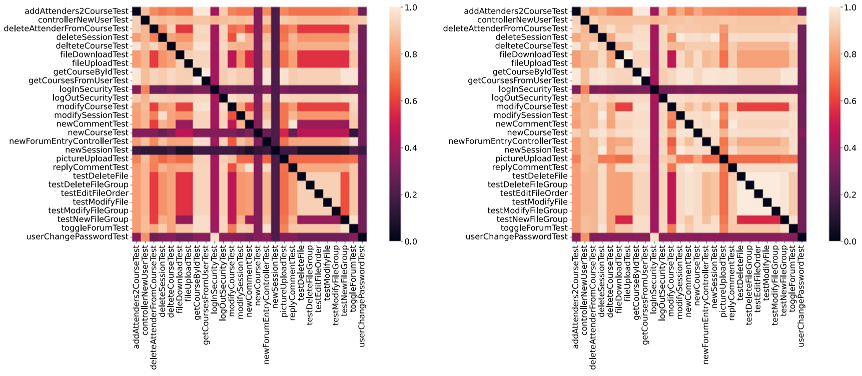
(a) Minimum Similarity Score. (b) Maximum Similarity Score.

Fig. 11. *Subject*: Fullteaching; *Domain*: `invokes`; *Criterion*: `nonemptyIntersection`.



(a) *Domain*: `endpoint`. (b) *Domain*: `invokes`.

Fig. 12. *Subject*: TrainTicket; *Criterion*: `nonemptyIntersection`.

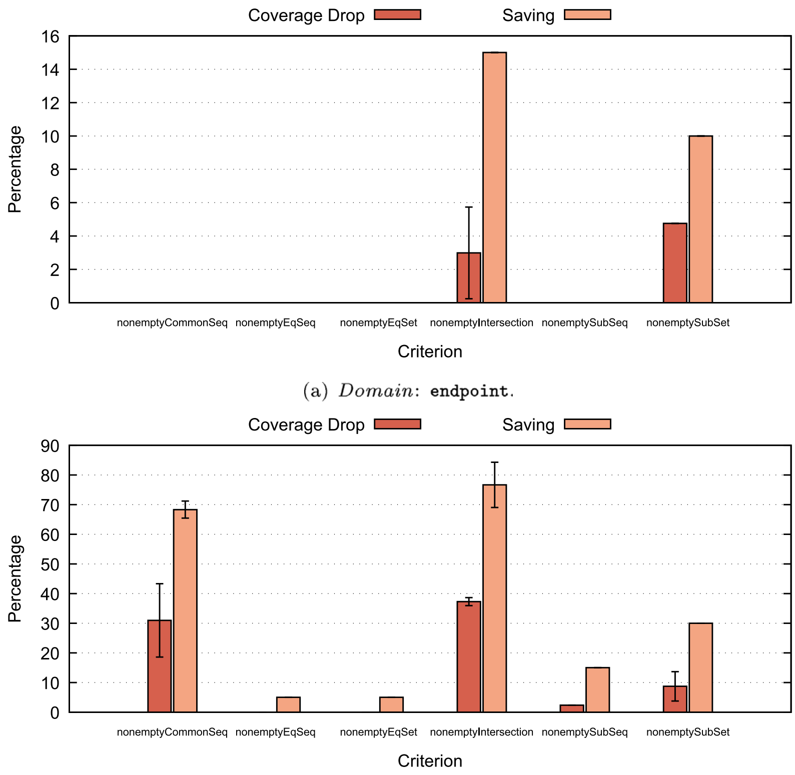


(a) Minimum Similarity Score. (b) Maximum Similarity Score.

Fig. 13. *Subject*: Fullteaching; *Domain*: `invokes`; *Criterion*: `nonemptyCommonSeq`.

to gather more stringent similarity information. An analysis based on `nonemptyCommonSeq` (and based on sequences in general) could enable a more fine-tuned selection of test programs in the considered governance framework for regression testing.

Overall, the criteria provide results that are comparably different. `nonemptyEqSet` is a stronger similarity criterion, which anyhow leaves out many test programs from the suite. `nonemptyIntersection`, on the other hand, includes a larger number of test programs while being less "categorical" about the similarity between test programs. `nonemptySubSet` and `nonemptyCommonSeq` capture capabilities of both criteria. Concerning RQ1, we can conclude that the criteria can detect similarities between test programs to various degrees, which can be beneficial depending on the current phase of the application's lifecycle. For example, when dealing with testing during feature development, the `nonemptyEqSet` criterion might help determine 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 in 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 reentrance bugs—multiple test programs that invoke methods from the same package of the application might be run concurrently.



(a) *Domain*: `endpoint`.

(b) *Domain*: `invokes`.

Fig. 14. *Subject*: Fullteaching.

Answer to RQ1: The proposed criteria can detect implicit similarities between test programs to various degrees: some criteria are more inclusive and they highlight coarse-grained implicit similarities, while others are more conservative as they report only very narrow similarities. Overall, the implicit similarities extracted from test programs can be used to support decisions in the context of a governance framework (as also argued when answering RQ3 below). However, the actual benefit they provide likely depends on the current phase of the application's lifecycle and needs the development of specific techniques, whose design is beyond the scope of the present paper.

