Architecture for the Automation of Live Testing of Cloud Systems

Oussama Jebbar
Gina Cody School of Engineering and Computer Science
Concordia University
Montreal, Canada
ojebbar@encs.concordia.ca

Ferhat Khendek
Gina Cody School of Engineering and Computer Science
Concordia University
Montreal, Canada
ferhat.khendek@concordia.ca

Maria Toeroe
Ericsson Canada Inc.
Montreal, Canada
maria.toeroe@ericsson.com

***Abstract*— Live testing is performed in the production environment. In such environment, test activities have to be orchestrated properly to avoid interferences with normal usage traffic. Conducting live testing activities manually is error prone because of the size and the complexity of the system as well as the required complex orchestration of different tasks. Furthermore, it would be impossible to react to failures and contain them in due time without automation. Live testing requires a high level of automation. This automation comes with several challenges especially in contexts such as cloud and zero touch networks because of the diversity of the software composing them. In this paper we discuss the challenges of automating live testing for cloud systems. We propose an architecture that relies on a modeling framework to decouple the specification of testing activities from the platforms needed to conduct them. We propose a solution for conducting testing activities on a live system according to such a specification.**

Keywords—live testing, cloud, UML Testing Profile, test architecture, automation

I. INTRODUCTION

Live testing is the activity of testing a system in the production environment without disturbing its usage. Many activities such as fault prediction [21], regression testing after a live update, composing web services [11], etc., are expected to rely on live testing in the future. Live testing can be either deployment time testing or service time testing [4, 3]. Deployment time testing is performed when a system is deployed but it is not yet serving users. Service time testing is performed while the system is serving its users.

Live testing requires a high level of automation. The test preparation phase does not only consist of deploying the test configuration, but also on setting up test isolation countermeasures to reduce potential disturbances. In addition, executing a crashing error revealing test case may lead to failures that need to be contained and recovered from in a timely manner. Testing related activities such as test planning, test execution and orchestration, test completion, etc., require automation for live testing to be conducted successfully.

Automating live testing for systems such as cloud systems has several challenges. The software involved in building such systems, as well as the test cases used to validate them often come from different sources; therefore, one cannot expect to interact-with/test all parts of the system using the same runtime environment (the same technology), or the same methodology (passive, active, metamorphic, etc.). Moreover, the diversity of the sources of software and test cases implies that they often use different configuration management platforms for test preparation and/or software reconfiguration. The dynamicity of cloud systems is yet another challenge for the automation of live testing of cloud systems as the frequently changing state of the system may jeopardize the applicability of test cases at certain points of time due to unmet preconditions for instance. Online testing methods [20] are well suited to deal with such challenges; however, they remain unsupported by the commonly used test runtime environments. In addition, existing test runtime environments are concerned only by test execution without considering test preparation and completion. Furthermore, they usually do not support multiple test case description languages. Thus, there is a need for a solution to tackle test automation of cloud systems.

Testing involves many activities such as planning, preparation, execution, and completion [19]. Reducing human intervention in these activities is a step forward towards automating live testing for cloud systems. In this paper we propose an architecture for such automation. This architecture is composed of two main building blocks, the Test Planner and the Test Execution Framework, each of which is responsible for a subset of the testing activities. To deal with the challenges related to the diversity of platforms encountered in cloud systems, we use UML Testing Profile (UTP) [16] to provide a platform independent representation of all the artifacts involved in the information flow of our proposed architecture. We map relevant elements in the artifacts involved in our architecture to UTP concepts. We also propose an execution semantics that the Test Execution Framework will associate with each model element that represents an entity involved in tests execution and orchestration. The execution semantics is not only useful for automatic orchestration of testing activities, but also for tracking the progress of these activities and reacting to them appropriately.

978-1-7281-8913-0/20/$31.00 ©2020 IEEE
DOI 10.1109/QRS51102.2020.00030

The rest of this paper is organized as follows. In Section II we review the related work and go through the challenges of the automation of live testing. We provide an overall picture of our proposed solution and its components in Section III. Section IV describes the first building block of our proposed solution, i.e. the Test Planner, and the method it uses for test suite generation. In Section V we map the concepts in our solution to the UTP concepts. In Section VI we show how this mapping can help automating testing activities in production by associating execution semantics to UTP concepts. We conclude in Section VII.

II. Related work and Challenges

Several architectures have been proposed for tests orchestration. The authors in [9] highlight three tasks that such an architecture should be able to perform: 1) observation, i.e. ability to collect information; 2) stimulation, i.e. ability to stimulate the system; and 3) reaction, i.e. an event, such as a detected error, should trigger a reaction at the level of the system under test (SUT) as much as it does at the level of the test system itself. In other words adaptation at the level of the SUT should lead to adaptation at the level of the test system too. The architectures mentioned in the literature can be classified into passive tests vs active tests orchestration architectures. [9 and 11] describe architectures that can be used to manage passive tests (monitoring) in production. [1, 3, 5, 6, 7, 10, and 12] propose solutions that can be used for active tests orchestrations. Another classification of these architectures can be established based on what triggers the testing activities; from this perspective, one can distinguish between interactive architectures and event driven architectures. Interactive architectures are the ones in which testing activities are triggered by human intervention (e.g. system administrator). Such architectures include [6, 7, and 13] as they launch testing activities when an administrator submits a request through a GUI or a CLI. Event driven architectures rely on events, such as the expiration of a timer, to trigger testing activities. The most commonly considered event is a system reconfiguration as it requires regression testing to evaluate the new state of the system. Reconfigurations can be simple adaptations such as a change in the binding of web services (e.g. the work in [9]); or, they can be additions or removals of one or more components (e.g. [5, 10]). Other types of events such as the expiration of timers to establish periodic checks [1, 3, 12], when the component is being looked up or called [1, 3], when an error is detected and the fault needs to be localized [2], etc., are also considered in the literature in event driven architectures. These architectures and solutions are used to orchestrate various types of tests. Yardstick [6] for instance, is used for pre-deployment testing of infrastructures' performance, capacity, availability, and the infrastructure's ability to properly run lifecycle operations on Virtual Network Functions (VNFs) and Network Services. Fortio operator [7] is yet another tool proposed by the Kubernetes community to run load tests on microservices in a Kubernetes managed environment. NetFlix Chaos Monkey [12] is one of the commonly discussed projects when it comes to live testing. It enables performing resiliency testing through fault injection in the production environment. Gremlin [14] is another tool, which is developed by IBM, for resiliency testing. Although it was not evaluated for live testing, but it is claimed to be easily portable to a production environment. Unlike Chaos Monkey, Gremlin injects faults at network level and not code level thus allowing for a better applicability across different technologies.

