Inferring Relations Among Test Programs in Microservices Applications

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

Guglielmo De Angelis
IASI–CNR
Rome, Italy
guglielmo.deangelis@iasi.cnr.it

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

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

***Abstract*—The emergence of the microservices-oriented architectural style calls for novel methodologies and technological frameworks that support the design, development, and maintainance of applications structured according to this new style. In this paper, we consider the issue of designing suitable strategies for the governance and the automation of testing activities within the microservices paradigm. We focus on the problem of discovering relations between test programs that help avoiding to re-run all the available test suites each time one of its constituents evolves. We propose an analysis technique, based on symbolic execution of test programs, which is able to collect information about the invocations of local and remote APIs performed when running such programs. Symbolic execution enables the analysis of sets of executions corresponding to different input data, and hence it is also suitable for parametric test programs. The information extracted by symbolic execution is processed by a rule-based automated reasoning engine, which infers dependencies and similarities among test programs. In particular, test programs are considered similar if they involve the same microservice instance, or they connect to the same remote API, or they locally activate overlapping APIs, or they raise similar kinds of errors. We show the viability of our approach by presenting a case study within the context of a real-world microservice application that implements an open-source educational platform.**

***Index Terms*—software testing, microservices architecture, test program similarity, symbolic execution, automated reasoning.**

I. Introduction

Microservices architectures promote the construction of software systems as distributed software units, each one abiding by the *single responsibility principle* [1]. 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 [2]. Also, the principles of microservices architecture 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 different microservices (i.e., their design, development, deployment, or update) should require minimal (or even zero) coordination effort with each other, possibly limited to immediate dependencies.

In order to take full advantage of this new architectural style, novel methodologies and new technological frameworks are needed for designing, developing, and maintaining microservices applications. In particular, testing activities demand for appropriate strategies and tools for achieving satisfactory levels of automation within the microservices paradigm. Current trends on testing of microservices architectures are highlighting that specific responsibilities can be assigned to each test phase: from unit testing, to integration, and contract testing, up to end-to-end testing [3].

In the microservices architectural style, each testing strategy aims to provide confidence of correctness on each microservice, or on a set of them. As detailed in [3], unit testing has the responsibility for validating the internal behaviour of a microservice. Integration tests have the responsibility for validating the communication paths and interactions among the composed microservices—their goal is to check that each microservice can communicate with others. As such, they are often considered as a fast feedback toward integration. Similarly, contract tests have the responsibility for checking interactions at the boundary of the application asserting that each microservice in the application meets the *contract* [4] expected by an external consuming service. Finally, end-to-end tests have the responsibility for checking if the considered system-as-a-whole achieves its intended goals. In general, end-to-end tests focus on testing the message-passing between the services, but they could also include checks validating the correct configuration of extra network infrastructures (e.g., firewalls, proxies, load-balancers).

In addition, both technical and managerial independence of microservices admit a dynamic scenario for applications built with this paradigm: the evolution of one or more constituents could take place according to several governance schemata opening to different degrees of challenges about the validation of the resulting system [5], [6]. Specifically, continuous evolution 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 [7] 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.

Relevant empirical considerations related to Cohn's metaphor of the Testing Pyramid [8] hinder the testing of any software system, but the decentralised nature of microservices solutions make them more difficult to mitigate. Both the industry and the academia proposed many successful approaches for unitary testing, and many of them result in effective technologies contributing toward automation in software testing. However, the context changes when addressing testing activities toward the *top of the pyramid*.

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 [9], or ii) selecting some test cases [10], or iii) prioritising those expected to yield earlier fault detection [11], [12]. 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 specification for the considered microservices and, in some cases, the lack of source code could hamper the direct application of such testing techniques [13].

In addition, regression testing activities have to cope with the maintenance and the evolution of the regression test suites [14]: augmenting their significance by deriving new test cases from existing ones, or by inferring a better understanding of the considered software system by leveraging evidences from the test cases. Among the others, the observation of the actual interactions among microservices instances are investigated as a means of contributing to the evolution of regression testing test suite [15]. Also, test cases have been proposed as viable solution for checking compliance of contracts across service releases [16].

This work contributes to the governance of regression testing, taking into account the specific context of microservice applications. One relevant information that is 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 focusing on a different testing strategy; if they target the same testing goal and strategy, but using 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 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 running again all test programs in an order fixed in advance.

Specifically, this work assumes that a set of test cases 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 they are provided by the integrator of the overall application. Also, it relies on symbolic execution techniques [17] in order to gather information about the behaviour of a test program and the interactions it establishes among the microservices in the referred application. This symbolic approach allows the exploration of sets of concrete executions, and it also allows us to handle parametric tests in a very natural way. The information extracted via symbolic execution is processed using logic-based reasoning techniques [18] in order to establish similarity relations.

Our reference implementation has been bundled in the Hyperion package, which is publicly available as open source software[1].

The rest of the paper is organised as follows. In Section II we discuss the related work. Our overall approach to extract relations among test programs is depicted in Section III. The technique used to extract information from test programs is described in Section IV, while the rules to determine the similarity are presented in Section V. Section VI presents the results of an experimental case study. Section VII draws the conclusions of this work.

