Live Testing of Cloud Services

Oussama Jebbar

A Thesis

in the Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy in Software Engineering at

Concordia University

Montreal, Quebec, Canada

January 2023

CONCORDIA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Oussama Jebbar

Entitled: Live Testing of Cloud Services

and submitted in partial fulfillment of the requirements for the degree of

Doctor Of Philosophy Software Engineering

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. Anjan Bhowmick	Chair
Dr. Ferhat Khendek	Thesis Supervisor
Dr. Maria Toeroe	Thesis Supervisor
Dr.	Thesis Supervisor
Dr. Juregen Rilling	Examiner
Dr. Yann-Gael Gueheneuc	Examiner
Dr. Jamal Bentahar	Examiner
Dr.	Examiner
Dr. Nadjia Kara	External Examiner

Approved by

Dr. Leila Kosseim , Graduate Program Director

«Date» 17/01/2023

Dr. Mourad Debbabi , Dean

Abstract

Live Testing of Cloud Services

Oussama Jebbar, Ph.D.

Concordia University, 2023.

Service providers use the cloud due to the dynamic infrastructure it offers at a low cost. However, sharing the infrastructure with other service providers as well as relying on remote services that may be inaccessible from the development environment create major limitations for development time testing. Modern service providers have an increasing need to test their services in the production environment. Such testing helps increase the reliability of the test results and detect problems that could not be detected in the development environment such as the noisy neighbor problem. Furthermore, testing in production enables other software engineering activities such as fault prediction and fault localization and makes them more efficient.

Test interferences are a major problem for testing in production as they can have damaging effects ranging from unreliable and degraded performance to a malfunctioning or inaccessible system. The countermeasures that are taken to alleviate the risk of test interferences are called test isolation. Existing approaches for test isolation have limited applicability in the cloud context because the assumptions under which they operate are seldom satisfied in the cloud context. Moreover, when running tests in production, failures can happen and whether they are due to the testing activity or not the damage they cause cannot be ignored. To deal with such issues and manage to quickly get the system back to a healthy state in the case of a failure, human intervention should be reduced in the orchestration and execution of testing activities in production. Thus, the need for a solution that automates the orchestration of tests in production while taking into consideration the particularity of a cloud system such as the existence of multiple fault tolerance mechanisms.

In this thesis, we define live testing as testing a system in its production environment, while it is serving, without causing any intolerable disruption to its usage. We propose an architecture that can help cope with the major challenges of live testing, namely reducing human intervention and providing test isolation. Our proposed architecture is composed of two building blocks, the Test Planner and the Test Execution Framework. To make the solution we are proposing independent from the technologies used in a cloud system, we propose the use of UML Testing Profile (UTP) to model the artifacts involved in this architecture. To reduce human intervention in testing activities, we start by automating test execution and orchestration in production. To achieve this goal, we propose an execution semantics that we associate with UTP concepts that are relevant for test execution. Such an execution semantics represent the behavior that the Test Execution Framework exhibits while executing tests. We propose a test case selection method and test plan generation method to automate the activities that are performed by the Test Planner. To alleviate the risk of test interferences, we also propose a set of test methods that can be used for test isolation. As opposed to existing test isolation techniques, our test methods do not make any assumptions about the parts of the system for which test isolation can be provided, nor about the feature to be tested. These test methods are used in the design of test plans. In fact, the applicability of each test method varies according to several factors including the risk of test interferences that parts of the system present, the availability of resources, and the impact of the test method on the provisioning of the service. To be able to select the right test method for each situation, information about the risk of test interference and the cost of test isolation need to be provided. We propose a method, configured instance evaluation method, that automates the process of obtaining such information. Our method evaluates the software involved in the realization of the system in terms of the risk of test interference it presents, and the cost to provide test isolation for that software.

In this thesis, we also discuss the feasibility of our proposed methods and evaluate the provided solutions. We implemented a prototype for the test plan generation and showcased it in a case study. We also implemented a part of the configured instance evaluation method, and we show that it can help confirm the presence of a risk of test interference. We showcase one of our test methods on a case study using an application deployed in a Kubernetes managed cluster. We also provide proof of the soundness of our execution semantics. Furthermore, we evaluate, in terms of the resulting test plan's execution time, the algorithms involved in the test plan generation method. We show that for two of the activities in our solution our proposed algorithms provide optimal solutions; and, for one activity we identify in which situations our algorithm does not manage to give the optimal solution. Finally, we prove that our test case selection method reduces the test suite without compromising the configuration fault detection power.

Acknowledgments

I would like to thank my family for their everlasting and unconditional love, support, and encouragement. I would like to thank my brothers Yassine, Mohamed Khalil, and Younes for being there for me when I needed it most. I would also like to thank my parents for encouraging me and supporting me throughout this journey.

I would like to express my gratitude to my supervisors Dr. Ferhat Khendek and Dr. Maria Toeroe for their patience, support, and guidance they have given me throughout this thesis. Thank you very much for making me feel welcome in your research team three times and giving me a chance to grow both as an individual and a researcher.

I would like also to thank Dr. Abdeslam Ennouaary, for being a teacher, a friend, and a mentor. Thank you for sharpening the engineer within the human without compromising the human within the engineer.

I would like to thank my colleagues in the MAGIC team for the good memories.

I would like to thank Concordia University, Ericsson Canada, and NSERC for the funding and the facilities they put at my disposal to achieve this work.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1. Context and motivations

Cloud computing is a commonly adopted paradigm due to the flexibility of resource provisioning and the improved resource utilization it enables. Whether it is dedicated (private cloud) or shared (public cloud), a cloud is a complex system due to its scale and the configurations involved in its setup and management. This complexity raises the question of how the difference between the test environment (development, lab, staging) and the cloud as a production environment may impact software engineering activities such as software testing.

Software testing is a set of activities conducted to facilitate discovery and/or evaluation of properties of one or more software products [79]. In legacy systems, software was deployed in a relatively small scale and dedicated infrastructures. In such systems, the tester can make accurate assumptions about the production environment and the results of testing in the development environment can still hold in the production environment. In the cloud, however, testing in the development environment is not enough anymore. In [64] for instance, Google reports a 43 minutes outage of the compute engine service due to a bad configuration. Although the configuration passed all the tests pre-deployment, unexpected errors manifested once it was deployed in production. The incident in [68] presents a similar yet more elaborated issue encountered by Microsoft Azure as a configuration and code change errors not detected prior to the deployment led to an almost 3 hours outage of several services. These errors manifested because of the new code and configuration being exposed to a specific traffic pattern not known/considered pre-deployment. To reveal such scenarios, services hosted in clouds must be re-tested in their production environments as 1) the multiple configurations involved in a cloud system lead to differences between the configurations used in the test environment and the ones deployed in production [65]; 2) clouds may be subject to unexpected scenarios (requests, traffic patterns, different dependencies in different sites) that may not have been covered by testing activities.

Testing a system in production is defined as executing tests in the production environment without causing any disturbance to the system's usage. It is generally challenging, and it becomes more so as the tolerable disruption constraints become more stringent as in the case of carrier-grade services. Test interferences between test traffic and production traffic are among the main challenges of testing a system in production as they result in violations of one or more of the functional or non-functional requirements of the cloud service. Test interferences may be due to internal characteristics of the services being tested (statefulness, nature of interactions, etc.), or the resources shared among services hosted in the same environment. Test interferences can manifest at the level of the services being tested or at the level of other services that share resources with them. A concrete example of such cases can be found in [66] where Google describes how testing an experimental Google App Engine feature released for Java and Go applications caused instances of Java, Go, and Python applications running on Google App Engine to malfunction for almost 20 minutes. The countermeasures taken to alleviate the risk associated with test interferences are known as test isolation. Some test methods provide test isolation by using only components that can be safely tested in production, i.e. they incorporate specialized modules called Built-In Test modules (BITs) that provide test isolation [1, 3, 4]. Other test methods rely on other mechanisms such as cloning (used in canary releases [8] and gradual rollout [44] test methods), snapshot and restore, resource negotiation, or scheduling tests when test interferences are less likely to happen [2].

We define live testing as testing a system in its production environment without causing any intolerable disturbance to its usage. Live testing does not only facilitate tests' execution in production, but it also enables the carrying out of other testing activities in production such as test design and test planning. Live testing of services, especially the ones deployed in dynamic environments such as clouds, has one main challenge: test interferences. A solution that properly deals with this challenge must: 1) minimize human intervention in testing activities, especially tests execution and orchestration, in order to tackle the complexity of the system and react in a timely manner to any mishappen that may take place; and 2) incorporate adequate test isolation methods.

To reduce human intervention in testing activities, one must propose a solution for automation. The diversity of sources of software as well as the test cases poses the main challenge to automation. In fact, test cases coming from different vendors may require different test architectures and test execution environments to be deployed. Therefore, the preparation for testing activities must be conducted differently based on the source of the test case and the software involved in the realization of the service under test.

Test isolation approaches such as BIT may not be applicable in the cloud context due to the diversity of sources of software components that are part of the system. Other cloning-based techniques such as canary release and gradual rollout still have limitations such as ignoring the difference between the environment of the instance to be tested and its clone which can negatively impact the reliability of the test results. [63] reports an incident in which a canary tested configuration turned out to be erroneous after being propagated to the rest of the system. The failure, according to [63], was due to the fact that the feature was canary tested in network locations where some of its scenarios were not executed. These uncovered scenarios have been executed in other network locations after the change was propagated system-wide. Had the new feature been tested in all possible network locations, this problem could have been avoided or at least contained earlier. Similarly, [67] reports on an issue caused by a bug in a feature that was not detected using gradual rollout. Although we cannot confidently claim that this could have been avoided if the feature had been tested in other locations under different conditions, we cannot rule it out as a possibility either.

1.2. Contributions

The goal of this work is to devise a solution for live testing of cloud services. To do so, we have outlined the following research objectives for this work:

- **Research Objective#1**: design an architecture to automate the orchestration of testing activities such as test planning, test design, and test execution in the production environment. As part of the design, a list of the artifacts in play will be compiled, as well as how these artifacts are used in each testing related activity. In addition, the identification of the building blocks of this architecture will guide the choice of the right building block to perform each activity that is needed for live testing to be conducted properly.
- **Research Objective#2**: propose a modeling framework to model the artifacts in play in the architecture. Such modeling helps decouple the behavior of the building blocks of the architecture from the platforms used to realize the System Under Test or the technologies used to develop test cases.
- **Research Objective#3**: propose test methods that may help overcome the limitations of existing test isolation techniques if any.
- **Research Objective#4**: to properly use the test methods developed in Research Objective #3, one has to know how each part of the system may be impacted by the coexistence of test traffic and production traffic as well as how parts of the system may

be impacted by the test methods. In this research objective, we aim to automate the process of obtaining the information needed to make such decisions.

- **Research Objective#5**: The architecture's building blocks exchange artifacts and process them in order to conduct testing activities in the production environment. These activities include test case selection, test planning, and test orchestration and execution. In this research objective, we aim to propose the behaviors we associate with each building block for these activities to be properly conducted in the production environment.

We achieve Research Objective #1 by proposing an architecture for live testing of cloud services. Our proposed architecture is composed of two building blocks, a Test Planner and a Test Execution Framework. The Test Planner is responsible for test planning activities such as test case selection and test plan generation. The Test Execution Framework is responsible for orchestrating and executing the test cases during the test sessions and providing test isolation. This architecture uses UML Testing Profile (UTP) as a modeling framework. To deal with Research Objective #2 we propose extensions to UTP to include some relations between concepts that are relevant for live testing. We map the concepts in the extended version of UTP to the concepts of our architecture to allow for modeling the artifacts involved in our architecture.

To address Research Objective #3, we assessed the existing test isolation techniques from the state of the art and the state of the practice. We concluded that these test isolation techniques have limitations, namely: 1) making assumptions about the components being tested (such as the inclusion of BIT module), 2) making assumptions about the features being tested (such as the case of canary testing which can only test existing features as it relies on the production traffic), and 3) not taking into consideration the diversity of runtime environments that may exist in a cloud system. To overcome such limitations and at the same time provide test isolation, we propose four test methods, the single step test method, the big flip test method, the small flip test method, and the rolling paths test method. Our test methods do not make any assumptions about the components/features being tested as they rely on the ability of the environment to snapshot/clone components, and relocate the load from one component to another. Such assumptions are supported in all cloud environments. Furthermore, our test methods ensure test isolation by using specific instances of the component for test purposes and testing only under a limited number of test configurations simultaneously as the resources' availability and the risk of test interference allow it. We used the rolling paths test method in a Kubernetes managed test bed on a simple REST API service, and the case study showed that the rolling paths actually helped avoid test interferences as opposed to when it was not used and no test isolation was implemented.

The test methods we devised for meeting Research Objective #3 help avoid test interferences, however, the suitability of each one of them to each situation is to be decided based on the risk of test interference and the availability of resources. The configured instance evaluation method we proposed to meet Research Objective #4 can help automate the process of obtaining such information. This information includes the risk of test interference that parts of the system may present as well as the cost of test isolation. The cost of test isolation is defined in terms of the time needed to snapshot/clone a component and the time needed to relocate the load from one component to another. The configured instance evaluation method can be conducted offline (not in production) and helps confirm the presence of test interferences that may manifest at the level of the behavior of parts of the system (violations of functional requirements) or at the level of resource consumption (violation of non-functional requirements). It relies on a scenario generation activity that generates samples of requests to mimic production traffic and test traffic, thus recreating conditions similar to the ones under which the system may be tested in production. A prototype of the scenario generation was implemented, and one of the scenarios it created revealed test interferences when tried manually.

To meet Research Objective #5 we propose solutions to automate the following test activities:

- Test case selection: we propose a method for test case selection for each test session depending on the reason for its initiation. Moreover, given the role configurations play in the operation of a cloud system, we propose a method for configuration-based regression test suite reduction. This method aims to focus the testing effort on finding configuration errors as they are the easiest to fix in production as opposed to source code errors. Our method for configuration-based test suite reduction relies on a classification of configuration parameters into deployment environment dependent configuration parameters and deployment environment agnostic configuration parameters. Such classification can help classify requirements based on the type of configuration parameters involved in their realization. Using this classification of requirements, the mapping of test cases to the requirements they validate, and the history of verdicts of previous runs of these test cases we perform the regression test suite reduction. We also propose a fault model for configuration faults and show that our test suite reduction method maintains the full fault detection power based on this fault model.

- Test planning: to automate test planning we propose a method to automate the generation of test plans. In this process, we automate the selection of test configurations to use in a test session, the selection of the test method to use for each test run as well as the automation of test runs ordering. The goal of test plan generation is to generate a test plan, the execution of which takes minimum time and does not induce unacceptable disturbance. We evaluated the algorithms of the relevant activities in this method and we identified when the solution they yield is optimal and when it is not. We also implemented a prototype of the test plan generation method.

- Test orchestration and execution: test plans that are generated by the Test Planner must be executed with minimum human intervention. To allow for such automation, we propose an execution semantics that we associate with UTP concepts. This execution semantics represent the behavior that the Test Execution Framework will exhibit when it encounters the associated model element in the test plan. Our execution semantics takes into consideration all possible outcomes of running a test run, namely: when the test run is executed and a verdict is obtained, when a failure at the level of the test component occurs during the execution of a test run, and when a failure occurs and we cannot identify whether it is at the level of the test component or at the level of the test item. We also propose how the Test Execution Framework should deal with each outcome in a way to maintain the consistency of the runtime state of the system. We also show that the mapping between the UTP model elements and their associated behaviors is sound.

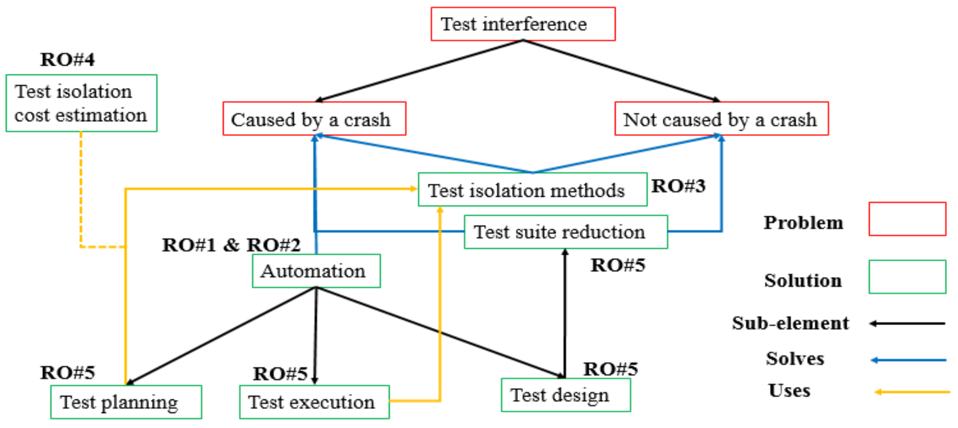


Figure 1-1 Contributions and their mapping to the research objectives

The solutions we propose towards achieving the research objectives we outlined for this work can be put together to provide an integrated solution for live testing. As summarized in Figure 1-1, test interferences which are the main challenge of live testing can be either caused by a crash or not. As a result, to address the risk of test interferences one can use various strategies. A first example of such a strategy is to reduce the chances of test interferences taking place, which we aim at with Research Objective #5 via test suite reduction. Another example of a strategy that can be used is to isolate the traffic to avoid the impact of testing on the production traffic, which we aim at with Research Objective #3, using test isolation methods. Furthermore, one can ensure that testing activities are conducted in the production environment in an adequate way and that any unfortunate event is handled in a timely manner. This strategy manifests in our work as the automation we propose to achieve Research Objectives #1, #2, and #5. The automation of test planning helps ensure that the testing activities are conducted properly, and the automation of test execution ensures fast recovery of the system from the impact of test interferences when they occur. The test isolation methods we propose to achieve Research Objective #3 play an important role in this automation. In fact, they are used during test execution to provide test isolation, and they are selected during test planning to guide the test execution. Finally, the test planning requires extra information to be able to guide properly the test execution by selecting the right test isolation method for each situation. The solution for the estimation of the test isolation cost we propose to achieve Research Objective #4 aims at providing such information.

1.3. Organization

The rest of this thesis is organized as follows. Chapter 2 lays out the background for this work to introduce the terminology we use in this document, and gives a review of the related work. Chapter 3 introduces the architecture we propose and the modeling framework that goes with it, its content was partially published in [74]. Chapter 4 presents the Test Execution Framework which uses the test methods we propose for test isolation and the execution semantics we propose to automate tests orchestration in production. The content of Chapter 4 was partially published in [73, 74]. Chapter 5 covers the method we propose for configured instance evaluation for live testing which was published in [91]. Before concluding in Chapter 7, we present the contributions on the Test Planner. This work includes the automation of test case selection which was partially addressed in [21, 72, 74] as well as the automation of test plan generation which is published in [92].

Chapter 2

Background and related work

In this chapter, we present a background on cloud systems and the current practices used to test them in production. We also present UML Testing Profile (UTP). Then we review the state of the art and state of the practice related to this thesis in the related work section.

2.1. Background

In this section, we provide a background on cloud systems. We also present the practices that are used to test a system in its production environment. Finally, we provide an overview of the UML Testing Profile (UTP).

2.1.1. Cloud systems

In this section, we give an overview of cloud systems and present the terminology we use throughout this thesis. Given the importance of configurations for cloud systems, we present the types of configurations involved in the composition of a cloud system and how these configurations bind the runtime states in which a cloud system can be.

2.1.1.1. Overview of configurable software and configurations

To speed up the time to market their services, service providers, nowadays, rely on cloud computing. In this model, service providers can serve multiple tenants using shared resources (between their tenants) and shared infrastructures with other service providers. Such infrastructures are provided by infrastructure providers that can host applications from different service providers. To accommodate various tenants' requirements, service providers use configurable software. Configurable software is software that cannot be used without a

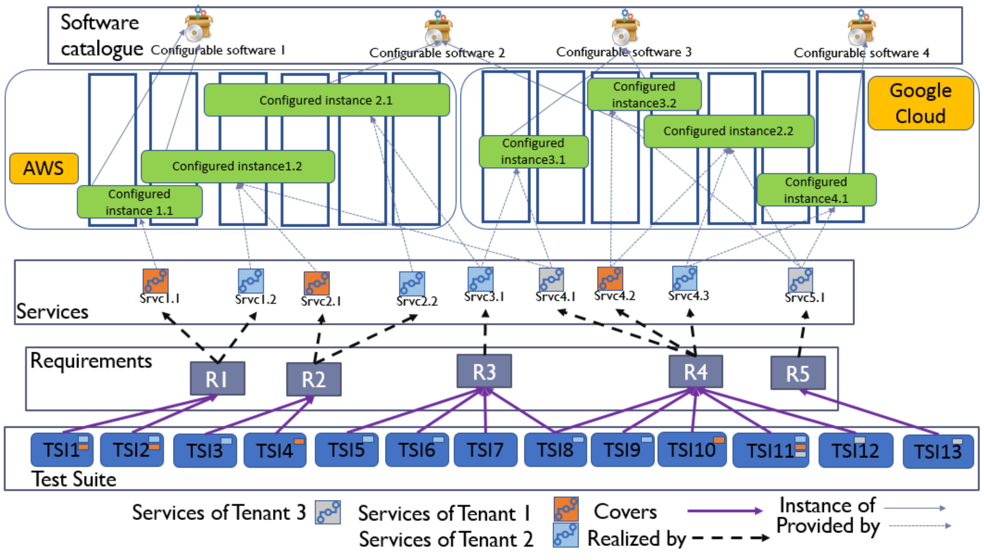


Figure 2-1. Example of cloud system

configuration. A ***configuration*** is a set of (key, value) pairs in which the keys represent parameters of the configurable software's functions. The values of such parameters do not change during the system's normal operation, e.g. no bug is detected, or no new feature is requested. We refer to these parameters as ***configuration parameters***.

Tenants request functionalities (functional requirements) with certain characteristics (non-functional requirements). To satisfy the tenants' requests, the service provider may use an already deployed configured instance (e.g. in multi-tenancy architectures as Configured instance 2.2 is shared between tenants in Figure 2-1.) or may create new configured instances (e.g. in single tenancy architectures as Configured instance 1.1 and Configured instance 1.2 in Figure 2-1.). A configured instance is a system resulting from configuring a configurable software. A configured instance can be used, as-is, to satisfy a refined subset of the requirements (reflecting the purpose of the configured instance) towards the configurable software. The set of configured instances that a service provider owns and manages composes the cloud system of that service provider.

Tenants may have the same or different requirements. A tenant requirement is realized using a ***service instance*** resulting in as many service instances as there are tenant requirements (e.g. in Figure 2-1. R1 is realized by Srvc1.1 for tenant 1 and by Srvc1.2 for tenant 2), i.e. one service instance per requirement per tenant. To provide a service instance, the service provider may use one (e.g. Srvc1.1 of Figure 2-1.) or more configured instances (e.g. Srvc2.1 of Figure 2-1.). Moreover, a service provider may or may not use configured instances of the same configurable software to provide service instances that realize the same requirement for different tenants. In other words, a requirement R may be realized for a tenant using a service instance provided by a configured instance of configurable software CS1, while that same requirement R is realized for another tenant using a service instance provided by a configured instance of configurable software CS2 (Requirement R3 of Figure 2-1. for instance). In addition to sharing configured instances, tenants may also share ***deployment environments*** (e.g. service instances of different tenants deployed in the same clouds, same datacenters, or same machines). On the other hand, a single tenant may have his service instances deployed in the same or in different deployment environments.

A configuration is used to realize a refined subset of the requirements of the configurable software that the configured instance is supposed to satisfy. Depending on the architecture of the configurable software, one or more configuration may be needed to realize the requirements. One configuration, the application configuration, is used to realize the requirements of the tenant in a single tenant architecture. In a multi-tenant architecture, one configured instance may be shared between multiple tenants. Therefore, we end up with an application configuration that is processed by the configured instance and shared between all the tenants that are using this instance; and, a tenant configuration that is managed by the configured instance and specific to each tenant.

Table 2-1 illustrates various requirement refinements for an e-commerce application and the configuration settings to satisfy these requirements. The configuration may be used to control the presence/absence of features of the configurable software in the configured instance (the first case). The configuration may also be used for the concretization of generic features of the configurable software (second and third case). The last case covers the refinement of the accessibility aspects (access to the services exposed by the configured instance or the service that the configured instance may need to provide its services properly). These settings in a configuration may be straightforward (e.g. the function refinement in the second case of Table 2-1), or may rely on the expertise of a configuration designer to properly map the requirement to the configuration (e.g. refinement of an interaction with the environment as in the third case). To put changes in application configurations into effect, one often needs to restart the configured instances. The same does not usually apply to tenant configurations as they need to change dynamically.

Table 2-1 Examples of refinement of configurable software's requirements and their manifestation in the configuration

	Requirement on the configurable system	**Requirement on a configured instance**	**Manifestation in the configuration**
Activation/deactivation	Payment per credit card is an optional feature in the configurable system.	An e-commerce application that allows payment per credit card.	A Boolean configuration parameter set to “true” or “1” (e.g. cc_payment=true).
Refinement of a function	The number of items per cart should not exceed a pre-set number.	An e-commerce application that allows for up to 10 items per cart.	An integer configuration parameter set to 10 (e.g. max_item_per_cart=10).
Refinement of an interaction	The duration for which the system waits for confirmation of a monetary transaction should not exceed a pre-set number.	An e-commerce application for which the processing of the payment should not exceed 1 minute.	An integer configuration parameter set to a duration in a time unit (e.g. if time unit is seconds, we can have a configuration parameter set as follows: transaction_timeout=60).
Refinement of accessibility	The credit card payment if activated should be accessible using TCP through a pre-set port.	An e-commerce application that allows payment per credit card.	An integer configuration parameter holding the port number (e.g. cc_service_port=1025)

2.1.1.2. Configurations at runtime

When instantiated, a configured instance yields a set of ***components*** that provide the actual service instance. The number of such components, their locations, their interactions with components of other configured instances, the policies that govern the number of such components, etc. are aspects set using deployment configurations. A configured instance may be deployed on several physical servers or virtual machines known as the ***nodes***, i.e. its components may be running on any of the nodes on which it is deployed. Figure 2-2 depicts such a setup, configuredInstance1 for instance has one component running on node N2 while configuredInstance2 has two components running on nodes N2 and N3. Such a design is usually used for capacity and/or fault tolerance purposes. Therefore, at any moment of the system's lifespan, components of configured instances may be running on the same nodes, on different nodes, bounded to the same or different components of another configured instance, etc. The ***runtime configuration state*** of the system is defined by the locations of the components of each configured instance, their numbers per configured instance, and their binding information. The system's deployment configuration defines the set of runtime configuration states in which a system can be, and it is also called the viability zone when talking about self-adaptive systems [23]. When the system is in a given runtime configuration state, each component is located on a specific node, in a specific network, sharing that node with a set of components from other configured instances.

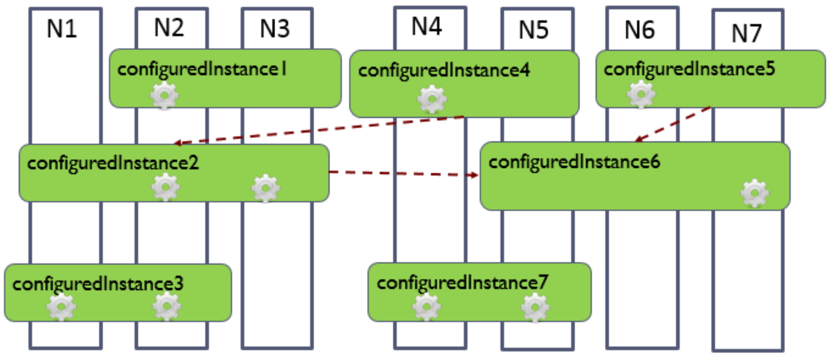


Figure 2-2 Closer look at configured instances at runtime

To maintain a requirement or satisfy a new one, administrators often must change the configurations. Since a service instance is provided using one (e.g. Srvc 1.1 in Figure 2-1.) or more (e.g. Srvc 3.1 in Figure 2-1.) configured instances, the service instance may be associated with different configurations for the different configured instances involved in its provisioning. A service instance provided using two configured instances for example, can be associated with an application configuration and have a tenant configuration for the first configured instance, and associated with an application configuration and a deployment configuration for the second configured instance. These configured instances are then put together to provide the service instance. A service instance is "reconfigured" when any of the configurations involved in its provisioning undergoes a change. In addition, two service instances are said to be provided the same way – they are alike – if the set of configurable software used to create the configured instances involved in the provisioning of these two service instances is the same.

To ensure compliance with the requirements of the service instances, service providers also design test cases or online test design techniques to perform acceptance tests on their cloud systems. The test cases and the test design techniques in the test suite, ***Test Suite Items (TSIs)***, cover all the requirements that should be fulfilled by the cloud system. A given TSI that covers a given requirement may or may not be applicable to all the service instances that realize this requirement. In fact, since service instances that realize the same requirement are not always alike, the service provider may have different TSIs that cover the requirement depending on the configurable software of the configured instances involved in the provisioning of the service instances realizing this requirement. As shown in Figure 2-1 a TSI may be applicable to a service instance only if provided using configured instances of specific configurable software (e.g. in Figure 2-1. TSI7 and TSI8). A TSI may also be applicable to all service instances realizing a requirement the TSI covers, such as TSI5 in Figure 2-1. Such TSIs may be used to ensure the compliance of the behavior or interface of the service instance to a given standard. A TSI may cover one or many requirements. TSI1 for instance covers one requirement R1, and TSI6 covers multiple requirements R2 and R3.

2.1.2. Testing in production

Testing in production, also referred to as runtime testing, is defined as executing tests on a system in its operational environment without causing any disturbance to the system's usage [1]. In other words, runtime testing aims at testing a system in production without that leading to a violation of a system's functional or non-functional requirement due to the coexistence of test traffic and production traffic. Runtime testing can be conducted in one of two phases: 1) **deployment time testing** [1] which is testing a system when it is first deployed in production but before it starts serving actual users, and 2) **service time testing** [1] which takes place while the system is serving its actual users.

Depending on the goals of conducting it, runtime testing can be done in different ways. Blue/Green deployment [46], for instance, is used to validate a system before it starts receiving production traffic. It consists of maintaining two identical environments (one is the blue environment and the other is the green environment); at any moment, one of the two environments is serving the users and the other is idle. When a new feature is added, it is deployed in the idle environment, and tested, when all tests pass the idle environment starts receiving the production traffic and the active environment becomes idle. Other methods of testing such as canary release [44], gradual rollouts [44], or shadow releases [45] are used to validate newer versions of existing features before releasing them. They consist of redirecting a portion of the production traffic to the new versions and comparing performance with the old version. If the new version outperforms the old version, then the new version is released, and the old version is retired. A/B testing is yet another method used for running tests in the production environment, its purpose is to evaluate user appreciation of new features. In this method, a new feature is released only to a portion of system users; if the new feature manages to satisfy a pre-specified fit criterion without any problems it is then released to all the users.

Test interferences are the main issue with runtime testing. Test interferences are violations of a system's functional or non-functional requirement due to the coexistence of the test traffic and the production traffic. When testing a component of the system, test interferences can manifest either at the level of the component under test, not at the level of the component under test but within the boundaries of the system under test, or at the level of external systems with which the system under test interacts. Many reasons may be behind a component presenting a risk of test interferences including the kind of state the component maintains, its resource consumption patterns, and the nature of interactions in which this component is involved. ***Test isolation methods*** are the countermeasures taken to avoid test interferences. Test isolation methods are based on many techniques such as using Built-In-Test modules within the components of the system, separating or maintaining a component state through cloning or snapshots, monitoring resource consumption, pre-empting testing activities, or scheduling the tests within timeframes when the risk of interferences is minimal (e.g. during maintenance windows).