The approaches discussed so far indeed have the potential to be adapted to safe use in a production environment (as not all of them deal with test interferences); however, they remain limited to specific test types (load, performance, resiliency, etc.), and/or to specific test items (infrastructure, specific software, etc.). As a result, a test engineer who uses these approaches will have to make their own test scripts to schedule these tests and orchestrate them; which can be time and effort consuming as it has to be done every time the system needs to be tested. It can also be error prone due to the complexity and size of the production systems. Furthermore, of the three tasks mentioned in [9], these architectures are capable of stimulating and observing the SUT; however, their reaction capabilities are limited to reactions at the level of the SUT only. [8 and 10] are some of the few works that address how the test system should be maintained in reaction to an adaptation or a change in the SUT. The approach proposed in [8] reacts at the level of the test system by changing a label that it assigns to test cases which can be either ACTIVE or INACTIVE. A test case label may switch if the test case, for instance, was applicable under a previous service bindings, but when the binding has changed it became inapplicable. At every test sessions, the architecture is only allowed to select from test cases that have the ACTIVE label. The work in [10] relies on a different approach with the aim of decoupling the test case specification from the test case implementation. As a result, at every test session all the test cases are eligible to be chosen; however, in response to a system adaptation or reconfiguration, the architecture changes how that test case is implemented by associating it with a different set of test tasks (which are concrete implementations of test cases).

From this review of the related work we can identify a number of challenges that need to be addressed to automate live testing. They can be summarized as follows:

- **Challenge#1**: Test cases come from different sources (software vendor, test teams, third party). They may be written in different languages and require different runtime environments for execution. On the other hand, converting all available test cases to one programming or modeling language is unpractical (at the moment as it has to be done manually), and sometimes even unfeasible (test cases whose logic is inaccessible for the system owner).
- **Challenge#2**: Test cases coming from different vendors typically imply also that the test configurations for these test cases are deployed differently. Therefore setting up the test configurations may require different environments or tools depending on the test case.
- **Challenge#3**: Test cases may be of different nature. Passive tests such as monitoring, active tests, or metamorphic tests; they all differ in the way their configurations are deployed, how they

should be executed, and how/when their verdicts are generated/fetched.

- **Challenge#4**: Existing tools of test case execution such as ETSI TTCN [17] Test Architecture, assume that only the test cases are executed in the SUT, however, one may need to use an online testing method to test some properties of the system. Online testing [20] is a viable solution to deal with non-determinism and the dynamicity of the runtime state of modern systems such as clouds. Therefore, a proper solution for automation of live tests needs to support them.

Today these challenges are not properly dealt with in practice as we have shown in this section. In fact, system maintainers who encounter them end up either orchestrating the tests manually, or creating ad hoc scripts that will solve the problem for a few test sessions at best. These methods of test orchestration are unpractical because they are not reusable, error prone as these scripts are often made manually, and rarely deal with all the various failures that may take place during the testing activities.

III. An Architecture for Automation of Live Testing

In this paper we propose an architecture for the automation of live testing of cloud systems. We aim at tackling the challenges we identified in the previous section.

The first step for dealing with these challenges is to decouple the representation of the system and the artifacts involved in the testing activities from any platform. The second step is to specify how this representation is processed and used to achieve the goals, i.e. automate testing activities in production.

Testing activities such as planning, preparation, execution, completion, etc., need to be automated for live testing to be properly conducted. Unlike test planning, other activities such as preparation and execution depend heavily on the platform of the SUT. In other words, due to platform dependencies a good solution for test execution for microservice based systems for instance may not be good enough for ETSI Network Function Virtualization (NFV) [22] based systems. However, a good solution for test planning is reusable regardless of the platform of the SUT provided the SUT can be modeled at the right level of abstraction. Therefore, when designing a solution to automate live tests one has to take into consideration to what extent each activity may depend on the target platform on which the solution would be applied to be considered a good solution to automate. Taking this into consideration, we grouped activities that heavily depend on the target platform into one of the building blocks of our solution; and grouped the activities that may be reused across several contexts into another building block. Therefore, we propose the architecture shown in Fig. 1 to automate testing activities in production. The architecture is composed of the Test Planner (reusable building block) and the Test Execution Framework (target platform dependent block). The Test Planner is responsible of generating the test package. The test package is generated as a response to an event taking into consideration the test cases in the test repository and the current state of the system (system information). The test package is then fed to the Test Execution Framework which executes it on a live system while maintaining the disruption level within a tolerable range.

A. System information

This artifact represents the knowledge about the SUT that is required to properly conduct testing activities. It includes system configuration, runtime state, available software packages and their properties, and data collected or used by other management frameworks (availability management, scalability management, virtualization management, etc.).

