A Proposal to Orchestrate Test Cases

Boni García
Universidad Rey Juan Carlos
Calle Tulipán S/N
28933 Móstoles, Spain
boni.garcia@urjc.es

Francesca Lonetti
CNR-ISTI
Via Moruzzi 1
56124 Pisa, Italy
francesca.lonetti@isti.cnr.it

Micael Gallego
Universidad Rey Juan Carlos
Calle Tulipán S/N
28933 Móstoles, Spain
micael.gallego@urjc.es

Breno Miranda
Federal University of Pernambuco
and *CNR-ISTI*
Via Moruzzi 1
56124 Pisa, Italy
bafm@cin.ufpe.br

Eduardo Jiménez
Universidad Rey Juan Carlos
Calle Tulipán S/N
28933 Móstoles, Spain
edu.jg@urjc.es

Guglielmo De Angelis
CNR-IASI
Via dei Taurini, 19
00185 Roma, Italy
guglielmo.deangelis@iasi.cnr.it

Carlos Santos
Universidad Rey Juan Carlos
Calle Tulipán S/N
28933 Móstoles, Spain
carlos.santos@urjc.es

Eda Marchetti
CNR-ISTI
Via Moruzzi 1
56124 Pisa, Italy
eda.marchetti@isti.cnr.it

***Abstract*—This paper presents the concept of test orchestration, understood as a novel way to select, order, and execute in parallel a group of tests. Our view of test orchestration can be seen as a process in which different test cases are organized, assembled and executed following a topology that determines how their executions coordinate. We distinguish two types of orchestrations techniques: i) *verdict-driven*, which organizes tests using their outcome (i.e., passed or failed) to drive the workflow; and ii) *data-driven*, in which test data (input) and test outcomes (output) are handled within the graph. Both approaches are being implemented in the project ElasTest, an open source platform aimed to simplify the end-to-end test process of large software systems.**

***Index Terms*—Software testing, test composition, test parallelization.**

I. Introduction

Since the early origins of software engineering, the "divide and conquer" principle (i.e., dividing a problem into its parts, solving them separately and creating a global solution by the combination of the partial ones) has been one of the main strategies for practitioners [1]. Being a complex and costly activity, software testing has also extensively applied such principle, for example in input domain partitioning [2], and mostly with the goal of reducing the test suite [3]. However, the potential of this simple strategy has not been fully exploited for managing complex test sequences within one test session. In this piece of research, we hypothesize that simpler test cases can be organized to create a complex compound test case into one *test orchestration*.

Our contribution in this area is the creation of a new notion of test orchestration as a process through which different test cases are organized, assembled and executed following a topology (i.e., graph) of dependencies that determines how their executions coordinate. This contribution is materialized through a number of novel concepts that include: i) the formalization of the test cases that can be interconnected for creating the topology; ii) the creation of an orchestration notation enabling testers to specify explicitly (i.e., edges and nodes) how the orchestration takes place; and iii) the creation of an engine interpreting that notation for orchestrating tests. All these contributions are being proposed and developed in the context of an European Union funded project called ElasTest [4]. In this project, we are creating a cloud platform designed for helping developers to test large software testing. For this, ElasTest is based on three principles: i) instrumentation (i.e., customization of the System Under Test (SUT) infrastructure so that it reproduces real-world operational behavior); ii) test orchestration (i.e., to combine intelligently test cases for creating a more complete test suite); and iii) test recommendation (i.e., to use machine learning and cognitive computing for recommending testing actions and providing testers with friendly interactive facilities for decision taking). This paper covers the second principle.

The remainder of the paper is structured as follows. Section II provides a comprehensive review of the theoretical background about test composition, parallelization, and orchestration languages. Then, we propose different theoretical approaches aimed to model our notion of test orchestration in Section III. This proposal is being implemented in ElasTest, as an ElasTest internal component called ElasTest Orchestration Engine (EOE). Thus, Section IV presents an overview of ElasTest, putting the accent in this EOE component. Section V presents a case study aimed to validate the *verdict-driven* test orchestration approach. After that, Section VI discuss the contributions of this paper by means of a SWOT (Strengths, Weaknesses, Opportunities, Threats) analysis. Finally, the conclusions of this paper together with the future work is summarized in Section VII.

This work has been supported by the European Commission under project ElasTest (H2020-ICT-10-2016, GA-731535); by the Regional Government of Madrid (CM) under project Cloud4BigData (S2013/ICE-2894) cofunded by FSE & FEDER; by the Spanish Government under project LERNIM (RTC-2016-4674-7) cofunded by the Ministry of Economy and Competitiveness, FEDER & AEI; and by the GAUSS national research project, funded by the MIUR under the PRIN 2015 program (contract 2015KWREMX).

II. Background

This section derives a comprehensive snapshot of existing work in related areas to the concept of test orchestration presented in this paper. To that aim, three different areas are studied: i) test composition, i.e., existing approaches to combine tests; ii) test parallelization, i.e., run test cases in parallel; and iii) orchestration languages, i.e., exiting notations to describe processes, pipelines or workflows.