2.1.3. UML Testing Profile

UML Testing Profile (UTP) [27] is an OMG standardized modeling language that is also part of the UML ecosystem. UTP's purpose is to enable the design, visualization, specification, analysis, and documentation of artifacts used in various testing approaches. UTP covers the following aspects of testing:

- Test planning: which includes both test analysis and test design. Test analysis covers concepts that enable the modeling of artifacts that justify the testing activities, such as mapping test cases or test sets to the test requirements or test objectives they achieve. Test design is more concerned with how the test cases that achieve these objectives are obtained. It includes concepts that cover test design techniques, their required inputs and expected outputs (test data, test cases, etc.), and the test objectives achieved by the test cases they generate.
- Test architecture: This mainly includes concepts that help model the architecture used during the execution of each test case. This includes concepts that enable the specification of test items (component or system under test), test components, test items configurations, and test components configurations.
- Test behavior: covers concepts used to model the behaviors and scheduling of test cases. It includes high level concepts such as test case and test procedure which are used to encompass the behavior of a self-contained test case. Moreover, it proposes concepts that capture the atomic actions involved in the description of the behavior of a test case or test procedure (stimulus creation, property checking, logging, etc.). Test behavior modeling is not the only aspect covered in this section of the standard; in fact, UTP also enables modeling the phases at which each action takes place. The standard proposes three phases. The setup phase includes actions for the preparation for the execution of the test case or test procedure, the main phase which is the phase at which

the test case or test procedure will be executed, and the tear down phase at which test completion activities will be conducted (cleanup of test architecture, logs storing, etc.).

- Test data: which covers concepts that enable the modeling of how test data is specified, provided, and how they relate to each other.
- Test evaluation: mainly offers concepts that enable the modeling of arbitration specifications, i.e. how a test case or test action verdict is generated. In addition, it also enables the modeling of how the logging is done during tests.

2.2. Related work

In this section, we present the related work that addresses configuration testing, automation of testing activities in the production environment, and the handling of test interferences.

2.2.1. Configuration testing

After a reconfiguration, the system administrator needs to perform regression tests to evaluate the compliance of the system (under the new configuration) to its requirements. Typically, the regression test suite is a subset of the set of TSIs available for the administrator and that is used to validate the cloud system. Selecting the regression test cases can be done using different strategies:

- Retest all [24]: in this case, all the TSIs will be in the regression test suite.

- Change based test case selection [26]: in this case, only impacted configured instances will be subject to tests. Several impact analysis methods were proposed in the literature.
- Test case prioritization [22, 23]: in this case, the whole set of TSIs is considered, and the test cases are executed in the order of their priority. The execution continues as long as the resources allow it. Resources can be time (test window), hardware/virtual resources, cost, etc. The subset of test cases that will be executed depends on the priority and the resources available for the regression test.
- Test case minimization [25]: in this case, the same subset of TSIs is used after each reconfiguration.

Misconfigurations are another type of configuration errors that is addressed in the literature from a configurable system testing perspective and not from configured instance testing perspective. In other words, works that addressed misconfigurations aim at evaluating the configurable system's response to a misconfiguration rather than linking a requirement violation to a misconfiguration. Syntactic misconfigurations humans can make were addressed in ConfErr [17]. The authors built a psychological model of the mistakes a configuration designer may make; and used it to generate misconfigurations and evaluate how the configurable system responds to them. MisconfDoctor [20] on the other hand, from automatically generated misconfigurations, builds a comprehensive set of logs that can be used to diagnose misconfigurations. The authors address both mistakes on a single configuration parameter known as simple syntactic misconfigurations; and multiple parameters faults which may violate constraints that should hold between configuration parameters. To generate misconfigurations, MisconfDoctor uses a classification of configuration parameters based on their types and applies different rules to each type of configuration parameters to generate misconfigurations. [51] addresses robustness testing of configurable software. The authors relied on a fault model that classifies faults into configuration dependent robustness faults and configuration independent robustness faults. Their main goal is to test the configurable software's error handling mechanism knowing that such mechanisms often depend on the configurations.

The work in [19] presents a survey of configuration fault localization methods. It highlights that white box approaches are commonly adopted in testing configurable software and configured instances. Authors in [9] propose the use of coverage criteria for regression test case selection. After a reconfiguration, the authors propose improving quality assurance activities by selecting test cases that cover new items that were not covered by the test cases under the previous configuration. Static code analysis techniques have also been used for configuration fault localization. The solutions proposed in [10, 11] rely on dynamic and static slices of configurable software and explore their similarities to localize crashing errors and non-crashing errors [11]. Non-crashing errors can be implicitly considered requirement violations. Text based techniques were also used in configuration fault localization. The work in [49] proposes ConfEx which is a framework that can discover configuration files in a system and identify misconfigurations if any.

The work in [12, 18] uses a black box approach to test the system after a reconfiguration. They classify the components of a component-based system taking into consideration how these components were impacted by the change. The authors then propose different strategies to select test cases to cover each component based on how impacted it is by the change.

Similarities between code slices, architectures, or requirements are considered a common way to address the test suite reduction problem for Software Product Lines (SPL) testing. SPLs, just like configurable software, enable the definition of a core reference architecture (configurable software) that can be concretized into custom architectures called products (configured instances) that realize specific requirements. [13] presents an extensive survey on test suite reduction for testing SPLs and their products. The approaches that were studied explore the similarities between an SPL and its products as well as similarities between products of the same SPL to reuse test cases and eliminate redundant ones. The work in [14] proposes a method to select from the SPL test cases a set of test cases to test the product taking into consideration the similarities between the requirements satisfied by the SPL and those for which the product was devised. This work was extended in [15] to include dataflow analysis to eliminate redundant test cases. The authors in [16] propose a solution to reduce the number of test cases necessary to test a new product based on its similarities with previous products. Such reduction relies on the use of the differences between the architectures of the products (both new products and previously tested products) to select test cases that only target the parts of the architecture that were not tested previously. They follow a white box testing approach that has no consideration for the requirements for which the SPL or the product were made.

Few works in the literature addressed testing configured instances with respect to the requirements for which they were devised. The work in [48] for instance proposes principles to reuse the test cases, which were used to test the configurable system, to perform acceptance testing on configured instances. The principles relied on abstracting the configuration parameter values in the test cases and only concretizing them at testing time. A major limitation of such an approach is the absence of a guarantee that the test steps will still be applicable after the configuration parameters have been concretized in the test case. ConfAdvisor [54] proposes a runtime verification-based approach to meet the performance requirement of configured instances. The solution proposed in this proposes deployment configurations and application configurations changes, advice, that are inferred using human provided rules. The authors also mention that such rules can be automatically fed to ConfAdvisor if successfully extracted by tools such as PracExtractor [56].

2.2.2. Test interference and test isolation

Testing a system in production attracted the attention of both industry and academia as it may have different purposes. Some works such as [55, 58] use it to detect noisy neighbor problems for cloud hosted applications. Other works such as [35] for instance, propose to test a system in production to predict faults based proactive adaptation. In the solution they propose, a service-based application is monitored, and a set of test cases are selected taking into

Table 2-2 Comparison of test isolation methods from the literature

	Component Under Test agnostic	Test case agnostic	Feature Under Test agnostic	Service time testing	Deployment time testing	Violations of functional requirements	Violations of non-functional requirements	Resource requirements
Scheduling [2]	Yes	Yes	Yes	Yes	Yes	Minor	Minor	None
Cloning [2]	Yes	Yes	Yes	Yes	Yes	None	Yes	Yes
Pre-emption [2]	No	Yes	No	Yes	Yes	None	None	None
Resource negotiation [2]	Yes	Yes	Yes	Yes	Yes	Yes	None	Trade off
BIT modules [1,3]	No	No	No	Yes	Yes	None	None	None
Canary releases [44, 8, 42]	Yes	Yes	No	Yes	NA	Yes (limited)	Yes (limited)	None
Gradual rollout [44, 42]	Yes	Yes	No	Yes	NA	Yes (limited)	Yes (limited)	None
Dark launches [45, 42]	Yes	Yes	No	NA	Yes	None	None	None
Blue/green [46]	Yes	Yes	Yes	NA	Yes	None	None	Significant

consideration the usage profile built considering the monitor data. Running the selected test cases against the service-based application, the administrator can anticipate problems that may require a change in the services bindings. Service composition as addressed in [41 and 47] is yet another activity that can benefit from testing a system in production. The work in [41] proposes incrementally testing service compositions in production to be able to roll them back early enough if any problem is detected. The authors also propose to guide the selection of test cases based on information that was monitored in production and trace analysis. The work in [50] showed that although runtime verification may not be very helpful in identifying GUI related bugs, it remains useful for validating the backends of applications and their resilience to network related issues especially when hosted in the cloud. [35, 40, and 51] rely on testing systems in production for different purposes, however, they did not propose any solutions to conduct such testing without disturbing the system. To achieve such a goal one has to: 1) have a solution to alleviate the risk of test interferences, 2) have a solution to identify parts of the system that present a risk of test interference, and 3) have a solution to assess the cost of test isolation.

Test isolation methods are the countermeasures taken to reduce the disturbance experienced by the system due to the coexistence of the test traffic and the production traffic, i.e. alleviate the risk of test interference. Table 2-2 summarizes the test isolation methods from the literature, their uses, and limitations. For each method, it shows whether it applies to all components or just to a specific type of components, whether it can be used with any test case, and whether it can be used for testing any feature of the system. Table 2-2 also shows whether the test isolation method is used/was designed for service time testing, deployment time testing, or both. Furthermore, it shows for each isolation method if when used there will be still a risk of violations of the system's functional requirements, non-functional requirements, or both. Some test isolation methods such as canary release and gradual roll out [44] have a limited risk as the number of users who will be exposed to this risk is limited. Finally, the table shows for each test isolation method whether it is resource demanding or not. Some test isolation methods require extra resources, some more than others, other test isolation methods do not require extra resources; resource negotiation [2] however makes a tradeoff between the availability of resources and the ability to conduct testing. The author in [2] proposed a set of test isolation methods to alleviate the risk associated with test interferences, however, in this work the author did not show any sign of handling crashes that can result during testing whether a test case was successful or not. The main focus of the work in [2] was to assess the runtime testability of the SUT provided a set of available test isolation methods; and, propose methods for test case selection that can balance between reducing the cost of runtime testability and the cost of runtime diagnosis. The authors in [1, 3] propose alleviating the risk of test interferences using Built-In modules called Built- In Tests (BITs) to be able to test a system in production. The work presented in [5] proposes the use of the methods mentioned in [1, 2, and 3] to avoid test interferences. Moreover, the authors extended TTCN Test Architecture to orchestrate test case execution in production. Although the solution proposed in this work allows automation of test case execution and test configuration deployment, it remains limited to the use of TTCN as a language for test case specification in specific environments (OSGi managed JAVA systems). Other methods for running tests in the production environment such as canary releases [44, 41], gradual rollouts [24], and dark launches [45] leverage the use of production traffic for testing purposes. In canary releases a new version of an existing feature is released to a subset of customers for a period of time, the new version will be then released to all the customers only if it does not show any problems during this testing period. Unlike in canary releases, in gradual rollouts the percentage of customers using the new version will keep increasing in small steps (5% for instance) until the new version takes over (in case no problems were revealed while it is being tested). Dark launches consist of deploying the new version without releasing it to any customers; however, the production traffic is duplicated, and the behavior of the new version is compared to the behavior of the old version. If no problems are detected during the testing period, the new version will be made visible to the users. As they rely on duplicating the production traffic for testing purposes, these methods have limited applicability as they can only be used to test new versions of an existing feature, i.e. they are not applicable for new features for which there is no production traffic to duplicate. The same applies to methods such as Simplex [37] which is used for dependable live upgrades and testing of real-time embedded systems. Blue-Green deployment [46] is a technique that is used to enable zero downtime live upgrade and testing. It consists of maintaining two identical production environments, Blue and Green, of which only one is being used to handle the production traffic and the second remains idle. When it is time for an upgrade, the idle environment will be upgraded, and tested, if the new setup passes all the tests the production traffic will be redirected to the idle environment and the active environment will go to the idle state. The major problems with this approach are that it comes at a high cost from resource perspective, it poses the challenge of maintaining two environments in synch, and it is not applicable for large scale systems.

The literature on live testing such as [1, 2, 3, 4] tackled to some extent the problems related to test isolation; however, they relied on human provided information on the risk of test interferences that components may present when tested in production as well as the cost of test isolation to alleviate that risk.

Other types of interferences, such as database interference, can be considered similar to the problem we are addressing and present similar challenges. Isolation in databases, whether it is transaction isolation or snapshot isolation, is classified into levels of isolation taking into consideration the interferences it can avoid, i.e., interferences related to read, write, commit, and abort operations. Elle [82] is a checker that can assess the level of isolation of database management systems. To achieve this goal, the authors proposed a method to generate "histories" (i.e., simulation of transactions coming from different clients), then infer a dependency graph from the generated histories and check if it exhibits certain patterns associated with anomalies resulting from interferences. In this work, the authors proposed how to generate the histories. Moreover, to detect the interferences they relied on patterns that should appear in the dependency graph.

Another phenomenon that can be compared to test interferences/isolation is the correctness attraction addressed in [83] and which can be used to assess the quality of testing strategies [84] or to automate program repair [85]. Correction attraction is the ability of a program to maintain correct behavior even when it is subject to runtime perturbation (perturbation to its internal state as defined by the values of its variables). From this definition, one can see that enhancing a component with test isolation mechanisms is similar to improving its correction attraction. [83] proposes a protocol, Attract, to assess the correction attraction of programs based on their source code. One of the challenges they addressed is the generation of runtime perturbation that will allow for the assessment of correction attraction. They proposed a set of perturbation operators that act on variables based on their types. Furthermore, to evaluate the correction attraction they compared the result of the execution under perturbation with the results obtained with a non-perturbed run of the same program with the same input. [86] followed a similar approach, however, the perturbation was limited only to switching execution branches. Other kinds of perturbations, such as value alteration, were not considered. [85] deals with other types of perturbation in order to automate repair. The authors considered environment induced errors and the automation of their repair by the alteration of thread execution schedule, allocating more memory, or denying some client requests.

2.2.3. Automation of testing activities in production

The automation of testing activities in production consists of automating test orchestration and test planning. In the following we go through the literature that addressed both these aspects of automating testing activities.

2.2.3.1. Automation of test orchestration

Several architectures have been proposed for tests orchestration. The authors in [38] 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 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. [38 and 41] describe architectures that can be used to manage passive tests (monitoring) in production. [1, 3, 5, 6, 7, 59, and 64] 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 38] 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 an elapse 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 [38]); or, they can be additions or removals of one or more components (e.g. [5, 59]). Other types of events such as elapsing of timers to establish periodic checks [1, 3, 64], 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 event-driven architectures in the literature.

The architectures and solutions from the literature can also be classified based on the types of tests they orchestrate. 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 [44] is one of the commonly discussed projects when it comes to the tests that are often conducted in the production environment. It enables performing resiliency testing through fault injection in the production environment. The strategy that Netflix follows to deal with test interferences consists of running Chaos Monkey during business hours (Monday to Friday, 9:00 am to 3:00 pm); thus ensuring that the workload is relatively low on the one hand, on the other hand having the engineers on-site in case their intervention is needed. Gremlin [43] is another tool, which is developed by IBM, for resiliency testing. Although it was not evaluated for use in the production environment, 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. These approaches presented so far indeed have the potential to be adapted to be safely used 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.), specific test items (infrastructure, specific software, etc.). As a result, a test engineer who uses these approaches will have to make his 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 is to be tested, and it can be error prone due to the complexity and size of the production systems.

Of the three tasks mentioned in [38], stimulating and observing the SUT are the capabilities the most commonly addressed by the architectures from the literature. However, their reaction capabilities are limited to reactions at the level of the SUT only. [34 and 40] are some of the few works that address how the test system should be maintained as a reaction to an adaptation or a change in the SUT. The approach proposed in [34] 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 binding, but when the binding changed it became inapplicable. At every test session, the architecture is only allowed to select from test cases that have the ACTIVE label.

The work in [40] relies on a different approach with the aim of decoupling test case specification from test case implementation. As a result, at every test session, all the test cases are eligible to be chosen; however, as a 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).

2.2.3.2. Automation of test planning

Automation of test planning is also often addressed in the literature. In addition, the concept of a test plan has more than one definition depending on the community. As a result, the concerns of test plan generation methods in the state of the art depend heavily on the definition used in each work.

The work in [75] uses “test plan” to refer to the test suite. It proposes a method for test suite generation, more specifically test suite generation for GUI applications. A broader definition of a test plan is used in [78], it is seen as an artifact that documents the test scope, test configurations, and test cases used to validate a new version of a product. [78] thus describes a CASE tool that can be used to generate a test plan to document testing of a new version of a product from the test plans that document testing of older versions of that same product.

ISO29119-1 [79] defines a test plan as a detailed description of test objectives to be achieved as well as the means and schedule for achieving them, organized to coordinate testing activities for some test item or a set of test items. Furthermore, ISO29119-2 [80] defines the test planning process as the process used to develop a test plan. Test planning according to this standard consists of several steps among which we find, identification and analysis of risks, identification of risk mitigation approaches, designing test strategy, and determining the test schedule. [77] covers a fair share of these steps as it proposes a method that starts from user configured test parameters, which include at least one test objective, to generate a test execution plan. The test execution plan is a set of actions that achieve the test objective, and that are generated by applying a set of rules (derived from the user provided input and some pre-set rule templates). This method does not explicitly handle the creation of the test schedule, although it can be adapted to achieve that. However, it can select the test actions and determine the test resources (tester, runtime environment, etc.) needed to execute them. [76] also addresses the test resource allocation problem by estimating the duration (man/hour) needed to test a new product. When such a duration is fixed as a budgetary constraint for instance, [76] generates an estimation of the risk associated with testing for that duration based on the defects that will be detected.

The definitions we use coincide to some extent with the definitions proposed in the ISO standard. In fact, the test plan we propose includes the test objective (of the test session) as well as the means to achieve it (TSIs and test configurations). Furthermore, the test plan generation approach that we propose covers creating the test schedule as well as identification of risks (applicability check of test methods as proposed in Chapter 6) and risk mitigation approaches (the test methods we use in our approach as proposed in Chapter 4) and the design of test strategy (test method selection as proposed in Chapter 6). Only a few test planning methods proposed in the literature design test strategies to mitigate the risk of test interferences. The work in [3], for example, proposes a set of algorithms to select test isolation countermeasures that will minimize the cost of implementation. This selection is also balanced with the cost of testing, i.e. how early defects are detected, and the cost of diagnosis, i.e. accuracy of the fault localization. The work in [5] also handles the risk of test interferences using the same test isolation methods as [3]. In addition, it also addresses the resource allocation concern of test planning by minimizing the resource consumption impact of testing in the production system.

Chapter 3

Architecture for live testing of cloud services

In this chapter, we propose an architecture to automate testing activities in the production environment. This architecture helps achieve Research Objective #1 which is about designing an architecture to automate testing activities in the production environment as well as the artifacts involved in this architecture. We also present our extension to UTP and show how we use it to model the artifacts involved in the proposed architecture. Such modeling helps meet Research Objective #2 which is about the use of a modeling framework that can help decouple the behavior of the building blocks of the architecture from the platforms used to realize the system under test.

3.1. Overall architecture

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 System Under Test (SUT). In other words, a good solution for test execution for microservice based systems for instance may not be good enough for ETSI Network Function Virtualization (NFV) 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 testing one must 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 group activities that heavily depend on the target platform into one of the building blocks of our solution; and group the activities that may be reused across several contexts into another building block.

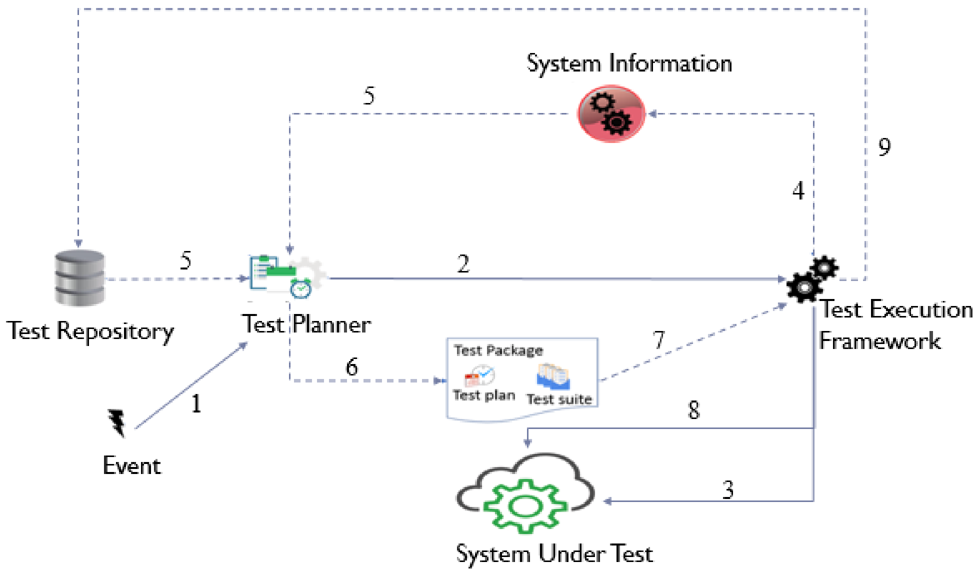


Figure 3-1 Abstract Architecture for live testing

Figure 3-1 captures the information flow and the exchange of artifacts involved in a typical test session when carried out in our architecture. The test session is triggered by an event. Events that may trigger testing include an elapse of a timer for periodic tests, reconfiguration as it requires regression tests, or even a test request issued by an administrator or third-party software. The event is received by the Test Planner which then requests System Information from the TEF. System information which consists of the system's configuration and its current runtime state is then read from the System Under Test (SUT) by TEF and returned to the Test Planner. The Test Planner then will have all the artifacts it needs to generate the Test Package. In addition to System Information, the Test Planner uses also the content of the Test Repository to generate the Test Package. The Test Repository contains the test cases and test design techniques available for the tester. To enable better use of the test cases, the Test Repository also contains metadata related to these test cases such as execution time, history of verdicts, test goals they achieve, test runtime frameworks they need for their execution, etc... Moreover, to allow for good test planning, the Test Repository includes also information about the software that compose the cloud system which includes whether instances of each software present a risk of test interference or not, and information needed to calculate the cost of test isolation.

The Test Planner then generates the Test Package which is composed of a Test Suite and a Test Plan. The Test Suite contains Test Suite Items (TSIs) which are test cases and test design techniques that will be used to respond to the received event. The Test Plan is a road map that will be followed by TEF to execute the TSI runs needed to respond to the received event. When the Test Package is generated it is then fed to TEF which executes it on the SUT.

After the execution, as new verdicts were acquired and new data was collected, TEF then updates the metadata associated with the TSIs in the Test Repository.

3.1.1. System information

This artifact composes the knowledge about the system under test that the tester should have in order to properly execute (and sometimes even design) the test suite items. It includes system configuration, runtime state, available software packages and their properties, and data collected or used by other management frameworks (availability management, elasticity management, virtualization management, etc.).

3.1.2. Test Repository

The test repository stores test suite items (TSIs), test cases and test design methods, along with the test goals they achieve. A test goal is a testable aspect of the system under test which may be targeted/achieved by a TSI. 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). The test repository also holds information collected about the test cases such as the history of execution times, fault exposure rates, etc. The test suite items and the test goals in the repository come from developers, software vendors, or the system administrator. Similarly, the test repository also holds information about test design methods such as their nature (online vs offline), runtime environment, test design input, etc. 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.

3.1.3. Event

An event is what can trigger a test session. We managed to establish the list of event types that one may need to test a system live. The list goes as follows:

- Periodic event: some parts of the system need to be checked periodically as even though unchanged; they may be impacted by changes in their environment or other factors such as aging. A periodic test helps ensure that some subsystems are always correctly functioning in the production environment. This type of events 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 validated through 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 need for regression tests either: 1) by registering to the notifications of the configuration manager; or 2) by being invoked directly by the configuration manager after a live reconfiguration. Another solution may involve the configuration manager creating the event and sending it to the test planner.
- 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 the addition of new TSIs 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 TSIs that were not used together before, which may reveal new errors as well.

- Test request: to keep our architecture generic, we propose the concept of test request to cover 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: sometimes a set of system checks may need to be run together for a period of 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 request. It consists of a set of test goals to be achieved. This type of test requests is mainly practical for test goals that are achieved using test design methods 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 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 third-party software.

3.1.4. Test package

Consists of a test suite and a test plan. The test suite includes the TSIs that were selected from the test repository to respond to the received event. The test plan specifies the ordering of running TSIs under their associated test configurations as well as how they should be executed. The details of the execution of the TSIs include test configurations, test isolation methods, contingency plans to fix/contain crashing errors detected during the tests, etc.

3.2. Modeling framework

In this section, we propose extensions to UTP in order to make it capable of capturing the concepts and relationships needed for live testing. We also present how UTP in its extended version can be used as a modeling framework to model the various artifacts involved in the architecture we proposed.

3.2.1. Extensions to UTP

The concepts standardized in UTP to model test design techniques were only introduced to help with the documentation and traceability of test design activities. However, testing some aspects of cloud systems in production may seem sometimes unpractical without the use of online testing methods as these testing methods allow the generation of test cases as the tests are being executed and they allow to properly react to adaptations of the systems that take place at runtime [36]. As a result, we propose some minor extensions to UTP which can be summed up in the following:

- 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 method.
- 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.
    - TestCaseLog: this concept already exists in UTP, however, it needs to be associated with a TestDesignTechnique element too. In fact, some online testing methods generate new test cases instead of selecting from existing ones. Therefore, for the TestDesignTechnique elements that model these online test methods one needs to specify a TestCaseLogStructure which will be a template for the logs of the test cases that will be generated; and TestCaseLog elements

which will capture the logs of the generated test cases every time the online test method is invoked.

3.2.2. Mapping to the modeling framework

The mapping between the concepts we propose and the ones defined in UTP is shown in Table 3-1. This mapping enables expressing test plans as TestExecutionSchedules that run UTP TestCases. UTP TestCases consist of one or more TSIs provided by the vendor or the developer (along with a test configuration) enhanced with some test isolation countermeasures that need to be set up before the execution of the TSI (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).

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 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 [28]. 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, similar to behaviors, it also allows the usage of other languages. Test configurations may be specified in various languages such as ansible playbook language [29], puppet DSL [30], chef DSL [31], 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.

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 environments about which the system maintainer has enough knowledge.

Table 3-1 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 repository	TestDesignTechnique
Test suite item in the test plan	ProcedureInvocation (e.g. TestProcedure, or TestDesignTechnique invocation) in the main phase of a TestCase
TSI run	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 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

3.3. Summary

In this chapter we presented the architecture we propose for live testing of cloud services. Our architecture is composed of two building blocks, the Test Planner and the Test Execution Framework. Moreover, artifacts such as the system's configuration, the events, and the test repository, are needed for our building blocks to be able to fulfill their purposes. The purpose of the Test Planner is to respond to the events which trigger testing activities. The events we have proposed are expressive enough to be used in various situations. The periodic event and the addition of a test goal for instance are events that can be configured within an implementation of this architecture, and as a result, the Test Planner responds to them systematically. Events such as a reconfiguration or a test request serve as means of interaction between an implementation of this architecture and other entities in the system such as configuration managers, third-party software, or human administrators. The Test Planner responds to events by generating a test package. The test package is composed of a test suite and a test plan that are passed to the Test Execution Framework which executes and orchestrates the tests during a test session.

We have presented the extensions to UTP that we propose to make it suitable to capture all aspects of testing that we may need such as the use of online testing methods. To decouple our proposed architecture from the platforms involved in the realization of the cloud system, we proposed the use of UTP, along with the extensions we have proposed, as a modeling framework for the artifacts involved in our architecture.

The architecture we proposed helped us identify the artifacts that are relevant to the automation of testing activities in the production environment. Moreover, the building blocks of our architecture provide a basis for an implementation that does not depend heavily on the platform and technologies involved in the realization of the cloud system nor the technologies used to develop the TSIs. This platform independence was also achieved using the extended version of UTP to model the artifacts involved in our architecture. Finally, our architecture has building blocks that are capable to stimulate, observe, an adapt to the system as the system changes. Such characteristics make it fit to automate testing activities in the production environment as it can be seen from the behaviors we associate with the building blocks and which are described in Chapter 4 and Chapter 6.

Chapter 4

Test Execution Framework

In this chapter, we propose a set of test methods that can provide test isolation. We aim in Research Objective #3 to propose solutions to overcome the limitations of existing test isolation techniques. Therefore, we propose these test methods to meet Research Objective #3. These test methods are part of the behavior executed by the Test Execution Framework. Research Objective #5 targets partially such behavior as it aims to devise behaviors to automate testing activities in the production environment including test orchestration and execution. Towards achieving this research objective, we propose also in this chapter the behavior to be enacted by the Test Execution Framework in the form of execution semantics.

4.1. Test methods

We aim for solutions capable of covering all runtime configuration states. Therefore, the coverage of runtime configuration states must be incorporated into our test methods. Furthermore, our solution should be independent of the configured instances under test (not assuming any capabilities of the configured instances under test); and, independent of the test cases being executed and the features being tested (like testing only features for which we have production traffic). To meet these goals, we assume that the environment in which the configured instances run has the capability: 1) to snapshot the components of the configured instances composing the system. The snapshot image that is taken should be enough to clone these components. The environment has also the capability to clone a component from a snapshot; and, 2) to relocate service assignment from one component to another. These assumptions are aligned with the cloud paradigm. Due to containerization, snapshotting and cloning, for instance, can be done independently of the technologies used to realize the configured instances.

Building our solution based on the assumptions we made is reasonable as similar assumptions were made in recent works such as [62]. Moreover, tools such as CRIU [71] enable the snapshotting and cloning of processes running in various container technologies such as Docker [32] and LXC [69]. Furthermore, production-like setups containerize even infrastructure services (kubelet and kubeproxy for Kubernetes [33], nova and neutron for

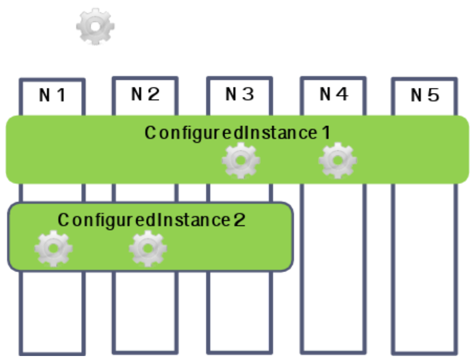


Figure 4-1. Setup for the illustrative example

openstack [70], for instance) which makes this assumption applicable also to infrastructure services. Service relocation is a feature also supported by cloud orchestrators. Such a feature is usually needed to meet QoS requirements such as availability and service continuity. However, not all orchestrators may support service continuity.

4.1.1. Illustrative example

Figure 4-1 shows a setup we will use to explain some concepts used in our test methods. It is composed of two configured instances that share three out of five nodes. The two configured instances share nodes N1, N2, and N3, while nodes N4 and N5 can only be used by ConfiguredInstance1. At the time of the testing, ConfiguredInstance1 and ConfiguredInstance2 have two components each.

To cover all runtime configuration states of the system, one should reproduce these states during testing. Therefore, during testing, a component can be either a serving component, a component under test, or a test configuration component. *A **serving component*** is a component that handles production traffic. A ***component under test*** is a component that receives the test traffic. A ***test configuration component*** is a component, which is not under test, which receives a duplicated production traffic but does not respond to the actual users of the system. Such components are used to recreate the runtime configuration state corresponding to the component being tested. A runtime configuration state is recreated by setting up the ***environments*** for the components under test. The environment of a component under test is defined by its location, on which node it is instantiated, and the test configuration components collocated with it in that location. We define a ***test configuration*** as the set of environments associated with the components under test needed for the TSI run. When a service that is composed of more than one service instance is being tested, the combination of environments associated with the components under test needed to test that service is referred to as the ***path***. Note that path here refers to the path through the environments that make the test configurations and not the path as used in white box testing. The set of paths that need to be covered for testing a service composed of a single service instance provided by ConfiguredInstance2 includes but is not limited to:

- Path1: {location: N1, collocation: comps of {ConfiguredInstance1}}. If we start from the runtime configuration state illustrated in Figure 4-1, this will imply instantiating a test configuration component for ConfiguredInstance1 on N1, while having the component of ConfiguredInstance2 on N1 as the component under test.
- Path2: {location: N2, collocation: comps of {ConfiguredInstance1}}. If we start from the runtime configuration state illustrated in Figure 4-1, this will imply instantiating a test configuration component for ConfiguredInstance1 on N2, while having the component of ConfiguredInstance2 on N2 as the component under test.
- Path3: {location: N3, collocation: comps of {ConfiguredInstance1}}. If we start from the runtime configuration state illustrated in Figure 4-1, this will imply instantiating a component of ConfiguredInstance2 on N3 as the component under test while

ConfiguredInstance1 already has its component on N3 handling production traffic, so it is a serving component.