II. Related Work

A recent systematic study surveyed specific techniques for testing microservices architecture-based applications [19]. Some of these techniques propose the use of formal methods (e.g., model checking) for automated test case generation [20], [21]. However, at present, the issue of inferring dependencies and similarities among test programs has received very little attention, especially with respect to their structural or behavioural analysis.

In the context of automated software testing, symbolic execution has been largely used as an effective technique for finding errors in software applications and for generating high-coverage test suites [22]–[29]. This technique, which was first introduced in the mid 1970's [30], [31], has been conceived to exercise a software system by searching for potential configurations/states violating a given set of assertions.

In the literature, the basic idea of symbolic execution is strongly related to techniques for *bounded model checking* of software, which use *satisfiability modulo theories* (SMT) solvers for checking that a specified program property is not violated by any execution path, up to a given length bound [32].

The symbolic exploration of the program states often requires the generation of very complex combination of constraints. The resolution of these constraints frequently leads pure symbolic approaches to suffer from severe scalability issues. *Concolic* approaches mitigate such a risk by combining symbolic evaluation with concrete execution, along with, in some cases, random data generation [23]–[26], [29].

In the implementation of our technique, we use the Java Bytecode Symbolic Executor (JBSE) [33], a symbolic Java Virtual Machine able to 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, e.g., various forms of dependency or similarity, between test programs, 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.

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

Some techniques for *relational verification* make use of *constraint logic programming* (i.e., logic programming augmented with constraint solving) to verify relations between programs [34], [35]. However, the kind of properties targeted by relational verification are very strong (in general, undecidable) relations, such as full functional equivalence, while here we focus on test programs and we are interested in much weaker dependency and similarity relations based on suitable abstractions of the finite set of paths generated by symbolic execution. In this respect, our work bears some relationships with symbolic execution techniques for crosschecking optimised versions of data-parallel programs with respect to the optimised ones [36].

III. Overall Approach

Structuring a governance framework for regression testing includes the formulation of both strategies and policies that Quality Assurance Teams can use while making decisions on testing campaigns or in order to guide the root-cause analysis of issues that have been spotted. Such policies can be either enforced offline by planning the regression testing activities (e.g., test suite reduction, test case selection/prioritisation), or rather online while dynamically driving the test case execution by means of a test case orchestration [37]. In both cases, the role of test suites dependencies/similarities is crucial, as they enable the declaration of flows of test cases that could be run in sequence, parallel, or alternative combination [6].

Dependencies among regression test cases could emerge because of explicit declarations by the software engineering teams in build automation tools or continuous-integration (CI) frameworks. Also, similarities can be inferred from available software artifacts by means of some (semi-)automatic mining procedures. For example, with respect to the domain microservices applications, the test cases in a set of integration test suites could be considered similar if they all explicitly involve the same set of microservices. Also, all the unitary tests for a given microservice ms_i could be considered related to integration tests that involve ms_i. Indeed, if an integration test fails spotting a system regression, it could be meaningful to check if any regressions also occurred in the microservices it refers. Similar criteria can be also formulated among regression test cases/suites targeting all the test levels (e.g., contracts, end-to-end). Finally, even the data used during the execution of test cases can be referred to as a source of similarity. In the following, test programs for microservice 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;
3) raise similar kinds of errors.

Microservices are distributed components whose interactions take place across some abstraction of the network interface, and in most of the cases abiding by the REST architectural style [1]. Test programs opening connections against the same remote APIs are acting as test drivers for the same type of microservices or, in some cases, among

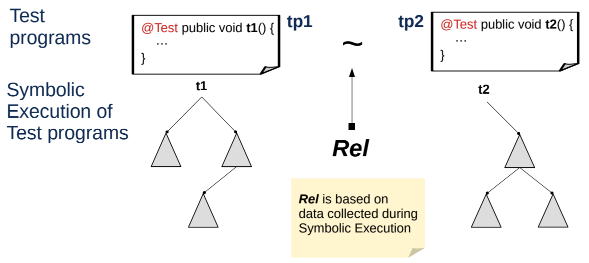

Test programs
@Test public void t1() {
...
}
tp1
~
tp2
@Test public void t2() {
...
}
Symbolic Execution of Test programs
t1
t2
Rel
Rel is based on data collected during Symbolic Execution

Fig. 1: Symbolic Execution and Analysis of Test Programs.

the same instances. Such connections to remote APIs can be directly coded in the test program by means of basic frameworks providing 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 to look for similarity among those test programs that locally activate overlapping APIs. In addition, item 2 is also considered useful when looking for test programs that converge on to a cohesive set of activities: for testing purposes, but also about the configuration of the test environment, or about their referred assertions. Finally, as a further refinement of the considerations about local libraries invocations, the presented approach leaves testers with the possibility to leverage dependencies on those test program implementations that raise, catch, or handle similar types of exceptions (see item 3).

In some cases, the microservices may interact by adhering to some asynchronous communication schema; for example their interactions could be mediated by means of some publish-subscribe middleware remotely shared among the different microservices. In this cases, the analysis of the interactions may require digging information from the payloads exchanged in the shared channels. In this sense, such a kind of communication schema is out of the scope of the referred scenarios.