B. Test Repository

The test repository stores test cases and test design methods along with the test goals they achieve. Test goals may be of various granularity ranging from the exercise of a path in a software (same granularity as test requirement in UTP); to system wide acceptance testing (same granularity as test objective in UTP). Other test case related information such as history of execution times, fault exposure rates, etc., are also stored in the test repository. Similarly, the test repository has information about test design techniques such as their required inputs and whether they are online or offline test design techniques. The test cases, test design techniques (both online and offline), and the test goals in the repository come from developers, software vendors, or the system administrator. The extra information is collected after each test session as the test case execution environment allows it. This extra information is then used to update the test repository accordingly.

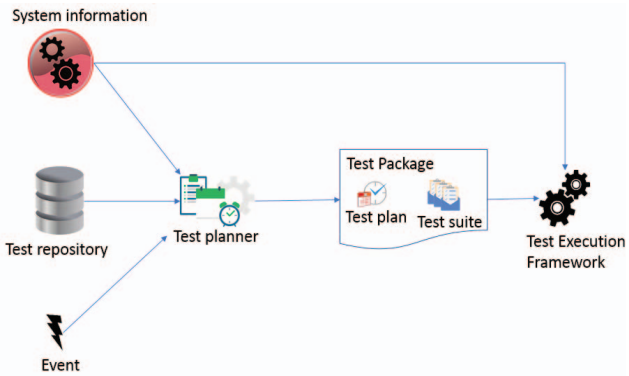
C. Event

An event is what triggers a test session. Events are configured at the level of an instance of this architecture, or created by the system maintainer or by a third party software to launch an on demand test session. Events created by third party software are of several types including events triggered by other management frameworks (for some specific types of regression testing, or to assist in root cause analysis for instance); as well as events created by some system components to ensure they are using services from the right components as stated in [3].

D. Test package

A test package consists of a test suite and a test plan. The test suite includes test suite items which are test cases and/or test

Fig. 1. Overall abstract architecture



design techniques that were selected from the test repository to respond to the received event.

The test plan provides the road map of the test execution and specifies a set of partially ordered Test Suite Item runs (TSIs), each TSI is an application of one or more test suite items under a given test configuration. This approach allows for grouping according to different criteria. For example, grouping together TSIs if setting up a test configuration is costly and/or disruptive, and there are more than one test suite items that apply to it. In this case one needs to setup this configuration only once during a test session and run all the test suite items that apply to it before tearing it down. In other cases TSIs applicable to the same test configuration may be grouped separately based on the criticality of the services they may impact.

Information on how the tests should be executed such as test configurations, test preparations e.g. isolation countermeasures, contingency plans to fix/contain crashing errors detected during the tests, etc., are also included in the test plan.

E. Test Planner

The Test Planner is responsible of generating the test package in response to a received event. Taking into consideration the received event, the Test Planner starts by selecting the test goals which respond/correspond to that event. After test goals selection, the Test Planner proceeds with the selection of test cases or test design techniques that can achieve the selected test goals. The selected test suite items will compose the test suite in the test package. The Test Planner relies on the information in the test repository, especially the mapping between test cases/test design techniques to the test goals they achieve, to select these tests cases and test design techniques. Later in this paper we describe our proposal for how this test suite is made based on the received event.

To complete the test package, the Test Planner generates a test plan. This test plan is generated taking into consideration the extra information available about each selected test case as well as the knowledge available about the system (system information). This information helps to decide which isolation method and contingency plan to use for each test case or set of test cases. As a result, test plan generation is a complicated process that involves making some decisions that if made inappropriately may lead to unnecessary, or even intolerable, disruptions in the system. The test plan generation is out of scope of this paper.

The generated test package is then given to the Test Execution Framework to execute. After the execution, the Test Planner updates, when applicable, the test repository using the information collected about each executed test case. More details about test suite generation are provided in the Section IV.

F. Test Execution Framework (TEF)

The Test Execution Framework (TEF) takes a test package as input and executes it on a live system. Executing a test suite consists of running the test suite items according to the test plan. The test plan is composed of partially ordered TSIs (test suite item runs) each of which combines a test configuration with at least one test suite item. Such grouping allows flexibility in scheduling TSIs according to their test configurations to enable a less intrusive test execution.

In order to automate testing activities in production, TEF implements different execution semantics for each one of the aforementioned concepts as it will be described in Section VII.

IV. The Test Planner and Test suite generation

The Test Planner's responsibilities are mainly related to test planning and test design activities. This clearly reflects on the main artifacts it is responsible of generating, i.e. the test package which is composed of a test suite and a test plan. In this section we go through our proposed approach for the automation of test design activities. We aim to automate the test planning in our future work.

A. Types of events

The Test Planner generates the test package in response to an event. As a result, the test design activities are conducted according to the received event. We identified the following list of event types:

- Periodic event: some parts of the system need to be checked periodically. In fact, those parts even though unchanged, they may be impacted by a change in their environment. A periodic test helps ensuring that some subsystems are always functioning correctly in the production environment. This type of events, such as for healthcheck, is specified as a set of test goals and a period of how frequently they should be achieved.
- Change in the system: a reconfiguration is usually a reason to perform regression tests. Therefore, a change in the system is considered an event that can trigger a live test session. The Test Planner can be made aware of a change either: 1) by registering to the notifications of the configuration manager; or 2) by being invoked directly by the configuration manager after a given reconfiguration.
- New test goal: addition of a new test goal to the repository should trigger a test session to achieve the new test goal. In fact, adding a new test goal may be accompanied by addition of new test case(s) the execution of which may reveal errors that were not detected previously. It is also possible that the new test goal is achieved using a combination of test cases that were not used together before, which may reveal new errors as well.
- Test request: to avoid limiting the applicability of our architecture, we propose the concept of test request to include the cases not covered by the previous types of events. Therefore, a test request may be submitted by an administrator or a third-party software. This test request can be of one of the following types:
    - Used as an aggregation of periodic events: the administrator may want to run some system checks together for a period of a time. In this case, a test request can be used to aggregate the periodic events associated with these system checks.
    - A set of test goals to achieve: at any point an administrator or a third party software may

initiate a test session by submitting this type of test requests. It is composed of a set of test goals to be achieved, which are selected from the repository. This type of test requests is mainly practical for test goals that are achieved using test design techniques and not test cases as there is another type of requests that can be used to invoke specific test cases.