A. Test composition

The concept of test composition with the aim of increasing its effectiveness while reducing the overall costs and effort has already been addressed in the literature. For instance, [5] and [6] are based on letting developers create elementary test cases involving simple predicates (e.g., "insert authentication pin", "user is authenticated", "user is blocked"). Then, the testing system composes the execution of these test cases for deducing the validity of logical formulae, which are also provided by the tester (e.g., "if after inserting authentication pin, authentication fails three times, user should be blocked"). This type of approach enables testers to reduce test code to the test cases and the formulae. However, there are not well established methodologies on how to generate such cases and a strong theoretical background (e.g., temporal logic) is requested from developers to do it. In addition, the computational complexity may be prohibitive for large systems where the number of cases may be huge. Due to this, compositional testing has traditionally only been used for testing small software systems.

Combinatorial testing [7] aims at reducing the testing complexity and costs through an approach involving: i)modeling the SUT as a set of input factors; ii) generating a sample of the possible combinations of factors and values; and iii) creating and executing test inputs corresponding to that sample. Although combinatorial testing is being applied in relevant application domains [8] it still has relevant limitations preventing its seamless use in the testing of large software systems. Notably, it does not provide any notion of composition or sequencing of tests, and the problem of evaluating combinatorial explosions of factors in terms of testing cost or time is only recently investigated, e.g., by Demiroz and Yilmaz [9]. However, cloud resources are leveraged to perform combinatorial tests execution in parallel and identify faulty interactions through concurrent test algebra execution and analysis [10].

B. Test parallelization

Modern software codebases contain lots of individual test cases. The execution of these test suites takes relevant amounts of time, and as a result, development and release procedures tend to be time-consuming. In order to solve this issue, test parallelization has been proposed as a solution. Recently, Candido et al. [11] conducted an empirical survey on the impact of test suite parallelization in open source projects. The authors reported that only 19.1% of the projects analyzed use parallelization, being the major deterrent to its adoption the resistance concerning concurrency issues.

Existing approaches on test parallelization assume, either implicitly or explicitly, independence among tests being executed. This assumption is not always true in practice, since test executions in parallel can produce non-deterministic outcomes. Zhang et al. [12] investigate the existence of dependent tests in 5 popular open source projects, finding a total of 96 dependent tests, 95 of which would result in a false negative when executed out of order.

Additional current research efforts on test parallelization are focused on test dependency. Gambi et al. [13] present Cloud Unit Testing (CUT), a tool for automatically executing unit tests in distributed execution environments. This work is continued in PRADET, another tool for detecting problematic dependencies in a reasonable amount of time for projects with thousands of tests [14].

C. Orchestration languages

One of the closest approaches related to our rich concept of test orchestration is implemented in the Jenkins pipelines. A Jenkins Pipeline[1] is made up of several *steps*, and each step tells Jenkins what to do, serving as the basic building block for both declarative and scripted pipeline syntax. A Jenkins Pipeline is written using a Domain Specific Language (DSL) syntax based on Groovy [15]. Typically, the definition of a Jenkins Pipeline is written into a text file (called a `Jenkinsfile`) implementing the test workflow, including checking out the project's source control, executing tests, reporting, deploying, etc.

Another relevant language to describe cloud infrastructures is AWS CloudFormation[2]. It is based on JSON, and provides a common language to describe and provision all the infrastructure resources in AWS cloud environments. Moreover, the OpenStack Foundation has defined Heat[3], a project which implements an orchestration engine to launch multiple composite cloud applications based on templates. The latter are conceived as text files that are readable and writable by humans, and can be checked into version control, diffed, etc.

Regarding orchestration languages (not strictly related to testing), we can find different approaches. First, we could use TOSCA[4] (Topology and Orchestration Specification for Cloud Applications). TOSCA is an OASIS (Organization for the Advancement of Structured Information Standards) language to describe a topology of cloud based web services, their components, relationships, and the processes that manage them. Its first version is based on XML. Moreover, TOSCA implements a profile based on YAML. This profile has been adopted by several solutions, such as in:

- Cloudify[5] is an open source software cloud orchestration product. It implements DSL configuration files called blueprints which define the application's configurations, services and their dependencies. The cloudify blueprint

[1] https://jenkins.io/doc/book/pipeline/
[2] https://aws.amazon.com/cloudformation/
[3] https://wiki.openstack.org/wiki/Heat
[4] https://www.oasis-open.org/committees/tosca/
[5] http://cloudify.co/

files describe the execution plans for the lifecycle of the application for installing, starting, terminating, orchestrating and monitoring the application stack. Cloudify also supports configuration management tools like Chef, Puppet, or Ansible for the application deployment phase, as a method of deploying and configuring application services.

- Alien4Cloud[6] (Application LIfecycle ENabler for Cloud) is an open source TOSCA based designer and Cloud Application Lifecycle Management Platform. At the moment of this writing, the topology definition in Alien4Cloud can be done using simple profile in YAML v1.0 and also with the Alien4Cloud 1.3 DSL.
- Ubicity[7] is a Model-Driven Service Management technology aimed to simplify service management on cloud stack. Ubicity is also based on TOSCA YAML profile for describing the topology of cloud-based services.

Regarding workflow definition, a promising alternative is the Common Workflow Language[8] (CWL), which is a specification for describing analysis workflows and tools in a way that makes them portable and scalable across a variety of software and hardware environments. CWL documents are written in JSON or YAML. CWL documents are made up of different parts to define the workflow: metadata, environment, input and output parameters, and steps. This structure could fit with our rich notion of test orchestration. Nevertheless, at the moment of this writing, CWL does not allow advanced workflow steps, such as loops, conditional, or parallel tasks. Similar features are planned in the CWL backlog for next future releases.