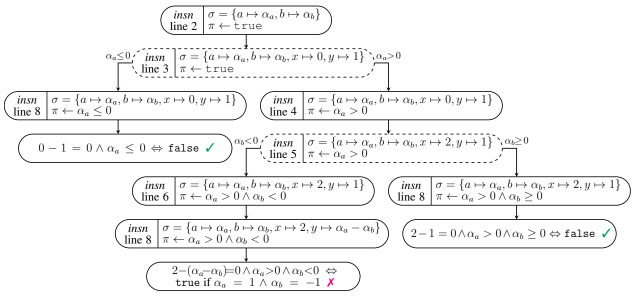
The identification of similarities among test programs is guided by an automatic analysis procedure scanning their implementations (e.g., available from source-code repositories). The analysis procedure assumes that test programs are clearly identifiable from the rest of the source code, for example by means of explicit JUnit annotations. Also, the proposed approach relies on the symbolic execution of each considered test program. The aim of this step is twofold: (i) to carve test data by exploring admissible executions subsumed by the test program; (ii) to exercise (parametric) test programs independently of their arguments.

Indeed, a parametric test program offers greater opportunities than a traditional test program. They arise from recognis-

[2]see: https://spring.io/microservices
[3]see: https://www.postman.com
[4]see: https://swagger.io/tools/swagger-codegen/

```java
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. 2: 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`
$\alpha_a \le 0$
$\alpha_a > 0$
insn line 3 | $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 0, y \mapsto 1\}$ $\pi \leftarrow$ `true`
insn line 8 | $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 0, y \mapsto 1\}$ $\pi \leftarrow \alpha_a \le 0$
insn line 4 | $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 0, y \mapsto 1\}$ $\pi \leftarrow \alpha_a > 0$
$0 - 1 = 0 \land \alpha_a \le 0 \Leftrightarrow$ `false` ✓
$\alpha_b < 0$
$\alpha_b \ge 0$
insn line 5 | $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 2, y \mapsto 1\}$ $\pi \leftarrow \alpha_a > 0$
insn line 6 | $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 2, y \mapsto 1\}$ $\pi \leftarrow \alpha_a > 0 \land \alpha_b < 0$
insn line 8 | $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 2, y \mapsto 1\}$ $\pi \leftarrow \alpha_a > 0 \land \alpha_b \ge 0$
insn line 8 | $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 2, y \mapsto \alpha_a - \alpha_b\}$ $\pi \leftarrow \alpha_a > 0 \land \alpha_b < 0$
$2 - 1 = 0 \land \alpha_a > 0 \land \alpha_b \ge 0 \Leftrightarrow$ `false` ✓
$2-(\alpha_a-\alpha_b)=0 \land \alpha_a>0 \land \alpha_b<0 \Leftrightarrow$ `true` if $\alpha_a = 1 \land \alpha_b = -1$ ✗

Fig. 3: Execution tree of the example program in Fig. 2. Dashed boxes correspond to states in which a branch is taken. The leave node marked with a ✗ is associated with a terminal state which violates the `assert()` in the example program.

ing that the parametric test program has an inherent ability to exercise the system under test in greater form. However, in traditional testing scenarios, parametric tests reduce this actual capability due to the possible small range of (concrete) parameters used to run the tests themselves.

Our overall approach is depicted in Fig. 1. The symbolic execution step explores admissible executions determined by the test programs implementations, and it produces assertions about the reached configurations. The analysis of such collected information follows the symbolic execution phase, and its objective is to reveal existing dependencies among test programs. Specifically the analysis phase is built upon a given set of inference rules which define the dependency criteria. As discussed in the following, an initial set of inference rules has been investigated within this work. Nevertheless, further dependency criteria can be covered by the approach simply developing additional inference rules or extending the existing ones. The execution of the available set of rules with tailored queries allows for specific consultancy of the assertions produced during the symbolic execution phase. The resolution of each query either checks if a dependency criterion is verified against several test program instances, or it proposes evidences that lead to satisfy a given dependency criterion.

IV. Carving Test Program Similarity

This section details the methodology, based on symbolic execution, which is leveraged to carve information from test programs. We also discuss our reference implementation.

A. Information Extraction Methodology

Unlike concrete execution, where a program is run on a specific input and a single control flow path is explored, the basic idea of symbolic execution is to allow variables to take on “symbolic values”, as well as concrete values. This characteristic of symbolic variables allows the simultaneous exploration of multiple paths that a program can take under different inputs. Every time that some condition is checked against a symbolic variable, a *branch* is taken, in the sense that multiple control flows are maintained at the same time by the *symbolic execution engine*.

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

$insn$ is the program point which 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. An SMT solver [38] checks whether there are any violations of the constraints along each explored path, and if the path itself is realisable. As an example, the symbolic execution of the code snippet in Fig. 2 generates the tree depicted in Fig. 3.

We pick symbolic execution as the main methodology to extract information from (parametric) test programs. Indeed, being able to observe all symbolic execution states across which a symbolic execution transits allows us to extract a large amount of information associated with what methods of the system under test are used, or more in general what parts of the system under test are exercised. The extracted information is represented as a set of logical assertions written as *Prolog facts* [18] enabling suitable rule-based reasoning techniques.

In more details, we extract from the traversed states facts related to *method invocation*. These facts track an invocation of a certain method within a certain class, from a certain caller. Given the nature of symbolic execution, these facts describe the *possibility* that, for certain concrete inputs to the test program, a certain method invocation could be materialised in a concrete execution. We recall that we are targeting *parametric tests*, therefore symbolic execution allows us to explore the whole class of inputs that could be fed into test programs, independently of the concrete input values specified by the programmer. The simplest representation of this fact can be the following:

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

where `invokes` is a Prolog predicate, `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. This type of Prolog fact, however, does not