- A set of fault-revealing test goals: when a fault revealing test goal is achieved, errors are detected if the fault to be revealed is present. The main purpose of a test request consisting of such test goals is to localize the faults behind these errors. Hence this type of test requests is mainly used to trigger system diagnostics. An administrator or a third party software may give the Test Planner a set of fault-revealing test goals and the Test Planner will have to generate a test package that can help localize the faults behind the errors that manifest when one or more of these test goals are achieved.
- A set of test cases: a set of test cases to be executed may be requested by the administrator or a third-party software.

B. *Test Suite Generation*

The test suite is one of the components of the test package that the Test Planner generates to respond to an event. Therefore, taking into consideration the received event, the Test Planner follows different strategies to select the test suite items that will compose the test suite. The method we propose for this purpose follows the following rules:

- If the received event is a periodic event happening for the first time, the Test Planner selects test cases and test design techniques that are able to achieve the test goals specified by this event. This is a straightforward process as the mapping between the test cases/test design techniques and the test goals they achieve is already stored in the test repository.
- If the received event is a periodic event and no reconfiguration or test goal addition happened since the last instance of this event, the Test Planner should reuse the same test suite from the last time an instance of this event occurred. We proceed this way for the reason that since the system has not undergone any change, this means that the same test cases/test design techniques are applicable and will be chosen for this instance of the event as for its previous instance. This is like a heuristic that we use to save some time in this activity as querying it may be time consuming.
- If the received event is a periodic event, and there was a reconfiguration or test goals were added since the last instance of this event; this event is treated as if it is happening for the first time. In fact, a change in the system or in the content of the test repository may require a different set of test suite items to achieve the same test goals as previous instances (before the change). Therefore, a reselection of test cases and test design techniques is deemed necessary in this case.
- If the received event is an addition of new test goals, then the Test Planner should select the test suite items that achieve the newly added test goals. The addition of a test goal leads to existing or new test cases/test design techniques be mapped to it. As a result, one needs to check if achieving this new test goal may reveal any errors that were not detected previously. Using the mapping information from the test repository, one can deduce the set of test cases/test design techniques needed to achieve this new test goal.
- If the received event is a reconfiguration, the Test Planner will use an approach for regression test case selection/generation to select the test cases. Several approaches may be used to deal with this kind of event. For instance, a test design technique that is stored in the repository may simply perform an impact analysis and come up with a set of regression test goals from the repository to be achieved. In this case the Test Planner will select test cases and test design techniques that achieve the selected test goals to compose the test suite. Another approach to deal with this event is by having a default regression test case selection/generation technique that will be invoked whenever a reconfiguration takes place.
- If the event is a test request:
  - If the test request is an aggregation of periodic events it is handled the same way as a periodic event.
  - If the test request consists of a set of test goals to be checked, the Test Planner selects the set of test cases and test design techniques to be used to achieve the requested test goals. This is done based on the information from the repository that maps each test goal to the test cases/test design techniques that achieve it.
  - If the test request consists of a set of fault-revealing test goals, the Test Planner first identifies other test goals that are related to the requested test goals and which (if achieved) can help localize the faults. After identifying those goals, the Test Planner selects test cases and test design techniques that are able to achieve the selected test goals. The identification of the related test goals can be done by invoking a default fault localization technique.
  - If the test request consists of a set of requested test cases, the test suite will be composed of the requested test cases.

V. MODELING FRAMEWORK

In this section we describe how we address the first step of dealing with the challenges outlined in Section II, i.e. decoupling

the representation of the system and the artifacts involved in testing activities from the target platform.

TABLE I. MAPPING THE ARTIFACTS IN THE ABSTRACT ARCHITECTURE TO UTP CONCEPTS

Abstract Architecture concepts	UTP concepts
Test repository	Set of pairs (TestContext, aggregate of test logs)
Test case as in the repository	TestProcedure
Test design technique as in the respoitory	TestDesignTechnique
Test suite item in the test plan	ProcedureInvocation (e.g. TestProcedure, or TestDesignTechnique invocation) in the main phase of a TestCase
TSI	TestCase
Test package	TestContext
Test suite	TestSet
Test plan	TestExecutionSchedule
Periodic event	Triplet (TestLevel, TestType, TestDesignInput)+ time data
Test request: test goals to be achieved	Subset of TestRequirements or TestObjectives that are in the test repository
Test request: fault revealing test goals	A set of of TestRequirements or TestObjectives (not necessarily in the test repository)
Test related metadata	TestLogs
Test related metadata specification	TestLogStructure
Test results	Verdicts (Pass, Fail, Inconclusive)
Failure detection during test execution	Verdicts (error, customized verdicts)
Test preparation including the setting up of isolation countermeasure	TestCase setup procedure invocation
Test completion including the cleanup of isolation countermeasure	TestCase teardown procedure invocation
Test goal	TestRequirement or TestObjective