III. Test Orchestration Approaches

In our approach, a test orchestration is understood as the interconnection of different tests expressed as a graph. The precise form of the graph (i.e., first one test, then this other one) is specified somehow by the tester. We propose two different types of test orchestration, which we refer to as *verdict-driven* and *data-driven*.

Fig. 1 depicts the verdict-driven approach. This notion of orchestration does not assume any constraints or model availability neither on the test cases nor in the execution topology nor in the SUT. Therefore, test are seen like black boxes (Fig. 1-a). Internally, tests exercise the SUT with some custom logic and assertions, and as a result provide a verdict (test passed or test failed).

A set of tests (i.e., a test suite) is typically executed in a Continuous Integration (CI) server or by a build tool. The order in which tests are executed is not predefined (Fig. 1-b). For example, the order of execution in JUnit tests is non-deterministic by default [16].

We propose a custom notation to select some of the tests within the test suite, ordering the execution as a graph. Moreover, we introduce conditional paths based on the test verdict (i.e., passed or failed) about previous test verdict in the graph (Fig. 1-c). Finally, as an advanced feature of this mode, we can also parallelize the execution of a number of tests, as depicted in Fig. 1-d. Again, we can use the verdicts of the parallel tests to feed a conditional, using logic operators (*OR*, *AND*, etc.) to create richer conditions.

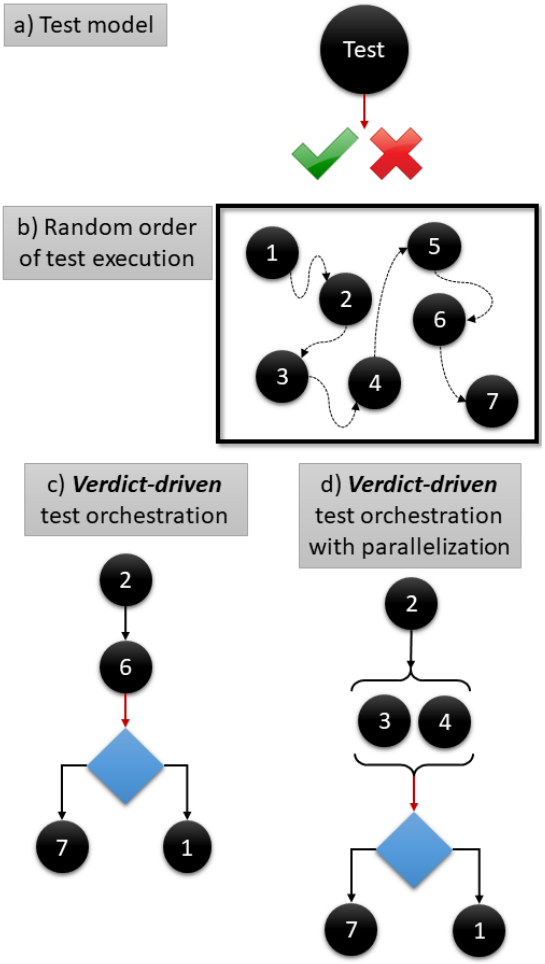

Fig. 1. Verdict-driven test orchestration

Then, Fig. 2 depicts the second approach, called data-driven. This approach is more advanced in the sense that test needs to be prepared to be interconnected, i.e., they should be composable. To achieve this, we put the accent into the test data (input) and the outcome (output). Therefore, a test is modeled as a set of input data which is incoming to the test, and as a result of the execution of the specific test's logic, some output data is generated (in addition to the usual test verdict, i.e., pass or fail). This concept is shown in Fig. 2-a, where the test is colored as green to differentiate to the "regular" test cases, used in the previous approach and represented as black colored circles. Again, these tests are executed in a CI server randomly (Fig. 2-b).

With this schema in mind, the test orchestration is richer in several ways. First, the output data of each test is used to

[6]https://alien4cloud.github.io/
[7]https://ubicity.com/
[8]http://www.commonwl.org/

feed the next test in the resulting graph. This is described in Fig. 2-c. Notice that the output data of each stage is used to feed the input of successive tests. Moreover, the output data can be used to control the workflow in conditional statements. In other words, not only the test verdict can be used to create logic conditions in the workflow, but richer operator conditions can be employed by comparing the test output with custom oracles. Moreover, constraints can be specified to the input or output data within the test, adding extra assertions to the test. Finally, tests can be parallelized in this approach as well as depicted in Fig. 2-d.