In the rest of this section, we will describe various test methods that can be used to test configured instances in production while avoiding the risk of test interferences. Note that these test methods are applicable at the configured instance level and not at the path level. That is, to test a service that is a composition of two or more service instances that are provided by different configured instances, we can use different test methods for the different configured instances on the path providing the service. Furthermore, we need to test the service for all the paths through which it can be provided.

4.1.2. Single step

The single step test method can be used to provide test isolation and it allows testing under more than one test configuration simultaneously. Moreover, it also allows the use of serving components for testing purposes. Executing a test case for a configured instance using the single step test method goes as follows:

1- Instantiate components under test to setup the paths that will be taken by the test traffic.

2- Instantiate test configuration components as needed to complete the creation of the environment under which the test case is to be executed.

3- Hit all the paths with the TSIs to be executed.

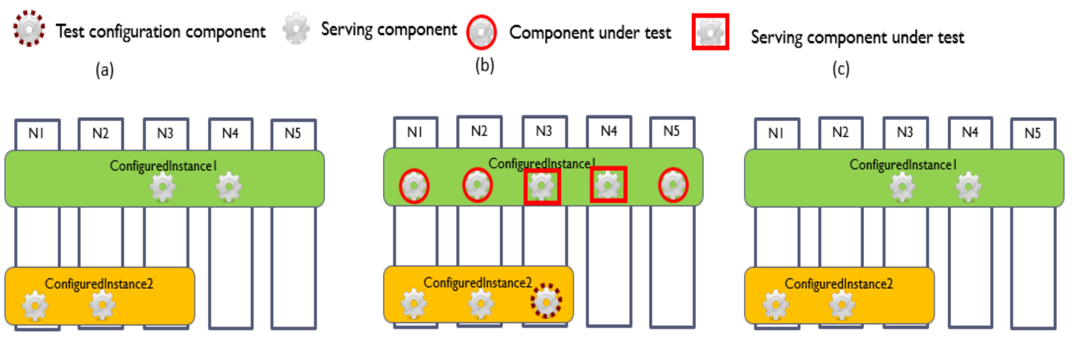


Figure 4-2 Testing iteration for the single step test method applied to ConfiguredInstance1

The single step test method can be used for configured instances with no potential risk of test interferences, i.e. when the testing activities have no impact on the configured instances' behavior. Figure 4-2 illustrates the iteration for using the single step method to test a service composed of one service instance which is provided by ConfiguredInstance1. From the configuration of ConfiguredInstance1, one can deduce that the service can be provided through one of the eight paths (that means also eight environments as we are in the case of a service composed of a single service instance). As shown in Figure 4-2, one can exercise five paths at a time using this test method. The figure shows five out of the eight paths tested. Since there is no potential risk of interferences, components of ConfiguredInstance1 can handle test traffic and production traffic at the same time. Therefore, test components are instantiated on an as-needed basis. Starting from the runtime configuration state in Figure 4-2 (a), in (b), for instance, for ConfiguredInstance1 we had to instantiate test components on nodes N1, N2, and N5, but for N3 and N4 we use the components which already handle production traffic, i.e. these components play two roles: serving component and component under test (marked as serving component under test). Similarly, the instantiation of test configuration components is not necessary as existing serving components help to create the environments under which one wants to test. However, in the absence of such components, one needs to instantiate test configuration components as it is the case for ConfigurationInstance2 on node N3 in Figure 4-2 (b). Figure 4-2 (c) shows the runtime configuration state after completing the single step test method. It is the same as in Figure 4-2 (a).

4.1.3. Small flip

The small flip test method also allows testing under multiple test configurations simultaneously, however, unlike the single step, it does not allow the usage of serving components for testing purposes. As a result, the small flip often requires more resources than the single step. Assuming we are testing a configured instance that, at the time of testing, requires k components out of the maximum n, where k <= n/2, the small flip test goes as follows:

1- Instantiate k components under test for the configured instance being tested on k available nodes.

2- Instantiate the test configuration components to create the test environments under which the k components under test are to be tested.

3- Test all the paths which can be exercised, and which involve the k instantiated components under test until all the paths have been covered

4- Take snapshots of the k serving components and replace the k components under test with k serving components cloned from the snapshots. Relocate the production traffic to the k new serving components.

5- Run the test cases for components on the rest of the nodes using the single step test method.

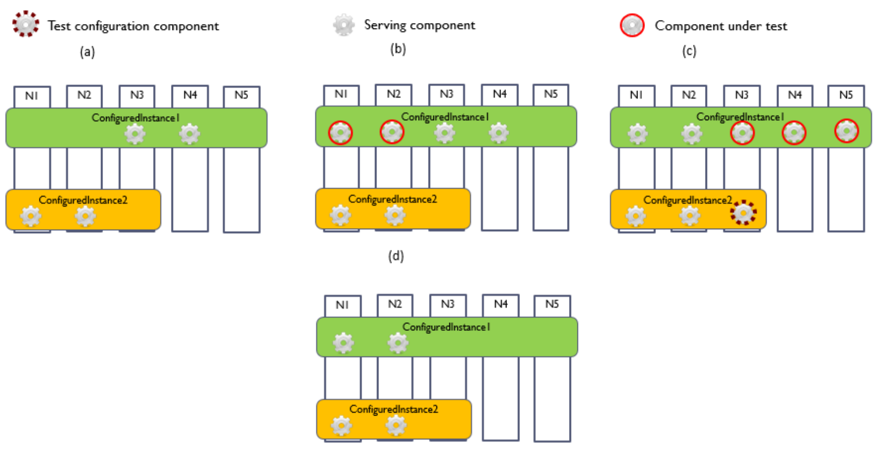


Figure 4-3 Testing iterations for the small flip test method applied to ConfiguredInstance1

The small flip test method can be used to test configured instances when there is a risk of potential test interferences; and, for which the number of currently required components is less than the number of unused nodes that can be used by this configured instance. For instance, the small flip can be used for a configured instance that can have components on six nodes and that, at the time of testing, requires only two components. Figure 4-3 shows the iterations of the small flip test method in the case of a service composed of a single service instance that is provided using ConfiguredInstance1. The small flip is used to test configured instances that present a potential risk of interferences, however, the resources available allow for exercising at least k paths in each iteration. In the first iteration shown in Figure 4-3 (b) two paths are exercised. The small flip results in one service relocation shown in Figure 4-3 (c) when used for only one configured instance involved in providing the service being tested. It shows the application of the single step test method to the remaining nodes. Figure 4-3 (d) shows the runtime configuration state after completion of the test. Note that the test of a service composed of more than one service instance and, therefore more than one configured instance (of the configured instances providing these service instances) is being tested using the small flip, may result in more than one service relocation for some configured instances.

4.1.4. Big flip

The big flip test method also allows testing under multiple test configurations simultaneously. Similarly, to the small flip, it does not allow the usage of serving components for testing purposes, even more so it relies on an extra configured instance to execute the test runs. The big flip test method of a configured instance goes as follows:

1- Create a new configured instance, the test configured instance, that has the same configuration as the configured instance to be tested. The components of the test configured instance are the components under test, while the components of the configured instance to be tested remain as serving components during testing activities.

2- Create test configuration components required to test each path and run the test case.

3- After completing the tests, take a snapshot of the original configured instance to be tested, replace the test configured instance with an instance cloned from the snapshot, relocate the production traffic to the test configured instance (which will make it the configured instance with the new serving components); and remove the original configured instance from the system.

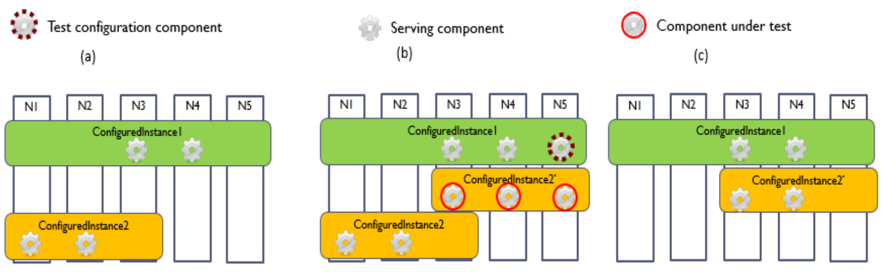


Figure 4-4 Testing iterations for the big flip test method when applied to ConfiguredInstance2

The big flip is the test method that induces the least service disruption and takes the shortest time. However, among the test method discussed here, it has the highest resource consumption. Figure 4-4 shows how the big flip test method is used to test a service composed of one service instance provided using ConfiguredInstance2. As shown in Figure 4-4 (b), a new instance of ConfiguredInstance2 (ConfiguredInstance2') is instantiated on nodes that can host its environments. Such nodes are identified by taking into consideration information from the system configuration such as software installed on each node, anti-affinity rules, etc. The new instance is then used to perform the tests under different paths (which are environments in this case as the service is composed of only one service instance). After the tests for all paths pass, the tested configured instance is replaced by one cloned from the original configured instance, which becomes the one with the serving components (service relocation from the original instance to the new one), and the original configured instance is terminated as shown in Figure 4-4 (c).

4.1.5. Rolling paths

The rolling paths executes the tests under a single test configuration at a time. Moreover, it also does not use serving components for testing purposes. The rolling paths test method applies to a single configured instance as follows:

1- Instantiate a component under test to create the path that will be taken by the test case.

2- Instantiate test configuration components as needed to setup the environments under which the test component should undergo testing.

3- Execute the test case on the created path.

4- Snapshot a serving component and replace the already tested component under test with a serving component cloned from the snapshot. Relocate the service to the new serving component.

5- Replace the snapshotted serving component with a component under test to create a new path to be tested.

6- Repeat from the second step until all the paths are exercised by the test case.

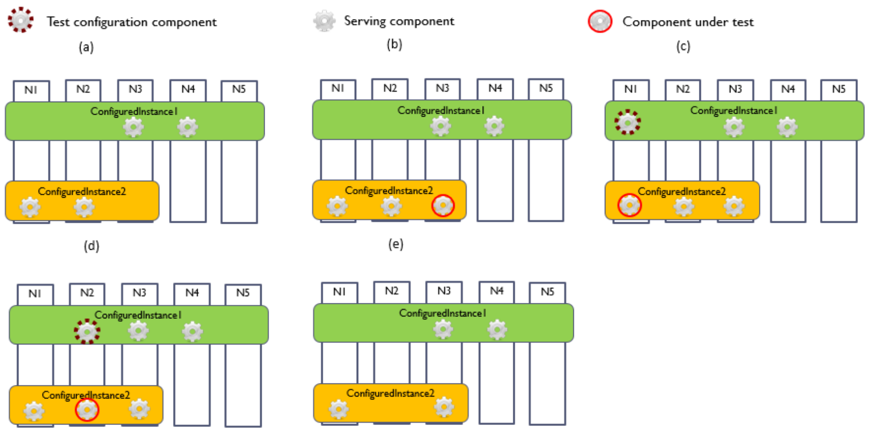


Figure 4-5 Testing iterations for the rolling paths test method applied to ConfiguredInstance2

The rolling paths test method can be used in situations where testing may impact the configured instance to be tested (therefore, serving components under test cannot be used), and the total cost of isolation mechanisms such as snapshot-and-clone and service-relocation is not too high in terms of causing intolerable disruption to the service. Figure 4-5 illustrates a few iterations of the rolling paths in the case of a service composed of a single service instance provided by ConfiguredInstance2. There are six paths to be exercised by the test case. Because of the potential risk of test interferences, test traffic must be isolated from production traffic. As a result, unless there are enough re-sources, one cannot test multiple paths at the same time. A single path is created at each iteration by instantiating the necessary components under test and test configuration components as shown in Figure 4-5 (b), (c), and (d). After executing the test case in each iteration, the service is relocated, and a next path is created for it to be exercised by the test case until all paths have been exercised. Figure 4-5 (e) shows that the runtime configuration state after the testing differs from the one in which the testing activity has started, i.e. Figure 4-5 (a).

4.1.6. Modeling of the test methods

The test method used for test isolation is an important element not yet mapped to UTP. The test methods are patterns in which the test runs can be arranged to isolate the test traffic from the production traffic. Each UTP TestCase combines a set of one or more test suite item (TSI) runs that are run under the same TestConfiguration, i.e. the same path. Fig. 1 shows a TestCase with three invoked behaviors: 1) a behavior to deploy the test configuration, i.e. setup the path to be exercised, 2) a behavior that invokes the TSIs to execute against this path, and 3) a behavior to tear down the test configuration. Note that UTP offers roles that can be assigned to such invocations, namely, setup, main, or teardown.

Since the test methods iteratively execute the TSIs against one or more paths (in each iteration), until all the required paths are exercised; these test methods can be modeled as CombinedFragments of UTP TestCases. The specification of a CombinedFragment, per test method, goes as follows:

- Single step: each iteration of a single step test method is modeled as a ParallelFragment. UTP TestCases corresponding to the paths of an iteration are

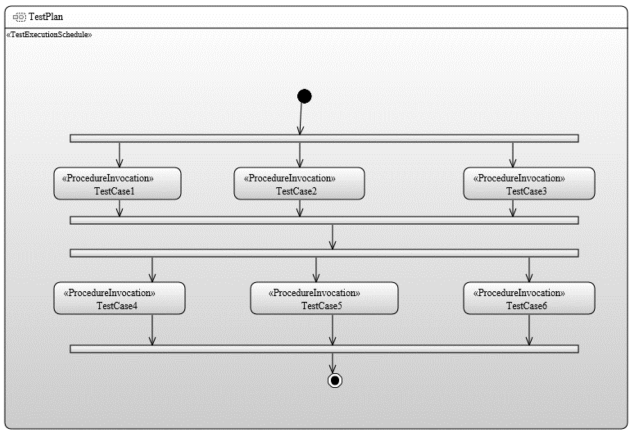

TestPlan
«TestExecutionSchedule»
«ProcedureInvocation» TestCase1
«ProcedureInvocation» TestCase2
«ProcedureInvocation» TestCase3
«ProcedureInvocation» TestCase4
«ProcedureInvocation» TestCase5
«ProcedureInvocation» TestCase6

Figure 4-6. Modeling of a single step test method

invoked in the fragment of that iteration. Fragments of the iterations are then put in sequentially. Figure 4-6 illustrates such a pattern for a situation where a set of TSIs is to be executed under six paths and the maximum number of paths that can be deployed at once is three. Therefore, the model ends up with a first parallel fragment that executes the TSIs under the first three paths and then deploys the next three paths to execute the TSIs against.

- Rolling paths: the rolling paths is modeled as a sequence of UTP TestCases Figure 4-7 illustrates a rolling paths test method modeled in UTP. In the rolling paths, only a single path can be setup at a time, therefore, the UTP TestCases are invoked sequentially. Each invoked UTP TestCase executes a set of TSIs against a test configuration (i.e. a path).
- Small flip: the small flip is modeled as two consecutive single steps (Figure 4-6), targeting two disjoint sets of paths, separated by a service relocation ProcedureInvocation.

- Big flip: the big flip is modeled as a single step (Figure 4-6) preceded by a

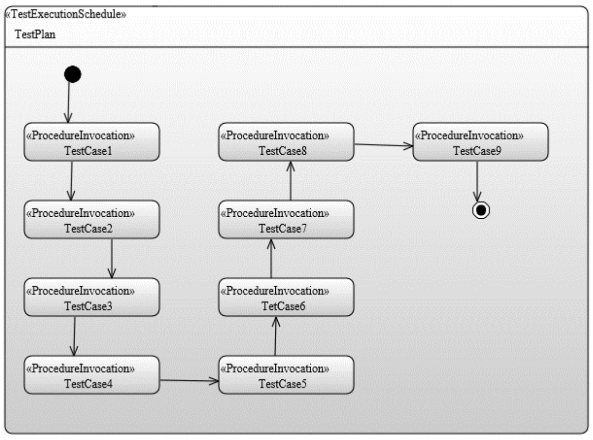

«TestExecutionSchedule»
TestPlan
«ProcedureInvocation» TestCase1
«ProcedureInvocation» TestCase2
«ProcedureInvocation» TestCase3
«ProcedureInvocation» TestCase4
«ProcedureInvocation» TestCase5
«ProcedureInvocation» TetCase6
«ProcedureInvocation» TestCase7
«ProcedureInvocation» TestCase8
«ProcedureInvocation» TestCase9

Figure 4-7. Modeling the rolling paths test method

ProcedureInvocation that sets up the test CI; and, followed by a ProcedureInvocation that relocates the service and removes the old CI.

4.1.7. Test method combined with coverage criteria

A cloud service needs to be tested against all possible runtime configuration states of the system providing it. For cloud services we redefine the notion of "*test case passed*" to "*a test case passed only when it passed in all possible applicable runtime configuration states*". Testing a service against all its applicable runtime configuration states is necessary, however, it is very costly. Let us consider a service composed of only one service instance provided/protected by a configured instance where the components can be on any one of ten dedicated nodes (not shared with any other configured instance). For a normal functional test case, one will have to run this test case ten times to test this service against all its applicable runtime configuration states (once per node). In the case of a stress test case, and if we assume the maximum size of the configured instance is four components, the test case will need to be executed 210 times (C_{10}^{4}times, without considering the scaling steps). These numbers increase with the number of service instances. Thus, covering all the runtime configuration states may be impossible for complex and large systems such as cloud systems.

4.1.7.1. Boundary environments

To tackle the problem, we propose testing against a representative set of runtime configuration states. A runtime configuration state is described via the environments in which each service instance is provided; as a result, identifying the representative set of runtime configuration states consists of identifying the environments that describe the runtime configuration states in this set. Because any environment has two elements, i.e. location and collocation, such environments can be derived in two mutually non-exclusive ways:

- Collocation-wise: for a given set of locations on which a configured instance is deployed, one can identify the environments with the largest collocations per location which we call boundary environments. In other words, the collocation set of an environment that has that same location is a subset of the collocation set of the boundary environment. Two environments are said to have the same boundary environment if they have the same maximum collocation and equivalent locations, i.e. same network, and hosts of identical specifications.
- Location-wise: for a given set of collocations of components of a given configured instance and a maximum number of N components, one can identify various

assignments of N collocations to N locations as allowed by the configuration. Such assignments are what we call mixtures of environments. Such assignments may or may not allow the reuse of collocations as the configuration allows. Two mixtures are said to have equivalent assignments if their assignments involve the same set of collocations in equivalent locations with the same numbers of occurrences of each collocation per location class.

By identifying the boundary environments in a system one can group the nodes of the system into groups that have the same boundary environment. Similarly, by identifying mixtures of environments one can group them into mixtures that involve the same set of collocations with the same number of occurrences of each collocation per location class. Our main idea is to use such groupings (location-wise and collocation-wise) to group runtime configuration states into equivalence classes taking into consideration the environments they involve. In other words, test runs should cover runtime configuration states that involve: 1) boundary environments, and 2) mixtures that were derived from collocations of boundary environments and that involve as many boundary environments as possible. The rationale behind this method is based on the following assumptions:

- Boundary environments present the worst case of resource sharing under which one can put the component under test; therefore, if a property holds under the boundary environment it will hold under all its sub-environments.

- Boundary environments allow for grouping nodes into equivalence classes. As a result, one node that replicates the boundary environment is considered representative of all the equivalent nodes that can host that boundary environment. This assumption enables us to reduce the number of paths, and as a result, the number of runs the test case should go through.

4.1.7.2. Coverage criteria

Using the boundary environments, one can define the set of paths that should be exercised by the test case. Such a set depends also on the nature of the test case itself. A functional test case will only need to target the boundary environments of the configured instances. However, for a stress test, for instance, one needs to use mixtures of boundary environments as well as check how the service behaves when these are chained with various mixtures of the other configured instances involved in the test case. Figure 4-8 shows an example in which a service composed of two service instances that are provided (and protected) by two configured instances (ConfiguredInstance1 and ConfiguredInstance2) undergoing a stress test. Figure 4-8 (a) shows that ConfiguredInstance1 can have up to three components and ConfiguredInstance2 can have up to two components. If we assume all nodes are identical and are in the same network, ConfiguredIntsnace1 components may serve under environments with any of four collocations (for instance, collocation: comps of {}). The same applies to ConfiguredInstance2. Furthermore, ConfigureInstance1 has three different collocations for the boundary environments, namely one shared with only ConfiguredInstance2 (for example on N1), one shared with only ConfiguredInstance3 (for example on N4), and one shared with both (on N3). Similarly, ConfiguredInstance2 has two collocations for the boundary environments, one shared with only ConfiguredInstance1 and one shared with both ConfiguredInstance1 and ConfiguredInstance3. Figure 4-8 (b) to (g) capture some mixtures under which the service must be tested (as it may be provided under these mixtures).

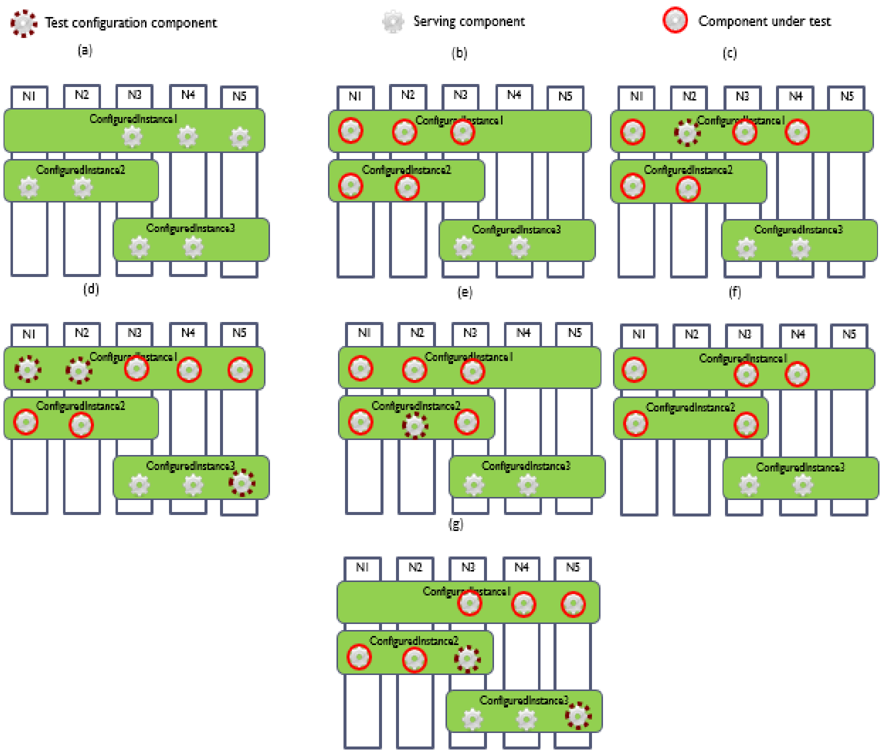


Figure 4-8. Some mixtures and their combinations against which a service provided by ConfiguredInstance1 and ConfiguredInstance2 should be tested

One can use various coverage criteria of the boundary environments as well as their mixtures in order to define the paths a test case must exercise. Among these coverage criteria, we believe the following are the most relevant. They are ordered in the descending order of their respective error detection power:

- All boundary environments mixtures (resp. boundary environments) paths: in this coverage one identifies first all possible mixtures of boundary environments (resp. boundary environments), then tests on all the paths that chain the mixtures (resp. boundary environments) of the configured instances being tested.
- Pairwise boundary environments mixtures (resp. boundary environments): in this coverage one identifies all possible mixtures of boundary environments (resp. boundary environments), then generates a set of paths such as each pair of identified mixtures (resp. boundary environments) is in at least one path. To do so, one can generate a covering array of strength two for the identified mixtures (resp. boundary environments) considering each configured instance as a factor and each mixture (resp. boundary environments) of a configured instance as a level of the factor representing that configured instance.
- All boundary environments mixtures (resp. boundary environments): in this coverage, one aims at testing a set of paths in which each mixture of boundary environments (resp. boundary environments) is used at least once.

Based on the assumptions we made, one can use any of these criteria to reduce the number of runs of a test case while maintaining an acceptable level of error detection power. To run a functional test case against the service composed of the service instances provided by the configured instances in Figure 4-8, one needs twenty runs for that test case to cover all runtime configuration states (without varying the location, taking into consideration the location one will end up with ninety six runs), as compared to six runs using all boundary environments paths and pairwise boundary environments coverage criteria, or three runs for all boundary environments criterion. A similar reduction in the number of runs can be achieved for test cases that may involve environments mixtures such as stress tests.

Figure 4-8 captures the case of a stress test of a service that is composed of two service instances. The first service instance is provided by ConfiguredInstance1 while the second service instance is provided by ConfiguredInstance2. Assuming all the nodes are identical (in terms of specifications), the factor of location then can be dismissed while identifying the boundary environments, and one can focus on the collocations only. As a result, we identify the set of boundary environments for ConfiguredInstance1 as {BE1.1={location: {N1, N2}, collocation: {components of ConfiguredInstance2}}, BE1.2={location: {N3}, collocation: {components of ConfiguredInstance2, components of ConfiguredInstance3}}, BE1.3={location: {N4, N5}, collocation: {components of ConfiguredInstance3}}}. Similarly, the boundary environments for ConfiguredInstance2 can be identified as the set {BE2.1={location: {N1, N2}, collocation: {components of ConfiguredInstance1}}, BE2.2={location: {N3}, collocation: {components of ConfiguredInstance1, components of ConfiguredInstance3}}}. The example in Figure 4-8 also assumes that ConfiguredInstance1 is to be tested with three components while ConfiguredInstance2 is to be tested with two components (the mixture widths). Considering this assumption and the possible location for each boundary environment, we can identify the mixtures under which the testing must take place. These mixtures are identified by associating with each boundary environment a number of components under test to be run under it keeping in mind that the total number of components under test must be equal to the mixture width. As a result, the set of mixtures that corresponds to ConfiguredInstance1 is {Mx1.1 = {BE1.1: 1, BE1.2: 1, BE1.3: 1}, Mx1.2 = {BE1.1: 1, BE1.3: 2}, Mx1.3 = {BE1.1: 2, BE1.3: 1}, Mx1.4 = {BE1.1: 2, BE1.2: 1}, Mx1.5 = {BE1.2: 1, BE1.3: 2}}. Similarly, the set of mixtures for ConfiguredInstance2 is {Mx2.1 = {BE2.1:2}, Mx2.2 = {BE2.1: 1, BE2.2: 1}}. Figure 4-8 shows testing under test configurations that make use of these mixtures. Figure 4-8 (b), for instance, illustrates how the test should be run under the test configuration that combines Mx1.4 for ConfiguredInstance1 and Mx2.1 for ConfiguredInstance2.

4.1.7.3. Revisiting the test methods

The test methods as described previously run a test case against all possible paths while reducing the impact of testing activities on the production traffic. We have shown that this can be time-consuming and may induce some intolerable service disruption due to service relocation for some methods. We proposed the concept of boundary environments along with the coverage criteria to reduce the number of paths under which a test case is to be run. Using these concepts, the proposed test methods can be revisited to run the test cases only on paths generated given an environments-coverage criterion. Such enhancement help reduce the disruption induced by live testing on the one hand by reducing the number of runs of the test case regardless of which test method is used for isolation, which reduces the time needed for testing. On the other hand, by reducing the number of service relocations when the small flip or rolling paths is used which reduces the service disruption.

Other test methods can be enhanced as well. Canary releases can be enhanced by placing the components that expose the new version of the feature under test on nodes that represent different boundary environments. Furthermore, as the rollout is progressing, the placement should aim to cover relevant mixtures of boundary environments as new users are being redirected to the new version. Had such an approach been used, the problem in [63] could have been detected earlier before retiring the older version and the damage could have been contained. The same applies to the gradual rollout method.

4.1.8. PoC and case study

In this section, we will use a case study and a setup to showcase the feasibility of our test methods and the difference they could make when used in production.

The setup we use is a Kubernetes cluster composed of one master node and two worker nodes. Kubernetes is a container orchestration and management middleware, and the container runtime we are using along with Kubernetes in this setup is Containerd [53]. Our test methods rely on the ability of the environment to snapshot and clone components of configured instances. CRIU is the tool that is commonly used to provide this functionality for container virtualization. Although the releases of Kubernetes do not offer integration with CRIU [71], we have used a modified version of Kubernetes [90] that supports snapshotting and cloning of

Pods using CRIU. It also supports Pod live migration which is done through a series of snapshot and clone operations.

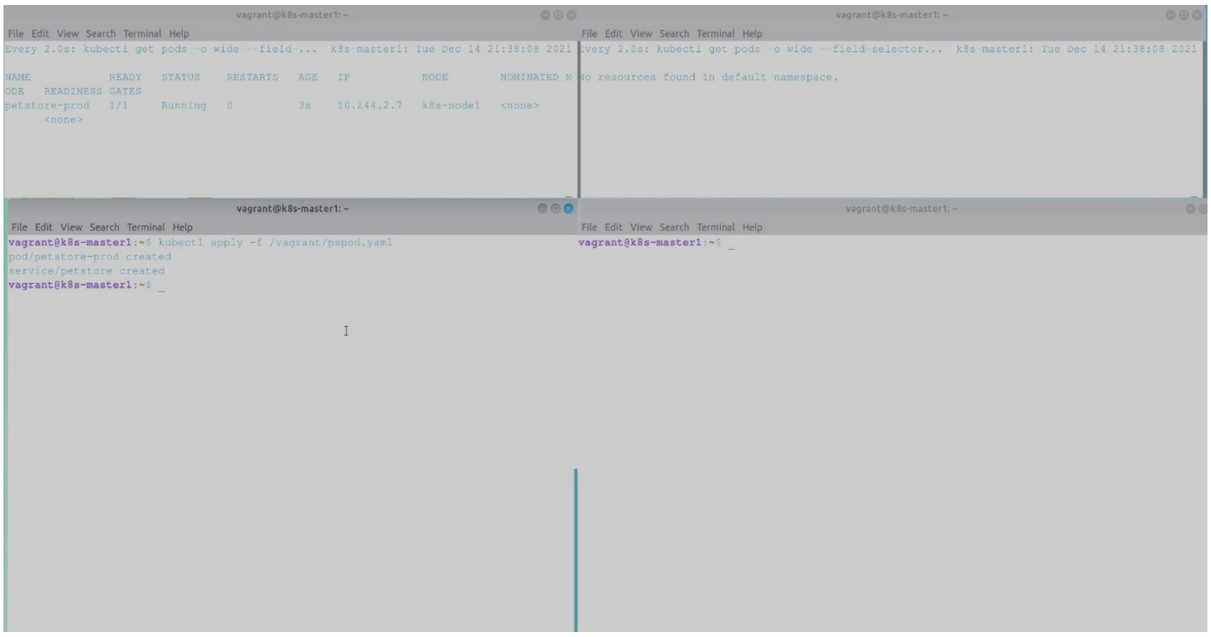
The application we have used for this case study is the pet store application. Pet store exposes a REST API that allows the management of an inventory of pets, management of users, and allows users to make orders to purchase pets. We deployed the pet store application in the Kubernetes cluster. The pet store application is deployed on both workers and should have one running pod at all times. In other words, the pod that provides the pet store application can be running on either one of the worker nodes in our cluster.