Modeling techniques, especially UML, are commonly used for platform independent modeling of systems. Therefore, we propose to use UML to provide a platform independent representation of our targeted systems and artifacts. Note that this representation is not only needed to model the artifacts and to have an exchange format for them (between the building blocks of our architecture), but also to track the progress of the testing activities and accordingly invoke appropriate behaviors. We chose UML Testing Profile (UTP) [16] to model all the artifacts in play in our architecture including the test plan. The choice of UTP was based on the following rationales:

- UTP covers a wide range of testing activities and it is aligned with industry testing standards [18, 19].
- UTP is a UML profile, and therefore some of the inherited concepts from UML allow for flexibility, extendibility, and interoperability with existing tools.
- The flexibility offered by UTP to model verdicts and arbitration specifications allows capturing of runtime errors. As a result, using UTP will not limit the capability of the architecture to detect failures and react to them.
- UTP allows the reuse of all concepts used in UML to model behaviors, including the use of UML's CombinedFragment. Therefore, it supports a wide range of patterns of scheduling such as sequential, parallel, alternative, etc., which allows us to deal with **Challenge#3**.

UTP offers plenty of opportunities to automate the orchestration of testing activities in production. However, some minor modifications may be needed to cover all bases. These minor modifications can be summarized as follows:

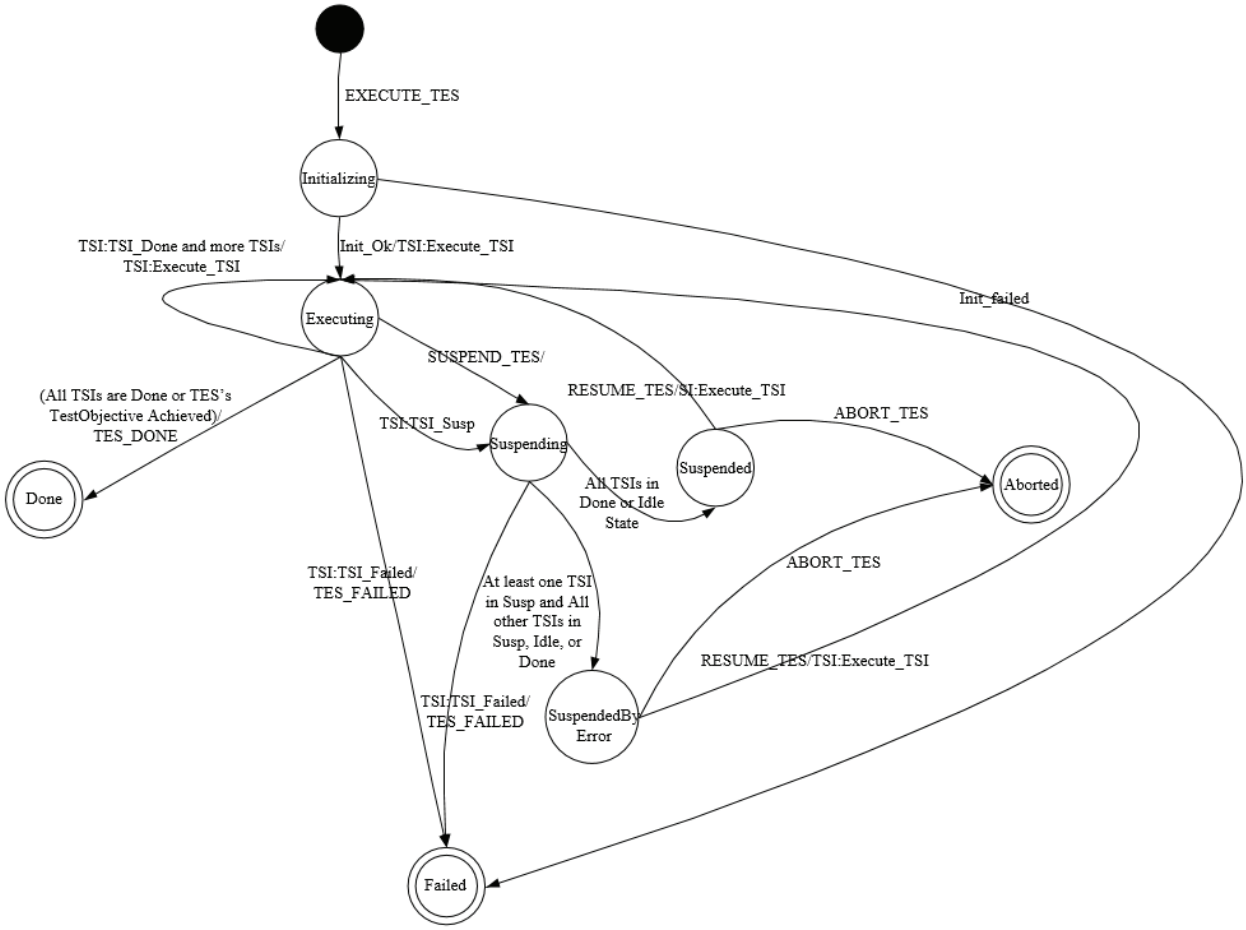
- UTP as-is does not allow associating an ArbitrationSpecification with a ProcedureInvocation element. This constraint needs to be relaxed if we want to be able to arbitrate actions taken in the setup phase (for setting up isolation countermeasures), teardown actions, or the execution of test generation actions when invoking a test design technique.
- UTP offers concepts to model test logs. However, these test logs are associated only with test cases and test procedures. In online testing methods, one needs to log the test design activities too. Therefore, we may need two types of logs to be associated with a TestDesignTechnique element:
    - TestDesignTechniqueLog: to log test design activities. The structure of this log is specified by a TestDesignTechniqueLogStructure element associated with the TestDesignTechnique.
    - TestCaseLog: if the online testing method generates new test cases the TestCaseLog is also associated with the TestDesignTechnique element. The structure of the logs of the generated test cases is specified by the TestCaseLogStructure element associated with the TestDesignTechnique.