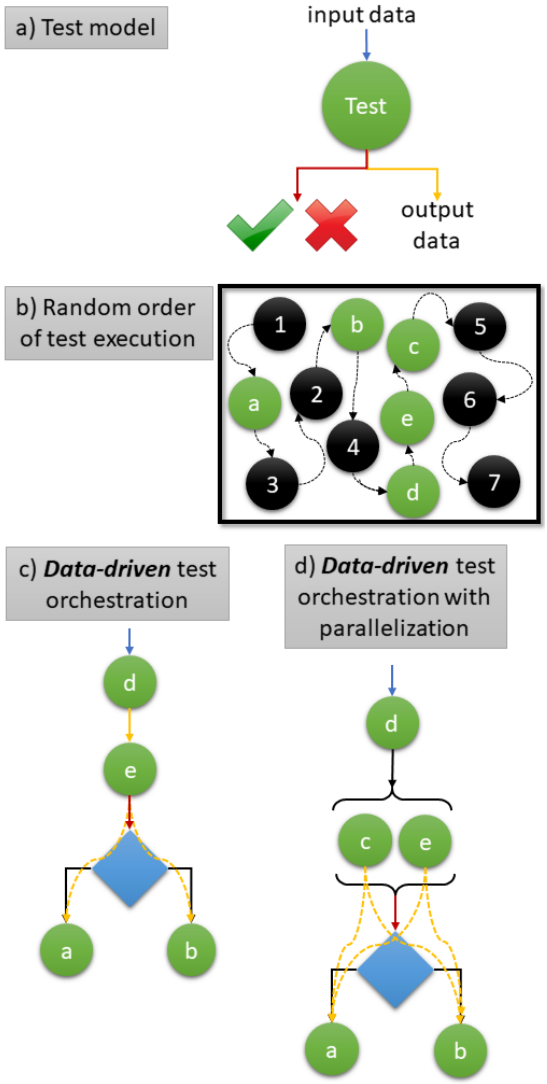

Fig. 2. Data-driven test orchestration

IV. Test Orchestration in ElasTest

The test orchestration approach as defined in previous section is being implemented in the context of the ElasTest project. The aim of this section is two-folded. On the one hand, it presents an overall of the ElasTest platform. On the other hand, it provides the details of the orchestration mechanism within ElasTest.

A. *ElasTest in a nutshell*

ElasTest[9] is an open source[10] platform aimed to ease the end-to-end testing activities for different types of distributed applications and services, allowing developers and testers to assess their cloud applications in an elastic, and integrated environment. ElasTest manages the full testing lifecycle, deploying and monitoring the SUT, executing the end-to-end tests and exposing the results to software engineers and testers.

In order to understand how ElasTest works, it is worth to review its internal architecture, depicted in Fig. 3. First of all, we find the ElasTest Test Orchestration and Recommendation Manager (ETM), which is the access point to the platform. It manages all other components exposing different interfaces for consumers, such as a web GUI, a command line interface, and also an interface with a custom Jenkins plugin.

ElasTest follows a microservices approach, and the component which is responsible for discovering and operating the different services that ElasTest make available to tests is called ElasTest Service Manager (ESM). This component is based on the Open Service Broker API (OSBA)[11] for discovering, registering and unregistering services within the platform. RabbitMQ[12] is used as messaging queue for the events communication among the different services.

One of the key aspects handled out of the box by ElasTest is related with data management. During its operation, ElasTest gathers different sources of data from test execution, including SUT logs, different types of metrics -including SUT resource consumption, packet-loss in the network traffic, or node failures, among others-, or custom files issued by services -e.g., browser/mobile session recordings carried out by EUS-. The component responsible for the persistence layer is called ElasTest Data Manager (EDM), and it has been built on the top of on MySQL[13] as relational database, Elasticsearch[14] as search engine, and Alluxio[15] as virtual distributed storage system.

The ElasTest Instrumentation Manager (EIM) provides the capability of instrumenting the SUT to inject potential system failures like packet-loss, network bandwidth adjustments to emulate real conditions, CPU bursting, and node failures, to name a few. To that aim, Beats[16] agents are installed together with the SUT.

Finally, the ElasTest Platform Manager (EPM) is the component responsible of isolating the ElasTest services from the underlying infrastructure. The supported cloud infrastructures

[9]http://elastest.io/
[10]https://github.com/elastest/
[11]https://www.openservicebrokerapi.org/
[12]https://www.rabbitmq.com/
[13]https://www.mysql.com/
[14]https://www.elastic.co/
[15]https://www.alluxio.org/
[16]https://www.elastic.co/products/beats