We showcase the rolling paths test method to show that it allows testing the pet store application without test interferences. To achieve this goal, we use the following three scenarios:

- Scenario 0: this scenario is the baseline, we will use it to watch the normal behavior of the application in the absence of test traffic. In this scenario, we will use a production traffic that consists of one POST request to add a new pet to the inventory, followed by 30 GET requests (a request every four seconds) that fetch the data of that newly added pet.
- Scenario 1: in this scenario, we will simulate testing the pet store application in production without implementing any test isolation. We will use the production traffic used in Scenario 0, and our test traffic will be a single test case that deletes the newly added pet. Such a scenario is very common in practice as in some cases service

providers need to use tenant data and tenant configuration in order to troubleshoot the problems reported by the users.

- Scenario 2: in this scenario, we will simulate testing the pet store application in production using the rolling paths test method. The test traffic and the production traffic used in this scenario are the same as in scenario 1.



```
Every 2.0s: kubectl get pods -o wide --field-...  k8s-master1: Tue Dec 14 21:38:08 2021

NAME           READY   STATUS    RESTARTS   AGE   IP           NODE        NOMINATED N
ODE   READINESS GATES
petstore-prod  1/1     Running   0          3s    10.244.2.7   k8s-node1   <none>
      <none>
```

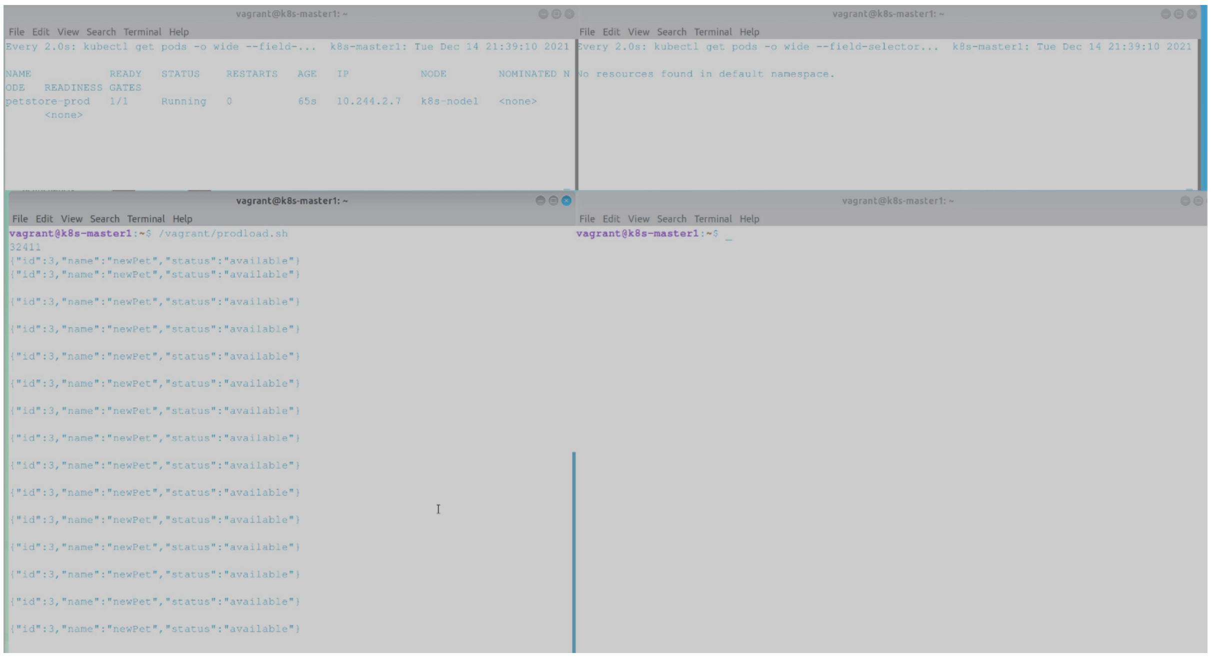
```
Every 2.0s: kubectl get pods -o wide --field-selector...  k8s-master1: Tue Dec 14 21:38:08 2021

No resources found in default namespace.
```

```
vagrant@k8s-master1:~$ kubectl apply -f /vagrant/pspod.yaml
pod/petstore-prod created
service/petstore created
vagrant@k8s-master1:~$ _
```

```
vagrant@k8s-master1:~$ _
```

Figure 4-9. Pods locations in the initial state of the cluster

Figure 4-9 was used to track the state of our cluster. The terminal in the top left corner tracks the pods running on the first worker node. The terminal in the top right corner tracks the pods running in the second worker. We use the bottom left corner for tracking the responses to production traffic. The figure also shows the initial state of the cluster as the pet store application has one pod running on the first worker of our Kubernetes cluster.



```
Every 2.0s: kubectl get pods -o wide --field-...  k8s-master1: Tue Dec 14 21:39:10 2021

NAME            READY   STATUS    RESTARTS   AGE   IP           NODE        NOMINATED N
ODE   READINESS GATES
petstore-prod   1/1     Running   0          65s   10.244.2.7   k8s-node1   <none>
      <none>
```

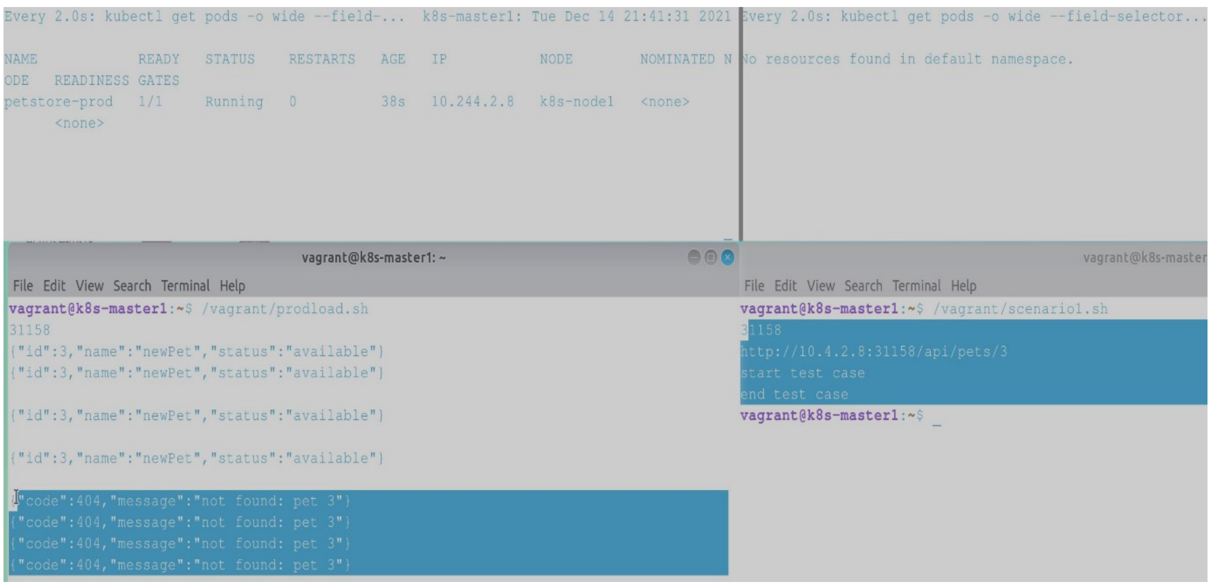
```
Every 2.0s: kubectl get pods -o wide --field-selector...  k8s-master1: Tue Dec 14 21:39:10 2021

No resources found in default namespace.
```

```
vagrant@k8s-master1:~$ /vagrant/prodload.sh
32411
{"id":3,"name":"newPet","status":"available"}
{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}
```

```
vagrant@k8s-master1:~$ _
```

Figure 4-10. State of the cluster and responses to production traffic in scenario 0

Figure 4-10 captures the output for scenario 0. On the one hand, we notice that the response to the GET requests always returns the information of the pet. On the other hand, we notice that the pet store application has its pod always running in the same worker node.



```
Every 2.0s: kubectl get pods -o wide --field-...  k8s-master1: Tue Dec 14 21:41:31 2021

NAME           READY   STATUS    RESTARTS   AGE   IP           NODE        NOMINATED N
ODE   READINESS GATES
petstore-prod  1/1     Running   0          38s   10.244.2.8   k8s-node1   <none>
      <none>
```

```
Every 2.0s: kubectl get pods -o wide --field-selector...

No resources found in default namespace.
```

vagrant@k8s-master1: ~

File Edit View Search Terminal Help

```
vagrant@k8s-master1:~$ /vagrant/prodload.sh
31158
{"id":3,"name":"newPet","status":"available"}
{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"code":404,"message":"not found: pet 3"}
{"code":404,"message":"not found: pet 3"}
{"code":404,"message":"not found: pet 3"}
{"code":404,"message":"not found: pet 3"}
```

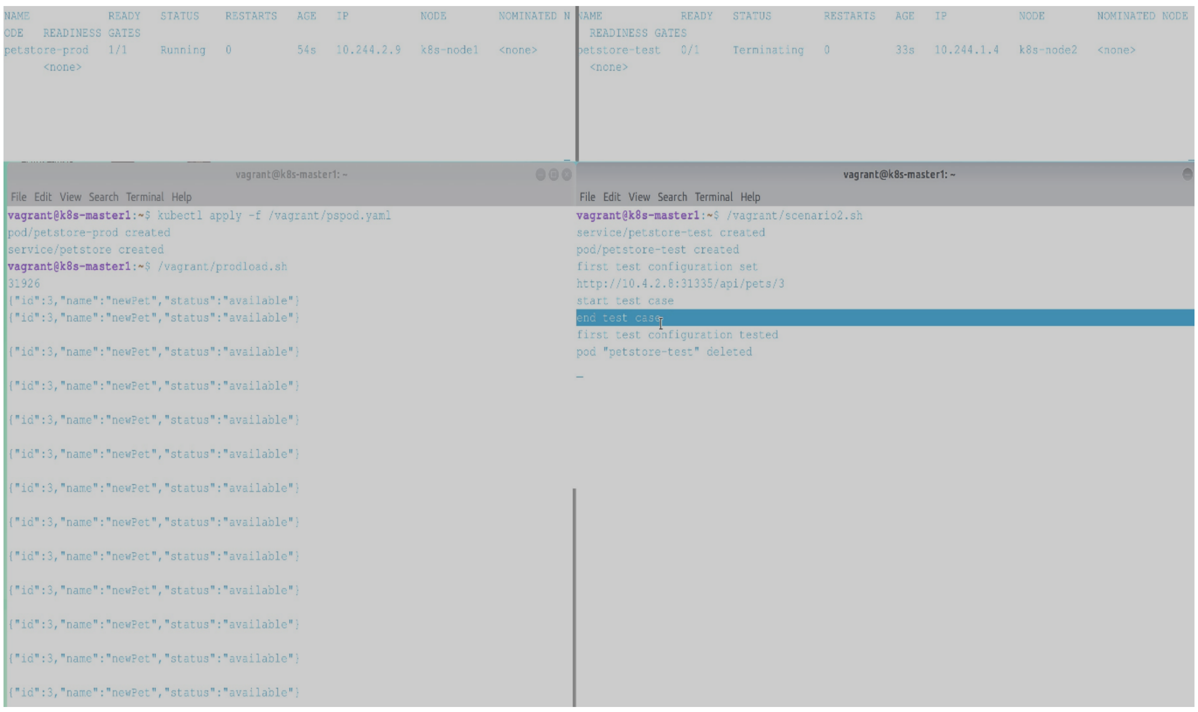
vagrant@k8s-master

File Edit View Search Terminal Help

```
vagrant@k8s-master1:~$ /vagrant/scenario1.sh
31158
http://10.4.2.8:31158/api/pets/3
start test case
end test case
vagrant@k8s-master1:~$ _
```

Figure 4-11. State of the cluster and responses to production traffic in scenario 1

Figure 4-11 captures the output for scenario 1. We notice that the new pet was added successfully, and the GET requests were able to return the information of the pet during the first few requests. However, once the test case was executed, we notice that the GET requests started returning the "pet not found" error. This deviation from the output of scenario 0 is what constitutes test interferences.



```
NAME            READY   STATUS     RESTARTS   AGE   IP           NODE        NOMINATED N
ODE   READINESS GATES
petstore-prod   1/1     Running    0          54s   10.244.2.9   k8s-node1   <none>
      <none>
```

```
NAME            READY   STATUS        RESTARTS   AGE   IP           NODE        NOMINATED NODE
  READINESS GATES
petstore-test   0/1     Terminating   0          33s   10.244.1.4   k8s-node2   <none>
  <none>
```

vagrant@k8s-master1: ~

File Edit View Search Terminal Help

```
vagrant@k8s-master1:~$ kubectl apply -f /vagrant/pspod.yaml
pod/petstore-prod created
service/petstore created
vagrant@k8s-master1:~$ /vagrant/prodload.sh
31926
{"id":3,"name":"newPet","status":"available"}
{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}
```

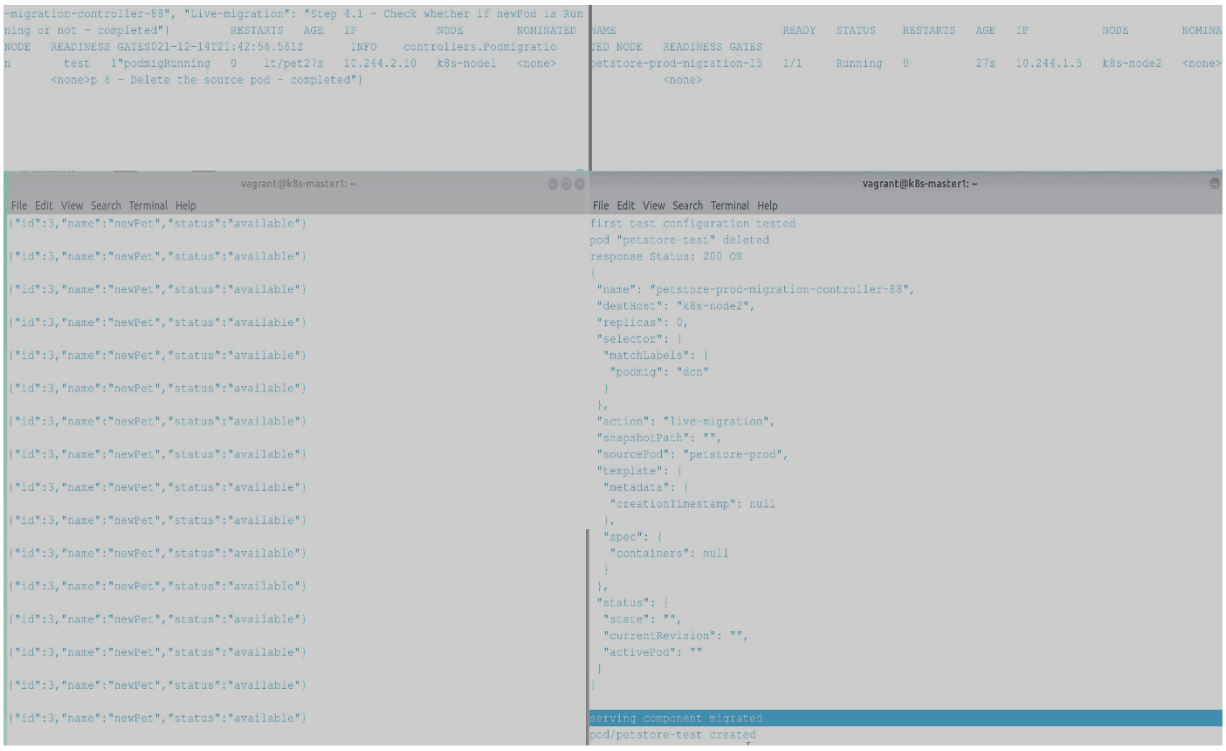
vagrant@k8s-master1: ~

File Edit View Search Terminal Help

```
vagrant@k8s-master1:~$ /vagrant/scenario2.sh
service/petstore-test created
pod/petstore-test created
first test configuration set
http://10.4.2.8:31335/api/pets/3
start test case
end test case
first test configuration tested
pod "petstore-test" deleted

_
```

Figure 4-12. State of the cluster and responses to production traffic in scenario 2 when testing the first test configuration

Figure 4-12 captures the output for scenario 2. We notice that initially the pet store application pod is running on the first worker node. Once the testing is started, a pod under test was created on the second worker node which illustrates testing under the first test configuration. Once the test case was done, the pod under test was removed, the pet store application pod was migrated to the second worker, and a pod under test was created on the first worker node to test under the second test configuration as one can tell from the differences between Figure 4-12 and Figure 4-13. From these two figures, we also notice that the GET requests were successful during the whole test session and the output matches the baseline as opposed to scenario 1. This shows that the rolling paths test method helped avoid test interferences during this test session.



```
-migration-controller-88", "Live-migration": "Step 4.1 - Check whether if newPod is Run
ning or not - completed"}        RESTARTS   AGE   IP           NODE        NOMINATED
NODE   READINESS GATES021-12-14T21:42:56.561Z       INFO    controllers.Podmigratio
n        test   1"podmigRunning   0    lt/pet27s   10.244.2.10   k8s-nodel   <none>
       <none>p 6 - Delete the source pod - completed"}
```

```
NAME                          READY   STATUS    RESTARTS   AGE   IP           NODE        NOMINA
TED NODE   READINESS GATES
petstore-prod-migration-15   1/1     Running   0          27s   10.244.1.5   k8s-node2   <none>
           <none>
```

```
{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}

{"id":3,"name":"newPet","status":"available"}
```

```
first test configuration tested
pod "petstore-test" deleted
response Status: 200 OK
{
 "name": "petstore-prod-migration-controller-88",
 "destHost": "k8s-node2",
 "replicas": 0,
 "selector": {
  "matchLabels": {
   "podmig": "dcn"
  }
 },
 "action": "live-migration",
 "snapshotPath": "",
 "sourcePod": "petstore-prod",
 "template": {
  "metadata": {
   "creationTimestamp": null
  },
  "spec": {
   "containers": null
  }
 },
 "status": {
  "state": "",
  "currentRevision": "",
  "activePod": ""
 }
}

serving component migrated
pod/petstore-test created
```

Figure 4-13. State of the cluster and responses to production traffic in scenario 2 after production pod migration and testing under the second test configuration

4.2. Execution semantics for UTP and its validation

In this section, we describe our approach to achieve **Research Objective #5**, 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.

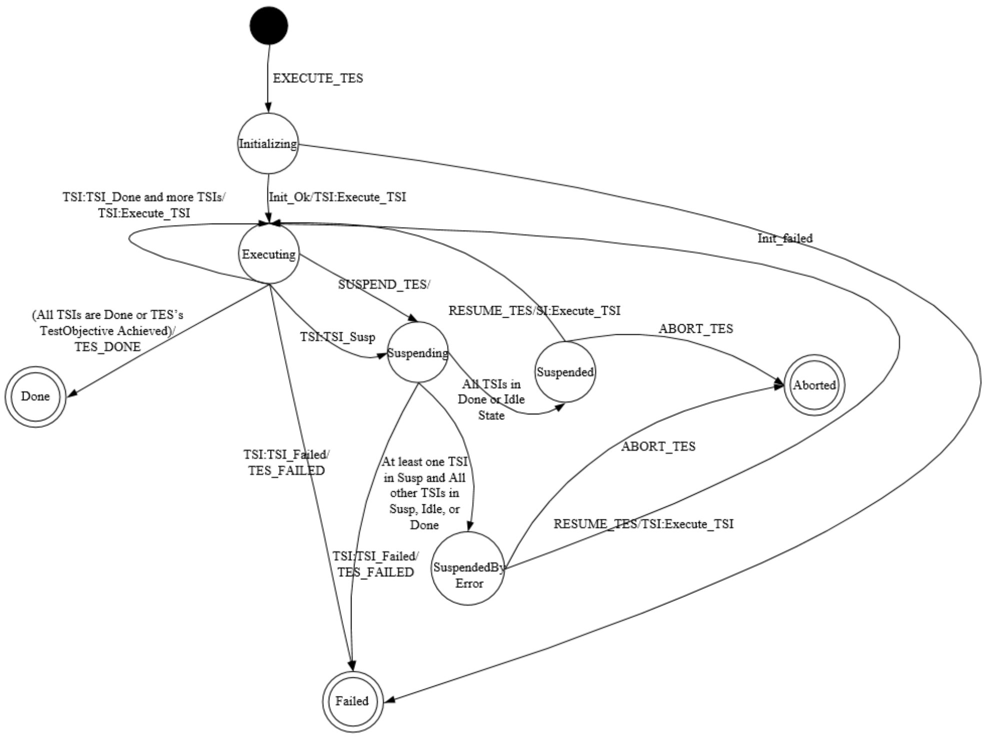

EXECUTE_TES
Initializing
Init_Ok/TSI:Execute_TSI
Init_failed
TSI:TSI_Done and more TSIs/ TSI:Execute_TSI
Executing
SUSPEND_TES/
RESUME_TES/TSI:Execute_TSI
(All TSIs are Done or TES's TestObjective Achieved)/ TES_DONE
TSI:TSI_Susp
Suspending
Suspended
ABORT_TES
Aborted
Done
All TSIs in Done or Idle State
TSI:TSI_Failed/ TES_FAILED
At least one TSI in Susp and All other TSIs in Susp, Idle, or Done
ABORT_TES
RESUME_TES/TSI:Execute_TSI
TSI:TSI_Failed/ TES_FAILED
SuspendedBy Error
Failed

Figure 4-14 Execution semantics of the TestExecutionSchedule

4.2.1. Execution semantics

The execution semantics we propose serves several purposes such as:

- 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 semantic is the TestExecutionSchedule which is also a runtime object that is used by the TEF to track and control a test session. It is composed of a set of partially ordered TSIs runs. A TSI run is modeled 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 can receive four administrative operations: EXECUTE, SUSPEND, RESUME, and ABORT (Figure 4-14.).

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

- sequentially,
- using a CompoundProceduralElement which allows for UTP 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 UTP 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 TestProcedure running or multiple

TestProcedures 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 UTP TestCase are decided based on the construct it belongs to and the target(s) of the control flow link(s) that have the completed UTP TestCase as the source.

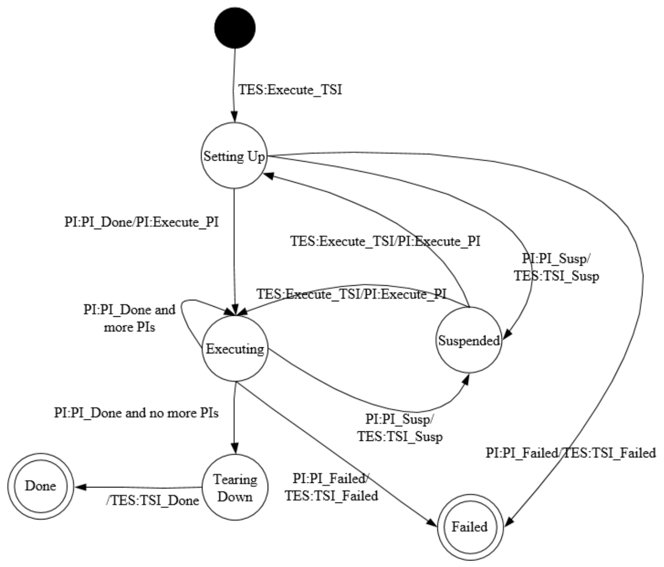


Figure 4-15 Execution Semantics of UTP TestCase

While the TestExecutionSchedule is in the Executing state, the TEF keeps invoking UTP TestCases using the Execute_TSI message. From the initial Idle state, the invoked UTP TestCase goes to the Executing state (Figure 4-15) via a Setting Up state and completes after a Tearing Down state. In all these states the UTP TestCase invokes different procedures composing the UTP TestCase using the Execute_PI message. According to the UTP TestCase state these procedure invocations (Figure 4-16 And Figure 4-17) can be setting up/tearing down a test configuration and/or test isolation countermeasures, or running a test case or a test design technique. In the latter case, the invocation is for a TestDesignTechnique with the execution semantics shown in Figure 4-17; in the other cases the procedure invoked follows the execution semantics shown in Figure 4-16, 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, and 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, and goes to the Failed state. In any case, the procedure notifies the UTP TestCase about the result, which then proceeds depending on the procedure's state. If the procedure's state is

- Done: the UTP TestCase proceeds to the next invocation(s);
- Suspended: the UTP TestCase goes to the Suspended state, and therefore the TestExecutionSchedule also goes to the SuspendedByError state;
- Failed: the UTP 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 UTP 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 UTP Testcase that is in Executing state goes to the Failed state while the TestExecutionSchedule is in the Suspending or the Executing state, the TestExecutionSchedule

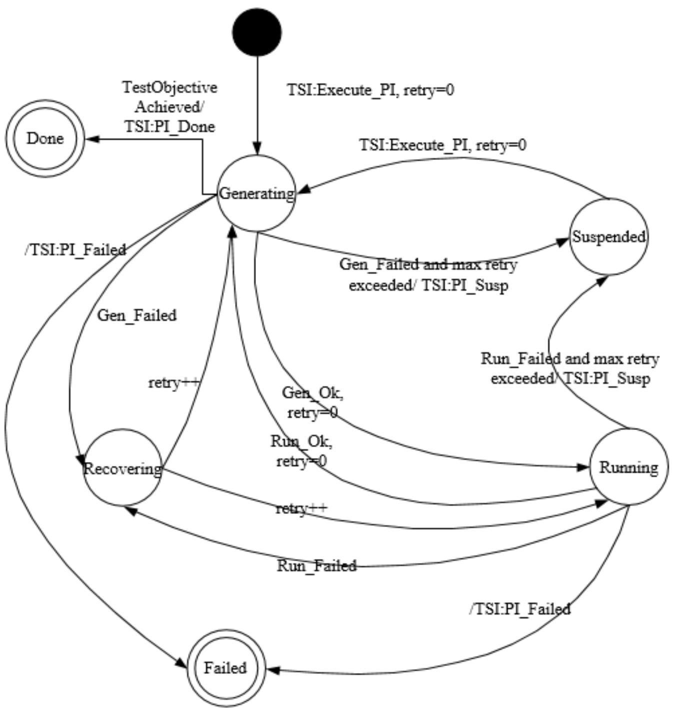


Figure 4-16 Execution semantics of the TestDesignTechnique invocation

goes to the Failed State and the whole test session will be deemed as failed.

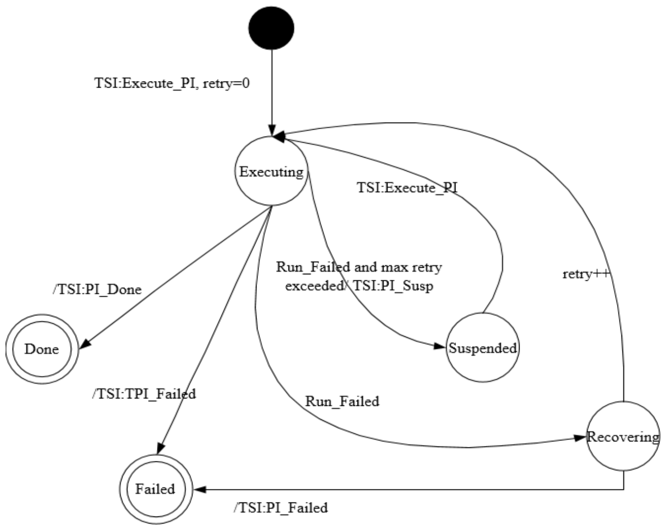


Figure 4-17. Execution semantics of ProcedureInvocation and TestProcedure Invocation

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 UTP TestCases. As described from the Suspending state the TestExecutionSchedule may go to the Suspended state, to the SuspendedByError state, or the Failed state depending on the results of the currently running UTP 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.

4.2.2. Validation

In this subsection, we aim to prove the soundness of our execution semantics. In other words, we will show that if two test plans are equivalent, the behavior that TEF will exhibit for both of them will be similar.

Definition 4-1: a test run is a triplet $(Tproc, TestConf, index)$ such that $Tproc$ is a TSI to be invoked, $TestConf$ is a test configuration under which the TSI is to be invoked, and $index$ is an integer indicating the number of occurrence of the pair $(Tproc, TestConf)$ in the test plan up until this test run.

Definition 4-1 allows us to capture all aspects related to a test run. In fact, the TSI along with the test configuration allow us to capture the test goal being achieved by the test run. The index of the test run allows us to capture the confidence level in the verdict obtained. The confidence in a verdict is approximated as the ratio between the number of times a verdict was obtained over the number of times the test run was executed. As a result, the more times a test run is executed the more confidence we can establish in the verdict obtained.

Definition 4-2: a test plan is said to be valid if its corresponding UTP model satisfies the following conditions:

- Between two sets of model elements that capture the deployment of two different test configurations, there should be a set of model elements that capture the tear down of the first test configuration.

- Between the set of model elements that capture the deployment of a test configuration and the set of model elements that capture its tear down, the TestProcedure model elements invoke only TSIs that hit sub-paths of the call path of that test configuration.
- Between the set of model elements that capture the deployment of a test configuration and the set of model elements that capture its tear down, at least one TestProcedure model element invokes a TSI that hits the call path of that test configuration.
- Invocation of TSIs respects the precedence relationships between TSIs.

Definition 4-2 specifies what makes a test plan valid. The first condition ensures that test configurations for test runs, executed in sequence, are not compromised as it mandates that the test configuration of a test run can only be deployed after the tearing down of the previous test configuration if any. The second condition ensures that TSIs are only invoked under test configurations that are setup along their corresponding call path. The third condition ensures that extra components under test that are not needed are not setup during the execution of a valid test plan.

Definition 4-3: two test runs are said to be equivalent if they have the same index, and invoke the same TSI under test configurations that have the same settings for the components under test in the call path that is exercised by the TSI.

Taking into consideration the rationale for Definition 4-1, we defined the equivalence of test runs in terms of the goal they achieve. As a result, equivalent test runs according to Definition 4-3 are test runs that achieve identical test goals with the same level of confidence.

Definition 4-4: two valid test plans $TP1$ and $TP2$ are said to be equivalent if and only if:

- For each test run in $TP1$ there is an equivalent test run in $TP2$, and
- The partial order of test runs in $TP1$ is the same partial order of their corresponding equivalent test runs in $TP2$.

From the above definitions, we can see that valid test plans are equivalent when they achieve the same set of test goals with the same corresponding confidence levels. The processing of test plans by TEF consists of exhibiting for each encountered model element the corresponding behavior as specified by the execution semantics. As a result, we can have the following definitions as well.

Definition 4-5: for a set S of model elements in the test plan UTP model, $System(S)$ is the state model obtained by associating with each model element in S its corresponding state model as specified in the execution semantics and:

- Perform concatenation of state models when model elements are put in sequence in S.
- Perform parallel composition of state models when model elements are put in a ParallelFragment in S, or they are nested.

Definition 4-6: for a given test plan TP, $System(TP)$ is the state model obtained by applying Definition 4-5 to the UTP model corresponding to the test plan TP.

From these definitions, we can then define the soundness of the mapping we have established between UTP model elements and their corresponding behaviors in the execution semantics.

Definition 4-7, *Soundness:* our mapping is sound if and only if: for any two equivalent test plans $TP1$ and $TP2$, $System(TP1)$ and $System(TP2)$ are weakly bisimilar.

Weak bisimilarity is a reasonable equivalence relation to use as opposed to strong bisimilarity. In fact, the equivalence between test plans was defined from a testing perspective only and not from the perspective of the disturbance their executions may induce or the amount of resources they may require. Therefore, the state models of equivalent test plans may have different numbers of occurrences of some actions (which justifies the difference in the disturbance for instance). Such differences stem from the fact that although the equivalent test plans achieve the same test goals with the same corresponding levels of confidence, the design decisions may have been made differently when devising each test plan. Therefore, the equivalence in behaviors exhibited by TEF when executing each test plan may prevail only when these differences in the set of actions are silenced or hidden and considered as internal actions. Such silencing can only be used with weak bisimilarity.