This extension will help us deal with the representation and modeling aspect of **Challenge#4**. The remaining aspect of this challenge related to the behavior is dealt with at the level of TEF and will be described later in this document.

The mapping between the concepts we propose, and the ones defined in UTP is shown in Table I. This mapping enables expressing test plans as TestExecutionSchedules that run UTP TestCases. UTP TestCases consist of one or more test cases provided by the vendor or the developer (along with a test configuration) enhanced with some isolation countermeasures that need to be set up before the execution of the test case (which is a UTP TestProcedure), and that need to be torn down at the end. UTP TestProcedures may be modeled using UML concepts. UTP also offers the possibility of specifying TestProcedures using other languages as OpaqueBehavior (a concept inherited from UML). Therefore, this mapping helps us properly deal with **Challenge#1**.

Test goals that are associated with test cases in the repository are modeled as UTP TestRequirement. Test goals that are associated with test design techniques are modeled as UTP

Fig. 2. Execution semantics of the TestExecutionSchedule



TestObjective. The main difference between the two is that a TestRequirement is a contribution of a test case towards achieving a TestObjective. However, a TestObjective is defined as the stopping criterion of testing activities. Both TestObjective and TestRequirement can be specified informally using natural language, or formally using a machine understandable language such as ETSI TPlan [15]. Expressing TestRequirements and TestObjectives using formal languages may open the door for further processing of these model elements and make them more suitable for other purposes of live testing such as diagnostics.

Test configurations in UTP include modeling the configuration of the test component as well as the configuration of the test item (system or component under test). Two patterns are proposed in UTP specification to model these configurations, the one we are recommending is modeling these configurations as constraints. Although UML has a language for constraints specifications, but similar to behaviors, it also allows the usage of other languages. Test configurations may be specified in various languages such as ansible playbook language, puppet DSL, chef DSL, etc.; as a result, one may use this feature of UML to specify test configurations as constraints expressed in languages that deployment management engines can process. Therefore, such use of UTP is useful for dealing with **Challenge#2**.

The mapping will also allow us to detect failures during execution as the verdict type provided by UTP allows it. To address this, we propose using the UTP provided verdict "error" as a concept to model failures of actions for which it is not clear whether the problem during the execution is caused by a problem in the test component or the test item. Moreover, UTP allows the creation of user customized verdicts. In our opinion, since the implementation of a test component may be part of an implementation of this architecture; the implementer can draw up a list of possible problems that can occur to the test component (test component's failure modes), create their customized verdicts, and then the implementation of this environment can decide which actions to take to recover the failed test component based on the customized verdict that was issued. Note that this approach can also be used with test components of some test environment about which the system maintainer has enough knowledge.

VI. TEST EXECUTION FRAMEWORK

In this section we describe our approach to address the second step of dealing with the challenges outlined in Section II, i.e. the behavior associated by the building blocks of the architecture to each element of the artifacts representation. Therefore, this section will mainly focus on the TEF, and how it processes the artifacts generated by the Test Planner according to the execution semantics we propose. This execution semantics is used for:

- The automatic orchestration and control of testing activities in production.
- Tracking the progress of testing activities by monitoring the state of each runtime object involved in the orchestration of testing activities. The TEF, through this tracking, is then able to orchestrate the testing activities and becomes aware of any mishaps that may take place during this orchestration.

The first test plan model element with which we associate an execution semantics is the TestExecutionSchedule which is also a runtime object that is used by the TEF to track and control a test session. It is a composed of a set of partially ordered TSIs. A TSI is modelled in our test plan as a UTP TestCase, its setup and teardown phases are composed of ProcedureInvocation elements (to setup/teardown the test configuration); and its main phase is composed of a set of invocations of TestProcedures and/or TestDesignTechniques. A TestExecutionSchedule is able to receive four administrative operations: EXECUTE, SUSPEND, RESUME, and ABORT (Fig. 2.).

EXECUTE is the only operation that can be invoked on a TestExecutionSchedule when it is first created. Upon the invocation of this operation the TestExecutionSchedule moves to the Initializing state, and the TEF performs all preparations necessary for the whole TestExecutionSchedule. After the preparations the TestExecutionSchedule moves to the Executing

Fig. 3. Execution semantics of ProcedureInvocation and TestProcedure Invocation

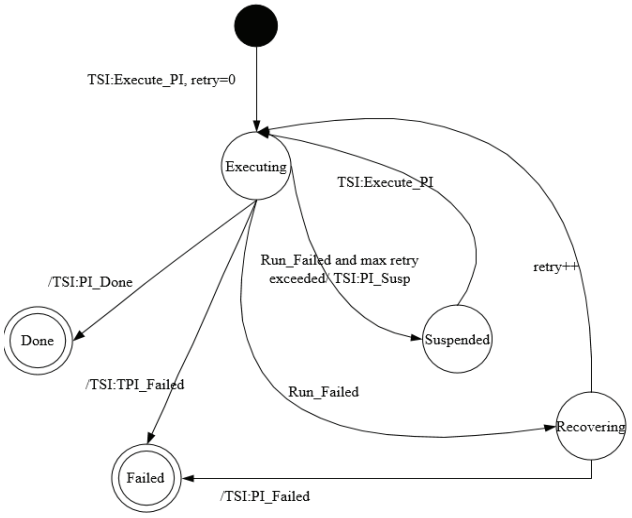
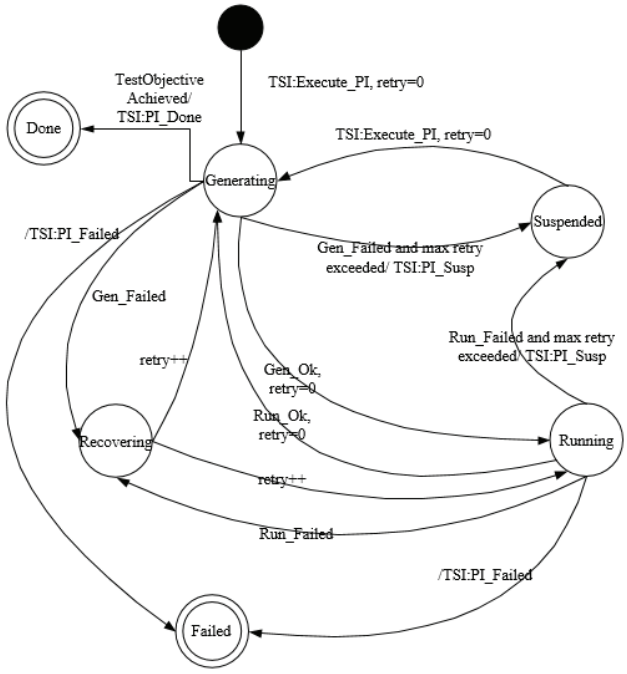