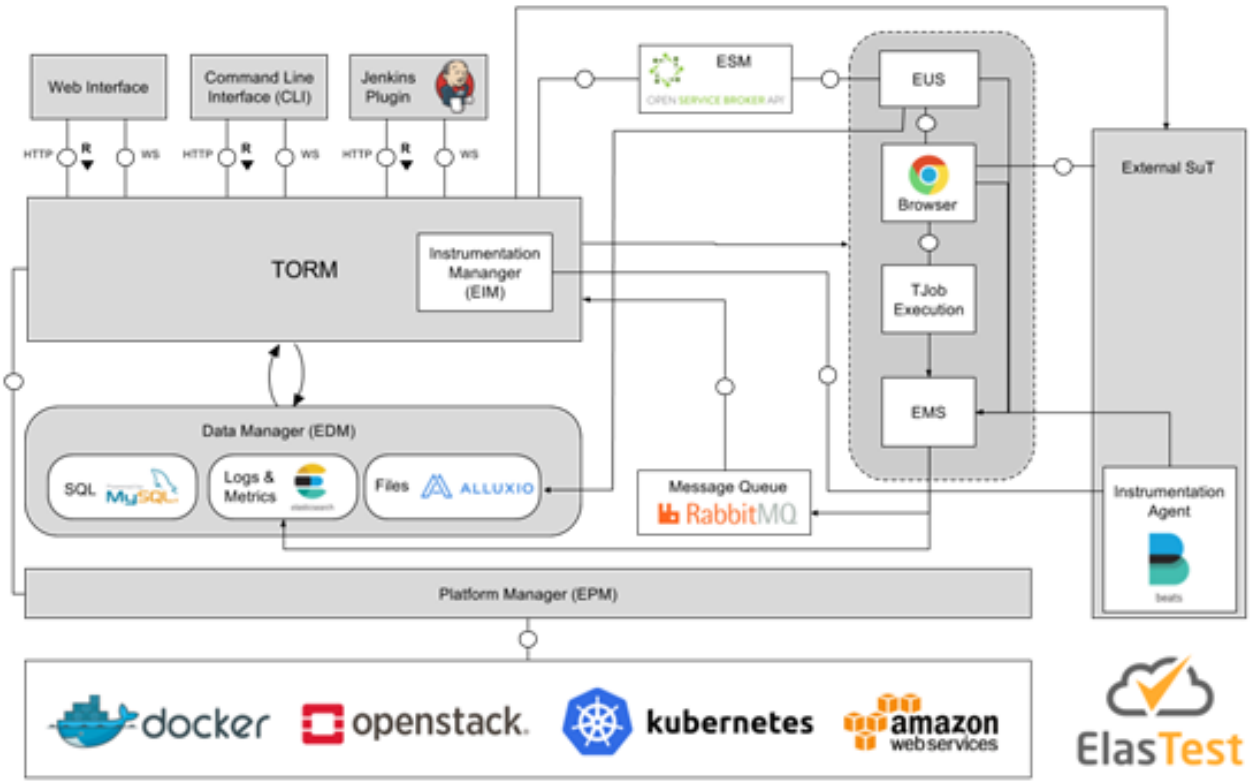

Fig. 3. ElasTest architecture

are OpenStack[17], Amazon Web Services[18] (AWS), Docker[19] and Kubernetes[20]. Moreover, Open Baton[21] is used for orchestrating the SUT and the network services within the ElasTest platform [17].

B. ElasTest Orchestration Engine

In order to implement the concept of orchestration as defined in Section III, first of all we need to select a strategy to define a graph of interconnected tests. As introduced in Section II-C, there are different alternatives for creating workflows and orchestration languages. Due to its flexibility, we leverage the DSL notation of Jenkins pipelines, both for verdict and data-driven approaches. Concretely, we have implemented a Jenkins *shared library* which exposes an simple API to orchestrate jobs. Job is the name given to single execution units in a CI server such as Jenkins, typically composed by one or several tests. The orchestration Jenkins library was been implemented in Groovy language. It is open source an available on GitHub[22]. It provides a high-level class called *orchestrator* which exposes the following methods:

- *runJob(String jobId)*: Method to run a Jenkins job given its identifier (*jobId*). The execution of the will be declared as an stage in a Jenkins pipeline. This method returns a boolean value: *true* if the execution of the job finishes correctly and *false* if fails.
- *runJobDependingOn(boolean verdict, String job1Id, String job2Id)*: This method allows to run one job given a boolean value (typically a verdict from another job). This boolean value is passes in the first argument (called *verdict* in the method signature). If this value job with identifier *job1Id* is executed. Otherwise it is executed *job2Id*.
- *runJobsInParallel(String... jobs)*: This method allows to run a set of jobs in parallel. The jobs identifier are passes using a variable number of arguments (*varargs*).

Moreover, the *orchestrator* class can be configured using the following different options. First, different exit condition for the orchestration can be selected. To that aim, a Groovy enumeration with the following options is provided:

- *EXIT_AT_END*: The orchestration finishes at the end (option by default). This means that even though an intermediate job fails, the orchestration continues until the end of the graph.
- *EXIT_ON_FAIL*: The orchestration finishes when any of the jobs fails.
- *EXIT_ON_PARALLEL_FAILURE*: The orchestration fin-

[17]https://www.openstack.org/
[18]https://aws.amazon.com/
[19]https://www.docker.com/
[20]https://kubernetes.io/
[21]https://openbaton.github.io/
[22]https://github.com/elastest/elastest-orchestration-engine/

ishes when any a set of parallel jobs fails.

Finally, the condition used to give a verdict about parallel jobs can be also configured. There are two options:

- *AND*: Using this option, the verdict of a set of jobs executed in parallel is *true* only if all the jobs finishes correctly. This is the default option.
- *OR*: Using this option, the verdict of a set of jobs executed in parallel is *true* when at least one of the jobs finishes correctly.

An example of orchestration notation using the orchestrator Jenkins library is shown in the following listing. In this example we can see how the library is configured at the beginning. After that, the graph of jobs is declared. A job identified as *myjob1* is executed first place. According to the next sentence, if the verdict of the execution of this job is success, then *myjob2* is executed. Otherwise *myjob3* is executed. After that, a set of jobs is executed in parallel: *myjob4* and *myjob5*. The result of this execution if computed when both jobs finished, and in this example it will be based using the *OR* boolean operation (as configured at the beginning of the orchestration). To conclude, a manual condition is defined using the result of the previous parallel job execution.

```
@Library('OrchestrationLib') _

// Config
orchestrator.setContext(this)
orchestrator.setParallelResultStrategy(
    ParallelResultStrategy.OR)
orchestrator.setExitCondition(
    OrchestrationExitCondition.EXIT_ON_FAIL)

// Graph
def result1 = orchestrator.runJob('myjob1')
orchestrator.runJobDependingOn(result1,
    'myjob2', 'myjob3')
def result3 = orchestrator.
    runJobsInParallel('myjob4', 'myjob5')

if (result3) {
    orchestrator.runJob('myjob6')
    orchestrator.runJob('myjob7')
}
else {
    orchestrator.runJob('myjob8')
}
```

Next, we need to implemented a software artifact able to understand this notation and carry out the actual orchestration of tests. We call this component ElasTest Orchestration Engine (EOE), following the ElasTest naming conventions. This component has been implemented as a new ElasTest's microservice, and lives together with the rest of the components as described in Fig. 3. Internally, EOE has been implemented in Java, and it is deployed as a Docker container within ElasTest. The structure and relationship with other components within ElasTest is illustrated in Fig. 4 and it is explained in the next paragraphs.