```java
public void f() {
    for(int i = 0; i < 1000; i++)
        g();
}
```

Fig. 4: Effect of the presence of calls within loops.

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

Fig. 5: Effect of recursion.

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

Fig. 6: A fragment generating a sequence of `invokes` equivalent to that of Fig. 5.

carry the information about the symbolic execution path where the invocation takes place, among the various ones generated by different symbolic inputs to the considered test program. The `invokes` predicate must be then augmented to keep track of the point in the symbolic execution tree at which a certain method call is observed. Thus, we introduce a list of *branching points*, which can be regarded as a linear representation of a path in the branching tree. The predicate thus becomes:

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

Let us now consider the example code snippet in Fig. 4. Here, we find repeated invocations to `g()`, which will in turn generate multiple `invokes` facts. One could argue that every single `invokes` fact generated by a call to `g()` is actually a different incarnation, and should be therefore considered different. To enforce this difference, we expand the `invokes` predicate as follows:

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

where `SeqNum` is a monotonic counter which is incremented every time that an `invokes` fact is generated, while `ProgramPoint` is a unique identifier of the location of the method call in the original program. Therefore, every invocation of `g()` in the example in Fig. 4 will bear the same value for `ProgramPoint` and a different value for `SeqNum`, thus allowing us to disambiguate invocations within iterations.

To complete the construction of the `invokes` predicate, let us consider the example provided in Fig. 5. Here, depending on the (either concrete or symbolic) value of the method parameter `count`, a different set of methods is invoked. If the example program is invoked as `a(1)`, a sequence of predicates will be generated, corresponding to the invocations of `b()`, `a(0)`, `b()`, `c()`, `c()`, all appearing as being called from `a()`. The problem, here, is that the same sequence of facts could be also generated by the example program in Fig. 6. The two programs are inherently different though, and cannot be described by the 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 the impossibility to distinguish between different "invocation contexts" in the `invokes` facts. To overcome this limitation, we enhance the predicate as follows:

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

where `FrameEpoch` is an additional monotonic counter which is handled as follows. Every time that 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 that 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.

Finally, we might consider two invocations to the same method as similar if they observe the same symbolic path condition, of if they have the same set of parameters. To this end, the final incarnation of the `invokes` predicate becomes:

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

B. Reference Implementation

To support symbolic execution, we have relied on JBSE [33], a symbolic Java Virtual Machine which allows us to carry out symbolic execution starting from any method within any class.

In order to support the generation of `invokes` facts that enable similarity analysis of complex microservice-based architectures, we have developed a tool (targeting the JUnit testing framework) which enumerates, for a certain test suite, every method which is annotated as `@Test`. We therefore initially build a hashmap that, for every class, allows the retrieval of a set keeping all test programs, which will be the entry points for instances of symbolic execution. To be consistent with the JUnit specification, we also build an additional hashmap to keep track of the set of all methods annotated as `@Before`. These methods will be invoked right before giving control to the test program.

After having discovered all the test programs belonging to the test suite of the microservice-based application, our tool starts an instance of the JBSE virtual machine for every enumerated test program. During the execution, JBSE is configured so as to give back control to our tool once specific conditions are met, in order to carry out information logging activities. In particular, if we traverse a symbolic state associated with the invocation of a method:

- we extract all the information required to assemble an `invokes` fact, and cache it in an in-memory data structure;

```
wrapper method body:
{
   TPClass kl = new TPClass();
   kl.setUp(); // @Before
   kl.testCase(); // @Test
}
```

Fig. 7: Method injected to account for `@Before` annotations.

- we increment the monotonic counters used in the predicates, and we push the frame epoch counter on the frame stack.

Once the execution of a test program completes, we emit Prolog facts on a file on disk, to prepare for the symbolic execution of the next test program in the pool.

As mentioned above, special care is taken to execute test programs that belong to a class in which methods annotated as `@Before` are present. In this case, to mimic the behaviour of JUnit, we perform the following tasks. First, we generate on the fly a method similar to the skeleton provided in Fig. 7. Here, after having created an instance of the test class, we invoke (in any order, which is compliant with the JUnit behaviour) all methods annotated as `@Before`, which were discovered in the previous method enumeration phase. Then, the actual test program is invoked. To generate this artificial method, we rely on Javassist [39].

Nevertheless, running a symbolic execution starting from this artificially-generated method would generate an amount of information which is not strictly related to the test program. Therefore, we rely on a form of *concolic* execution [26], which is essentially a "mixed" concrete/symbolic execution. In particular, we instruct JBSE to rely on a "guided" execution: we launch an additional concrete JVM, starting from an artificial `main()` program which loads via reflection our generated wrapper method, and invokes it. By relying on JDI, we set a breakpoint on the method associated with the test program. Once that breakpoint is hit, JBSE takes back control and runs the same code—up to the breakpoint—by making decisions using the actual branches which were taken in the concrete execution. In this way, we can quickly reach the entry point of our test program, without having to explore execution paths which are not relevant for the extraction of similarity information.

V. Rules for Similarity

The data set consisting of `invokes` facts generated by the symbolic execution of the test programs can be used to conduct a variety of analysis tasks on the functional behaviour of such programs. These tasks may be performed by inspecting the execution traces of the test programs to discover which methods of the system under test are invoked, or the sequences of direct method invocations performed by a caller method to discover which implementations of a specific operation are being tested.

In the following, we show how *Prolog rules* [18] can be conveniently used to sieve through the sets of `invokes` facts at different levels of abstraction, and shape them into suitable data structures to be used within queries for reasoning about the similarity among test programs.

A. From Invokes to Endpoints

In order to inspect the execution traces of a test program, our system provides the predicate `trace(TP,Trace)` (shown in Fig. 8), which relates a test program `TP` to an execution trace `Trace` of the method annotated as `@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).
```