Fig. 4. Execution semantics of the TestDesignTechnique invocation



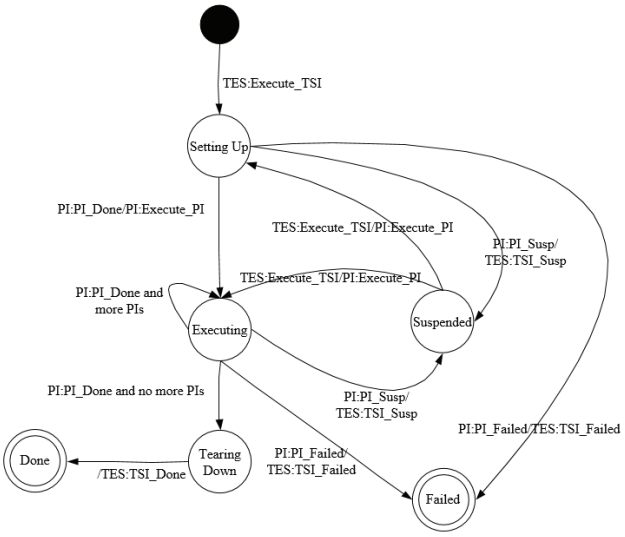
state, and the TEF starts invoking TestCases according to the specification of the test plan. TestCases can be specified:

- sequentially,
- using a CompoundProceduralElement which allows for TestCases to be executed in parallel, or
- as alternatives based on specified conditions (like switch blocks in programming).

Just like any UML specified behavior, a TestExecutionSchedule can use any combination of these facilities to model the schedule for the TestCases. The same also applies to the invocations within a TestCase in the TestExecutionSchedule (a.k.a ProcedureInvocations, such as TestProcedure and TestDesignTechnique invocations). Therefore, at any time one can have either a single TestCase running, or multiple TestCases running at the same time. In the TestExecutionSchedule the partial order is specified through a control flow, which is specified by control flow kind of links among the above constructs. The invocations to be made after the completion of a TestCase are decided based on the construct it belongs to and the target(s) of the control flow link(s) that have the completed TestCase as source.

While the TestExecutionSchedule is in the Executing state, the TEF keeps invoking TestCases using the Execute_TSI message. From the initial Idle state, the invoked TestCase goes to the Executing state (Fig. 5.) via a Setting Up state, and completes after a Tearing Down state. In all these states the TestCase invokes different procedures composing the TestCase using the Execute_PI message. According to the TestCase state these procedure invocations (Fig. 3. And Fig. 4.) can be setting

Fig. 5. Execution semantics of UTP TestCase



up/tearing down a test configuration and/or isolation countermeasures, or running a test case or a test design technique. In the later case, the invocation is for a TestDesignTechnique with the execution semantics shown in Fig. 4.; in the other cases the procedure invoked follows the execution semantics shown in Fig. 3., this includes the invocation of TestProcedures. When the execution of a procedure stops, its associated arbitration specification is invoked still in the Executing state. This leads to the creation of a verdict. If the verdict is None, PASS, FAIL, or INCONCLUSIVE, the invocation is deemed as successful, the invoked procedure goes to the Done state. If the verdict is a customized verdict, the TEF should be capable of taking recovery actions depending on the received customized verdict, because customized verdicts are produced only if the failure of an action was caused by the failure of a test component. The TEF then tries to recover from the failure and reinvoke the failed action. If this retrying exceeds a pre-specified number of times, the procedure goes to the Suspended State. Finally, if the produced verdict is ERROR, the procedure is deemed as failed, goes to the Failed state. In any case the procedure notifies the TestCase about the result, which then proceeds depending on the procedure's state. If the procedure's state is:

- Done: the TestCase proceeds to the next invocation(s);
- Suspended: the TestCase goes to the Suspended state, and therefore the TestExecutionSchedule also goes to the SuspendedByError state;
- Failed: the TestCase also fails and notifies the TestExecutionSchedule. As a result the whole test session is deemed as failed.

If the TestExecutionSchedule goes to the Suspending state, the TEF waits for all the currently running TestCases to either complete (Done state), in which case it moves into the Suspended state; or if any TestCase is suspended then the TestExecutionSchedule moves to the SuspendedByError state. Once the TestExecutionSchedule is in the SuspendedByError state, the administrator can either fix/repair the system and resume the test session, or abort the test session. If a TestCase that is in Executing state goes to Failed state while the TestExecutionSchedule is in the Suspending or the Executing state, the TestExecutionSchedule goes to the Failed State and the whole test session will be deemed as failed.