EOE is in charge of handling test orchestrations, both verdict and data-driven within ElasTest. To that aim, EOE uses as input the DSL orchestration language introduced before. As a result, EOE is aware of the amount of tests to be executed and its relationships in terms of conditional paths and test data (in the case of data-driven). After parsing the DSL notation, EOE performs in a different way for verdict and data-driven orchestration.

Regarding verdict-driven orchestration, EOE basically starts tests in sequence in synchronous fashion. That means that it starts the first test, wait until it finishes, and then the next one. EOE is also capable of executing tests in parallel if required.

Regarding data-driven orchestration, EOE works in a more complex fashion. In this case, as depicted in previous sections, tests are supposed to be composable, and for that reason, these tests need to be created beforehand following some guidelines, as follows:

- Just before the actual test starts, the test sends a message to EOE asking for permission to execute the test logic. In other words, the test is paused until EOE gives the grant to be started. At this point, EOE also injects the input data in the test.
- Just before the test instance is disposed, the test sends a message to EOE informing the output data together with the test result.

The idea is that EOE starts all the test at the beginning of the execution. This way every test is able to resolve its dependencies, pausing the execution just before the actual test. After that, EOE sends the proper signal to start the test execution in the proper order (established in the DSL workflow). Before this signal, the input data is injected in the test. If the test is intermediate, this input data will be provided by the output data of the previous test. Both output data and test verdict should be sent at the end of the tests. This data can be used in the EOE (according to the workflow) to decide next test. Of course, the SUT is always the same among the different test executions. The idea is that the state of the SUT is evolving from some initial condition through the different steps according to the DSL orchestration.

Moreover, EOE behaves as a proxy for ElasTest's services, called Test Support Services (TSS) in the ElasTest jargon. The idea is that EOE intercepts these calls to share sessions between all the tests. For example, and supposing that the tests in the orchestration are using a browser provided by the EUS, the browser is shared between all the tests. In terms of the W3C WebDriver protocol [18], this simply implies to create a browser session at the beginning (identified with a unique identifier, *sessionId*), and this identifier is shared among all requests in different tests. Only in the last tests (those that end at the leafs of the graph) this session will be closed.

The implementation of the text fixture to be implemented in the test side will be basically some calls to the EOE, pausing the test at the beginning, and handling the input and output data. We provide a reference implementation using the

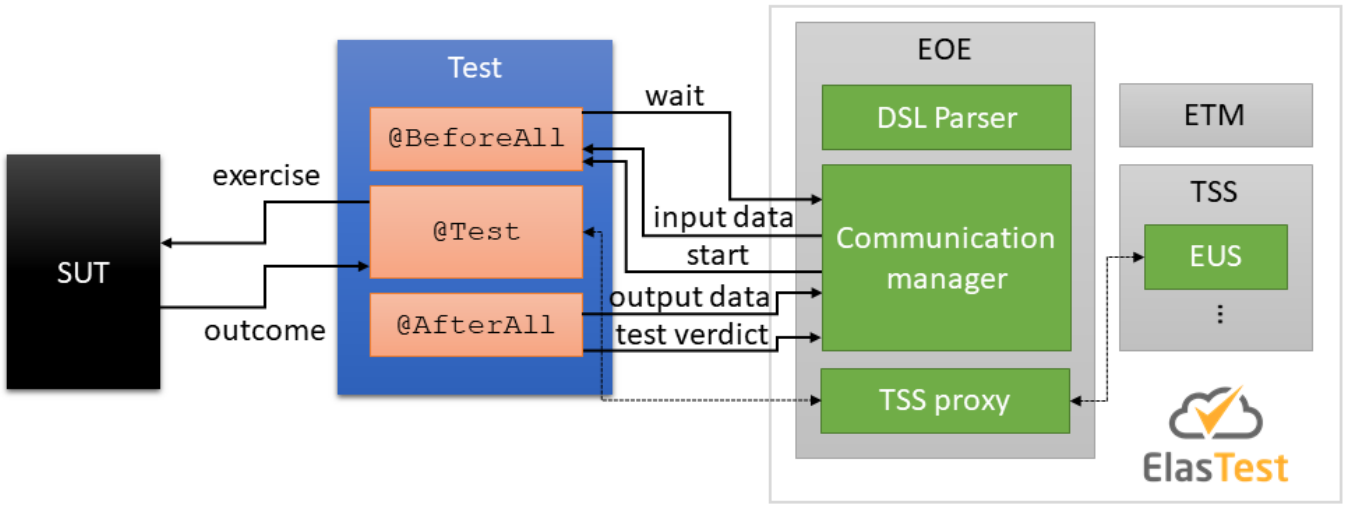

Fig. 4. EOE schema

extension model of JUnit 5 [19]. This way, "composable tests" based on JUnit 5 are as shown in the following listing. Notice that the input data can declare some default value in order to be executed as single instances (i.e., outside the orchestration workflow). These data are later overridden by EOE in the actual orchestration execution.

```
@ExtendsWith(ElasTestExtension.class)
class TJob1Test {

    @InputData
    String in1 = "default-value1";

    @InputData
    int in2 = 20;

    @InputData
    boolean in3 = false;

    @OutputData
    String out1

    @OutputData
    int out2

    @Test
    void myTest() {
        // my test logic
    }

}
```

V. Case Study

In order to carry out an initial experimental validation of the presented approach, we have carried out a case study using a real application as target. Concretely, we use an application called *Full Teaching* application, which is educational web platform based on OpenVidu[23], an open source videoconferencing framework based on WebRTC.