Fig. 8: Prolog rule that defines `trace(TP,Trace)`.

Suppose that, instead of execution traces, we want to analyse the sequences of direct method invocations performed by a caller method annotated as `@Test` in the test program. Our system provides the predicate `invoke_sequence(TP,Caller,ISeq)` (shown in Fig. 9), which relates a test program `TP` and a caller method `Caller` with a sequence of method invocations `ISeq` performed directly by `Caller`. The sequence of method invocations `ISeq` consists of `invokes` facts whose first and fourth arguments are `TP` and `Caller`, respectively.

```
invoke_sequence(TP,Caller,ISeq) :-
  first_invokes(TP,Caller,FirstInvokes),
  iseq(FirstInvokes,ISeq),
  ISeq = [FirstInvokes|ISeqTail],
  last_invokes(TP,Caller,LastInvokes),
  last(ISeqTail,LastInvokes).
```

Fig. 9: Prolog rule that defines `invoke_sequence(TP, Caller,ISeq)`.

Therefore, to get the traces generated by the symbolic execution of a given test program, we can collect the answers to the query `trace(TP,X)`, where `TP` is bound to the test program name and `X` is an unbound variable. Similarly, to compute the sequences of method invocations generated by a given caller in a test program, we can collect the answers to the query `invoke_sequence(TP,Caller,X)`, where `TP` and `Caller` are bound to the test program name and the caller method, respectively, and `X` is an unbound variable.

The answers provide values for `X`, that is, lists of `invokes` facts, that can be analysed for discovering similarity relations between test programs by using suitable helper predicates.

In particular, when testing microservice applications, it is useful to have predicates that: (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 of the form:

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

```
1    filter([],XSchema,XSelFun,XExtFun,YName,[]).
2    filter([X|Xs],XSchema,XSelFun,XExtFun,YName,[Y|Ys]) :-
3        eval_sel_fun(X,XSchema,XSelFun), !,
4        eval_ext_fun(X,XSchema,XExtFun,YArgs),
5        Y =.. [YName|YArgs],
6        filter(Xs,XSchema,XSelFun,XExtFun,YName,Ys).
7    filter([X|Xs],XSchema,XSelFun,XExtFun,YName,Ys) :-
8        filter(Xs,XSchema,XSelFun,XExtFun,YName,Ys).
```

Fig. 10: Prolog rule that defines `filter(Xs,XSchema, XSelFun,XExtFun,YName,Ys)`.

representing that the method `Caller` of the test program `TestProgram` invokes the remote API identified by `URI`, using the HTTP method `HTTPMethod`.

Fig. 10 shows the Prolog rule defining the predicate `filter(Xs,XSchema,XSelFun,XExtFun,YName,Ys)`, which performs the operations (1)–(3) on a given list of `invokes` facts.

From an input list `Xs`, the predicate `filter` generates a list of terms `Ys` satisfying the following conditions. Any element `Y` of `Ys`: (i) is a term with functor `YName`, (ii) is obtained from a term `X` in `Xs` for which all predicates in the list `XSelFun` hold (that is, `XSelFun` is list of functions for selecting elements form `Xs`), and (iii) the i-th argument of `Y` is obtained by applying to `X` the i-th predicate in the list `XExtFun` (that is, `XExtFun` is a list of functions for generating the arguments of `Y` using the information extracted from the selected `X`). The second argument `XSchema` of `filter` is a ground term that defines the structure of any term in `Xs` by assigning labels to its arguments. The mapping between the arguments of `XSchema` and the arguments of terms in `Xs` provides an easy way to get any argument of a term in `Xs` by using the corresponding label in `XSchema`.

In particular, the recursive rules of `filter`, that is, rules starting at lines 2 and 7 in Fig. 10, take the head `X` of `Xs` and perform the following operations. The predicate `eval_sel_fun` in the second rule of `filter` checks if the predicates in `XSelFun` hold for `X`. If so, the predicate `eval_ext_fun` generates the list `YArgs` from `X` and makes use of the Prolog "univ" operator "`=..`" (line 5) to construct a new term `Y`, named `YName`, whose arguments are the terms in `YArgs`; otherwise the third rule of `filter` applies and `X` is ignored. The Prolog "cut" predicate "`!`" (line 3) prevents the application of the third rule whenever the second rule applies.

The code snippet in Fig. 11 shows how to query the `invokes` data set to select those facts that represent an invocation to a remote API, and generate the corresponding `endpoint` facts.