7.2 Stability of the similarity criteria

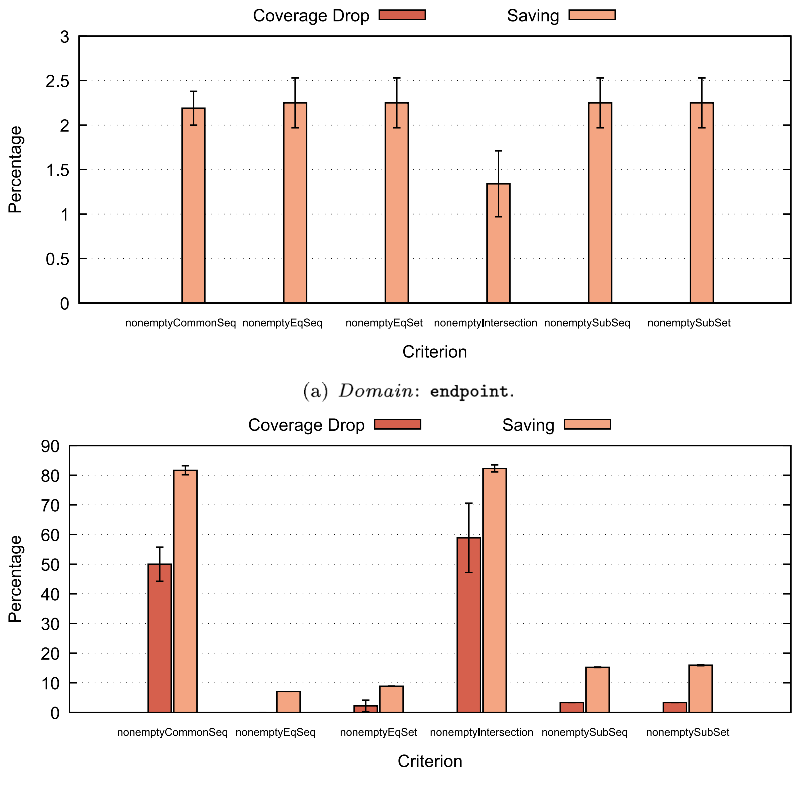
In this part of our empirical evaluation, we explicitly pursue an answer to RQ2. We have not considered symbolic execution traces for this part of the analysis precisely due to their symbolic nature. Indeed, we focus only on traces generated by instrumented execution because they provide a single score for each test program. This approach is helpful when studying the stability of the criteria because it removes a possible source of instability related to the multiple paths explored by the symbolic execution rather than to the criteria themselves. Indeed, symbolic traces would explore execution paths that might not be taken by the actual execution of the test program. Moreover, we target a comparison with a randomly-selected test suite built without considering the symbolic trace. Building *TS-small* based on symbolic execution would lead to incomparable results.

As mentioned in Section 6.1, we focus on multiple *TS-small* subsets of the original test suite and study the coverage variance to consider a similarity criterion as stable. We have set the similarity threshold s_{min} to 0.5 for both applications as an intermediate value, allowing a non-minimal number of test programs to be deemed similar. Our experimental assessment has shown that if the threshold is changed, the trends in the experimental results are comparable, although with different slopes related to the increased/reduced number of test programs included. Coverage data have been obtained by relying on JaCoCo (Hoffmann et al., 2022). We report the result of this experiment when considering three different *TS-small* subsets for each configuration, in Figs. 14 (for Fullteaching) and 15 (for TrainTicket).

We observe from the results that the coverage drop is pretty stable in all configurations as far as the `endpoint` domain is concerned. The only exception is in the Fullteaching case when relying on the `nonemptyIntersection` criterion. As discussed above, `nonemptyIntersection` is the less stringent criterion. Therefore, the *TS-small* suite built based on this criterion is likely to offer a more significant number of selection possibilities. The high variability is also related to many test programs overlapping in the Fullteaching test suite concerning the invoked endpoints. Therefore, a totally-random selection with a larger degree of freedom can produce the observed high variability.

Interestingly, many criteria provide a 0% coverage drop in Fullteaching. This result is an early indication of the capability of the proposed criteria to support the exclusion of test programs that are likely to exercise the same parts of the SUT. This behaviour is more evident in the TrainTicket case (see Fig. 15).

Concerning the `invokes` domain, we observe more variability. Besides the already mentioned `nonemptyIntersection` (particularly in TrainTicket), also `nonemptyCommonSeq` shows high variability. The reason for this variability can be found in Fig. 16, where we report the associated heatmap. Indeed, from the results, we note that the largest part of the test programs is considered similar, with scores that are around 0.5. Therefore, also



(a) *Domain*: `endpoint`.

(b) *Domain*: `invokes`.

Fig. 15. *Subject*: TrainTicket.

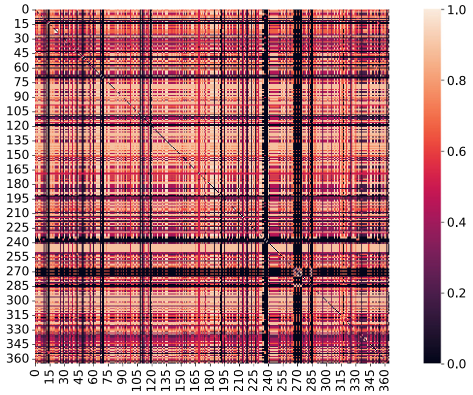


Fig. 16. *Subject*: TrainTicket; *Domain*: `invokes`; *Criterion*: `nonemptyCommonSeq`.

in this case, given the large number of test programs and similar values for the scores, the degree of freedom is quite high.