Concerning test, *Full Teaching* is assessed using a complete test suite implemented in JUnit 4 with different types of tests, including unit, integration, and end-to-end. At the time of this writing, the total number of tests in *Full Teaching* is 87. This large test suite is good news for the *Full Teaching* team in terms of coverage and level of confidence to avoid regressions in the codebase. On the other side, it has a relevant side-effect which impacts directly to the agility of the development process. Due to the fact all tests are executed in the Jenkins server supporting the CI process, developers need to wait until one patch is merged in the codebase.

To avoid this problem, our orchestration library has been used. As shown in the right side of Fig. 5, a Jenkins pipeline implementing an orchestration has been created. In this orchestration, a group of tests has been selected. A *smoke test* is going to be the first one. A smoke test case is the first to be run by testers before accepting a build for further testing. Failure of a smoke test case will mean that the software build is refused. The name of smoke testing derives electrical system testing, whereby the first test was to switch on and see if it smoked. This type of tests is done for accepting a build for further testing. A failure of this test will mean that the software build is refused, due to the fact that the orchestration has been configured using the *EXIT_ON_FAIL* option. After that, a group of relevant functional tests has been selected. These tests, executed as Jenkins jobs, is executed in parallel using the method *runJobsInParallel* of the orchestrator library. To rest of the initial tests of the *Full Teaching* test suite is executed using another job configured using a nightly job.

As a result, the orchestrated job shows a relevant reduction of time to be executed compared to the complete test suite.

[23]http://openvidu.io/

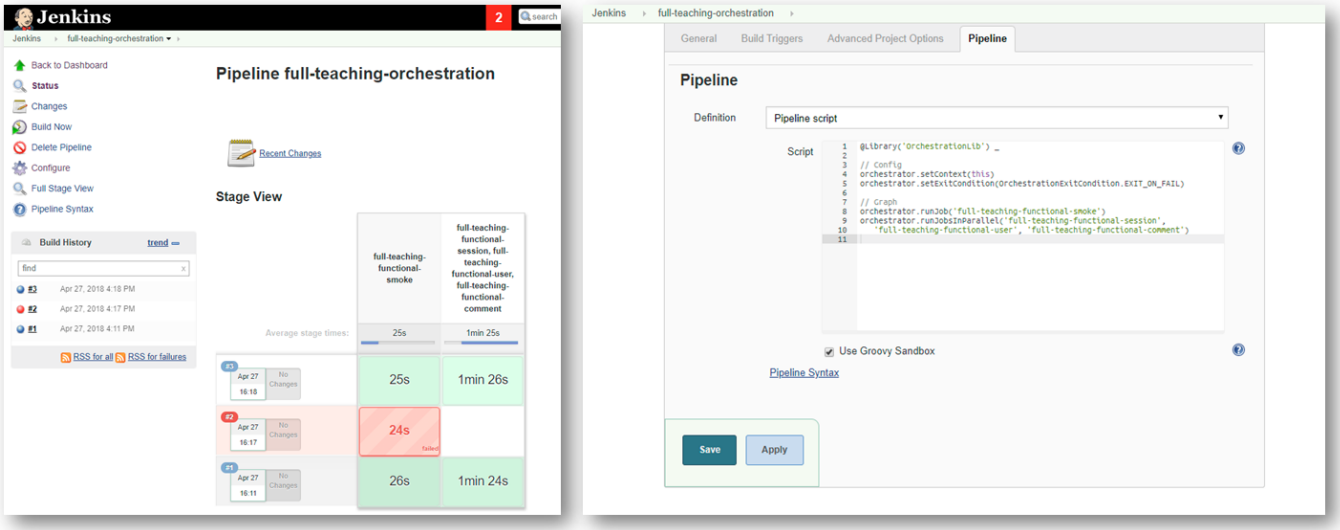

Fig. 5. Using orchestration Jenkins library in Full Teaching application

The proper selection of the smoke test together with critical functional test cases allows to the *Full Teaching* team to have a good level of confidence to merge patches in the development branch in a short amount of time. As can be seen in the left side of Fig. 5, all the orchestrated test takes less than 2 minutes to be completed. In addition, if the smoke test fails at the beginning, no further tests are executed and the job is declared as failed.

VI. Discussion

In order to analyze the contributions of this work, this section presents a SWOT (Strengths, Weaknesses, Opportunities, Threats) analysis of the proposal.

- Strengths:
  - Out concept of orchestration is aligned with current trends in software testing research, at least in the parallelization domain.
  - To implement verdict-driven orchestration, existing test codebases can be reused for selecting and parallelizing tests.
- Weaknesses:
  - To implement data-driven orchestration, tests needs to be implemented specifically. In other words, we cannot reuse existing codebases since tests need to be composable according to some guidelines.
  - Tests are managed individually. This is not completely aligned with the notion of *job* handled in the most of CI servers (e.g., Jenkins). In a job, typically a group of test are executed.
- Opportunities:
  - Our view of test orchestration is novel in the state of the art, and we aim to create a complete theory around this concept.
- Threats:
  - The concept of orchestration still need to prove its value for practitioners. At the moment of this writing there is a lack of validation of this approach in real scenarios.