To prove weak bisimilarity, first, we need to identify the actions that we are going to silence. Therefore, the actions to be silenced are the following:

- The actions inside the state models of ProcedureInvocation, TestProcedure invocation, and TestDesignTechnique invocation state models. The rationale behind this is that these actions are only used to ensure the successful execution of the invocation and properly react to failure if any. The only action that will not be silenced in these models is TSI:Execute_PI as it represents the triggering of the invocation and an interaction with the state model responsible for this triggering.
- All the actions in the UTP TestCase state model as these actions only serve for a medium of interaction between the TestExecutionSchedule and the invoked procedures on the one hand. On the other hand, they serve to scope/contain the failed executions, i.e. failures manifest only at the level of the state model associated with the UTP TestCase related to the failed invocation and not all the invocations in the test plan.
- If the test plan contains a set of consecutive test runs with test configurations that correspond to the same call path, silence all actions in the state models related to test configuration deployment and test configuration removal except for the actions of the state models related to the deployment of the first test configuration that is deployed and the state models related to the removal of the last test configuration to be removed.
- If the two equivalent test plans contain each a sequence of test runs such that:
    - For each test run in the sequence of the first test plan there is an equivalent test run in the sequence of test runs in the other test plan, and

- For each sequence of the two sequences, there is a test configuration used in a test run in that sequence such that all the test configurations used in the test runs in that sequence are subconfigurations of that test configuration.
- Then the actions to be silenced are the following:
    - all actions in the state models related to test configuration deployment and test configuration removal except for the actions of the state models related to the deployment of the first test configuration that is deployed and the state models related to the removal of the last test configuration to be removed, and
    - Actions not in common between the state models related to the deployment of the first test configurations in the two sequences of test runs, and actions not in common between the state models related to the removal of the last test configurations in the two sequences of test runs.

We have silenced the last two groups of actions as they do not contribute to the achievement of the next goal. In fact, they constitute extra actions for early preparation to achieve future test goals, but they are irrelevant as such at those stages of the execution of the test plan. As a result, they can be considered as internal actions at that stage.

We will demonstrate the soundness of our mapping from the UTP model elements to the execution semantics using the bisimulation game. Let $System(TP1)$ and $System(TP2)$

be the systems associated with the equivalent test plans. A bisimulation game consists of two players, attacker and defender, and goes as follows:

- The game starts from a pair of states $(s1, t1)$ such that $s1$ is a state in $System(TP1)$ and $t1$ is a state in $System(TP2)$. Such a pair is referred to as the current configuration.
- The game goes iteratively, in each iteration the attacker chooses a model, $System(TP1)$ or $System(TP2)$ and an action to take in that model. The defender is then left to choose the other model, the System that was not chosen by the attacker, and must take the same action in that model. Once both players have taken their actions, the current configuration becomes $(s2, t2)$, representing the states where those actions can take. A silent action from the attacker does not count as an action since the defender can respond to that action by just taking no action and remaining in the same state.
- The game stops when one of the players does not have actions to take, or when it is obvious that the game will go infinitely.
- The attacker wins when the defender cannot take an action, the defender wins when the game is deemed to go infinitely or when the attacker cannot take any more actions.
- If the attacker has a universal winning strategy, the two systems are not weakly bisimilar. If the defender has a universal winning strategy the two systems are weakly bisimilar.

Lemma 4-1: let $System(TP1)$ and $System(TP2)$ be the systems associated with two equivalent test plans. In a bisimulation game, if the current configuration represents states in which a TestProcedure or a TestDesignTechnique is being invoked, the defender always has an action to take.

Proof: let (s, t) be the current configuration. Since from state s and from state t a TestProcedure or a TestDesignTechnique is being invoked, the game can go one of the paths below:

- The attacker follows the action taken when the invocation fails. In this case, the defender can do the same as both systems will go to the Suspending or SuspendedByError states. Actions for failure management are the same since they are defined at the level of the TestExecutionSchedule, as a result, they do not pose a problem.
- The attacker follows the action taken when the invocation succeeds, in this case, we have the following subcases:
    - The next action is an invocation of another TestProcedure or TestDesignTechnique. In this case, the defender can take the same action. If the defender does not have that option that means that the test plans are not equivalent since after the current test run the two test plans invoke two nonequivalent test runs (because they invoke different TSIs, i.e. not the same partial order of test runs).
    - The next action cannot be an action related to a setup of a test configuration, otherwise, the test plan will not be valid. In fact, before the invocation of TSIs

actions for the deployment of test configurations are triggered, as a result, the removal of test configurations should be triggered before any more test configurations deployment could be done.

- The next action is an invocation of a test configuration removal action. The defender cannot have as a next action a TestProcedure or TestDesignTechnique invocation action, otherwise, this will constitute an extra run, and then the test plans will not be equivalent. As a result, the next action for the defender will be a test configuration removal action. This test configuration removal action will be the same as the one taken by the attacker. Otherwise, this will mean that before this test configuration removal action, the two test plans have actions that setup two test configurations that have different settings for at least one component under test along the call path of one of the TSIs. Therefore, the defender will not have this action only when the test plans are not equivalent.

Lemma 4-2: let $System(TP1)$ and $System(TP2)$ be the systems associated with two equivalent test plans. In a bisimulation game, if the current configuration represents states in which a test configuration deployment (resp. removal) action is being invoked, the defender always has an action to take.

Proof: let (s, t) be the current configuration. Since from state s and from state t a test configuration deployment (resp. removal) action is being invoked, the game can go one of the paths below:

- The next action taken by the attacker is the same as the action taken to get to the current configuration, i.e. the game got to (s, t) by a test configuration deployment (resp. removal) action and the next action is also a test configuration deployment action (resp. removal). In this case, the defender should be able to pick the same action as well, otherwise, the actions in the two systems are deploying test configurations along two different call paths, and therefore the test runs they are associated with are non-equivalent.
- The next action taken by the attacker is not the same as the action taken to get to the current configuration, in this case, we have two subcases:
    - A test configuration deployment action cannot be followed by a test configuration removal action (otherwise the test plan is not valid). Therefore, the attacker would pick a TestProcedure or TestDesignTechnique invocation action. If the defender does not have the possibility of doing the same action, then the test plans do not have the same partial order, or their test runs are not equivalent.
    - A test configuration removal action cannot be followed by a TestProcedure or TestDesignTechnique invocation action (otherwise the test plan is not valid). Therefore, the action that will be picked by the attacker is a test configuration deployment action. If the defender does not have that option, this means that the next test configurations to be deployed do not have the same setting for

components under test along the call path. Therefore, the next test runs in both test plans are not equivalent.

- For the case of following the path when the test configuration deployment action fails, we use the same argument as in Lemma 4-1.

To prove the soundness of our mapping we need to prove that if two test plans TP1 and TP2 are equivalent, then $System(TP1)$ and $System(TP2)$ are weakly bisimilar. To do that we will use the bisimulation game and start from $(s0, t0)$ as a current configuration, such that $s0$ is the initial state of $System(TP1)$ and $t0$ is the initial state of $System(TP2)$:

- From $(s0, t0)$ the attacker can choose $System(TP1)$ or $System(TP2)$. And the only action it has is EXECUTE_TES. The same action will be available for the defender as well.
- After EXECUTE_TES the current configuration is (Initializing, Initializing). From this configuration, the attacker can only choose Int_Ok action, and then it will go to the state {Executing, Setting up, Executing} which represents the first action taken to deploy the first test configuration. The same action will be possible for the defender to take as well, and it will take the other System to a {Executing, Setting up, Executing} state.
- The current configuration now is a configuration to which Lemma 4-2 applies. Moreover, given the analysis in the proofs of Lemma 4-1 and Lemma 4-2, taking the actions from this current configuration will either take to a current configuration to which Lemma 4-1 applies, or to a current configuration to which Lemma 4-2 applies.

- If an attacker takes an action that takes it to a final state, then this action is either:
    - Failure management action such as ABORT_TES, in which case it will be possible for the defender to take that same action as well.
    - Or it will be through a silenced action, since TES:TSI_Done and TES:TSI_Failed come from the state model associated with UTP TestCase which we have silenced already.
- If the attacker takes the action to go to a final state, then it will have no actions to play in the next iteration.
- Therefore, the defender has a universal winning strategy.
- As a result, our mapping is sound.

4.3. Summary

In this chapter, we presented the test methods we propose for test isolation. Our test methods are widely applicable because of the nature of the assumptions made during their design. We presented also how our proposed test methods can be modeled using the modeling framework introduced in Chapter 3. We showcased the rolling paths test method on a case study and showed how it helps avoid test interferences. Furthermore, our test methods also help with dealing with the diversity of runtime environments encountered in a cloud system as they allow testing under multiple test configurations. To guide the choice of such test configurations we have proposed a set of environment coverage criteria that one can use for that purpose.

We completed the work on TEF as a building block of our architecture by proposing an execution semantics associated with the UTP concepts encountered in a test plan. Such execution semantics represent the behavior TEF exhibits when it encounters any of their associated model elements, and they enable better control over /tracking of the progress of tests execution during a test session. Existing approaches for automating test execution were designed to be used in the test environment which makes them not fit to be used to execute tests in the production environment. In fact, they do not take into consideration the failures that may happen during a test session. Our execution semantics, however, takes that aspect into consideration and proposes how to manage it. Such management can be complicated as different middleware that compose the system may be responsible for dealing with different types of failures. The use of customized verdicts in our case helped us distinguish between the failures TEF will handle, and the failures that it should leave for other middleware/human administrators to handle. Finally, we presented proof of the soundness of the mapping between the execution semantics and the model elements.

The execution semantics we proposed serves for tracking and controlling test execution which includes the deployment and tear down of test configurations and the execution of TSIs. Some behaviors were left as a choice for the implementer such as what happens to UTP TestCases in the initial state when the TestExecutionSchedule is aborted, or how the asynchronous execution of test runs is handled when the test objective is achieved. Because it associates the behaviors with UML model elements, our execution semantics is defined at a high level of abstraction which makes it a legitimate candidate for extensions to be used in other software management processes. However, in this thesis, we did not evaluate to which extent our execution semantics can support the expressiveness of UML.

Chapter 5

A method for configured instance evaluation for live testing

In this chapter, we propose a method for configured instance evaluation for live testing. To reduce human intervention in the process of test method selection, we aim in Research Objective #4 to automate the process of assessing the configured instance for the risk of test interference it may present, and how it may be impacted by each one of the test methods devised in Chapter 4 may have on parts of the system. The method we propose in this chapter helps meet this research objective as it provides an insight on the risk of test interference a configured instance presents, and an estimation of the cost to provide test isolation for that configured instance. We also present the implementation of scenario generation which is an important part of this method.

5.1. Introduction and challenges

Test interferences are the main challenge of live testing. We previously defined test interference as a loss, alteration, or degradation of a system's property due to the coexistence of production traffic and test traffic. The literature [1, 2, 3, 4] addressed some countermeasures that are taken to alleviate the risk of test interferences, test isolation, some of which rely on the ability of a developer to enhance the components of the system to become testable in production. The test isolation methods we proposed in Chapter 4 rely on the ability of the environment to clone/snapshot the components of configured instances and relocate the load. The implementation/usage of each of these methods comes at a cost. An insight on such cost can improve test planning activities as it would guide the choice of the suitable test isolation method for each situation.

Test interferences can manifest at the level of a configured instance's resource consumption patterns as well as its external behavior. Therefore, to estimate the cost of test isolation for a given configured instance, one needs to: 1) determine the risk of test interference this configured instance presents, and 2) estimate the cost of implementing test isolation countermeasures for this configured instance. Existing approaches rely heavily on human intervention to provide such insights. Such approaches have limited applicability for systems composed of closed-source components coming from different vendors such as Commercial-Off-The-Shelf (COTS) component-based systems.

In this chapter, we present a method to automatically determine if a configured instance presents a risk of test interferences manifesting at the external behavior level or at the resource consumption level, and to evaluate the cost of test isolation. The goal of our method is to confirm the presence of a risk of test interferences and not to confirm the absence of such risk;

and, to provide an insight on the test isolation cost. To achieve this goal, we had to address the following challenges:

- **Challenge#5-1**: test interferences may not always occur. In other words, the coexistence of specific production traffic and specific test traffic is what may cause an observable test interference. Therefore, at least one such combination of traffic needs to be identified and used for the test interferences to be observable. With limited insights about the configured instance being evaluated, identification of such combinations may be challenging. Furthermore, the generation of such combinations is seldom addressed in the literature as most approaches rely on record and replay solutions [52].
- **Challenge#5-2**: test interferences are hard to identify without prior knowledge of the internal behavior of the configured instance under evaluation. In fact, attention to (test) interferences that occur in production is usually raised by users. Such actors are not an option if the evaluation is to be done automatically. Thus, the need for a setup that allows for the detection of any deviance in the external behavior that may arise as a result of the existence of test traffic.
- **Challenge#5-3**: components in a system have different interfaces exposed using different protocols. To reduce human intervention in the evaluation process, the evaluation environment must have the capability to stimulate components regardless of the protocols they use.

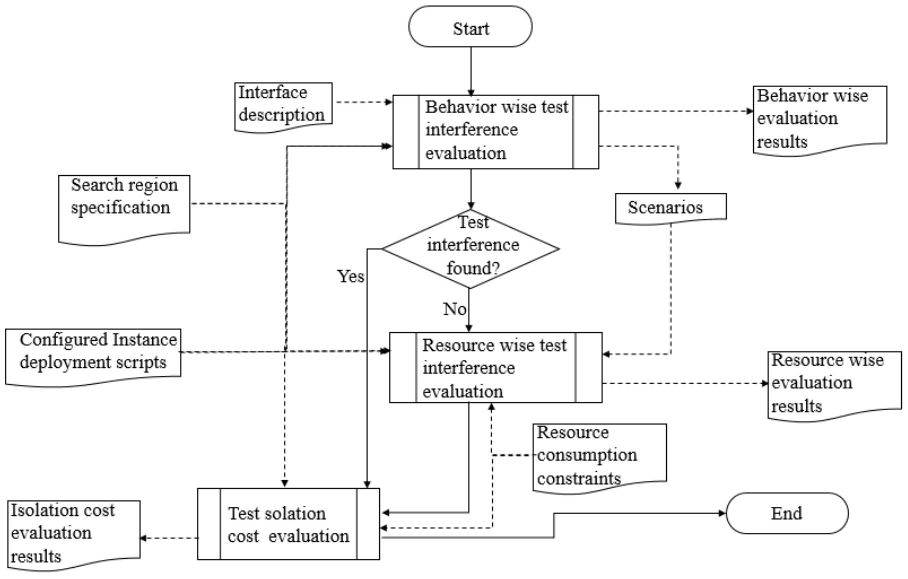
Finally, even when our method does not successfully identify any test interference revealing scenario, the user who performs the evaluation may still find useful the data we provide about the test isolation cost. Such data can guide a more vigilant user in choosing the most suitable test isolation countermeasures.

5.2. Overall approach

An instance of a configured instance (***CI instance***) in production may respond to requests (***CI event instances***) of various request types (***CI events***), with varying arrival rates and holding times, as applicable.

The ***sequences*** of CI event instances come from actors, (***sources***), that are supposed to interact with CI instances, which assign values to the parameters of the CI event. When a system is being tested in production, some of these CI event instances are part of the production traffic and some belong to the test traffic (depending on their source). Based on the value assignments, a sequence of CI event instances can be either correct/valid or an invalid sequence that the configured instance should handle properly. The production traffic received by a CI instance at the time of testing represents the conditions under which the CI instance is being tested. While the test stimulus is a sequence of arbitrary length of CI event instances. Both, the test traffic and the production traffic received by the CI instance are handled by the components of that instance and represent a ***scenario*** of testing the CI instance in production.

When in production, CI instances may receive valid or invalid sequences as production traffic. Evaluating a configured instance under all possible conditions is costly (see unfeasible).



Start
Interface description
Behavior wise test interference evaluation
Behavior wise evaluation results
Search region specification
Scenarios
Test interference found?
Yes
No
Configured Instance deployment scripts
Resource wise test interference evaluation
Resource wise evaluation results
Resource consumption constraints
Isolation cost evaluation results
Test solation cost evaluation
End

Figure 5-1. Overall approach for configured instance evaluation

One way of reducing such cost is to perform the evaluation under a set of conditions that is representative of the conditions under which a configured instance may be tested. Similarly, the test stimulus needs to be representative of the test traffic that may cause test interferences when used in production. Furthermore, test interferences may manifest as incorrect responses to user traffic impacting the external behavior of the configured instance. Test interferences may also manifest as a service outage if crashing errors are not properly handled by the configured instance. In addition, varying the volume and load of the production traffic under which a configured instance is being tested may lead to test interferences that manifest at the resource consumption level.

Figure 5-1 captures the solution we propose for configured instance evaluation. Our solution takes four inputs: (a) the interface description of the configured instance under evaluation to capture the CI events and their parameters, (b) the resource consumption constraints to capture the resource consumption range of the configured instance in production (not to be confused with the amount of resources available in production), (c) the configured instance deployment scripts to instantiate the configured instance being evaluated and create the CI instance; and, (d) the search region specification to configure the evaluation process that also serves as the stopping criterion. Such configuration includes the number of times an experiment is to be performed, the maximum number of sources of CI event instances that can be used (for a scenario generated in this evaluation), the maximum length for a sequence of CI event instances (for a sequence used in this evaluation), the interval of arrival rate and holding time (that a source uses to stimulate the instances in the evaluation setup) as well as the step by which these two values increment/decrement.

The evaluation goes through three steps. In the first step, our solution evaluates the configured instance for the risk of behavior-wise test interferences it may present when tested in production. In the second step, the configured instance is evaluated with respect to resource-wise test interference. In both steps, samples of test load and production load (scenarios) are generated and used to stimulate instances of the configured instance to check if the coexistence of these two loads causes a detectable test interference. In the last step, the cost of test isolation of the configured instance is evaluated. At the end of the third step, the evaluation will lead to three artifacts, the result of the behavior-wise test interference evaluation, which is empty if no test interference is found, otherwise, it gives the test interference revealing scenario; the result of the resource-wise test interference evaluation, which is empty if no test interference is found in this step or if this step is skipped, otherwise it gives the test interference revealing scenario; and the result of the test isolation cost evaluation, where the test isolation cost is defined as the time it takes to snapshot and to clone a component of an instance of the configured instance under evaluation, and the time needed to relocate the load from one to another component of this instance of the configured instance.

5.3. Behavior-wise test interference evaluation

Figure 5-2 captures the steps of behavior-wise test interference evaluation. Scenario generation is the first and most important step as it addresses Challenge#5-1. The scenarios used for behavior-wise test interference evaluation are generated in this step. Each scenario consists of a sequence of CI event instances that represents the test traffic, and a set of sequences of CI event instances that represent the production traffic. The scenario generation aims to generate sequences of CI event instances that enable a thorough evaluation of the configured instance. Such sequences are generated using constraints, inferred from the interface description of the configured instance (a), that may bind values assigned to parameters of CI event instances. The constraints may bind the values of parameters of CI event instances coming from the same source as well as CI event instances coming from different sources (including the test source). The ordering of CI event instances along with the used constraints are what makes a sequence valid or invalid. The generation of such sequences goes as follows:

- Inference of constraints: in this phase, the constraints that may bind the values of parameters are extracted from the description of the interface of the configured instance. Such constraints are extracted using the following rules:
    - Constraints can bind only parameters of the same type and same name. This rule extracts the operands of each constraint.
    - Constraints use operators based on the type and multiplicity of the parameters being constrained. An illustrative scheme for choosing the operator can consist of: 1) use the equal EQU operator and not-equal NEQU operator for operands of type string and multiplicity one, 2) use the operator equal EQU, lesser than LEQU, and greater than GEQU for operands of type numeral (int or float) or date and multiplicity one, and, 3) use all possible combinations for operands of type enumeration. Note that boolean operands are a special case of operands of type enumeration, 4) use inclusion and exclusion to constrain two parameters of the same type for which one has the multiplicity of one and the other has a multiplicity of one-to-many.

- Generation of scenarios: the generation of scenarios consists of, first, identifying the types of sources from which the configured instance may receive requests. This can be extracted from the interface description provided as input. A scenario is then created by generating a sequence of CI event instances per source and a test sequence for an additional source representing the test component.

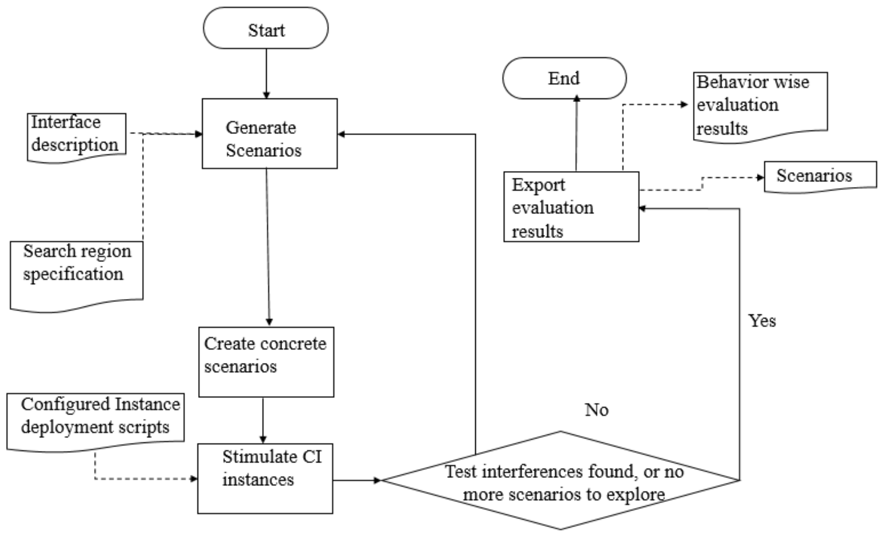

Start
End
Interface description
Generate Scenarios
Behavior wise evaluation results
Export evaluation results
Scenarios
Search region specification
Yes
Create concrete scenarios
No
Configured Instance deployment scripts
Stimulate CI instances
Test interferences found, or no more scenarios to explore

Figure 5-2. Behavior wise test interference evaluation

The search region specification configures the evaluation process by constraining its parameters. The parameters constrained during the behavior-wise test interference evaluation are the number of sources, the length of sequences, and the number of constraints. As a result, the scenarios are generated in batches. The first batch of scenarios is generated by using the minimum values (or 1 if the minimum is not specified) of the applicable parameters as constrained by the search region specification provided as input. If the scenarios of a batch fail to reveal any test interferences, the parameter values are incremented, and a new batch of scenarios is generated as the search region allows it, i.e., up to the maximum values provided in the search region specification.

Once a batch of scenarios is generated, they are turned into an executable form (concretized) and used to stimulate the CI instances. As long as no test interference was found and the search region (as specified in the input) has not been fully explored (with respect to the parameters relevant to this activity), new scenarios are generated and used to stimulate the CI instances to keep searching for test interferences revealing scenarios. Protocols of interaction may vary from one configured instance to another, and as result, the stimulation of CI instances should support multiple protocols. Tools such as [87, 88] can be used for such a purpose and as a result help address Challenge#5-3. These tools rely on input artifacts that guide their interaction with their targets, as a result, the creation of concrete scenarios will only consist of generating such artifacts from the scenarios that were generated in the scenario generation step. The concretization and stimulation follow an iterative process. Each iteration goes as follows:

- Pick a scenario that has not been used yet,
- Generate the artifacts that are needed to execute the scenario,
- Instantiate the CI instances to create the evaluation setup (Figure 5-4),
- Pass the generated artifacts as input to the tool used for stimulation such as Gym [87] or Tsung [88], and
- If a test interference is detected, stop.

5.4. Resource-wise test interference evaluation

Figure 5-3 shows the overall approach for the resource-wise test interference evaluation. Similar to the behavior-wise evaluation, scenario generation is an important step of this activity as it addresses Challenge#5-1. This step relies on the scenarios generated in the behavior-wise test interference evaluation and aims to generate test load and production load to check for deviations from expectations in the resource consumption of the configured instance under evaluation caused by the coexistence of the production and test loads. Therefore, the resource-wise test interference evaluation is driven by the constraints specified in the resource consumption constraints artifact provided as input. Note that these constraints bind resource consumption expectations of the configured instance, and they are not to be confused with the constraints inferred in the behavior-wise test interference evaluation, which bind values of parameters of CI event instances. Furthermore, these constraints specify the expected range of the amount of resources consumed in production by an instance of the configured instance and not the amount of resources available in production. Similar to behavior-wise test interference evaluation, the scenario generation for resource-wise test interference is an iterative process, and in each iteration, a batch of scenarios is generated. The generation of one such scenario goes as follows:

- Choose a scenario from the ones generated in the behavior-wise test interference evaluation.

- For each sequence in the chosen scenario, assign an arrival rate value and holding time value as applicable.
- The values of the arrival rate and holding time start from a minimum and keep increasing by a step. The minimum value and the step are specified in the search region specification.
- Once a test interference is detected or the search region is completely covered (for the arrival time and holding time) the resource-wise test interference evaluation stops.

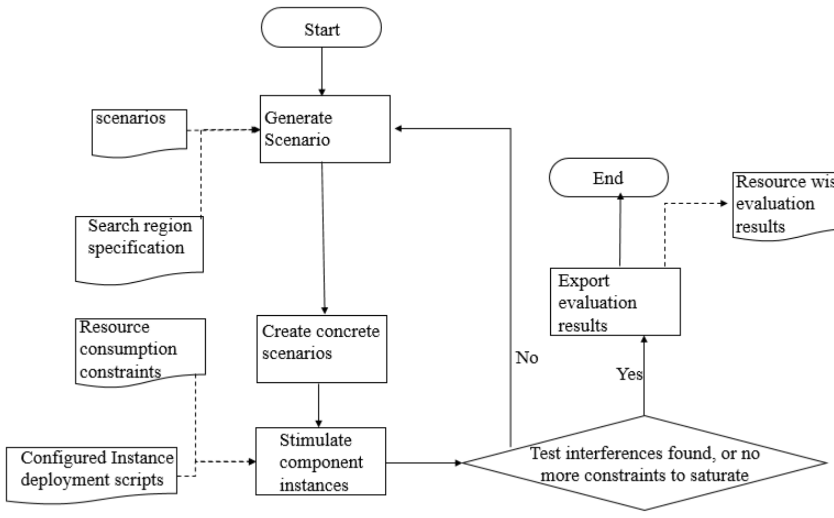

Start
scenarios
Generate Scenario
Search region specification
Resource consumption constraints
Create concrete scenarios
Configured Instance deployment scripts
Stimulate component instances
Test interferences found, or no more constraints to saturate
No
Yes
Export evaluation results
End
Resource wise evaluation results

Figure 5-3. Resource wise test interference evaluation

The creation of concrete scenarios and the stimulation of CI instances follow the same logic as in the behavior-wise test interference evaluation.

5.5. Arbitration and scenario classification

Arbitration is the process used to judge whether an observation represents a test interference or not. To achieve this goal, and address Challenge#5-2, the configured instance evaluation process relies on a setup and a scenario classification scheme to be used in both the behavior-wise and the resource-wise test interference evaluations.

Similar to how software testing relies on an oracle to decide whether a test case has passed or failed; configured instance evaluation should rely on an entity that plays a similar role to decide whether a test interference was observed or not. Such an entity should enable the decision on whether the coexistence of production traffic and test traffic caused a deviation in the behavior of a CI instance or not. In the proposed setup another instance of the same configured instance is used to play such a role. Therefore, the evaluation setup (Figure 5-4) is composed of two instances of the configured instance being evaluated:

- An instance under evaluation, which is an instance that will receive the test traffic and the production traffic simultaneously. This instance simulates an instance, of the configured instance under evaluation, being tested in production and allows for the observation of the behavior and resource consumption when the test traffic and the production traffic coexist.

- A reference instance, which is the instance that will be receiving the production traffic only. This instance represents a CI instance when it is receiving production traffic only. Therefore, it is the one capturing the normal behavior and resource consumption of the

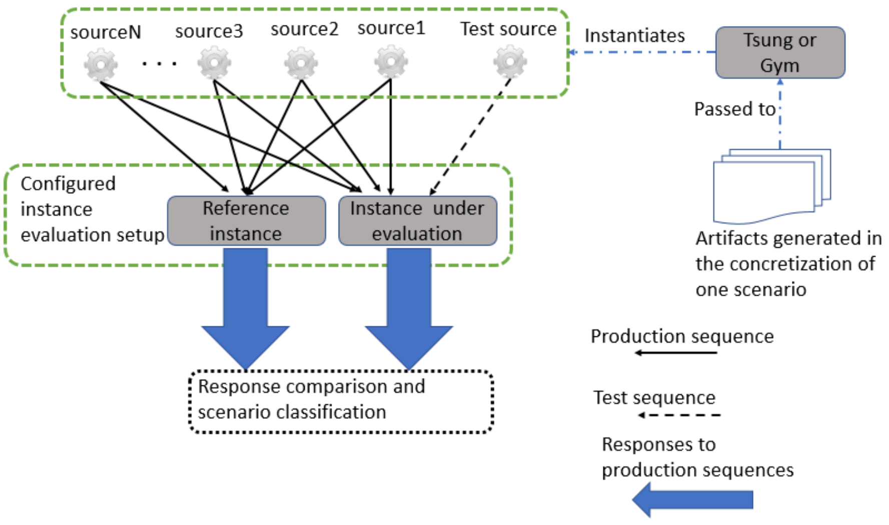

sourceN
source3
source2
source1
Test source
Instantiates
Tsung or Gym
Passed to
Configured instance evaluation setup
Reference instance
Instance under evaluation
Artifacts generated in the concretization of one scenario
Response comparison and scenario classification
Production sequence
Test sequence
Responses to production sequences

Figure 5-4. Configured instance evaluation setup

configured instance. As a result, we consider this instance to be always right and it will be playing the role of the oracle.

As a result, the sequences generated by the scenario generation and that stimulate the instance under evaluation and not the reference instance represent the test traffic and are coming from test sources. Similarly, the sequences that represent the production traffic are used to stimulate the instance under evaluation and the reference instance. Using these two instances one can easily pinpoint test interferences by observing how these two instances respond to the production traffic. This pinpointing is what we refer to as the scenario classification scheme and it goes as follows:

- If the responses (to all requests that simulate the production traffic) of both instances match, the scenario is deemed as behavior-wise test interference non-revealing. Moreover, if neither instance violates any resource consumption constraints, or if they violate the same set of constraints, the scenario is deemed as resource-wise test interference non-revealing.
- If the responses of the instances do not match, the scenario just exercised is deemed as behavior-wise test interference revealing scenario. Therefore, the configured instance being evaluated presents a risk of behavior-wise test interferences when tested in production.
- If the instance under evaluation violates more constraints of the resource consumption constraints than the reference instance, then the scenario just exercised is deemed as resource-wise test interference revealing.

5.6. Test isolation cost evaluation

The impacts the test interferences have on the system cause violations of functional or non-functional requirements. These impacts can be avoided by implementing test isolation measures. When such measures are implemented, they themselves may impact the system and lead to overhead or disturbances to the service. The nature of such impacts depends on the methods used for test isolation. In the test isolation cost evaluation, we consider the test methods described in Chapter 4 for test isolation. These test methods rely on the ability of the environment to snapshot/clone the components of a configured instance, and the ability to relocate the load from one component to another. Indeed, performing such operations on components that handle the production load will cause service disruption. Such disruption becomes more significant when the scope (number of components of the same configured instance subject to the operation simultaneously) of the operation grows or when the operation takes a longer time. The cost we consider is in terms of the time an operation takes for a component of a CI instance, and the evaluation aims to assess:

- The time it takes to snapshot a component.
- The time it takes to clone, from a snapshot, a component.
- The time it takes to relocate the load from one component to another component of that same instance of the configured instance.

The evaluation process that produces these measurements is an iterative process, each iteration goes as follows:

- Select an operation (i.e., snapshot, clone, or load relocation) for which the measurement is to be done.
    - Select a scenario randomly from the scenarios generated in the previous step(s)
    - Instantiate a CI instance according to the selected scenario.
    - Hit that instance with the production load of the scenario, i.e. the scenario is used without including the test source.
    - Perform the selected operation as applicable to a component serving production load and measure the time.