> **Answer to RQ2**: The most stable criteria are the most stringent ones (i.e., `nonemptyEqSet` and `nonemptyEqSeq`). For more inclusive criteria, the stability significantly depends on the nature of the test suite.

7.3 Quality of the similarity criteria

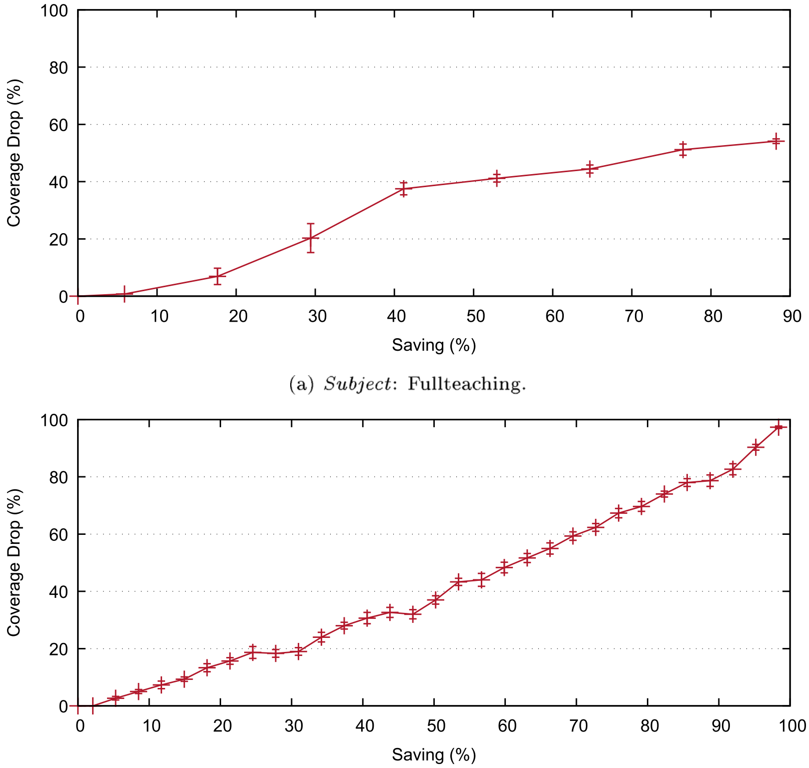
To answer RQ3, we consider the results provided in Fig. 17, where we report the average coverage drop obtained when a certain percentage of test programs is dropped. These results have been collected by running ten different randomly-constructed test suites (*TS-small-rnd*) for each point in the plots—we show average values and 90% confidence intervals. These data allow answering the following question: "*if you plan to skip x% test programs to save the time to run the test suite, what is the coverage drop that you must be prepared to observe if you have no way to make an informed selection of the test programs?*". As an example, in Fullteaching (Fig. 17(a)), if you skip 60% of the test programs, on average, you can observe a 40% performance drop. Similarly, if you skip 60% of test programs in TrainTicket (Fig. 17(b)), you should be prepared to face a 50% coverage drop.

Therefore, to answer RQ3, we want to determine whether, relying on the similarity criteria that we have proposed, it is possible to obtain a certain level of test program execution saving associated with an equal or lower coverage drop. In Figs. 14 and 15, we also present the test program savings we have observed when running the various *TS-small* test suites generated based on the similarity scores. By comparing the results in Figs. 14, 15, and 17, we notice that no domain/criterion configuration shows a coverage drop higher than in the case of a random selection of test programs, with the same amount of saving. Most notably, many configurations exhibit a lower coverage drop.

> **Answer to RQ3**: Without any prior knowledge of the test suite, the proposed similarity criteria can extract similarity information from the test programs in a way that can be effectively exploited in governance frameworks for regression testing, and specifically for test suite reduction.

8 Threats to validity

In the following, we discuss some of the threats that may potentially affect our empirical evaluation's validity, and thus the validity of the conclusions we have drawn.



(a) *Subject*: Fullteaching.

(b) *Subject*: TrainTicket.

Fig. 17. Results with *TS-random*.

8.1 Threats to construct validity

Any explicit or implicit assumption concerning the setup of the validation scenarios may lead to questionable conclusions. In the following, we discuss the more relevant decisions that may have impacted the interpretation of the observed outcomes.

Knowledge base carved from the test programs: Implicit dependencies across test programs are extracted by considering a well-defined but limited set of statements coded within the tests' implementations. Indeed, our approach considers only those statements that activate shared local/remote APIs. While presenting our work's research context and motivation, we discussed why these information sources could help study test programs' similarities in microservices applications. However, we agree that false-positive and false-negative similarities may result from this narrowed analysis of the test programs' traces. As we also acknowledged in Sections 7.2 and 7.3, these considerations become much more relevant when employing the symbolic execution traces. Thus we cannot exclude that a more sophisticated analysis of the behaviour exposed by test programs' implementation may lead to a finer observation of actual similarities.