The first argument (line 2) of the query in Fig. 11 is a list of `invokes` facts (e.g., obtained from `trace` or `invoke_sequence`). The second argument (lines 3–11) is a ground term that describes the structure of any `invokes`, that is, a mnemonic for its arguments. The third argument (line 12) is a singleton list consisting of the helper predicate `isHttpMethod` applied to the argument labelled as `callee` in the `invokes` that holds if the callee argument is a call to an HTTP method (that is, `filter` selects only those `invokes` facts that call a remote API through an HTTP method). The fourth argument (lines 13–16) is a list consisting of four helper predicates that extract from the selected `invokes`: (1) the name of the test program (`testProgram`), (2) the method that invokes the remote API (`method(caller)`), (3) the HTTP method (`httpMethod(callee)`, such as get and post), and (4) the first argument of the list of parameters of the HTTP method (`head(parameters)`), that is, the URI of the remote API. The extracted arguments are shaped into a new fact with functor name `endpoint` (line 17).

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

Fig. 11: Prolog query to generate `endpoint` facts.

```
1  matching_endpoint(E1,E2) :-
2    E1 = endpoint(TP1,Caller1,HTTPMethod1,URI1),
3    E2 = endpoint(TP2,Caller2,HTTPMethod2,URI2),
4    HTTPMethod1 == HTTPMethod2,                % (c1)
5    atom_string(URI1,U1), atom_string(URI2,U2),
6    rest_api_regex(REX),
7    re_match(REX,U1), re_match(REX,U2).     % (c2)
```

Fig. 12: Prolog rule that defines the predicate `matching_endpoints(E1,E2)`.

The data set consisting of the `invokes` facts collected during the symbolic execution can thus be enriched with the `endpoint` facts generated from traces (Fig. 13 shows an excerpt of this new data set) or sequences of method invocations.

B. Similarity Reasoning

Now, we are able to introduce a notion of "similarity between endpoints", which will be the basis of a similarity relation between test programs. Given two `endpoint` facts `E1` and `E2`, we say that they are *similar* if and only if:

(c1) they make use of the same HTTP method to invoke a remote API, and
(c2) their URIs match the same regular expression belonging to a list extracted from the test suite.

Indeed, the use of regular expressions is motivated by the fact that URIs may include some parameters related to the call site, which need to be ignored during their comparison.

Similarity between endpoints is evaluated by using the predicate `matching_endpoints(E1,E2)` shown in Fig. 12.

The term `REX` (line 6) is a string representing a regular expression in the Perl-Compatible Regular Expression (PCRE) format asserted in the Prolog database using `rest_api_regex` facts. Pattern matching is performed using the `re_match` predicate (line 7) provided by the SWI-Prolog `pcre` library, and `atom_string` (line 5) is a built-in predicate to convert atoms into strings—according to the definition of similarity between endpoints, the name of test programs, `TP1`, `TP2`, and the callers `Caller1`, `Caller2`, occurring in the `endpoint` facts are ignored.

Now, we can use the following predicate to reason about similarity of test programs:

```
similar_tp(EpSrc,SimCr,TP1,TP2,Es1,Es2)
```

where:

- `EpSrc` specifies the source of the `endpoint` facts,
- `SimCr` specifies the criterion to evaluate the similarity between the test programs `TP1` and `TP2`, and
- `Es1` and `Es2` are nonempty lists of `endpoint` facts generated from the `invokes` facts of `TP1` and `TP2`, respectively, that make `TP1` and `TP2` similar.

Without loss of generality, we assume to use `endpoint` facts generated from the execution traces.

Among the many similarity criteria that can be defined, in this paper we consider the criterion called "`nonemptyIntersection`", for which two test programs `TP1` and `TP2` are *similar* if and only if there exist two nonempty lists `Es1` and `Es2` of `endpoint` facts, extracted from the traces of `TP1` and `TP2`, respectively, and two `endpoint` facts `E1` in `Es1`, `E2` in `Es2`, such that `matching_endpoints(E1,E2)` holds.

In order to quantify the *degree* of similarity between test programs with respect to lists of endpoints, our tool also provides the predicate `similarity_score(SimCr, Es1,Es2,Score)`, that computes a `Score` in $(0, 1]$ from any answer to the query `similar_tp(EpSrc,SimCr, TP1,TP2,Es1,Es2)`.

In particular, for `nonemptyIntersection`, the value of `Score` is computed as follows:

$$\texttt{Score} = \frac{|\text{matchingSet}(\texttt{Es1},\texttt{Es2})|}{\min(|\text{setOf}(\texttt{Es1})|, |\text{setOf}(\texttt{Es2})|)}$$

where:

- setOf(`L`) is the set of distinct elements occurring in the list `L`, and
- matchingSet(`L1`,`L2`) is the nonempty set of all elements `E1` in setOf(`L1`) such that there exists an element `E2` in setOf(`L2`) for which the predicate `matching_endpoint(E1,E2)` holds.

VI. Case study

We have carried out an evaluation of the viability of our approach by means of a case study based on a real-world application[5]. In particular, we have used a microservice-based application called *Fullteaching*[6], an educational platform based upon OpenVidu, which is an open-source videoconferencing system employing the WebRTC API [40].

Fullteaching provides an extensive test suite implemented using JUnit 4, which includes a total of 88 test programs. Among them, 29 tests require contacting remote URIs, either for integration or end-to-end testing purposes. These are the test programs used for our case study, as they involve the invocation of URIs using get, post, put, and delete methods. All RESTful invocations are managed through the `MockMvc` class by the Spring testing framework [41].