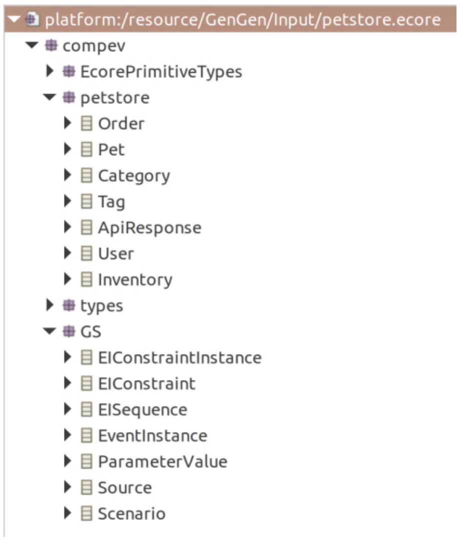
- For each operation, this process is repeated a number of times as specified in the search region specification. Such repetition is required to increase confidence in the measurements that were taken.

Once all the measurements (of all the repetitions) have been collected and exported, the user who performs the evaluation can choose how to use them. Such usage scenarios include (but are not limited to) using the average of all the repetitions, using the maximum of all the repetitions, or even training a machine learning model from the collected data in order to predict the duration of the operation at runtime.

5.7. Prototype

To show the feasibility of our approach, in this section we present an implementation of the scenario generation for the behavior-wise test interference evaluation of components that interact using REST. Accordingly, the interface description input is a swagger file of the API exposed by the configured instance. To model the scenarios, we use the Eclipse Modelling Framework (EMF). Therefore, we had to design a metamodel to which our generated EMF models will have to conform. This metamodel is composed of two parts (two packages): 1) a package that captures the general concepts used to describe scenarios. The content of this package is not specific to the configured instance being evaluated (GS package in Figure 5-5 and Figure 5-6); and, 2) a package that captures the concepts related to the API exposed by the configured instance being evaluated (petstore package in Figure 5-5), the content of this package is used to describe/specify CI event instances that are used in the scenarios and it is specific to the configured instance being evaluated. In order to obtain this metamodel we use an opensource tool [89] that converts swagger API descriptions into Ecore metamodels, then we append the content of the first package to the generated Ecore metamodel. Therefore, scenarios will be instances of the final metamodel. In order to generate such instances, we use the Epsilon Model Generation language (EMG).

The EMG modules are composed of two types of rules: 1) creation rules which are used to create instances of the metaclasses in the metamodel; and 2) matching rules which specify patterns that will be followed in order to create associations between the already created instances, or create instances of metaclasses for which we do not have creation rules. The EMG module to generate the scenarios is specific to the API of the configured instance being



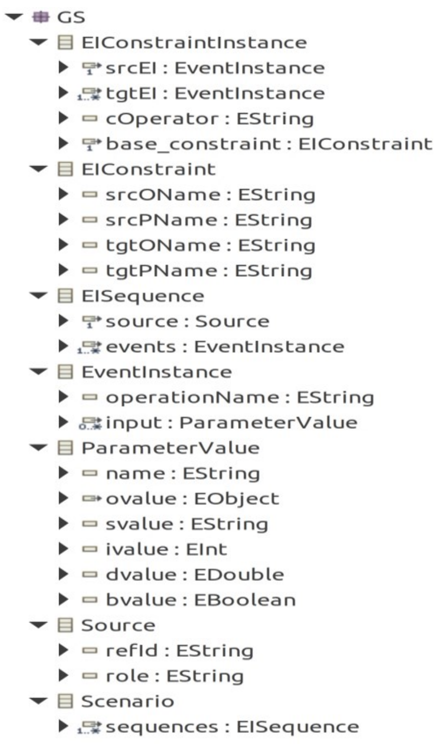
- platform:/resource/GenGen/Input/petstore.ecore
  - compev
    - EcorePrimitiveTypes
    - petstore
      - Order
      - Pet
      - Category
      - Tag
      - ApiResponse
      - User
      - Inventory
    - types
    - GS
      - ElConstraintInstance
      - ElConstraint
      - ElSequence
      - EventInstance
      - ParameterValue
      - Source
      - Scenario

Figure 5-5. Generated metamodel for PetStore API

evaluated. Therefore, we have created a Higher Order Transformation (HOT) using the Epsilon Generation Language (EGL) to generate the EMG module that will generate the scenarios. The

HOT takes as input the Ecore metamodel previously obtained and then generates the EMG module that is used to generate scenarios. The EMG module is composed of the following rules:

- Creation rules for all the metaclasses in the package that is specific to the configured instance being evaluated (e.g. package petstore in Figure 5-5).
- A matching rule that creates sequences of CI event instances that have constraints that bind parameters of operations which are of primitive data types.
- For each attribute of a metaclass that types a parameter of an operation, create a matching rule that creates CI event instances that bind that attribute to parameters of operations which are of primitive datatype (same type as per the scenario generation logic).
- For each reference metaclass, create a matching rule that creates CI event instances that bind attributes of the reference to parameters of operations which are of primitive datatype (same type as per the scenario generation logic).
- Each matching rule creates all kinds of constraints as they apply to that attribute (EQU, NEQU, LEQU, GEQU, etc.).



```
GS
  EIConstraintInstance
    srcEI : EventInstance
    tgtEI : EventInstance
    cOperator : EString
    base_constraint : EIConstraint
  EIConstraint
    srcOName : EString
    srcPName : EString
    tgtOName : EString
    tgtPName : EString
  EISequence
    source : Source
    events : EventInstance
  EventInstance
    operationName : EString
    input : ParameterValue
  ParameterValue
    name : EString
    ovalue : EObject
    svalue : EString
    ivalue : EInt
    dvalue : EDouble
    bvalue : EBoolean
  Source
    refId : EString
    role : EString
  Scenario
    sequences : EISequence
```

Figure 5-6. GS package details

Executing the generated EMG module yields a model for the configured instance being evaluated which is an instance of the metamodel in Figure 5-5. A sample model is shown in Figure 5-7, it shows the number of constraints that were inferred as well as the number of scenarios that were generated. The scenario expanded in Figure 5-7 is a scenario of an EQU type of constraint between the petId parameter of getPetById operation, and the petId parameter of deletePet operation. This specific scenario was tried manually and led to test interferences that manifested in the external behavior.

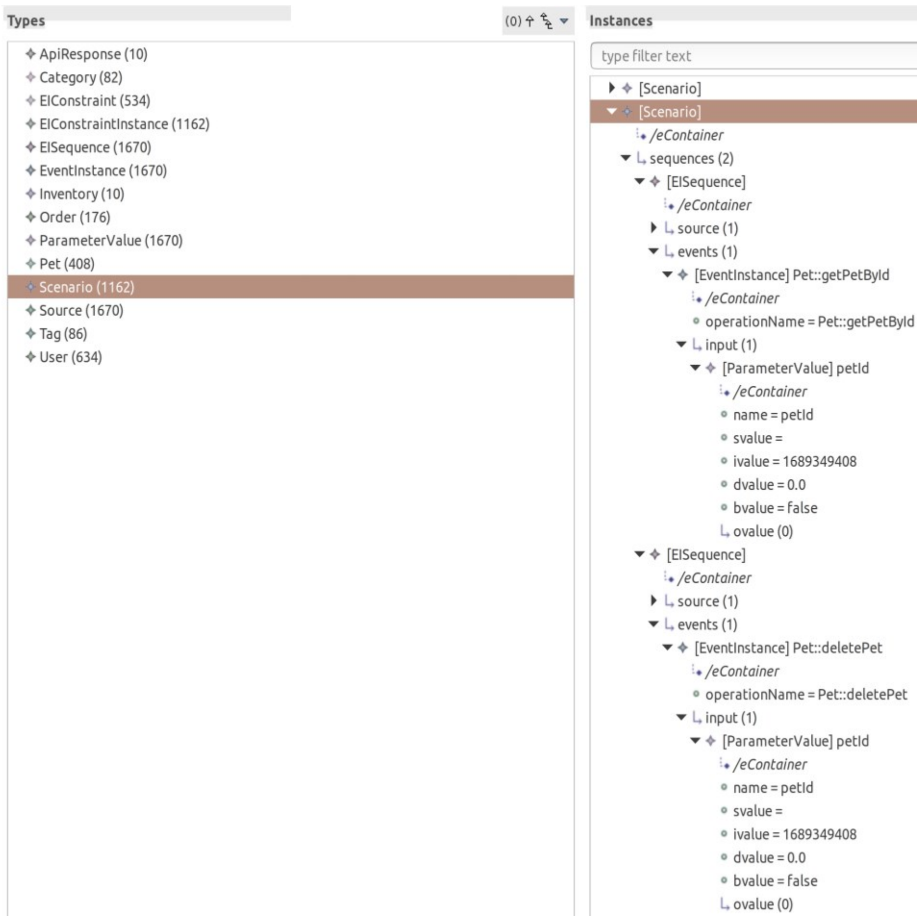

```
Types                                        (0)
✦ ApiResponse (10)
✦ Category (82)
✦ ElConstraint (534)
✦ ElConstraintInstance (1162)
✦ ElSequence (1670)
✦ EventInstance (1670)
✦ Inventory (10)
✦ Order (176)
✦ ParameterValue (1670)
✦ Pet (408)
✦ Scenario (1162)
✦ Source (1670)
✦ Tag (86)
✦ User (634)

Instances
type filter text
▸ ✦ [Scenario]
▾ ✦ [Scenario]
    /eContainer
  ▾ sequences (2)
    ▾ ✦ [ElSequence]
        /eContainer
      ▸ source (1)
      ▾ events (1)
        ▾ ✦ [EventInstance] Pet::getPetById
            /eContainer
            operationName = Pet::getPetById
          ▾ input (1)
            ▾ ✦ [ParameterValue] petId
                /eContainer
                name = petId
                svalue =
                ivalue = 1689349408
                dvalue = 0.0
                bvalue = false
                ovalue (0)
    ▾ ✦ [ElSequence]
        /eContainer
      ▸ source (1)
      ▾ events (1)
        ▾ ✦ [EventInstance] Pet::deletePet
            /eContainer
            operationName = Pet::deletePet
          ▾ input (1)
            ▾ ✦ [ParameterValue] petId
                /eContainer
                name = petId
                svalue =
                ivalue = 1689349408
                dvalue = 0.0
                bvalue = false
                ovalue (0)
```

Figure 5-7. Sample model of the generated scenarios

5.8. Summary

In this chapter, we presented the method we propose for configured instance evaluation. Our method can be used to confirm the presence of a risk of test interferences at the level of both the external behavior of the configured instance and its resource consumption. Scenario generation is an important activity in this method as it allows to recreate the conditions under which test interferences can be observed. Furthermore, our method also provides an estimate on the time it takes to snapshot/clone a component of the configured instance as well as the time it takes to relocate the service from one component to another. Such estimates can be used in various situations to calculate an estimated impact of each one of the test methods described in Chapter 4. The risk of test interference and the estimation of test isolation cost are valuable information that can be used in test method selection which is an important activity in test planning.

The information obtained using the method we proposed in this chapter is important as it can guide the activity of test planning. Furthermore, estimates on the time needed for snapshotting/cloning components as well as the time needed for service relocation can also guide the planning of other software management activities that are performed in the production environment such as live upgrade. The method we proposed relies heavily on scenario generation to recreate the conditions under which test interferences may occur/be observed. An alternative to scenario generation could have been capture and replay which consists of capturing actual production traffic and using it to stimulate the instances (replay). Although such an approach brings to the picture stimulus that is closer to reality than the scenarios we generate, the conditions under which it will allow the evaluation to be conducted will remain limited to the traffic used in the evaluation. Finally, our proposed method helps confirm the presence of test interferences that manifest at the level of the CI receiving both the test traffic and the production traffic. However, test interferences may also manifest at the level of a CI that is not receiving the test traffic. The relevance of our method to such a situation is yet to be investigated.

Chapter 6

A Test Planner for live testing of cloud services

In this chapter, we present the behavior we associate with the Test Planner. To automate testing activities for live testing, we aim in Research Objective #5 to define the behaviors enacted by the building blocks of the architecture we proposed. To achieve this research objective, we propose in this chapter the behaviors enacted by the Test Planner to generate the test package. We first present the method we propose to automate test case selection. Given the role configurations have in the cloud environment, we present a configuration-based test suite reduction method that can help focus the testing effort on finding configuration errors. We also present the method we devised to automate test plan generation.

6.1. Test case selection approach

The Test Planner is responsible for generating the test package. Therefore, the behavior we associate with the Test Planner serves for the automation of the test package generation. Such automation consists of automating the generation of the test suite and the generation of the test plan. In this section, we address the automation of the test suite generation.

6.1.1. Overall approach

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 strategies we propose for this purpose follow the rules below:

- If the received event is a periodic event happening for the first time, the Test Planner selects test cases and test design techniques that can 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 because 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 can 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.

6.1.2. Configuration based regression test suite reduction

The reconfiguration event is an important event as it usually triggers regression testing. Although one can use any regression test case selection method to respond to that event, reducing the set of selected test cases can be beneficial. Using a reduced test suite in production implies less disturbance on the one hand. On the other hand, time and resources would be better spent if used to check for errors that often cause issues in production such as configuration errors. In this section, we present our approach for reducing the regression test suite while maintaining the configuration error detection power.

6.1.2.1. Fault Model

Configuration designers may make various types of faults. Misconfigurations for instance are the type of faults that is addressed the most in the literature. Misconfigurations happen when the configuration designer sets a configuration parameter to a value that makes the resulting configuration invalid. In other words, the value of the configuration parameter does not match the type or the format it should; or it violates a constraint with the values of other configuration parameters. Misconfigurations manifest differently at runtime depending on the configurable software in question, they can manifest as crashing errors (when the configuration parameter causes the configured instance to crash), and they can also lead to a configured instance that cannot be instantiated at all (e.g. if a configurable software validates the configuration at instantiation time). In this thesis, we assume that the configuration is valid, and we question whether the cloud services meet the requirements. In other words, we address the question: Are we using the right configuration to meet the requirements? To the best of our knowledge, none of the previous work addressed this kind of errors. Therefore, we start by proposing a fault model to classify the different mistakes a configuration designer may make, and which can lead to a violation of one or more requirements.

In a cloud system, configured instances may interact with one another and share resources to realize the requirements. The configurations play an important role in setting these interactions and resource sharing. Moreover, configurations may also specify the behaviors the configured instances exhibit at runtime. As a result, a wrong configuration may lead to violations of the requirements caused by wrong interactions, flawed resource sharing, or badly parametrized behaviors. Therefore, we propose the following types of configuration faults:

- Hooking fault: a hooking fault occurs when a configured instance is set to interact with: 1) the wrong client, which can be internal or external; or 2) the wrong resource, which can be physical, virtual, or logical resource such as another configured instance. This kind of faults may occur at the level of the deployment configuration or at the level of the application configuration. Examples of such faults include: a configured instance is set to request a service instance from another configured instance, but the second configured instance does not expose that service instance; a configured instance that is set to write to a folder to which it does not have permissions, etc.
- Grouping/sharing fault: a grouping/sharing fault occurs when multiple configured instances are wrongly set to use the same resource or to be treated the same way under certain circumstances. Examples of such faults include: two configured instances that expose service instances on the same port are hosted on the same node; or multiple configured instances with largely different scaling requirements are set to scale at the same pace, etc.
- Dimensioning fault: a dimensioning fault occurs when a configured instance, for example, is not "big enough" for its purpose, i.e. does not run enough processes to meet its functional and non-functional requirements. This kind of faults can occur at the level of the deployment configuration. Examples of such faults include: not enough number

of processes to handle the maximum load, not enough number of processes to handle the maximum number of concurrent sessions, not enough redundancy to meet the availability requirement, too many underutilized instances, etc.

- Behavior setting fault: a behavior setting fault occurs when a behavior is falsely activated, deactivated, or refined. This kind of faults can occur at the level of both application and deployment configurations. Examples of such faults include: setting a timeout in a way that a non-functional requirement may be violated, setting a parameter of a function in a way that a functional requirement may be violated such as setting the maximum withdrawal limit to a wrong value in a banking application.

6.1.2.2. Classification of configuration parameters

A configured instance realizes a refined subset of the requirements of a configurable software. Therefore, one can notice similarities between configurations of configured instances that realize the same requirements. These similarities stem from the fact that to realize a specific requirement, certain functionalities of the configurable system need to be refined in certain ways. The differences between configurations, in this case, are a result of deployments in different environments (OS, libraries, network, etc.) on one hand as mentioned in [1, 19, 49]; on the other hand, different deployment environments can have different types/amounts of resources that can be allocated to the configured instance. Taking these aspects into consideration, we propose the following classification for configuration parameters:

- Deployment environment agnostic configuration parameter: a configuration parameter which is independent from the deployment environment of the configured instance. In other words, to satisfy a requirement the value of this parameter can be set independently from the environment where the configured instance is deployed. Typically, it has the same value (this can also be a range of values) for all configured instances that satisfy the requirement in question.
- Deployment Environment dependent configuration parameter: a configuration parameter for which the value, to satisfy a requirement, depends on the environment of the configured instance.

6.1.2.3. Input Artifacts

The regression test suite reduction method takes as input the following artifacts:

a) The configuration of the cloud system: is composed of the configurations of all the configured instances that compose the cloud system. These configurations are sets of key-value pairs. The configuration parameters are identified in a configuration as IdOfInstance-IdOfProduct-NameOfParam (e.g. configuredInstance1.1-configurableSoftware1-confParam1, configuredInstance1.1-configurableSoftware1-confParam2, configuredInstance1.2-configurableSoftware1-confParam1).

b) The classification of configuration parameters as deployment environment agnostic or deployment environment dependent. In this classification, the configuration parameters are referenced as IdOfProduct-NameOfParam (e.g.

configurableSoftware1-confParam1, configurableSoftware 1-confParam2, configurableSoftware2-confPram1). Thus, the class of a configuration parameter can be found by removing the instance Id from the Id of the configuration parameter in the configuration and finding in the classification the class associated with the remaining sequence. For instance, to find the class of the configuration parameter configuredInstance1.1-configurableSoftware1-confParam1 we need to look up the class of configurableSoftware1-confParam1 in the configuration parameters' classification.

c) The impact matrix of the cloud system: maps every requirement that is realized by the cloud system to the configuration parameters (of specific configured instances) that are related to it. In this matrix, the configuration parameters are identified according to (a), i.e. as they are in the cloud system configuration.

d) The initial regression test suite: the initial set of test cases selected for regression testing. These test cases can be selected using a regression test case selection method; for example, if the administrator has a set of potentially impacted requirements after an impact analysis, the test suite will be composed of all the test cases that cover at least one of the impacted requirements.

e) The requirement-service matrix: maps every requirement to the service instances used to realize the requirement.

f) The applicability matrix: maps every test case in the acceptance test suite to the service instances to which it is applicable. Since the regression test suite is a subset of the acceptance test suite, this artifact will always be the same (as long as the acceptance test suite has not changed) and can be fetched from the test repository, regardless of the regression test suite selection method that was used to come up with the regression test suite. Thus, reducing the effort to come up with a new applicability matrix every time the regression test case selection method changes.

g) The change: is the set of changes that were made to the configuration. This includes the list of configured instances that were newly instantiated, terminated, or reconfigured by changes in configuration parameters values, software updates, etc.; the configuration parameters are identified according to (a).

h) Test case runs history: captures the relevant information of successful test case runs. Such information is usually stored in the test repository and consists of triplets of: 1) a test case; 2) a requirement this test case covers; and 3) the values of the configuration parameters that were used to test this requirement in the test case run. The configuration parameters in this artifact are identified according to (b). In a fully automated environment, this artifact would be updated after every test session, i.e. entries associated with new successful test runs would be added.

6.1.2.4. Classification of Requirements and Service Instances

From the relationships between the requirements and the configuration parameters (c), and the classification of configuration parameters (b):

- Environment agnostic requirements: they map to deployment environment agnostic configuration parameters only, if any,
- Environment dependent requirements: they map to deployment environment dependent configuration parameters only, and they are about system interfaces or for interacting with the environment.
- Composite requirements: they map to deployment environment agnostic as well as deployment environment dependent configuration parameters.

To classify further the requirements based on how they are impacted by a change (g), first, we need to classify the service instances realizing the requirements. The applicability matrix (f) maps the test cases to the service instances to which they are applicable, and the requirement-service matrix (e) maps every requirement to the service instances realizing it.

Using this information, we can put a service instance in one of the following three categories:

- A service instance directly impacted by the change: a service instance which was reconfigured; and, to which at least one regression test case is applicable.
- A service instance indirectly impacted by the change: a service instance which was not reconfigured; and, to which at least one regression test case is applicable.
- A service instance not impacted by the change: a service instance which was not reconfigured; and, to which no regression test case is applicable.

Based on the classification of the service instances, the requirements can be classified from the perspective of the impact as follows:

- Directly impacted requirements: requirements that are realized by at least one directly impacted service instance.
- Indirectly impacted requirements: requirements that are not realized by any directly impacted service instances and that are realized by at least one indirectly impacted service instance.
- Unimpacted requirements: requirements that are realized only by service instances that are not impacted by the change.

6.1.2.5. Test Suite Reduction Rules

After classifying the requirements, we first reduce the applicability matrix according to the following rules:

- Rule #1: For a requirement which is not impacted by the change (i.e. unimpacted requirement), all the service instances realizing it are removed from the applicability matrix.
- Rule #2: For each test case TC that covers requirement R, remove the applicability link between a service instance S, which realizes R, and TC if:
    - R is directly or indirectly impacted, environment dependent, and there is an entry in the history (h) consisting of TC, R, and a set of configuration parameters identical to the set of configuration parameters that are owned/associated with

S and that impact R.
Or

- R is directly impacted, environment agnostic, and there is an entry in history (h) consisting of TC, R, and a configuration (set of configuration parameters and their values) identical to the configuration owned/associated with S and that impact R.

After applying these two rules to reduce the applicability matrix, we apply the following rules to try to reduce it further before selecting the final set of test cases:

- Rule #3: Remove any service instance with no applicable test cases after applying the first two rules, and which realizes environment agnostic requirements.
- Rule #4: Remove any service instance from the applicability matrix if it realizes an environment dependent requirement and:
    - After applying the first two rules, it has no applicable test cases; and
    - The service instance shares configuration parameters with another service instance that realizes a composite requirement (that is directly or indirectly impacted by the change). These configuration parameters impact both requirements, i.e. the environment dependent requirement and the composite requirement
- Rule #5: If a service instance realizes an environment dependent requirement and Rule#4 does not apply to it, it gets its applicability links back.

Finally, we select from the initial regression test suite all test cases that apply to at least one service instance left in the applicability matrix. The selected test cases will be run on each service instance to which they apply in the reduced applicability matrix (not the original).

6.1.2.6. Semi-formal proof

As a first step, we established, as shown in Table 6-1, the link between the classes of requirements and the kinds of faults that can cause their violations. The simplest case is the environment agnostic requirements as their violation can only be caused by a behavior setting fault (specifically a setting fault of an environment agnostic configuration parameter). This stems from the fact that the other three types of faults involve deployment environment dependent configuration parameters and therefore can impact either environment dependent or composite requirements. Environment dependent requirements are by definition impacted by deployment environment dependent configuration parameters only, and are about the system interfaces for interacting with the environment; therefore, their violations can only be caused by a grouping/sharing fault or a hooking fault. Finally, violations of composite requirements can be caused by a fault of any of the above kinds. Moreover, a behavior setting fault that causes a violation of a composite requirement may involve environment agnostic as well as environment dependent configuration parameters.

We need to prove that if the original regression test suite can detect a configuration fault, the reduced test suite can detect it too. For this end, we make the assumption (Assumption #1) that hooking and grouping/sharing faults due to a given configuration parameter will manifest at the level of all the requirements impacted by this configuration parameter, i.e. for each one of these requirements at least one test case covering it will fail.

Table 6-1. Classes of requirements and types of faults that may cause their violations

	Environment dependent requirement	Environment agnostic requirement	Composite requirement
Hooking fault	X		X
Grouping/sharing fault	X		X
Dimensioning fault			X
Behavior setting fault		X	X

We proceed hereby with the different configuration faults.

Case 1: Dimensioning faults

Dimensioning faults can only violate composite requirements. The original regression test suite and the reduced test suite contain the same set of test cases for covering composite requirements. Therefore, they can detect the same set of dimensioning faults.

Case 2: Behavior setting faults

Behavior setting faults can violate composite requirements and environment agnostic requirements.

Using the same reasoning as for Case 1, we can conclude that the reduced regression test suite and the original regression test suite can detect the same set of setting faults that may jeopardize composite requirements.

Environment agnostic requirements are requirements that are only impacted by environment agnostic configuration parameters. Therefore, if a test case which covers an environment agnostic requirement passed for a service instance S which realizes this requirement and which is provided using a configured instance of a configurable system CS, the test case will also pass for all service instances to which the test case is applicable and which realize this requirement and are provided using configured instances of CS using the same values of the configuration parameters that impact that environment agnostic requirement. As a result, behavior setting faults which may violate an environment agnostic requirement can only be detected using test cases which were never used to validate a service instance which realizes the same requirement and is provided using the same configuration. Such a test case can either be a newly added test case, or a test case which was never selected for regression testing; i.e. when the service instance has been provided using the current values of the configuration parameters that impact the requirement. This could be the case as some regression test case selection methods select different regression test suites on each run depending on the change. These test cases, according to our reduction rules, remain in the reduced test suite as their applicability links to the service instances are never removed, because their entries in the runs history (h) do not satisfy the conditions of Rule #2. Therefore, the reduced test suite and the original test suite can detect the same behavior setting faults that may jeopardize environment agnostic requirements.

Case 3: Hooking faults and Grouping/sharing faults

Hooking faults and grouping/sharing faults may violate both composite requirements and environment dependent requirements. Since the reduced test suite and the original test suite use the same set of test cases to cover the composite requirements, the reduced test suite will be able to detect hooking and grouping/sharing faults that can be detected by the original test suite and that may jeopardize composite requirements. Using Assumption #1, and the fact that we keep the test cases that were not used before for another configured instance realizing the same requirement, we can deduce that the reduced test suite and the original test suite can detect the same faults that may jeopardize environment dependent requirements. In fact, evaluating the compliance of the system to an environment dependent requirement is only a matter of ensuring that the interactions with the environment are done properly.

6.2. Test plan generation approach

The test plan is the road map followed by the Test Execution Framework to execute the TSI runs in the test suite. Just like the test suite, the test plan is an important component of the test package that is generated by the Test Planner. As a result, the Test Planner should be capable of generating the test plan as it does for the test suite. In this section, we propose a method to automate the generation of the test plan. This test plan is enacted by the Test Execution Framework.

6.2.1. Overall approach

The goal of test plan generation is to design a test plan that enables the execution of TSIs under the required test configurations while maintaining the disturbance level within an acceptable range. Moreover, test plan designers may strive to reduce the disturbance induced and the time taken by testing activities to make such disturbance less noticeable or more tolerable.

Given a test suite and a set of test configurations against which each TSI is to be run, the number of the resulting test runs will always be the same for this combination regardless of how the test plan was designed. The cost we consider in this thesis consists of the time taken and disturbance induced by the execution of a test plan and it can be broken down into: 1) a cost endured by running the TSIs; and 2) a cost endured by the setup and teardown of test configurations (setting up and removal of paths). Assuming that the former will be the same per TSI for all test plans involving a given test suite and a given set of test configurations. Thus, improvements can only be achieved by playing with the latter, i.e. the cost of setting up and tearing down test configurations. To illustrate this observation by an example we can consider the case of two TSIs, TSI1 and TSI2 along the same call path. TSI1 should be run against the

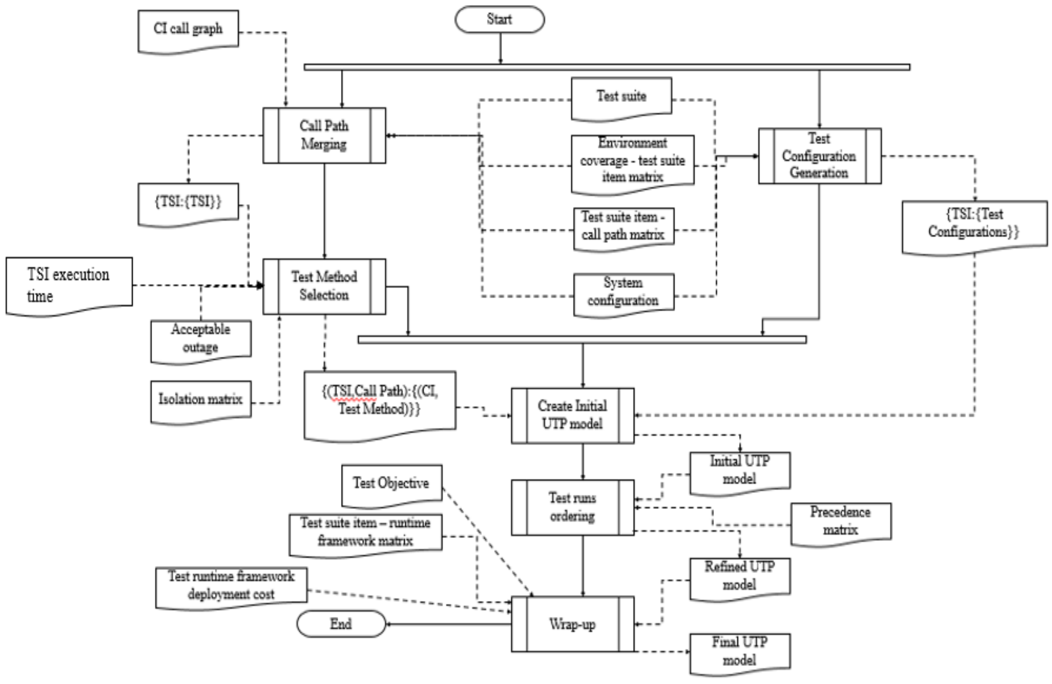


Figure 6-1. Overall Test Plan Generation Approach

set of test configurations {TestConf1, TestConf2, TestConf3}. TSI2 should be run against the set of test configurations {TestConf4, TestConf5}. If we assume that when TestConf1 and TestConf5 have the same environments for all the components under test in common between the two test configurations, we can then compare the following two alternatives to design the test plan below:

- Alternative #1: in which we deploy TestConf1 and execute TSI1 and tear down TestConf1. Then, deploy TestConf2 and execute TSI1 and tear down TestConf2. Then, deploy TestConf3 and execute TSI1 and tear down TestConf3. And then do the same for TSI2 under its associated test configurations. In this design, we had five test runs, with TSI1 being executed three times, and TSI2 executed two times. And we had five test configuration deployments and 5 test configuration tear downs.

- Alternative #2: in which we deploy TestConf1 and execute TSI1, execute TSI2, then tear down TestConf1. Then, deploy TestConf2 and execute TSI1 and tear down TestConf2. Then, deploy TestConf3 and execute TSI1 and tear down TestConf3. Finally, deploy TestConf4 and execute TSI2 and tear down TestConf4.

It is clear that the second alternative is more efficient as it has fewer test configuration deployment/tear down related operations as opposed to the first design although both of them execute the same number of test runs, five. As a result, many activities that we propose in this approach focus mainly on reducing the number of times a test configuration is deployed, and its deployment/tear down time as it is the main factor that makes the difference between one test plan and another.

The test plan generation method is shown in Figure 6-1. It starts by generating the test configurations under which each TSI is to be run. This generation is based mainly on the system configuration and the environment coverage criterion (as described in Chapter 4) the test plan designer provided as input for each TSI. Call path merging is an activity of the test plan generation that can be carried out while test configurations are being generated. This activity is the first and most important step in reducing the number of times test configurations are deployed. It mainly relies on the intersections of the call paths on which each TSI is to be applied, as well as the environment coverage associated with each TSI. Based on such information one can identify TSIs that will be associated with test configurations that can be deployed at once. The goal of call path merging is to put such TSIs into the same group. Therefore, there exists a test configuration, associated with a TSI of a group, that will have to be deployed for all the TSIs of the group to have their runs executed. After the call paths merging activity, the test method selection activity selects the test method that will be used for each CI in each call path associated with a group from the previous activity. After completing the test method selection and the test configuration generation is done, an initial UTP model is created using the mapping in Table I, and by cleaning up any duplicate test runs which may result from previous activities (mainly call paths merging). The initial UTP model is then given to the Test runs ordering activity which achieves two goals: 1) orders the test runs based on the precedence relationships between their associated TSIs; and, at the same time, 2) orders the test runs based on their associated test configurations to reduce disturbance. After the test runs ordering activity, the test plan generation is wrapped up by selecting the test runtime framework for each test component that will be involved in the test session.