Inference rules: Similarly to the specific constructs in the test programs, the way we processed the information collected during the concrete/symbolic executions may concern the appropriateness of the analysis. Section 5 reports the set of inference rules which define the considered similarity criteria. In other words, these inference rules represent the core definitions of test programs' implicit dependencies. Clearly, a different set of similarity criteria could lead to different results and conclusions, but we can argue that the study is limited only to those criteria. However, even focusing on the given group of similarity criteria, we are aware that the current definitions of the rules in Prolog may suffer from typical implementation issues (e.g., related to the backtracking strategies). Such issues may lead to tiny differences between the outcome expected by the abstract formulation of the similarity criterion and the outcome returned by its implementation in a Prolog rule. The Prolog implementations referred to in this work are built, extended, and refined on a set of inference rules from previous works (De Angelis et al., 2021; De Angelis et al., 2021). In addition, before their exploitation in our empirical evaluation, we carefully analysed them in peer-reviewing sessions that also included dedicated testing activities that should have mitigated such risks.

8.2 Threats to internal validity

This class of threats to validity refers to those aspects that could have influenced the observed outcome.

Choice of the case study: The empirical evaluation referred to two different subjects. They have been selected because both projects are open-source, abiding by the microservices architectural style, and their development toolchain could be integrated easily into our reference implementation. Nevertheless, both subjects may have hidden or uncontrolled influences on the experimentation, and more case studies may be useful for a more thorough experimental evaluation of our approach.

Minimum degree of similarity: The validation methods adopted for answering RQ2 and RQ3 rely on the parameter s_{min} to cluster test programs for a given similarity criterion (see Section 6.1). In these specific settings, we considered two test programs similar if their score was higher than 0.5 over a range [0–1]. We are aware that a different tuning of this parameter impacts the presentation of both the quality and the stability of the similarity criteria. However, the results for both concepts are also influenced by the specific composition of the available test suites. Thus, we tried to make a neutral choice that could also overcome the quality/variety of the actual test suites in the considered subjects.

8.3 Threats to external validity

The expected scenario for a study is to draw conclusions to such an extent that they are valid also in other studies. However, generalised conclusions are difficult to achieve as various factors often threaten them.

Statistical power: Overall the study considered a set of 711 test programs: 29 from Fullteaching, and 682 from TrainTicket. Though the number is not small, it is also evident that it is insufficient to advocate a strong significance for the observed outcomes. In addition, the experimental evaluation only focused on two subjects. Thus our interpretations could be influenced by hidden aspects present in both subjects.

Generalisation: The results we collect in this study may strictly depend on the experiments we planned for the specific case studies we selected. Thus, we cannot draw fully general conclusions that claim the proposed approach can always provide valuable results for any application. We clarify that such a statement needs more extensive validation against different case studies.

In order to support other researchers to repeat our experience or replicate it with different subjects, we make available the whole artefacts developed within the context of this work (see the section titled: "Replication Package and Data Availability"). Also, the appendix details all the similarities of the test programs we produced during this study. We believe this information could support future works aiming to validate the generalisation of the outcomes observed in this study.

9 Related work

Regression testing is an interesting application context that has been intensively investigated, as reported, for instance, in the survey paper by Yoo and Harman (2012). Some of the works classified in the survey focus on discovering and processing test cases in a given test suite that *traverse* modifications in the original SUT. Among others, the survey reports on approaches that leverage analysis on control (Laski and Szermer, 1992; Rothermel and Harrold, 1997) or data (Gupta et al., 1992) flows, symbolic execution (Yau and Kishimoto, 1987), textual difference in source code of the SUT (Vokolos and Frankl, 1997). Our work complements these approaches by starting the analysis of similarity from the test program implementations. The results reveal additional information that can potentially be used in other common regression testing activities such as test case prioritisation, minimisation or selection.

A recent systematic mapping study by Waseem et al. (2020) surveys techniques for testing microservice-based applications. Among these, the paper by Sotomayor et al. (2022) targets open-source testing tools for microservices.

Some of the surveyed techniques propose the use of formal methods (e.g., model checking) for automated testing, specifically for test case generation, scheduling, and execution (Meinke and Nycander, 2015; Quenum and Aknine, 2018; Hillah et al., 2017; Camilli et al., 2018).

In order to improve the effectiveness of testing, Rahman et al. (2015) introduce a framework for parallel execution of tests that, by cutting down the execution time, enables frequent re-running of the entire test suite during the development of microservices-based applications. While we share the same goal, and indeed our framework can contribute to the design of flexible policies for effective and efficient regression testing, our approach is closer to those aimed at avoiding re-running all available tests when a small component of the SUT changes, and instead supporting the selection of tests that are useful to exercise the changed component. In order to retrieve subsets of test cases required to deal with microservice changes, Ma et al. (2019) propose a graph-based approach for analysing the dependencies among microservices based on their APIs. A dynamic analysis of microservices, based on their workload characteristics, is used by Schulz et al. (2019) to generate tailored test cases for exercising specific microservices of an application in isolation.