By the rules discussed in Section V, we have generated the `endpoint` facts that describe the URI(s) invoked by the test programs. An excerpt is provided in Fig. 13, 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`?"

With respect to the `nonemptyIntersection` similarity metric discussed above, in Fig. 14 we report similarity matrices (in the form of heatmaps) for all considered test programs[7]. Since our symbolic execution tool 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. 14a and Fig. 14b, respectively, the minimum and the maximum score values—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 is a symmetric matrix.

To understand the usability of this information in the context of a governance framework for regression testing, let us discuss some values related to Fig. 14a and Fig. 14b. If we consider the test program U, which tests login capabilities of Fullteaching users, we observe that it is associated with a minimum/maximum value of 0.5. The test programs we have taken into account are all associated with authenticated APIs: all test programs try to create a user (if it does not exist), authenticate it, perform some action, and then conclude the session. Therefore, U's similarity score is stable with respect to the other test programs, and it is set to a low value. In this sense, we cannot consider it much similar to other test programs. On the other hand, M is a test program which alters several aspects related to a whole course, thus interacting with many parts of the system under test. Due to its behaviour, it exhibits a high score value, which is stable. This allows us to conclude that it can be considered pretty similar to the other test programs.

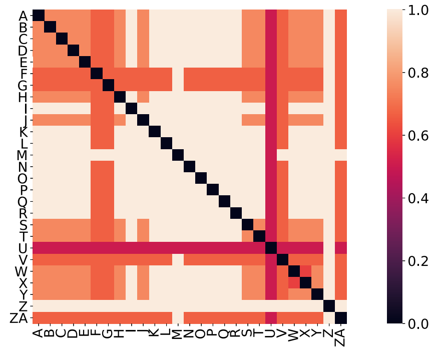
Let us now focus on test program A. If we compare the minimum and maximum scores of the pairs A–O and A–V, we may try to answer the question: "to which test is A most similar?". The pair A–O is associated with a minimum/maximum value of 1.0, while A–V has a minimum/maximum value of 0.75. We might conclude that, as far as endpoint

[5]The replication package is distributed with the source code of Hyperion
[6]https://github.com/OpenVidu/full-teaching