6.2.2. Input artifacts

The input artifacts of our method are as follows:

a) System configuration: composed of the configurations of all the CIs that compose the system.

b) Test suite: a set of test cases and test design techniques that will be used to achieve the test objective. Each element of the test suite is a TSI.

c) TSI application matrix: maps every TSI to a call path in the CIs call graph. The vertices of such a path represent the CIs that are targeted by the TSI.

d) Environment coverage-TSI matrix: maps every TSI to the environment coverage that its runs should achieve.

e) CIs call graph: a directed graph that captures functional dependencies between the CIs. Each vertex in this graph represents a CI in the system. An edge going from vertex V1 to vertex V2 means that the CI represented by V1 calls the CI represented by V2 in a realization of one of the services provided by the system. Each edge has a weight that represents the tolerance time of the CI represented by the source of the edge to the unavailability of the CI represented by the target of the edge. Such representation of the system can be extracted automatically using tools such as [81, 57, 58, 59, 60, 61]. Moreover, the weights of the edges of such a graph (i.e. the tolerance time) are usually part of the configuration and indicate when the outage is unnoticeable for the dependent CI.

f) Isolation cost matrix: associates with every CI in the system the time it takes: 1) to snapshot one of its components, 2) to clone one of its components from an existing snapshot, and 3) to relocate from one of its components to another the portion of the SIs a component provides. Such information can be provided either by the CI vendor or measured when the CI is first acquired by the service provider. In a fully automated context such as the one we describe in this thesis, such information should be in the test repository.

g) TSI execution time: associates with each TSI an estimate of the time one of its runs may take. This information is usually provided by the test developer. In a context that leverages automation such as our context, such information can be collected from previous runs of the TSI and stored in the test repository or set by a default value.

h) Test runtime framework deployment cost: associates with every test runtime framework the deployment alternatives and the time it takes to deploy it using each possible alternative. Such information is usually available at test case (or TSI) development time.

i) TSI test runtime framework matrix: associates with every TSI the runtime framework that is needed to execute the TSI. Such information is usually available at test case (or TSI) development time.

j) Acceptable outage: associates with every SI the duration for which it can be unavailable during the testing. A SI is said to be unavailable if it was inaccessible to a dependent SI for longer than the tolerance time of this dependent SI or if the availability manager reports it as unavailable (if the SI has no dependent). Note that an acceptable outage is the time spent by the SI in an outage that is noticeable, while the tolerance time is the time for which the disturbance of the SI is tolerable.

k) Test objective: the objective that needs to be achieved by the test session for which the test plan is to be generated. The test objective is specified in the event that triggered the test session.

l) TSIs precedence matrix: which is a matrix that maps each TSI in the test suite, to TSIs in the test suite that must be executed before it. Such information is usually available at TSI development time.

6.2.3. Test configuration generation

The test configuration generation activity generates test configurations under which the TSIs will be run. In a test configuration, each CI along the call path is assigned a mixture, such an assignment specifies the path under which the TSI run is to be conducted. The test configuration generation takes as input the test suite (b), the environment coverage-TSI matrix (d), the system configuration (a), and the TSI-call path matrix (c). The generation of test configuration is done with respect to the coverage criteria provided in (d) and they are amongst the ones defined in Chapter 4. The generation of test configurations that takes into consideration the environment coverage has three main steps: First, the boundary environments are identified from the configuration for the required coverage criterion. Then, the mixtures are created based on the set of boundary environments and the mixtures' width. Finally, the set of test configurations that satisfy the required criterion is created for the:

- All boundary environment mixtures paths coverage: as the cartesian product of the sets of mixtures of the CI along the call path.
- Pairwise boundary environment mixtures: as the covering array of strength two created by considering each CI as a factor and each mixture of a CI is a level of the factor associated with that CI.
- All boundary environments mixtures: as the covering array of strength one created by considering each CI as a factor and each mixture of a CI is a level of the factor associated with that CI.a greedy algorithm can generate a minimal set of test configurations that meets this criterion.

6.2.4. Call paths merging algorithm and evaluation

The call paths merging helps reduce the cost of testing by playing with the first factor that contributes to the cost which is the number of times test configurations are deployed. Because more than one TSIs may have runs under the same test configuration, deploying such test configurations only once and invoking the TSIs is indeed a way to reduce the number of test configurations deployments which also reduces the number of test configuration related operations (for deployment and removal). In this section, we propose an algorithm for call paths merging and prove that the solution it yields is optimal, i.e. the grouping it outputs has the minimum set of test configurations to be deployed for running all the TSIs runs.

6.2.4.1. Algorithm

The call paths merging algorithm we propose takes as input the test suite (b), the TSI call path matrix (c), the environment coverage-TSI matrix (d), and the CIs call graph (e). The output of the call paths merging activity is a set of groups of TSIs, the runs of TSIs of each group under a given test configuration are invoked within the same UTP TestCase in the final UTP TestExecutionSchedule model. A call path is a path (as defined in graph theory) in the CIs call graph (e). We say that path A is a sub-path of path B if: 1) all vertices of A are also vertices of B; and 2) all edges of A are also edges of B. One can also say that B is a super-path to A. In S set of paths, a max-path is a path which is a super-path to all paths in S.

The call paths merging follows two rules. A path A can be merged with S set of paths only if:

- *A* is a super-path to the max-path of *S*, and the width of the mixtures in which *A* is to be covered is greater than or equal to the maximum width in which the max-path of *S* is to be covered.

  Or,

- *A* is a sub-path of the max-path of *S*, and there exists at least one mixture width in which the max-path of *S* is to be covered that is greater than or equal to the width of the mixtures in which *A* is to be covered.

Applying these two rules may result in two types of merges:

- Full merge: which is a merge that happens when the sub-path has a weaker environment coverage criterion than the super-path. It is called a full merge because the runs of the sub-path are covered by the runs of the super-path.
- Partial merge: which is a merge that happens when the sub-path has a stronger environment coverage criterion than the super-path. It is called partial merge because the runs of the super-path will not be enough to cover all the runs of the sub-path. As a result, the runs of the sub-path may be split over several super-paths, and some runs may need to be covered in addition, and will be executed together.

The goal of the call paths merging activity is to perform as many full merges and partial merges as possible, thus reducing the number of times some test configurations will be setup to execute the TSIs runs.

Algorithm 1 achieves the goals of the call paths merging activity, i.e. it applies as many full and partial merges as possible. From the algorithm, one can identify several possibilities of the merging. Lines 6-23 show the possible scenarios of the full merge. The full merge can either be a full merge while maintaining the same max-path (lines 6-14); or, a full merge in which the new TSI sets a new max-path for the group (lines 15-22). Similarly, partial merges are done in various forms (lines 24-42). The first scenario of a partial merge (lines 24-31) consists of distributing the runs of a TSI over several groups with max-paths that are super-paths to the call path of the TSI. The algorithm accounts for the case where such groups are not enough to cover all the runs of the TSI, thus the addition of another group (line 41) to cover the remaining runs. In the second scenario of the partial merge (lines 32-39), the max-path of the group to which the TSI is added is set by the newly added TSI. Therefore, this implies that all the runs of the new TSI are covered in this new group but runs of the TSIs of the old group (the group before the addition of the new TSI) may not be all covered. Therefore, the algorithm keeps the old group as well to account for the runs that will not be covered by the group after the partial merge.

Algorithm 1: Call Path Merging

```
CI_CG: (e),TS: Test Suite, EC_TSI: (d), TSI_CP: (c);
TSI_G: output = {};
while TS not Empty do
    cTSI = TS.first();
    if TSI_G not Empty then
        for t in TSI_G do
            if TSI_CP.get(cTSI).length == 1 then
                if TSI_CP.get(cTSI).isSubPath(TSI_CP.get(t)) then
                    if EC_TSI.get(cTSI).isOfLessWidth(EC_TSI.get(t)) then
                        TSI_G.get(t).add(cTSI); TS.remove(cTSI);break;
            if TSI_CP.get(cTSI).isSubPath(TSI_CP.get(t)) then
                if EC_TSI.get(cTSI).isWeakerThan(EC_TSI.get(t)) and EC_TSI.get(cTSI).isOfLessWidth(EC_TSI.get(t)) then
                    TSI_G.get(t).add(cTSI);
                    TS.remove(cTSI);break;
            if TSI_CP.get(t).isSubPath(TSI_CP.get(cTSI)) then
                if EC_TSI.get(t).isWeakerThan(EC_TSI.get(cTSI)) and EC_TSI.get(t).isOfLessWidth(EC_TSI.get(cTSI)) then
                    tmp = TSI_G.get(t);
                    tmp.add(cTSI);
                    TSI_G.put(cTS,tmp);
                    TSI_G.remove(t);
                    AdjustGroupingFull(TSI_CP,EC_TSI,cTSI,TSI_G);
                    AdjustGroupingPartial(TSI_CP,EC_TSI,cTSI,TSI_G);
                    TS.remove(cTSI);break;
        if TS.contains(cTSI) then
            isBeingMerged = false;
            for t in TSI_G do
                if TSI_CP.get(cTSI).isSubPath(TSI_CP.get(t)) then
                    if (not EC_TSI.get(cTSI).isWeakerThan(EC_TSI.get(t))) and EC_TSI.get(cTSI).isOfLessWidth(EC_TSI.get(t)) then
                        TSI_G.get(t).add(cTSI);
                        if not isBeingMerged then
                            isBeingMerged = true;
                if TSI_CP.get(t).isSubPath(TSI_CP.get(cTSI)) then
                    if (not EC_TSI.get(t).isWeakerThan(EC_TSI.get(cTSI))) and EC_TSI.get(t).isOfLessWidth( EC_TSI.get(cTSI) ) then
                        tmp = TSI_G.get(t);
                        tmp.add(cTSI);
                        TSI_G.put(cTSI,tmp);
                        AdjustGroupingPartial(TSI_CP,EC_TSI,cTSI,TSI_G);
                        TS.remove(cTSI);break;
            if isBeingMerged and TS.contains(cTSI) then
                TSI_G.put(cTSI,{cTSI});
                TS.remove(cTSI);
        if TS.contains(cTSI) then
            TSI_G.put(cTSI,{cTSI});
            TS.remove(cTSI);
    else
        TSI_G.put(cTSI,{cTSI});
        TS.remove(cTSI);
procedure AdjustGroupingPartial(TSI_CP,EC_TSI,tsi,grouping)
for nT in grouping do
    if TSI_CP.get(nT).isSubPath(TSI_CP.get(tsi)) then
        if not EC_TSI.get(nT).isWeakerThan(EC_TSI.get(tsi)) and EC_TSI.get(nT).isOfLessWidth(EC_TSI.get(tsi)) then
            grouping.get(tsi).addAll(grouping.get(nT));
procedure AdjustGroupingFull(TSI_CP,EC_TSI,tsi,grouping)
for nT in grouping do
    if TSI_CP.get(nT).isSubPath(TSI_CP.get(tsi)) then
        if EC_TSI.get(nT).isWeakerThan(EC_TSI.get(tsi)) and EC_TSI.get(nT).isOfLessWidth(EC_TSI.get(tsi)) then
            grouping.get(tsi).addAll(grouping.get(nT));
            grouping.remove(nT);
```

6.2.4.2. Evaluation

The goal of the proof is to demonstrate that the call paths merging using Algorithm 1 yields the maximum merging, i.e. the set of test configurations associated with the groupings is the minimum set of test configurations to be deployed for all the test runs to be executed. To prove this, we define a partial order amongst test runs based on the test configurations they involve. We represent this binary relation as a directed graph and show afterward that Algorithm 1 yields the sources of this graph as heads of the groupings. The number of sources is equal to the minimum number of test configurations to be deployed for all the test runs to be executed.

Definition 6-1: Let $TestConf1$ and $TestConf2$ be two test configurations, we say that $TestConf1$ is a sub-configuration of $TestConf2$ and write it $TestConf1 < TestConf2$ iff:

$$\forall CUT \in TestConf1,\ CUT \in TestConf2\ \wedge\ env(CUT, TestConf1) = env(CUT, TestConf2)$$

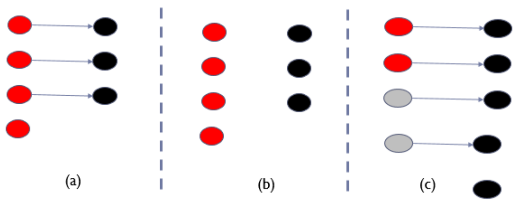


Figure 6-2. Example of graph that represents the binary relation of Definition6-1

$env(C, TestConf)$ returns the environment of the component under test C in the test configuration $TestConf$.

Figure 6-2 shows a representation of the partial order of the test runs according to the definition. The colors of the nodes distinguish the test cases involved in each test run, test case with red nodes, test case with black nodes, and test case with grey nodes. Each node represents a run of the test case against a specific test configuration. An edge between two nodes indicates that the test configuration of the target node is a sub-configuration, as per the Definition 6-1, of the test configuration of the source node. (a) shows the case in which we may end up with a full merge, as every test configuration under which the test case with the black nodes is to be run is a sub-configuration of at least one test configuration under which the test case with red nodes is to be run. In such case, the grouping will group the test runs of both test cases and the head of the grouping, i.e. the TSI associated with the max-path of this grouping, will be the test case with the red nodes. (b) captures two test cases that are run under unrelated test configurations, in this case, the minimum number of test configurations that need to be deployed for all test runs to be executed equals the total number of test configurations as no merging is possible. (c) shows the case in which we may end up with two partial merges as some test configurations under which the black test case is to be run are sub-configurations of test configurations under which the red or grey test case, moreover there are test configurations under which the black test case is to be run and which are not sub-configurations of any other test configuration. Such a case occurs when the environment coverage under which the black test case is to be run is stronger than the environment coverages under which the red test case and the grey test case are to be run. In this situation we end up with three groupings, the first one has the red test case as head, the second one has the red test case as head, and the third has the black test case as head in order to cover the test configurations that are not related to any other test configuration.

Lemma 6-1: given a test case T and a set of test configurations $TConfs$ under which T is to be run. The test configurations in $TConfs$ are unordered as per Definition 6-1.

In order to prove the optimality of Algorithm 1, we will proceed by induction:

- Case we have only one test case to be run under a set of test configurations. In this case Algorithm 1 will yield one grouping under which there is one test case, and as per **Lemma 6-1**, the number of test configurations to be deployed for the test runs to be executed will be equal to the number of sources in the graph representation of the ordering defined in **Definition 6-1** or this set of test runs.
- Let TCs be a set of test cases, each of which is to be run under a set of test configurations $TConfs(TC_i)$ for $1 \leq i \leq |TCs|$. Let G_n be the result of Algorithm 1 when applied to the test cases in TCs and their corresponding test configurations. Let OR_n be the directed graph representation of the ordering defined in **Definition 6-1** for the test runs obtained by running the test cases in TCs under their corresponding test configurations. Let's assume that G_n is optimal, i.e. the number of test

configurations to be deployed for the test runs to be executed given G_n equals the number of sources in OR_n. Let TC_{n+1} be a new test case that is to be executed under a set of test configurations $TConfs(TC_{n+1})$, and let OR_{n+1} be the directed graph representation of the ordering defined in Definition 6-1 for the test runs obtained by running the test cases in $TCs \cup \{TC_{n+1}\}$ against their corresponding test configurations. OR_{n+1} will be composed of the nodes and edges of OR_n plus additional nodes and edges to represent the new test runs and their relation to the test runs from TCs. Since OR_{n+1} and OR_n are directed graphs, there are a few ways into which such nodes and edges can be added:

- Case 1: all the nodes that were added have no incoming edges from or outgoing edges to any of the sources of OR_n, i.e. new nodes are added in a manner like Figure 6-2 (b). In this case OR_{n+1} will have $|TConfs(TC_{n+1})|$ more sources than the sources of OR_n. Moreover, lines 46 to line 47 of Algorithm 1 will be executed, in which case a new grouping will be added in which TC_{n+1} will be the head, meaning that executing the test runs given the new grouping will result in the deployment of $|TConfs(TC_{n+1})|$ more test configurations, which is equal to the number of sources in OR_{n+1}. As a result, the solution is still optimal.
- Case 2: a subset of the nodes that were added has outgoing edges to sources in OR_n, and such sources correspond to the test runs of a set of heads of

groupings in G_n, i.e. the case in Figure 6-2 (a) in which the nodes of the black test case existed in G_n and TC_{n+1} was the red test case. If TC_j for $1 \leq k \leq j \leq l \leq |TCs|$ is the set of heads of groupings in question, then OR_{n+1} has $|TConfs(TC_{n+1})| - \sum_{j=k}^{l} |TConfs(TC_j)|$ more sources than OR_n. Algorithm 1 will apply line 19 to apply a full merge when encountering the first head of such set, then loops through existing groupings in line 21 to line 22 to see if there is potential for any other full merge or partial merge. Therefore, the new solution will add $|TConfs(TC_{n+1})|$ (line 19) more test configuration deployment and subtract $\sum_{j=k}^{l} |TConfs(TC_j)|$ (lines 21-22) test configuration deployments, which corresponds to the number of sources in OR_{n+1}. As a result, the solution is still optimal.

- Case 3: the set of nodes that were added has outgoing edges to sources in OR_n, and such sources correspond to subsets of test runs of a set of heads of groupings in G_n, i.e. the case in Figure 6-2 (c) in which the nodes of the black test case and the red test case existed in G_n and TC_{n+1} was the grey test case. If TC_j for $1 \leq k \leq j \leq l \leq |TCs|$ is the set of heads of groupings in question, then OR_{n+1} has the same number of sources as OR_n. Algorithm 1 will apply line 36 to apply a partial merge when encountering the first head of such set, then loops through existing groupings in line 37 to see if there is potential for any other partial merges. Since it is a subset and not the full set of test runs that existed in OR_n, this will not change the number of sources although it does change the heads of some groupings. As a result, the solution is still optimal.

- Case 4: the set of nodes that were added has no outgoing edges to sources in OR_n, but has incoming edges from sources in OR_n, i.e. a manner like Figure 6-2 (a) in which TC_{n+1} is the black test case, or Figure 6-2 (c) in which TC_{n+1} is the black test case without the run represented by the extra node with no incoming or outgoing edges. In such situation OR_{n+1} has the same number of sources as OR_n. Algorithm 1 will either execute lines 6 to 14 to perform a full merge (same situation as Figure 6-2 (a)) or lines 27 to 31 to perform a set of partial merges (same situation as Figure 6-2 (c)). In the latter case, Algorithm 1 will also add an extra grouping (line 40), but such grouping will be filtered out in the wrap up as it will not contain any test run. As a result, the solution is still optimal.
- Case 5: a subset of the nodes that were added has incoming edges from sources in OR_n and another subset has outgoing edges to sources in OR_n. Let I_n be the set of test cases of which the runs are represented by sources that have outgoing edges to nodes that represent the runs of TC_{n+1}, and O_n the set of test cases of which the runs are represented by sources that have incoming edges from the new sources that represent the runs of TC_{n+1}. I_n and O_n are

both nonempty sets because if at least one of them is empty we end up in one of the cases 1-4. Obviously, the number of nodes that represent the runs of TC_{n+1} is at least as big as the number of sources corresponding to test runs of the test cases in I_n and O_n. If those numbers are equal, then the number of sources in OR_{n+1} is equal to the number of sources in OR_n. If those numbers are not equal then OR_{n+1} has $|TConfs(TC_{n+1})| - |\{ s \in sources(OR_n)\ such\ that\ TSI(s) \in I_n \cup O_n\}|$. Using Definition 6-1, one can deduce that in the process of creating OR_n the test runs of the test cases in O_n were merged in a partial merge with the runs of the test cases in I_n. As a result, Algorithm 1 will use lines 27 to 31 to perform a partial merge with the nodes that correspond to test cases in I_n without head change, and lines 32 to 38 to perform the partial merge with head change with nodes that correspond to test cases in O_n. Therefore, new sources will be considered only when the merge with the nodes in O_n is a full merge, and it will be the case in which new sources are added to compose the solution. Hence the optimality of the result.

6.2.5. Test method selection

After grouping the test runs in the previous activity, we obtain sets of test runs that are grouped together. For each group of TSIs, the test method selection activity selects the test methods that will be used. Since the test methods apply at the CI level, for each group of TSIs we assign a test method for each CI in the max-path of that group of TSIs. The test method selection activity takes as input the groups of TSIs from the call paths merging and their associated paths, the system configuration (a), the call graph of the CIs (e), the isolation cost matrix (f), and the acceptable outage (j). The selection of the test methods takes into consideration the availability of resources, the cost of isolation, the dependencies between CIs, and the amount of tolerable disturbance for each SI.

6.2.5.1. Applicability check of the test methods

A test method is said to be applicable to a given CI if it can be used for the test isolation for this CI without causing any unacceptable outage. As a result, more than one test method may be applicable to a CI. Furthermore, a test method may be used even though it causes outage, but this outage is acceptable. The applicability of the different test methods is determined as follows:

- The single step test method is applicable for CIs that do not present any potential risk of test interferences.
- The rolling paths test method is applicable to a CI if the time it takes to snapshot a component of that CI is less than the tolerance time of all the SIs that depend on the SIs provided by this CI and the service relocation time is also less than the tolerance time of all the dependents.

- The small flip is applicable whenever the rolling paths is applicable, and the configuration allows it. In other words, the small flip is applicable to a CI if there exists *S*, for which Equation (1) holds. In Equation (1) *env* is a boundary environment, *S* is a set of boundary environments of the CI, *N* is the width of the mixture of the environment coverage criteria, *Nodes*(*env*) is the set of nodes that can host boundary environment *env*, *K* is the number of components needed by the CI at the time of the testing, and *Nodes* is the set of all nodes on which the CI is deployed. Note that *K*, the number of components needed at the time of the testing, may not be known at the time of the design of the test plan. It can be estimated based on the operational profile of the CI, or its worst-case value can be used.

$$\sum_{env \in S} \min\left(N, |Nodes(env)|\right) \leq |Nodes| - K - \left\lceil \frac{\sum TestConfSetupTime + \sum TestExecutionTime}{coolDownPeriod} \right\rceil \times scalingStep \qquad (1)$$

- If resources permit the big flip can always be used. However, it is said to be applicable when the sum of the snapshot time, clone time, and service relocation time is less than the acceptable outage. It is also preferred (in comparison with other methods) when the snapshot time or the service relocation time is more than the tolerance time of at least one dependent.

<u>*6.2.5.2. Algorithm*</u>

The test method selection is an important activity of the test plan generation because the decisions made during this activity impact the disturbance induced by the test plan execution. On the one hand, the test method selection impacts the number of test runs that can be executed simultaneously which impacts the time it takes to execute the test plan. On the other hand, it impacts the number of service relocations for each CI which impacts the level of disruption induced by the test plan execution.

In order to keep the disruption acceptable and reduce the execution time, a test method can be selected for a given CI as follows:

- If only one test method is applicable (based on the applicability check), select that test method.
- If more than one test method is applicable, the precedence of test methods is single step, big flip, small flip, and last is the rolling paths.
- Conflicts (e.g. due to resource needs) between two CIs are solved by setting the preferred test method for the CI with the bigger number of mixtures.

Algorithm 2 enables the test method selection according to these rules. For each TSI group, it initializes the test method assignment to the empty set (Line 8). Then it assigns a test method to each CI to which only one test method is applicable (lines 9-12). It then sorts the remaining CIs in decreasing order of their mixture size (i.e. number of mixtures) (Line 14). It iterates through this sorted set of CIs and from the applicable test methods of each CI it assigns the most preferred one (lines 16-17, 19-20, 23-24, 27) and updates the set of available resources as it is used in subsequent iterations to select the test methods for the remaining

CIs. The getApplicableTM(…) function performs the applicability check to obtain the set of test methods that are applicable to a CI.

Algorithm 2: Test method selection

```
TSI_G: Test suite items groupings;
Is_M: Isolation matrix;
A_O: Acceptable outage;
SysConf: System configuration;
CI_TM: Test method assignment;
foreach g in TSI_G do
    Available_Resources = getAvailableResources(SysConf);
    CI_TM.put(g.path,{});
    for ci in g.path do
        if getAplicableTM(Is_M,A_O,ci,SysConf).size()==1 then
            CI_TM.get(g.path).add((ci,getAplicableTM(Is_M,A_O,ci,SysConf).first()));
            updateAvailableResources(CI_TM,Available_Resource);
    g.path.sortByMixtureSize();
    for ci in g.path do
        if getAplicableTM(Is_M,A_O,ci,SysConf).size()>1 and getAplicableTM(Is_M,A_O,ci,SysConf).size().contains(SingleStep) then
            CI_TM.get(g.path).add((ci,SingeStep);
            continue;
        if getAplicableTM(Is_M,A_O,ci,SysConf).size()>1 and getAplicableTM(Is_M,A_O,ci,SysConf).size().contains(BigFlip) then
            CI_TM.get(g.path).add((ci,BigFlip);
            updateAvailableResources(
            CI_TM,Available_Resource);
            continue;
        if getAplicableTM(Is_M,A_O,ci,SysConf).size()>1 and getAplicableTM(Is_M,A_O,ci,SysConf).size().contains(SmallFlip) then
            CI_TM.get(g.path).add((ci,SmallFlip);
            updateAvailableResources(
            CI_TM,Available_Resource);
            continue;
        CI_TM.get(g.path).add((ci,RollingPaths);
        updateAvailableResources(CI_TM,Available_Resource);
```

6.2.5.3. Evaluation

The goal of test plan generation is to design a test plan that has a reduced execution time while maintaining the disturbance acceptable. The test configurations deployment effort is the only aspect on which a test plan designer can act in order to achieve this objective. The precedence order among test methods and the strategy that we have adopted in our algorithm to solve conflicts, both compose the set of actions taken in our test method selection in order to reduce the test configurations deployment effort. Such reduction manifests as a reduced number of instantiations/removals of the components under test; and, a leveraged use of parallelism as compared to other alternatives for designing the test plan. As a result, the goal of this evaluation is to answer two research questions:

- **RQ1**: Does the precedence order we use in our test method selection help reduce the test configurations deployment time?
- **RQ2**: How good is the solution obtained by using our strategy for conflict resolution?

Table 6-2. Number of instantiations, removals, and service relocations per test method

Test method	Number of instantiations	Number of service relocations	Number of removals
Big flip	$\#CUTs$	$iterationCount$	$\#CUTs$
Small flip	$\#CUTs$	$iterationCount$	$\#CUTs$
Rolling paths	$iterationCount$	$iterationCount$	$iterationCount$

Table 6-2 shows the number of instantiations, removals, and service relocations needed to deploy the test configurations using each of the test methods we have proposed, these numbers are given in terms of the number of iterations and the number of components under test (CUTs) needed to deploy the test configurations. Table 6-3 gives the formulas for the number of iterations in terms of the number of test configurations when each test method is used. Finally, Table 6-4 gives the number of test configurations depending on the coverage criteria associated with each TSI.

Table 6-3. Iteration count per test method in terms of the number of test configurations

Test method	$iterationCount$
Big flip	$\dfrac{\#testConfigurations}{min_{i,\, CI\ in\ call\ path}(\frac{\#comp_i}{mw_i})}$
Rolling paths	$\#testConfigurations$
Small flip	$\dfrac{\#testConfigurationsOfFirstBatch}{min_{i,\, CI\ in\ call\ path}(\frac{\#comp_i}{mw_i})} + \dfrac{\#testconfigurations - \#testConfigurationsOfFirstBatch}{min_{i,\, CI\ in\ call\ path}(\frac{\#comp_i}{mw_i})}$

In our test method selection algorithm, when more than one test method is applicable, we give precedence to big flip, then small flip and rolling paths is the last choice. From Tables

6-2, 6-3, and 6-4 we see that the big flip and the small flip have the same number of instantiations, removals, and service relocations while the rolling paths has more instantiations and removals than the other two test methods. Moreover, knowing that the big flip allows for

Table 6-4. Number of test configurations based on the environment coverage that was used

	All boundary environments mixtures paths	**All boundary environment mixtures**
$\#testConfigurations$	$\prod_{i,\, CI\ in\ call\ path} C^{mw_i}_{\sum_{i \in BEs} \min\,(\#BE_i, mw_i)}$	$max_{i,\, CI\ in\ call\ path}(C^{mw_i}_{\sum_{i \in BEs} \min\,(\#BE_i, mw_i)})$
$\#testConfigurationsOfFirstBatch$	$\prod_{i,\, CI\ in\ call\ path} C^{mw_i}_{\sum_{i \in S} \min\,(\#BE_i, mw_i)}$	$max_{i,\, CI\ in\ call\ path}(C^{mw_i}_{\sum_{i \in S} \min\,(\#BE_i, mw_i)})$

more parallelism than the small flip, as seen from Table 6-3, we can deduce that our precedence order helps reduce the test configurations' deployment time which answers RQ1.

When two CIs can be tested using more than one test method but only one of them at a time can have the preferred test method, our algorithm assigns the preferred test method to the CI with the biggest number of mixtures. To answer RQ2, we will go case by case (based on the environment coverage used) and evaluate how good the resulting solution is, we will show optimality in certain cases, and give upper bounds for how bad a solution can be in other cases whenever possible. To achieve this goal, we will reason only on cases where the conflict is between the big flip and rolling paths test methods. In fact, the big flip and small flip have the same number of instantiations and removals as we can see from Table 6-2, and this is the only relevant factor for test plan's execution time. Therefore, choosing one or another does not make a difference in execution time as much as it does in resource consumption or disruption.

Moreover, the results that will hold for one will also hold for the other. In the rest of this proof, we are reasoning only on a single instance of the test method selection problem. In other words, we are not addressing the decisions made for the whole test plan, we are focusing on the decisions made only for a single call path, i.e. one grouping. This is reasonable as the test method selection for one call path is totally independent of the test method selection for another call path as the testing in our test plans targets one call path at a time.

Let G be the number of instantiations/removals associated with the CIs being tested using the rolling paths test method in our solution. Let OPT be the number of instantiations associated with the CIs being tested using the rolling paths test method in the optimal solution.

Case 1: all boundary environment mixtures

In this case, G and OPT are given by the formula in the first row second column in Table 6-4 considering only the set of CIs being tested using the rolling paths test method and not all the CIs in the call path. We will use an exchange argument to show that our solution is optimal in this case.

Let's assume the solution obtained by our algorithm is not optimal and see what kind of change we can perform to improve it. Let CI_i be a CI that will be tested using the big flip in our solution, and CI_j a CI that will be tested using the rolling paths in our solution. The only situation in which we can improve our solution by changing the test methods of CI_i and CI_j is if:

- CI_j is the CI with the maximum number of mixtures amongst all the CIs being tested using the rolling paths in our solution.
- CI_i has a number of mixtures less than the number of mixtures of CI_j.
- There are enough resources to test using the big flip CI_j and all the CIs that have a number of mixtures bigger than the number of mixtures of CI_j.

Because of the third point such exchange is not possible. If the third condition was met, the big flip would have been chosen for CI_j as per our algorithm. As a result, any exchange can only make our solution worse. Therefore, our solution is optimal.

Case 2: all boundary environment mixtures paths

In this case, G and OPT are given by the formula in the first row first column in Table 6-4 considering only the set of CIs being tested using the rolling paths test method.

Lemma 6-2: given a situation when test method selection is to be performed, the number of CIs tested using the big flip in our solution is at most equal to the number of CIs tested using the rolling paths in the optimal solution.

Proof: we will go case by case:

- Case 1: for all the CIs for which a test method is to be chosen, the optimal solution and our solution use different test methods. In this case, the number of CIs tested using the big flip in our test method equals the number of CIs tested using the rolling paths in the optimal solution.