However, the issue of inferring dependencies and similarities among test programs has received very little attention, especially compared to their structural or behavioural analysis. Indeed, instead of analysing microservices, our approach is based on the dynamic analysis (either concrete or symbolic) of the test suite to determine which microservices are tested (therefore, allowing the selection of only those required to test the modified component).

In the more general context of automated software engineering, the problem of identifying similarities among test programs is related to the broader issue of identifying similarities among generic programs (Walenstein et al., 2007), which has been studied for multiple purposes (and with different techniques), such as duplicate code detection (Sheneamer and Kalita, 2016), plagiarism detection or copyright infringement (Lancaster and Culwin, 2004), and code compression (Evans and Fraser, 2003).

The idea of using test case similarity to design effective testing strategies has been explored in several works (Noor and Hemmati, 2015; Wang et al., 2015; Hao et al., 2008; Ledru et al., 2012; Miranda et al., 2018). Noor and Hemmati (2015) propose an approach for prioritising test cases based on their similarity with those 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 uses concrete execution while we perform a logic-based similarity analysis on traces extracted via symbolic execution.

Another similarity-based approach for regression test case prioritisation is presented by Wang et al. (2015). The execution order of the test cases is scheduled based on the distance between them, where the notion of distance is defined concerning branch coverage. That paper evaluates six similarity measures and shows that Euclidean distance gives the best result through experiments on a few benchmark programs.

Test case similarity is defined by Ledru et al. (2012) based on the string distance between the test cases, and hence no notion of execution is considered. Also Miranda et al. (2018) base their test case prioritisation technique on similarity relations defined on test cases and not on their execution. To enforce scalability, the similarity is computed by algorithms usually applied in the context of big data processing.

Similarity has been exploited for fault-localisation in the paper by Hao et al. (2008), where the 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 the similarity of programs (not necessarily test programs) based on static analysis or fuzz testing, whereas we employ symbolic execution. In particular, Raman et al. (2017) use the call-dependency relation among program APIs to generate a trace of the API calling sequence. Wang and Wu (2017) present a method that uses fuzz testing for similarity analysis of binary code. The similarity score of two behaviour traces generated by fuzzing from two program functions is computed according to their longest common subsequence.

In automated software testing, symbolic execution has been largely used as an effective technique for finding errors in software applications and generating high-coverage test suites (Meudec, 2001; Visser et al., 2004; Cadar and Engler, 2005;

Godefroid et al., 2005; Sen, 2007; Cadar and Sen, 2013; Braione et al., 2018; Nivethithaa and Krishnapriya, 2018). This technique, which was first introduced in the mid 1970's, 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 SMT solvers for checking that a specified program property is not violated by any execution path up to a given length bound (Armando et al., 2009).

The symbolic exploration of the software of states requires the generation of a very complex combination of constraints. The resolution of these constraints frequently leads pure symbolic approaches to suffer severe scalability issues. *Concolic* approaches mitigate such a risk by combining symbolic evaluation with concrete execution and, in some cases, random data generation (Visser et al., 2004; Williams et al., 2005; Cadar and Engler, 2005; Sen, 2007; Godefroid et al., 2005).

In implementing our technique, we use the JBSE, a symbolic Java Virtual Machine which can deal with complex heap data structures. We also use a form of concolic execution to handle methods in charge of setting up the environment for a test program execution (e.g., `@Before` in JUnit). However, the main goal of our work is neither the search for errors nor the generation of test cases. In fact, we want to infer relations between test programs, e.g., various forms of dependency or similarity, and we do so by extracting high-level information from symbolic execution paths and states. Our approach is particularly suitable when dealing with parametric test programs.

Some techniques for *relational verification* make use of *constraint logic programming* (i.e., logic programming augmented with constraint solving) to verify relations between programs (Felsing et al., 2014; De Angelis et al., 2016). However, the kind of properties targeted by relational verification is 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 parallels symbolic execution techniques for crosschecking optimised versions of data-parallel programs against the unoptimised ones (Collingbourne et al., 2014).

10 Conclusions and future work

We have discussed a methodology to extract similarity relations among test programs for microservices applications. By dynamic analysis (i.e., either instrumented or symbolic), we can extract from a test suite relevant information about the methods called by the test programs. A set of Prolog rules processes this information to filter the execution traces of interest and generate additional facts to enlarge the knowledge base, e.g., to determine the endpoints that may be activated by running the various test programs. Other Prolog rules compute a similarity score according to multiple criteria. In two case studies, we have observed that our approach can generate a significant amount of information, which can be used in the context of a governance framework for regression testing, for example, by supporting decisions that could prevent the enforcement of a *retest-all* strategy.

Additionally, our empirical evaluation shows that the proposed criteria support the selection of test programs that can be stable and effective at automatically identifying what test programs shall be excluded. Nevertheless, the overall quality of the test suite offered by the application being tested plays a significant role in the selection power of the proposed approach.

Future work includes devising additional rules to cope with test suites that have a high overlap degree across different test programs. Moreover, we plan to exploit the similarity relations to support an online selection procedure that could quickly determine what test programs to execute after the outcome of a previous set of executed test programs is gathered.