The SUSPEND administrative operation is used to suspend a test session. Upon the reception of this administrative operation, the TestExecutionSchedule goes to the Suspending state and waits for all currently running TestCases. As described from the Suspending state the TestExecutionSchedule may go to the Suspended state, to the SuspendedByError state or to the Failed state depending on the results of the currently running TestCases. If in the Suspended and the SuspendedByError states, the administrator can either decide to resume the test session later using the RESUME operation, or abort the test session using the ABORT operation. In the latter case it is left to the administrator to perform any required teardown or clean-up actions.

VII. Conclusion

Live testing has become a necessity as the production environments have become bigger and more complex and impossible or unfeasible to recreate it in the test environments. The automation of test activities is a must for live testing among others due to the complexity of the production environment and the need for short reaction times. In this paper, we highlighted the challenges of automating live testing, and showed the limitations of existing approaches in addressing them as they are either limited to a specific target platform (e.g. TTCN), specific method of testing (active vs passive), or specific test types (performance, resiliency, etc.).

We proposed an architecture to enable the automation of testing activities in the production environment. We also proposed the use of UTP as the specification language for testing activities planning. We associated an execution semantics with the UTP concepts that are relevant to the automated orchestration of test activities. As part of the proposed Test Planner in our architecture, we outlined the main principles for a test suite generation method. As future work we plan to complete the work on the Test Planner by developing a method for automating the test plan generation. We also aim to apply this architecture for live testing of microservice based architectures.

Acknowledgment

This work has been partially supported by Natural Sciences and Engineering Research Council of Canada (NSERC) and Ericsson.

References

[1] D. Brenner, C. Atkinson, R. Malaka, M. Merdes, B. Paech, D. Suliman, Reducing verification effort in component-based software engineering through built-in testing, Inf. Syst. Front. 9(2–3) (2007) 151–162

[2] Alberto Gonzalez Sanchez, Cost Optimizations in Runtime Testing and Diagnosis. Phd Thesis, Delft University of Technology, September 2011

[3] Dima Suliman, Barbara Paech, Lars Borner, Colin Atkinson, Daniel Brenner, Matthias Merdes, Rainer Malaka. The MORABIT Approach to Runtime Component Testing. Proceedings of the 30th Annual

International Computer Software and Applications Conference (COMPSAC'06).

[4] M. Merdes, R. Malaka, D. Sulimani, B. Paech, D. Brenner, C. Atkinson, Ubiquitous RATs: How Resource-Aware Run-Time Test can improve Ubiquitous Software Systems. Proceedings of the 6th International Workshop on Software Engineering and Middleware, SEM 2006, Portland (USA), pp. 55-62.

[5] Mariam Lahami, Moez Krichen, Mohamed Jmaiel, Safe and efficient runtime testing framework applied in dynamic and distributed systems, Science of Computer Programming, 2016.

[6] Yardstick. https://wiki.opnfv.org/display/yardstick/Yardstick. Last visited, March 28th, 2020.

[7] Fortio operator. https://github.com/verfio/fortio-operator. Last visited, March, 28th, 2020.

[8] E. M. Fredericks, B. H. C. Cheng, Automated Generation of Adaptive Test Plans for Self-Adaptive Systems, In the proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, 2015.

[9] P. H. Deussen, G. Din, I. Schieferdecker, A TTCN-3 Based Online Test and Validation Platform for Internet Services, In the proceedings of the 6th International Symposium on Autonomous Decentralized Systems, ISADS 2003, pp. 177-184.

[10] M. Greiler, H. Gross, A. Van Deursen, Evaluation of Online Testing for Services – A Case Study, in the proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems, PESOS 2003, pp. 36-42.

[11] M. Elqortobi, J. Bentahar, R. Dssouli, Framework for Dynamic Web Services Composition Guided by Live Testing, in the proc. Of Emerging Technologies for Developing Countries, AFRICATEK, 2017. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol 206. Springer, Cham.

[12] Netflix – Siamian Army, https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey. Last visited, March 28th, 2020.

[13] M. Ali, F. De Angelis, D. Fani, A. Bertolino, G. De Angelis, A. Polini, An Extensible Framework for Online Testing of Choreographed Services, in Computer, vol. 47, no. 2, pp. 23-29, Feb. 2014

[14] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, V. Sekar, Gremlin: Systematic Resilience Testing of Microservices, In the proceedings of IEEE 36th International Conference on Distributed Computing Systems, ICDCS 2016, pp. 57-66

[15] ETSI ES 202 553 V1.2.1. Methods for Testing and Specification (MTS); TPLan: A notation for expressing Test Purposes

[16] Object Management Group. UML Testing Profile 2 (UTP2) Version 2.1

[17] Testing and Test Control Notation Version 3. www.ttcn-3.org

[18] International Software Testing Qualifications Board. www.istqb.org. Last visited March, 28th, 2020.

[19] ISO/IEC/IEEE: "Software Testing - The International Software Testing Standard", ISO29119. www.softwaretestingstandard.org. Last visited March, 28th, 2020.

[20] T. Cao, P. Felix, R. Castanet, I. Berrada, Online Testing Framework for Web Services, In the proceedings of the 3rd International Conference on Software Testing, Verification and Validation, ICST 2010, pp. 363-372.

[21] O. Sammodi, A. Metzger, X. Franch, M. Oriol, J. Marco, K. Pohl, Usage-based Online Testing for Proactive Adaptation of Service-based Applications, In the proceedings of the 53th IEEE Annual Computer Software and Applications Conference, COMPSAC 2011, pp. 582-587.

[22] ETSI Network Function Virtualization. https://www.etsi.org/technologies/nfv. Last visited March, 28th, 2020.