VII. Conclusions and Future Work

This paper proposes a novel way to select, order, and execute a group of tests from a test suite in parallel. We call this approach test orchestration, and we distinguish between two types of orchestration techniques. The first approach is called *verdict-driven* orchestration, and it allows to create test pipelines by modeling test as black-boxes, meaning that we only known its final verdict (i.e., passed or failed) after the execution. This boolean value can be used to create conditional paths within the orchestration graph. The second approach presented in this paper is called *data-driven*. It is more complex due to the fact that tests within the graph need to be composable, meaning that the test data (input) and test outcomes (output) needs to be imported and exported by the tests. The inconvenient of this approach is that new tests following these guidelines need to be created. On the other side, we can create richer test suites using the “divide and conquer” principle applied to testing.

Both approaches are being implemented in the ElasTest project. ElasTest provides an integrated solution for end-to-end test automation along the development life cycle, including test case management, deployment, instrumentation, and monitoring for different kind of applications, including web and mobile. Internally, ElasTest has been implemented following a microservices architecture based on Docker containers. The ElasTest component in charge of implementing the orchestration approaches is called ElasTest Orchestration Engine (EOE). This component is able to parse an orchestration

workflow based on the DSL Jenkins Pipeline, sequencing, and executing in parallel tests according to the DSL (provided by testers). In order to ease the development of composable test as required in the data-driven approach, the ElasTest project provides a reference implementation as a JUnit 5 extension.

This work is the first step in our vision to create a novel testing theory for sequencing, ordering, and parallelization applied to software testing. This is an ambitious goal, and so, there is still a long path ahead. The next step is to validate properly the initial implementation of EOE with real use cases. Moreover, we plan to investigate additional techniques (new or existing) to include automated assertions (i.e., the oracle problem [20]) applied to the output data in the data-driven orchestration approach.

REFERENCES

[1] M. V. Zelkowitz, A. C. Shaw, and J. D. Gannon, *Principles of software engineering and design*. Prentice Hall Professional Technical Reference, 1979.

[2] D. Hamlet and R. Taylor, "Partition testing does not inspire confidence (program testing)," *IEEE Transactions on Software Engineering*, vol. 16, no. 12, pp. 1402–1411, 1990.

[3] T. Chen and M. Lau, "On the divide-and-conquer approach towards test suite reduction," *Information Sciences*, vol. 152, pp. 89 – 119, 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020025503000604

[4] F. Gortázar, M. Gallego, B. García, G. A. Carella, M. Pauls, and I.-D. Gheorghe-Pop, "Elastestan open source project for testing distributed applications with failure injection," in *Network Function Virtualization and Software Defined Networks (NFV-SDN), 2017 IEEE Conference on*. IEEE, 2017, pp. 1–2.

[5] Y. Falcone, J.-C. Fernandez, L. Mounier, and J.-L. Richier, "A compositional testing framework driven by partial specifications," in *Testing of Software and Communicating Systems*. Springer, 2007, pp. 107–122.

[6] P. Daca, T. A. Henzinger, W. Krenn, and D. Nickovic, "Compositional specifications for ioco testing," in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 373–382.

[7] R. Kuhn, Y. Lei, and R. Kacker, "Practical combinatorial testing: Beyond pairwise," *It Professional*, vol. 10, no. 3, 2008.

[8] A. Orso and G. Rothermel, "Software testing: a research travelogue (2000–2014)," in *Proceedings of the on Future of Software Engineering*. ACM, 2014, pp. 117–132.

[9] G. Demiroz and C. Yilmaz, "Using simulated annealing for computing cost-aware covering arrays," *Applied Soft Computing*, vol. 49, pp. 1129 – 1144, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1568494616304161

[10] W. Tsai and G. Qi, *Combinatorial Testing in Cloud Computing*, ser. Springer Briefs in Computer Science. Springer, 2017. [Online]. Available: https://doi.org/10.1007/978-981-10-4481-6

[11] J. Candido, L. Melo, and M. dAmorim, "Test suite parallelization in open-source projects: a study on its usage and impact," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 838–848.

[12] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 385–396.

[13] A. Gambi, S. Kappler, J. Lampel, and A. Zeller, "Cut: Automatic unit testing in the cloud," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 364–367.

[14] A. Gambi, J. Bell, and A. Zeller, "Practical test dependency detection," in *Proceedings of the 2018 IEEE Conference on Software Testing, Validation and Verification*, ser. ICST 2018, 2018, acceptance rate: 25Available: http://jonbell.net/publications/pradet

[15] D. Ghosh, *DSLs in action*. Manning Publications Co., 2010.

[16] H. Do, G. Rothermel, and A. Kinneer, "Prioritizing junit test cases: An empirical assessment and cost-benefits analysis," *Empirical Software Engineering*, vol. 11, no. 1, pp. 33–70, 2006.

[17] G. A. Carella and T. Magedanz, "Open baton: a framework for virtual network function management and orchestration for emerging software-based 5g networks," *Newsletter*, vol. 2016, 2015.

[18] S. Stewart and D. Burns, "Webdriver," *Working draft, W3C*, 2017.

[19] B. García, *Mastering Software Testing with JUnit 5*. Packt Publishing, 2017.

[20] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2015.