CRediT authorship contribution statement

Emanuele De Angelis: Methodology, Software, Investigation, Writing – original draft. **Guglielmo De Angelis:** Conceptualization, Methodology, Investigation, Writing – original draft. **Alessandro Pellegrini:** Methodology, Software, Investigation, Writing – original draft. **Maurizio Proietti:** Conceptualization, Methodology, Investigation, Writing – original draft.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Guglielmo De Angelis reports financial support was provided by Government of Italy Ministry of Education University and Research. Guglielmo De Angelis is currently serving as a guest editor for a Special Issue in JSS.

Data availability

The replication package and the source code used in the empirical evaluation of this work are available at: https://github.com/IASI-SAKS/hyperion/releases/tag/journal.

Acknowledgements

This paper has been supported by the Italian MIUR PRIN2017 Project: SISMA (Contract 201752ENYB), and partially by the Italian Research Group: INdAM-GNCS.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.jss.2023.111674.

References

Armando, A., Mantovani, J., Platania, L., 2009. Bounded model checking of software using SMT solvers instead of SAT solvers. Int. J. Softw. Tools Technol. Transf. 11 (1), 69–83.

Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I., 2018. A survey of symbolic execution techniques. ACM Comput. Surv. 51 (3), 50:1–50:39.

Bertolino, A., De Angelis, G., Lonetti, F., 2019. Governing regression testing in systems of systems. In: Proc. of ISSRE Workshops, GAUSS. IEEE, pp. 144–148. http://dx.doi.org/10.1109/ISSREW.2019.00064.

Braione, P., Denaro, G., Mattavelli, A., Pezzè, M., 2018. SUSHI: a test generator for programs with complex structured inputs. In: Proc. of ICSE. ACM, pp. 21–24.

Braione, P., Denaro, G., Pezzè, M., 2016. JBSE: A symbolic executor for Java programs with complex heap inputs. In: Proc. of FSE. ACM, pp. 1018–1022. http://dx.doi.org/10.1145/2950290.2983940.

Bruneton, E., Lenglet, R., Coupaye, T., 2002. ASM: a code manipulation tool to implement adaptable systems. Adaptable Extensible Compon. Syst. 30 (19).

Bruno, M., Canfora, G., Penta, M.D., Esposito, G., Mazza, V., 2005. Using test cases as contract to ensure service compliance across releases. In: Proc. of ICSOC. In: LNCS, Vol.3826, Springer, pp. 87–100. http://dx.doi.org/10.1007/11596141_8.

Cadar, C., Engler, D.R., 2005. Execution generated test cases: How to make systems code crash itself. In: Proc. of Int. Workshop SPIN. In: LNCS, Vol. 3639, Springer, pp. 2–23.

Cadar, C., Sen, K., 2013. Symbolic execution for software testing: three decades later. Commun. ACM 56 (2), 82–90.

Camilli, M., Bellettini, C., Capra, L., Monga, M., 2018. A formal framework for specifying and verifying microservices based process flows. In: Cerone, A., Roveri, M. (Eds.), Software Engineering and Formal Methods. Springer International Publishing, Cham, pp. 187–202.

Cerny, T., Donahoo, M.J., Trnka, M., 2018. Contextual understanding of microservice architecture: current and future directions. ACM SIGAPP Appl. Comput. Rev. 17 (4), 29–45.

Chiba, S., 2000. Load-Time structural reflection in java. In: Bertino, E. (Ed.), ECOOP 2000 — Object-Oriented Programming. In: Lecture Notes in Computer Science, Vol. 1850, Springer Intrernational Publishing, Berlin Heidelberg, Germany, pp. 313–336. http://dx.doi.org/10.1007/3-540-45102-1_16.

Clemson, T., 2014. Testing strategies in a microservice architecture. URL https://martinfowler.com/articles/microservice-testing/.

Collingbourne, P., Cadar, C., Kelly, P.H.J., 2014. Symbolic crosschecking of data-parallel floating-point code. IEEE Trans. Softw. Eng. 40 (7), 710–737.

De Angelis, E., De Angelis, G., Pellegrini, A., Proietti, M., 2021. Inferring relations among test programs in microservices applications. In: Proceedings of the 15th IEEE International Conference on Service Oriented Systems Engineering. In: SOSE, IEEE, pp. 114–123. http://dx.doi.org/10.1109/SOSE52839.2021.00018.

De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M., 2016. Relational verification through horn clause transformation. In: Proc. of SAS. In: LNCS, Vol. 9837, Springer, pp. 147–169.

De Angelis, E., Pellegrini, A., Proietti, M., 2021. Automatic extraction of behavioral features for test program similarity analysis. In: Proc. of ISSRE Workshops, GAUSS. IEEE, pp. 129–136. http://dx.doi.org/10.1109/ISSREW53611.2021.00054.

Evans, W.S., Fraser, C.W., 2003. Grammar-based compression of interpreted code. Commun. ACM 46 (8), 61–66. http://dx.doi.org/10.1145/859670.859699, URL https://dl.acm.org/doi/10.1145/859670.859699.

Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M., 2014. Automating regression verification. In: Proc. of ASE. pp. 349–360.

García, B., Lonetti, F., Gallego, M., Miranda, B., Jiménez, E., De Angelis, G., Moreira, C.E., Marchetti, E., 2018. A proposal to orchestrate test cases. In: Proc. of QUATIC. pp. 38–46.