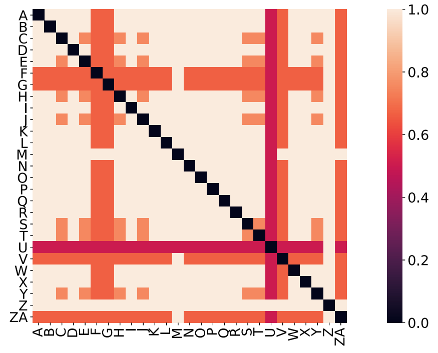
[7]The match with the test names in Fullteaching is in the replication package.

```
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. 13: Example `endpoint` facts generated by the rules for similarity.



(a) Minimum Similarity Score.



(b) Maximum Similarity Score.

Fig. 14: Similarity Matrices.

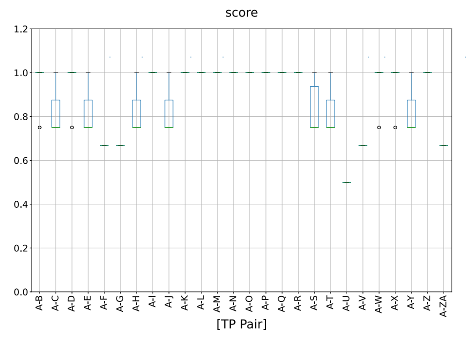
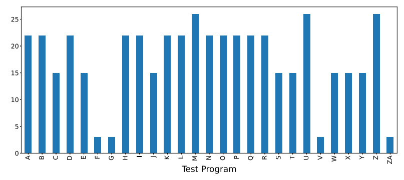
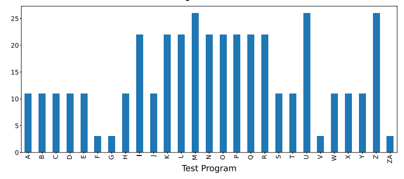


Fig. 15: Effect of Multiple Symbolic Traces



(a) Similarity Score $\geq$ 0.75.



(b) Similarity Score = 1.00.

Fig. 16: Number of Test Programs Deemed Similar.

invocations are concerned, A is more similar to O than V. On the other hand, if we compare the values of the pairs A–T and A–V, we observe for A–T a minimum value of 0.75 and a maximum value of 1.0 (depending again on the multiple symbolic execution traces observed by our tool), while A–V is stable at 0.67. In this case, we cannot conclude much on the similarity among A, T, and V.

However, if we observe the results in Fig. 15, we can extract more information. In the figure, we have selected a single test program (A) and we have displayed the dispersion of the similarity score with all the other test programs. By looking at these results, we might conclude that A is more similar to T than V, but not as much as we might imagine by looking at Fig. 14. It is also interesting to note that, for some pairs (e.g., A–K), there is no dispersion at all—this is also reflected in Fig. 14, where both the minimum and maximum values are the same. This can be related to the fact that in the symbolic execution tree there is only one feasible path for test program K. In contrast, for other test programs, there are multiple execution traces to compare, and therefore different similarity scores are derived.

In Fig. 16 we show the number of test programs that can be deemed similar by relying on our metric. In particular, for each test program, we show the number of other test programs that have a median similarity score among all symbolic execution traces above 0.75 (Fig. 16a), and exactly 1.00 (Fig. 16b). As expected, the number of test programs deemed similar decreases for higher median values. This is an additional indication of the versatility of our approach. Indeed, higher values of the similarity score might help at defining narrower governance policies that can be enforced for reducing regression test suites by selecting some test cases, skipping redundant ones, or prioritising those expected to yield earlier fault detection.

VII. Conclusions and Future Work

We have discussed a methodology to extract relations among (parametric) test programs for microservices applications. Through symbolic execution, we are able to extract from a test suite relevant information about the methods that may be called along some execution path—i.e., independently of the actual concrete inputs fed into the test programs. This information is processed by a set of Prolog rules, so as to determine the endopoints that may be activated by running the various test programs, and then used for computing a similarity score. In a case study, 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.

Future work will focus on a larger experimentation of the proposed approach against known benchmarks or other available microservices applications. In addition, future research directions will explore the formulation of alternative similarity rules and their comparative study, as well. Finally, we plan to apply the dependencies among regression test cases in the context of test orchestrations graphs (i.e., active test case selection and prioritisation), where the next set of test cases to execute are dynamically chosen taking into account either the observed test verdicts (i.e., test passed or failed), or the output data produced by earlier executions.

Acknowledgments

This paper has been supported by the Italian MIUR PRIN 2017 Project: SISMA (Contract 201752ENYB), and partially by the Italian Research Group: INdAM-GNCS.

References

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

[2] J. Lewis and M. Fowler, "Microservices, a definition of this new architectural term," Mar. 2014.

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

[4] H. Vocke, "The practical test pyramid," Feb. 2018.

[5] V. de Oliveira Neves, A. Bertolino, G. De Angelis, and L. Garces, "Do we need new strategies for testing systems-of-systems?" in *Proc. of SESoS*, 2018, pp. 29–32.

[6] A. Bertolino, G. De Angelis, and F. Lonetti, "Governing regression testing in systems of systems," in *Proc. of ISSRE Workshops, GAUSS*. IEEE, Oct. 2019, pp. 144–148.

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

[8] D. Spinellis, "State-of-the-art software testing," *IEEE Software*, vol. 34, no. 5, pp. 4–6, 2017.

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

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

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

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

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

[14] M. J. Harrold and A. Orso, "Retesting software during development and maintenance," in *Proc. of FoSM*, 2008, pp. 99–108.

[15] L. Gazzola, M. Goldstein, L. Mariani, I. Segall, and L. Ussi, "Automatic ex-vivo regression testing of microservices," in *Proc. of AST*. ACM, 2020, pp. 11–20.

[16] M. Bruno, G. Canfora, M. D. Penta, G. Esposito, and V. Mazza, "Using test cases as contract to ensure service compliance across releases," in *Proc. of ICSOC*, ser. LNCS, vol. 3826. Springer, 2005, pp. 87–100.

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

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

[19] M. Waseem, P. Liang, G. Márquez, and A. D. Salle, "Testing microservices architecture-based applications: A systematic mapping study," in *Proc. of APSEC*. IEEE, 2020, pp. 119–128.

[20] K. Meinke and P. Nycander, "Learning-based testing of distributed microservice architectures: Correctness and fault injection," in *Proc. of SEFM Workshops*, ser. LNCS, vol. 9509. Springer, 2015, pp. 3–10.

[21] J. G. Quenum and S. Aknine, "Towards executable specifications for microservices," in *Proc. of SCC*. IEEE, 2018, pp. 41–48.

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

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

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

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

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

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

[28] P. Braione, G. Denaro, A. Mattavelli, and M. Pezzè, "SUSHI: a test generator for programs with complex structured inputs," in *Proc. of ICSE*. ACM, 2018, pp. 21–24.

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

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

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

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

[33] P. Braione, G. Denaro, and M. Pezzè, "JBSE: A symbolic executor for Java programs with complex heap inputs," in *Proc. of FSE*. ACM, 2016, pp. 1018–1022.

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

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

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

[37] B. García, F. Lonetti, M. Gallego, B. Miranda, E. Jiménez, G. De Angelis, C. E. Moreira, and E. Marchetti, "A proposal to orchestrate test cases," in *Proc. of QUATIC*, 2018, pp. 38–46.

[38] C. Barrett, D. Kroening, and T. Melham, "Problem solving for the 21st century: Efficient solvers for satisfiability modulo theories," LMS and the Smith Institute, Tech. Rep. 3, June 2014.

[39] S. Chiba, "Load-time structural reflection in java," in *Proc. of ECOOP*. Springer, 2000, pp. 313–336.

[40] A. B. Johnston and D. C. Burnett, *WebRTC: APIs and RTCWEB protocols of the HTML5 real-time web*. Digital Codex LLC, 2012.

[41] R. Johnson, J. Hoeller, K. Donald, C. Sampaleanu, R. Harrop, T. Risberg, A. Arendsen, D. Davison, D. Kopylenko, M. Pollack *et al.*, "The spring framework reference documentation," The Spring Framework, Tech. Rep., 2016.