Gazzola, L., Goldstein, M., Mariani, L., Segall, I., Ussi, L., 2020. Automatic ex-vivo regression testing of microservices. In: Proc. of AST. ACM, pp. 11–20.

Godefroid, P., Klarlund, N., Sen, K., 2005. DART: directed automated random testing. In: Proc. of PLDI. ACM, pp. 213–223. http://dx.doi.org/10.1145/1065010.1065036.

Gupta, R., Harrold, M.J., Soffa, M.L., 1992. An approach to regression testing using slicing. In: ICSM. Vol. 92, pp. 299–308.

Hao, D., Zhang, L., Pan, Y., Mei, H., Sun, J., 2008. On similarity-awareness in testing-based fault localization. Automated Softw. Eng. 15 (2), 207–249. http://dx.doi.org/10.1007/s10515-008-0025-9, URL http://link.springer.com/10.1007/s10515-008-0025-9.

Harrold, M.J., Orso, A., 2008. Retesting software during development and maintenance. In: Proc. of Frontiers of Software Maintenance. pp. 99–108.

Hillah, L.M., Maesano, A.-P., De Rosa, F., Kordon, F., Wuillemin, P.-H., Fontanelli, R., Bona, S.D., Guerri, D., Maesano, L., 2017. Automation and intelligent scheduling of distributed system functional testing. Int. J. Softw. Tools Technol. Transf. 19 (3), 281–308. http://dx.doi.org/10.1007/s10009-016-0440-3.

Hoffmann, M.R., Janiczak, B., Mandrikov, E., Friedenhagen, M., 2022. Jacoco code coverage library.

Johnston, A.B., Burnett, D.C., 2012. WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web. Digital Codex LLC.

Kazmi, R., Jawawi, D.N.A., Mohamad, R., Ghani, I., 2017. Effective regression test case selection: A systematic literature review. ACM Comput. Surv. 50 (2), 29:1–29:32.

Khatibsyarbini, M., Isa, M.A., Jawawi, D.N., Tumeng, R., 2018. Test case prioritization approaches in regression testing: A systematic literature review. Inf. Softw. Technol. 93, 74–93.

Lancaster, T., Culwin, F., 2004. A comparison of source code plagiarism detection engines. Comput. Sci. Educ. 14 (2), 101–112. http://dx.doi.org/10.1080/08993400412331363843, URL http://www.tandfonline.com/doi/abs/10.1080/08993400412331363843.

Laski, J., Szermer, W., 1992. Identification of program modifications and its applications in software maintenance. In: Proceedings Conference on Software Maintenance 1992. IEEE Computer Society, pp. 282–283.

Ledru, Y., Petrenko, A., Boroday, S., Mandran, N., 2012. Prioritizing test cases with string distances. Autom. Softw. Eng. 19 (1), 65–95. http://dx.doi.org/10.1007/s10515-011-0093-0.

Lewis, J., Fowler, M., 2014. Microservices, a definition of this new architectural term. URL https://martinfowler.com/articles/microservices.html.

Ma, S.-P., Fan, C.-Y., Chuang, Y., Liu, I.-H., Lan, C.-W., 2019. Graph-based and scenario-driven microservice analysis, retrieval, and testing. Future Gener. Comput. Syst. 100, 724–735. http://dx.doi.org/10.1016/j.future.2019.05.048, URL https://www.sciencedirect.com/science/article/pii/S0167739X19302614.

Mariani, L., Papagiannakis, S., Pezzè, M., 2007. Compatibility and regression testing of COTS-component-based software. In: Proc. of ICSE. IEEE CS, pp. 85–95.

Meinke, K., Nycander, P., 2015. Learning-based testing of distributed microservice architectures: Correctness and fault injection. In: Proc. of SEFM 2015 Collocated Workshops. In: LNCS, Vol. 9509, Springer, pp. 3–10.

Meudec, C., 2001. ATGen: automatic test data generation using constraint logic programming and symbolic execution. Softw. Test. Verif. Reliab. 11 (2), 81–96.

Miranda, B., Cruciani, E., Verdecchia, R., Bertolino, A., 2018. FAST approaches to scalable similarity-based test case prioritization. In: Chaudron, M., Crnkovic, I., Chechik, M., Harman, M. (Eds.), Proceedings of the 40th International Conference on Software Engineering. ACM, pp. 222–232. http://dx.doi.org/10.1145/3180155.3180210.

Nivethithaa, K., Krishnapriya, V., 2018. A brief survey on symbolic execution test-selection techniques. Int. J. Comput. Sci. Eng. 06 (08), 81–85. http://dx.doi.org/10.26438/ijcse/v6si8.8185, URL http://www.ijcseonline.org/full_spl_paper_view.php?paper_id=480.

Noor, T.B., Hemmati, H., 2015. A similarity-based approach for test case prioritization using historical failure data. In: Proceedings of the 26th International Symposium on Software Reliability Engineering. In: ISSRE, IEEE, pp. 58–68. http://dx.doi.org/10.1109/ISSRE.2015.7381799, URL http://ieeexplore.ieee.org/document/7381799/.

Paterson, D., Campos, J., Abreu, R., Kapfhammer, G.M., Fraser, G., McMinn, P., 2019. An empirical study on the use of defect prediction for test case prioritization. In: Proc. of ICST. IEEE, pp. 346–357.

Quenum, J.G., Aknine, S., 2018. Towards executable specifications for microservices. In: Proc. of SCC. IEEE, pp. 41–48.

Rahman, M., Chen, Z., Gao, J., 2015. A service framework for parallel test execution on a developer's local development workstation. In: 2015 IEEE Symposium on Service-Oriented System Engineering. pp. 153–160. http://dx.doi.org/10.1109/SOSE.2015.45.

Raman, D., Bezawada, B., Rajinikanth, T.V., Sathyanarayan, S., 2017. Static program behavior tracing for program similarity quantification. In: Satapathy, S.C., Prasad, V.K., Rani, B.P., Udgata, S.K., Raju, K.S. (Eds.), Proceedings of the First International Conference on Computational Intelligence and Informatics. In: Advances in Intelligent Systems and Computing (AISC), Vol. 507, Springer, Singapore, pp. 321–330. http://dx.doi.org/10.1007/978-981-10-2471-9_31, URL https://link.springer.com/chapter/10.1007/978-981-10-2471-9_31.

Rothermel, G., Harrold, M.J., 1997. A safe, efficient regression test selection technique. ACM Trans. Softw. Eng. Methodol. (TOSEM) 6 (2), 173–210.

Schulz, H., Angerstein, T., Okanović, D., van Hoorn, A., 2019. Microservice-tailored generation of session-based workload models for representative load testing. In: 2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems. MASCOTS, pp. 323–335. http://dx.doi.org/10.1109/MASCOTS.2019.00043.

Sen, K., 2007. Concolic testing. In: Proc. of ASE. ACM, pp. 571–572.

Sheneamer, A., Kalita, J., 2016. A survey of software clone detection techniques. IJCA, Int. J. Comput. Appl. IJCA, 137 (10), 1–21. http://dx.doi.org/10.5120/ijca2016908896.URL http://www.ijcaonline.org/research/volume137/number10/sheneamer-2016-ijca-908896.pdf,

Sotomayor, J.P., Allala, S.C., Santiago, D., King, T.M., Clarke, P.J., 2022. Comparison of open-source runtime testing tools for microservices. Softw. Qual. J. http://dx.doi.org/10.1007/s11219-022-09583-4.

Spadini, D., Aniche, M., Bruntink, M., Bacchelli, A., 2018. Mock objects for testing java systems. Empir. Softw. Eng. 24 (3), 1461–1498. http://dx.doi.org/10.1007/s10664-018-9663-0.

Vahabzadeh, A., Stocco, A., Mesbah, A., 2018. Fine-grained test minimization. In: Proc. of ICSE. pp. 210–221.

Visser, W., Pasareanu, C.S., Khurshid, S., 2004. Test input generation with java PathFinder. In: Proc. of ISSTA. ACM, pp. 97–107.

Vokolos, F.I., Frankl, P.G., 1997. Pythia: A regression test selection tool based on textual differencing. In: Reliability, Quality and Safety of Software-Intensive Systems. Springer, pp. 3–21.

Walenstein, A., El-Ramly, M., Cordy, J.R., Evans, W.S., Mahdavi, K., Pizka, M., Ramalingam, G., von Gudenberg, J.W., 2007. Similarity in programs. In: Koschke, R., Merlo, E., Walenstein, A. (Eds.), Duplication, Redundancy, and Similarity in Software. In: Dagstuhl Seminar Proceedings, Vol. 6301, Schloss Dagstuhl, Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 1–8. http://dx.doi.org/10.4230/DagSemProc.06301.11.

Wang, R., Jiang, S., Chen, D., 2015. Similarity-based regression test case prioritization. In: Proceedings of the International Conference on Software Engineering and Knowledge Engineering. In: SEKE, KSI Research Inc., p. 6. http://dx.doi.org/10.18293/SEKE2015-115.

Wang, S., Wu, D., 2017. In-memory fuzzing for binary code similarity analysis. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. In: ASE, IEEE, pp. 319–330. http://dx.doi.org/10.1109/ASE.2017.8115645, URL http://ieeexplore.ieee.org/document/8115645/.

Waseem, M., Liang, P., Márquez, G., Salle, A.D., 2020. Testing microservices architecture-based applications: A systematic mapping study. In: Proc. of APSEC. IEEE, pp. 119–128.

Wielemaker, J., Schrijvers, T., Triska, M., Lager, T., 2012. Swi-prolog. Theory Pract. Logic Program. 12 (1–2), 67–96.

Williams, N., Marre, B., Mouy, P., Roger, M., 2005. PathCrawler: Automatic generation of path tests by combining static and dynamic analysis. In: Proc. of EDCC. In: LNCS, Vol. 3463, Springer, pp. 281–292.

Yang, Q., Li, J.J., Weiss, D.M., 2009. A survey of Coverage-Based testing tools. Comput. J. 52 (5), 589–597. http://dx.doi.org/10.1093/comjnl/bxm021.

Yau, S.S., Kishimoto, Z., 1987. Method for revalidating modified programs in the maintenance phase. In: Proceedings-IEEE Computer Society's International Computer Software & Applications Conference. IEEE, pp. 272–277.

Yoo, S., Harman, M., 2012. Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verif. Reliab. 22 (2), 67–120.