- Case 2: for some CIs, our solution and the optimal solution use the same test method. If we assume that the number of CIs being tested using the big flip in our test method is more than the number of CIs being tested using the rolling paths test method in the optimal solution, then to improve our solution we need to test more CIs using the rolling paths. This is not possible since our heuristic favors the big flip for CIs with a bigger number of mixtures. As a result, using the rolling paths more can only make our solution worse.

Definition 6-2: let CI_1, CI_2, ..., CI_n be a set of CIs with their associated numbers of mixtures $\#mx_1$, $\#mx_2$, ..., $\#mx_n$We define K such that: $\forall\, l \geq K, \forall i \in [|1,n|], \forall \left(\alpha_j\right)_{1\leq j\leq l} \in ([|1,n|]\backslash\{i\})^l,\ \#mx_i \leq \prod_{j=1}^{l} \#mx_{\alpha_j}$

In other words, K is the lowest number bigger than the maximum number of CIs that you can switch from the rolling paths to the big flip in exchange for one CI from the big flip to the rolling paths and improve G.

Let R be the number of CIs for which the rolling paths test method was chosen in our solution. Let B be the number of CIs for which the big flip was chosen in our solution. And let l be the number of CIs for which the rolling paths test method was chosen in the optimal solution. In the case when $l > K$, if we choose for each CI being tested using the rolling paths in our solution, K CIs from the ones being tested using the rolling paths in the optimal solution, and apply the inequality in Definition 6-2 while trying to diversify the CIs picked from the optimal solution as much as possible (using permutations), we can establish that

$$G \leq OPT^{\left\lceil \frac{R \times K}{l} \right\rceil}$$

Since we cannot know l without calculating the optimal solution, we will use B and Lemma 6-2 to identify when this inequality is satisfied. Table 6-5 captures the different upper bounds we can establish for our solution compared to the optimal solution. It is not obvious from the table but for $K = 2$ our solution is optimal. In fact, this can be proven using an exchange argument same as we did for the case of all boundary mixtures coverage. In some other cases, one can obtain a solution that is no worse than OPT^2 . In fact, if each CI in the call path cannot be collocated with any other CI in the call path, algorithms that are used for 2-approximations for the knapsack problem will yield such solutions.

Table 6-5. Upper bounds of our solution compared to the optimal solution

Case	**Upper bound**
$R > B \geq K$	OPT^{K^2}
$R < B \wedge B \geq K$	OPT^{K}
Other cases	The solution can be arbitrarily bad

6.2.6. Test runs ordering

The ordering of test runs has two goals: 1) ensuring that TSIs are invoked only when their preconditions are met, and 2) reducing the disturbance by reducing the impact of service relocations. In this section, we propose an algorithm for test runs ordering. We also formally prove that our algorithm yields the optimal solution with respect to the second goal of test runs ordering.

6.2.6.1. Algorithm

A test run is a combination of a TSI and a test configuration under which it will be invoked. The first goal of the test runs ordering is achieved by ordering the test runs based on the TSIs they invoke. The second goal is achieved by ordering the test configurations in a way that the more critical a CI is, the least service relocations it experiences. The ordering of test runs takes as input a test plan with test runs in arbitrary order and the test suite precedence matrix (l). This process performs the ordering using the following operators on the test plan:

- Ordering test runs by changing the order of invocation of TSIs within the same UTP TestCase.
- Ordering runs by changing the order of UTP TestCases to order test runs based on the test configurations they involve.
- Ordering test runs by changing the UTP TestCase within which the TSI is invoked.

The first step of ordering test runs activity, described below, helps to achieve the first goal. After this step is completed, a TSI – the following TSI – will be invoked under a given test configuration only after all TSIs – the leading TSIs – that should precede this TSI (as per the precedence matrix) have been invoked under that same configuration. Therefore, this step uses only the first and third operators and proceeds according to the following rules:

- If the subset of UTP TestCases in which the leading TSI is invoked includes the subset of UTP TestCases in which its following TSI is invoked, then order the invocations of the TSIs within the same UTP TestCase of the subset of UTP TestCases in which the following TSI is invoked in a way that the following TSI is always invoked after the leading TSI.
- If the subset of UTP TestCases in which the following TSI is invoked is a union of a subset of UTP TestCases in which the leading TSI is invoked and a subset of UTP TestCases in which the leading TSI is not invoked; then for the first subset of the union order the invocations of the TSIs within the same UTP TestCase in a way that the following TSI is invoked after the leading TSI, and follow up this subset with the second subset of the union in which the leading TSI is not invoked.
- If none of the above rules apply, the third operator is used to move invocations of the following TSI to the first UTP TestCase in which it can be invoked (as per the test configuration of that UTP TestCase) while maintaining the precedence constraint. Moving such invocation may lead to the violation of Equation 1 in case any of the CIs in

the max path is to be tested using small flip. When such a violation occurs, the test method of such CI will change from small flip to rolling paths.

To achieve the second goal of the ordering, the test runs need to be ordered based on the test configurations they involve to reduce the disturbance induced by service relocations. The solution proposed in this thesis is based on the following assumption: the more similar consecutive test configurations are, the less disruption the services endure. To assess similarities between test configurations, we represent them as assignments of mixtures to nodes along a call path. Therefore, the similarity between two test configurations is the number of nodes along a call path to which different mixtures are assigned in the two test configurations. The ordering of UTP TestCases that considers such similarities goes along the following line:

- For each call path, start from a random UTP TestCase as the current UTP TestCase.
- The next UTP TestCase should be the one most similar to the current UTP TestCase, i.e. the UTP TestCase that involves a test configuration that changes the least number of mixtures from the test configuration of the current UTP TestCase. If more than one UTP TestCases change the same number of mixtures of the test configuration of the current UTP TestCase, the TestCase that changes the less critical CIs is chosen, i.e. disturbing less critical CIs is preferred compared to disturbing more critical CIs.

Algorithm 3 can be used to order test runs according to the rules mentioned above. It first achieves the first goal of the ordering of the test runs by taking into consideration the precedence constraints (lines 4-31). Then it achieves the second goal of the ordering of the test runs by taking into consideration the test configurations the test runs involve (lines 33-43).

To achieve the first goal of test runs ordering, the first and third operators mentioned earlier are used. Lines 5-10 address situations in which the first operator is used to maintain precedence constraint by ordering TSI invocations within the same UTP TestCase. Lines 22-29 address situations in which the third operator is used in Line 25 to maintain the precedence constraints by ordering UTP TestCases in the UTP model. Lines 12-20 use both operators, the first operator in Line 14, and the third operator in line 16 in order to maintain the precedence constraints. Note that every time a precedence constraint is handled it is added to a set of constraints, toBeMaintained, as readjustments with respect to these constraints are needed every time test runs are moved around to satisfy a new constraint.

To achieve the second goal of test runs ordering, i.e. reducing the disturbance, Algorithm 3 sorts UTP TestCases that invoke the same set of TSIs based on the test configurations they involve. To do so, for each UTP TestCase tc in the UTP model, it starts first by finding the UTP TestCases that invoke the same set of TSIs as tc (Line 34). It places tc as the first UTP TestCase of that group, then places after tc the UTP TestCase that involves the configuration most similar to tc's configuration (lines 36-41). Every time a UTP TestCase is sorted it is removed from the set of UTP TestCases to be considered, this process keeps going until there are no more UTP TestCases to sort.

Algorithm 3: Test runs ordering

```
UTP_M: Initial UTP Model;
P_M: Precedence Matrix;
toBeMaintained = {};
foreach c in P_M do
    if UTP_M.TestCases.select(t |t.invokes(c.preceding)).includesAll(UTP_M.TestCases.select(t |t.invokes(c.following))) then
        for tc in UTP_M.TestCases.select(t |t.invokes(c.following)) do
            placeAfter(tc,c.following,c.preceding);
        readjust(UTP_M,toBeMaintained);
        toBeMaintained.add(c);
        continue;
    if UTP_M.TestCases.select(t |t.invokes(c.following)).includesAll(UTP_M.TestCases.select(t |t.invokes(c.preceding))) then
        for tc in UTP_M.TestCases.select(t |t.invokes(c.preceding)) do
            placeAfter(tc,c.following,c.preceding);
            for ftc in UTP_M.TestCases.select(t |t.invokes(c.following) and not t.invokes(c.preceding)) do
                placeAfter(UTP_M,ftc,tc);
        readjust(UTP_M,toBeMaintained);
        toBeMaintained.add(c);
        continue;
    if UTP_M.TestCases.select(t |t.invokes(c.following)).excludesAll(UTP_M.TestCases.select(t |t.invokes(c.preceding))) then
        for tc in UTP_M.TestCases.select(t |t.invokes(c.preceding)) do
            for ftc in UTP_M.TestCases.select(t |t.invokes(c.following)) do
                placeAfter(UTP_M,ftc,tc);
        readjust(UTP_M,toBeMaintained);
        toBeMaintained.add(c);
        continue;
tcs = UTP_M.TestCases;
foreach tc in tcs do
    toSort = UTP_M.TestCases.select(t |t.invokedTCs.forall(itc, tc.invokes(itc)));
    currentTc = tc;
    while toSort not Empty do
        next = MaxSimilar(currentTc.conf,toSort);
        placeAfter(UTP_M,currentTc,next);
        toSort.remove(currentTc);
        tcs.remove(currentTc);
        currentTc = next;
```

6.2.6.2. Formal proof of optimality of Algorithm 3

The goal of test runs ordering based on test configurations is to reduce disturbance. In this section, we prove that Algorithm 3 yields the optimal solution to achieve this goal by proving that:

- The solution obtained using Algorithm 3 reduces the number of service relocations configured instances endure, and
- in the solution obtained using Algorithm 3, more critical CIs endure fewer service relocations compared to less critical CIs.

Lemma 6-3: To move from one test configuration to another, it takes a service relocation of at least one CI being tested using rolling paths.

Lemma 6-4: Along a call path CIs being tested using big/small flip undergo one service relocation, CIs being tested using single step undergo zero service relocation, CIs being tested using rolling paths undergo a number of service relocations that depends on the ordering of test configurations.

In our proof we will first assume that we are testing a call path along which all CIs are to be tested using the rolling paths (because of **Lemma 6-4**), then we will generalize the result. To perform the first step, we will go case by case depending on the environment coverage criterion that was used:

- **Case#1: all boundary environments mixtures coverage**
    - When using this coverage criterion, each CI goes through at least its $\#BMxs$ (number of boundary environment mixtures).
    - The generation of test configurations is done by varying as many CIs as possible (the greedy algorithm mentioned in the test configuration generation subsection) until no CI has more mixtures to cover. As a result, the later test configurations are generated the more similar they are.
    - The number of test configurations generated is max ($\#BMxs$) as per Table 6-4
    - As a result, the algorithm will order the test configurations in reverse order of their generation.
    - And each CI will go through exactly $\#BMxs$ of service relocations which is the minimum.
- **Case#2: all boundary environments mixtures paths coverage**
    - The set of test configurations is the Cartesian product of the sets of boundary environment mixtures of the CIs along the path.
    - The algorithm we propose when used is equivalent to the following mapping:
        - Each CI is represented by a digit.

        - The most critical CI is the most significant digit (the one on the extreme left), and the least critical CI is the least significant digit (the one on the extreme right).
        - Each digit takes values in the range of $\#BMxs$ of its associated CI.
    - The output of the algorithm is the n-ary Gray code associated with this setting
    - Gray code is a single distance code.
    - As a result, going from one test configuration to another is done using a single service relocation (the minimum as per **Lemma 6-3**).
    - Moreover, because of how the CIs are prioritized (i.e. associated to digits).
    - The least critical CI will undergo $\frac{\#testconfigurations}{\#BMxs}$ service relocations.
    - The most critical CI will undergo its associated $\#BMxs$ of service relocations.

6.2.7. **Wrapup**

The wrapup activity helps complete the specification of the TestExecutinoSchedule. It takes as input the test objective (k), test runtime framework deployment cost (h), TSI-test runtime framework matrix (i), and the refined UTP model obtained from the test runs ordering activity. This activity starts first by adding the TestObjective to the UTP model, i.e. by creating the TestObjective model element and filling in its description attribute with the test objective given as input. Then it proceeds to choose the most suitable runtime framework deployment.

This is done first by identifying the runtime framework of the TSI from (i), then checking the deployment options of this runtime framework in (h) (whether the runtime framework can be deployed using a configuration manager, or using a VM image, or a container). Then the least disturbing option is chosen, and the order of precedence between the deployment options is container deployment, then VM deployment, then the deployment using a configuration manager when no other option is available. The wrapup activity also cleans up the UTP model from any UTP TestCase model elements that do not contain any test runs.

6.2.8. An illustrative example

We implemented a prototype of our approach for test plan generation. The implementation was done using the Epsilon family of languages. Each one of the activities outlined in Figure 6-1 is implemented as an Epsilon module. The test configuration generation, call path merging, test method selection, and the creation of initial UTP models are implemented using the Epsilon Object Language (EOL). The ordering of test runs is implemented using Epsilon Pattern Language (EPL). In this section, we will demonstrate the prototype through an illustrative example.

The system configuration we take as input for this example is shown in Figure 6-3. The system is composed of nine CIs, each one of them handling a certain number of SIs shown as smaller squares inside the big rectangle (four for CI5, two for CI2, CI3, CI7, and CI9, one for the rest of the CIs). The squares of SIs that contribute to the realization of the same requirement are shown with the same pattern as the requirement they help to realize, i.e. squares R1-R6. Some CIs depend on other CIs indicated by arrows (CI8 depends on CI7 for instance). From this configuration, one can also identify the boundary environments to consider. The boundary environments related to CI1, for instance, are Env1.1 (which has a component of CI1 collocated with a component of CI3), and Env1.2 (which has a component of CI1 collocated with a component of CI3 and a component of CI5). CI3 has the same set of boundary environments as CI1. For CI5, in addition to Env1.2, it has also another boundary environment Env5.2 (which has a component of CI5 collocated with a component of CI4 and a component of CI9). The CI call graph associated with this configuration is shown in Fig. 6. An edge between two CIs

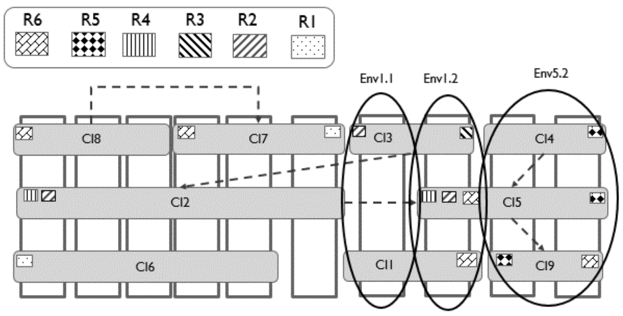


Figure 6-3. System Configuration of the illustrative example

indicates that the source CI provides at least one SI that depends on at least one SI of the target CI. It is taken as input in our prototype as an instance of the CI call graph metamodel. This metamodel allows capturing the concepts needed for a weighted directed graph.

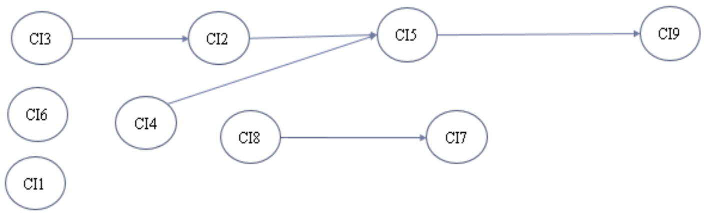


Figure 6-4. ConfiguredInstance call graph associated with the configuration of Figure 6-3

The TSI call path matrix is shown in Table 6-6. We associate each TSI with the set of SIs it traverses; from this information and the CI call graph, one can deduce to which call path each TSI applies. Certain TSIs apply to a single call path such as TC2 which applies only to the path CI3->CI2->CI5. Other TSIs may apply to more than one path such as TC1 which applies to CI8->CI7 and CI1. Such differences may arise when some TSIs aim to validate the service of a specific tenant and which realize a certain requirement (the case of TC2), while others aim to validate the realizations of a specific requirement for more than one tenant (the case of TC1). As a result, we need to append indexes to the TSI Ids to remove ambiguity (for

Table 6-6. The TSI call path matrix for the illustrative example

TSI	**call paths**
TC1	{CI8->CI7, CI1}
TC2	{CI3->CI2->CI5}
TC3	{CI4->CI5->CI9}
TC4	{CI2->CI5}
TC5	{CI4, CI5}

instance TC1-0 is the application of TC1 to CI8->CI7 and TC1-1 is the application of TC1 to CI1).

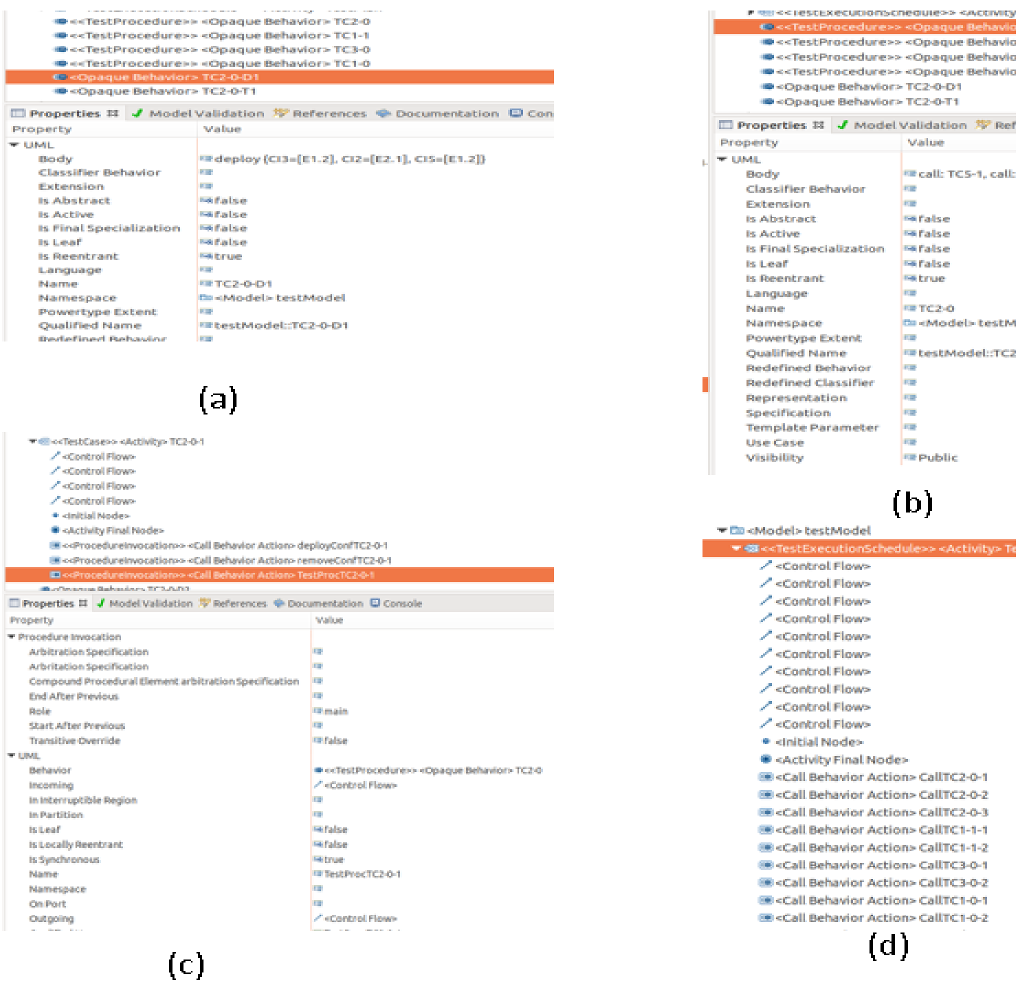
In this case study we consider a simple environment coverage case. The coverage

Table 6-7. Isolation matrix (time unit: seconds)

CI	**Risk**	**Snapshot**	**Clone**	**Load relocation**
CI1	1	0.3	1	0.001
CI2	1	0.7	1.2	0.01
CI3	0	0.4	1.3	0.03
CI4	0	10	10	5
CI5	1	5	5	3
CI6	0	10	9	13
CI7	1	4	5	3
CI8	0	1	1	1
CI9	1	0.1	0.1	0.1

criterion is the same for all the TSIs and it is the all boundary environments mixtures coverage. Moreover, we consider the case of mixtures of width one. As a result, the set of test configurations generated for each TSI should involve each mixture of width one (i.e. boundary environment) of each CI along the call path at least once. Table III shows an example of the isolation matrix. In this matrix, for each CI one can find in the first column whether the CI represents a risk of test interferences (1 means there is a risk of test interferences while 0 means there is no risk of test interferences), and, in the rest of the columns respectively, the time needed for snapshotting, cloning, and relocating the service. This information along with the acceptable outage guides the choice of the test method for each CI.

Figure 6-5 shows the results of running the test plan generation prototype. The model elements that are first created in the model are the OpaqueBehavior model elements. These elements can be either for deploying or for removing test configurations (non-stereotyped OpaqueBehavior model elements); or, for invoking TSIs in which case they will be stereotyped with the UTP stereotype TestProcedure. Taking into consideration the test runs needed for the test session, UTP TestCase model elements are created (as Activity elements stereotyped with TestCase). Each TestCase model element invokes other model elements in the following order: first, an OpaqueBehavior as setup (to deploy the test configuration), then an OpaqueBehavior stereotyped as TestProcedure as its main procedure to invoke the TSIs; and, finally it invokes an OpaqueBehavior as teardown to remove the test configuration. After the creation of TestCases, an Activity stereotyped as TestExecutionSchedule is created, and it invokes the created TestCases considering all the ordering constraints to respect. Figure 6-5(a) shows an OpaqueBehavior used to deploy a test configuration, which can be seen in the Body attribute of the OpaqueBehavior, i.e. "deploy {CI3:{E3.1}, CI2:{E2.1}, CI5:{E1.2}}". One can also notice how the groups manifest at the level of OpaqueBehavior elements that are stereotyped by TestProcedure. Figure 6-5(b) shows a TestProcedure that invokes the grouped TSIs TC5-1, TC4-0, and TC2-0. Activities stereotyped as TestCase are then created as shown in Figure 6-5(c) to capture the execution of a set of grouped TSIs under a specific test configuration. This grouping is done using CallBehaviorAction elements which invoke OpaqueBehaviors such as the ones shown in Figure 6-5(a) and Figure 6-5(b). Each invoked behavior has a role (setup, main, or teardown), taking into consideration that UTP only allows TestProcedures to be invoked as main. Finally, the TestCase elements are invoked within an activity stereotyped TestExecutionSchedule as shown in Figure 6-5(d).



```
<<TestProcedure>> <Opaque Behavior> TC2-0
<<TestProcedure>> <Opaque Behavior> TC1-1
<<TestProcedure>> <Opaque Behavior> TC3-0
<<TestProcedure>> <Opaque Behavior> TC1-0
<Opaque Behavior> TC2-0-D1
<Opaque Behavior> TC2-0-T1
Properties   Model Validation   References   Documentation   Con
Property                      Value
UML
  Body                        deploy (CI3=[E1.2], CI2=[E2.1], CI5=[E1.2])
  Classifier Behavior
  Extension
  Is Abstract                 false
  Is Active                   false
  Is Final Specialization     false
  Is Leaf                     false
  Is Reentrant                true
  Language
  Name                        TC2-0-D1
  Namespace                   <Model> testModel
  Powertype Extent
  Qualified Name              testModel::TC2-0-D1
  Redefined Behavior
```

(a)

```
<<TestExecutionSchedule>> <Activity
  <<TestProcedure>> <Opaque Behavio
  <<TestProcedure>> <Opaque Behavio
  <<TestProcedure>> <Opaque Behavio
  <<TestProcedure>> <Opaque Behavio
  <Opaque Behavior> TC2-0-D1
  <Opaque Behavior> TC2-0-T1
Properties   Model Validation   Ref
Property                      Value
UML
  Body                        call: TC5-1, call:
  Classifier Behavior
  Extension
  Is Abstract                 false
  Is Active                   false
  Is Final Specialization     false
  Is Leaf                     false
  Is Reentrant                true
  Language
  Name                        TC2-0
  Namespace                   <Model> testM
  Powertype Extent
  Qualified Name              testModel::TC2
  Redefined Behavior
  Redefined Classifier
  Representation
  Specification
  Template Parameter
  Use Case
  Visibility                  Public
```

(b)

```
<<TestCase>> <Activity> TC2-0-1
  <Control Flow>
  <Control Flow>
  <Control Flow>
  <Control Flow>
  <Initial Node>
  <Activity Final Node>
  <<ProcedureInvocation>> <Call Behavior Action> deployConfTC2-0-1
  <<ProcedureInvocation>> <Call Behavior Action> removeConfTC2-0-1
  <<ProcedureInvocation>> <Call Behavior Action> TestProcTC2-0-1
Properties   Model Validation   References   Documentation   Console
Property                                                    Value
Procedure Invocation
  Arbitration Specification
  Arbitration Specification
  Compound Procedural Element arbitration Specification
  End After Previous
  Role                                                      main
  Start After Previous
  Transitive Override                                       false
UML
  Behavior                                                  <<TestProcedure>> <Opaque Behavior> TC2-0
  Incoming                                                  <Control Flow>
  In Interruptible Region
  In Partition
  Is Leaf                                                   false
  Is Locally Reentrant                                      false
  Is Synchronous                                            true
  Name                                                      TestProcTC2-0-1
  Namespace
  On Port
  Outgoing                                                  <Control Flow>
```

(c)

```
<Model> testModel
  <<TestExecutionSchedule>> <Activity> Te
    <Control Flow>
    <Control Flow>
    <Control Flow>
    <Control Flow>
    <Control Flow>
    <Control Flow>
    <Control Flow>
    <Control Flow>
    <Control Flow>
    <Control Flow>
    <Initial Node>
    <Activity Final Node>
    <Call Behavior Action> CallTC2-0-1
    <Call Behavior Action> CallTC2-0-2
    <Call Behavior Action> CallTC2-0-3
    <Call Behavior Action> CallTC1-1-1
    <Call Behavior Action> CallTC1-1-2
    <Call Behavior Action> CallTC3-0-1
    <Call Behavior Action> CallTC3-0-2
    <Call Behavior Action> CallTC1-0-1
    <Call Behavior Action> CallTC1-0-2
```

(d)

Figure 6-5. Output UTP Model

6.3. Summary

The role of the Test Planner is to generate the test package that is composed of a test suite and a test plan. In this chapter, we presented how we automated these processes. We first presented how the Test Planner properly selects test cases to respond to various events. A system change is an important event as it triggers regression testing. Our method allows the use of any regression test case selection to respond to this event. Furthermore, our method includes a configuration-based approach to reduce the select set of test cases and focus the testing effort on finding configuration errors. Our test suite reduction method relies on a classification of configuration parameters into deployment environment dependent and deployment environment agnostic configurations parameters. Such classification allows the classification of requirements based on the configuration parameters involved in their realization. Such classification guides the selection of relevant test cases to use in regression testing. We have also proposed a fault model for configuration faults, and used it to semi-formally prove that our regression test suite reduction approach does not reduce the configuration fault detection power of the test suite. It is worth noting that our test suite reduction method is not safe, i.e. our method does not guarantee that for all the test cases in the reduced test suite, their pre-requisite test cases are in the reduced test suite as well. Moreover, this method was designed to reduce a test suite that is composed of test cases, its applicability to test suites that include online test methods cannot be confirmed and needs further investigations.

In addition to test design, test planning was yet another activity that we automated in this thesis. In this chapter, we presented our method for test plan generation. The goal of our method is to generate a test plan of which the execution takes minimal time without causing any intolerable disturbance. We based our method on the fact that for all test plans that achieve the same test objective, the time to execute tests is the same, and the only thing that makes the difference is the time for setting up and tearing down the test configuration. As a result, many activities in our solution such as the call paths merging and the test method selection aimed to reduce this value. Moreover, our solution for test runs ordering aimed to maintain the precedence between TSIs on one hand and reduce the time of going from one test configuration to the next on the other hand. We have proved formally that our algorithms for call paths merging and test runs ordering give the optimal solutions for the tasks they perform. The algorithm we have proposed for test method selection does not always yield the optimal solution, however, we have identified the cases when our solution is optimal and evaluated, depending on the situation, how bad our solution can be compared to the optimal solution. We have also implemented a Proof of Concept for our test plan generation method and showcased it on an illustrative example.

The test plan generation method we proposed relies on artifacts that are modeled using our modeling framework. As a result, our algorithms were defined at a high level of abstraction, that of UML, which makes them depend less on the technologies and platforms used to realize the system under test or used to develop the TSIs. Moreover, the use of model transformations allowed us to focus on the logic of our algorithms because the efficiency can improve as the transformation engines improve. Finally, the execution of the test plans we generate will operate on a batch of dependent call paths at a time, i.e. a call path and its sub-paths only. This design choice simplifies the problem and focuses the resource usage to find the best test isolation method for each CI in the call path. However, designing test plans of which the execution may target two independent call paths at a time is possible. Such a design has the potential to reduce the execution time further although it raises the number of conflicts as to resource allocation for test isolation. The advantages of such design as compared to ours were not looked into in this thesis.

Chapter 7

Conclusion and future work

Nowadays, services hosted in the cloud need to be tested in production. Such testing not only helps properly validate the code base of the software that provides the service but also helps detect errors related to the configurations. Automating the orchestration of testing activities in production and alleviating the risk of test interference are the two main pillars of any solution for live testing. Achieving these two capabilities in the cloud context is challenging. On the one hand, the diversity of technologies used in a cloud environment makes automation hard to achieve. On the other hand, the diversity of sources of software that compose a cloud system makes classic solutions for test isolation such as BIT inapplicable. Finally, the diversity of deployment/runtime environments in a cloud system makes test methods such as canary release as they are used now insufficient to properly test services offered by a cloud system although they can provide test isolation.

In this thesis, we dealt with these three challenges and proposed a solution for live testing. We proposed an architecture composed of two building blocks, the Test Planner and the Test Execution Framework, to automate testing activities in production. To play their roles in the architecture, these building blocks rely on various artifacts. System configuration contains information about the configuration of the system and its runtime configuration state. The events trigger test sessions. The test repository holds information about the test suite items and the risk of test interference associated with configured instances. Finally, the test package is an artifact that is automatically generated, and it is composed of the test suite and the test plan. To deal with the diversity of technologies used in a cloud system, we proposed extensions to UTP to make it suitable to our context and proposed the use of this extended version of UTP as a modeling framework for the artifacts involved in our architecture.

The Test Execution Framework is responsible for orchestrating and executing test runs in production. To achieve this goal, we proposed an execution semantics that the Test Execution Framework associates with relevant UTP model elements. This execution semantics helps track the progress of a test session and react to failures that may take place as the test runs are being executed. We also proposed a set of test methods that the Test Execution Framework uses to provide test isolation. These test methods rely on the ability of the environment to snapshot/clone components as well as the ability to relocate the load from one component to another. Therefore, they properly tackle the problems related to the diversity of sources of software that compose the cloud system as they provide test isolation without making any assumptions about the components targeted by the test. Moreover, these test methods also help deal with the challenges related to the diversity of deployment/runtime environments in the cloud as they allow testing under various test configurations.

The Test Planner is responsible for generating the test suite and test plan to be used in a test session. Given the role configurations play in the behavior and operation of cloud systems our test suite generation approach includes a configuration-based regression test suite selection solution. Such a solution helps focus the testing efforts toward finding configuration errors. Furthermore, we also proposed a solution to automate test plan generation. This includes automating, the selection of test configurations to be used in each test session as well as the ordering of test runs. Such ordering is based on the relationships between test suite items in terms of precedence on the one hand. On the other hand, it is based on the test configuration in order to reduce the disruption induced by the setting up of test configurations. Test plan generation also automates the selection of test isolation countermeasures to be used in each situation. To perform such selection, the test plan generation method requires information about the risk of test interference that the configured instances present when tested in production as well as the cost to implement test isolation for each configured instance. Such information can be obtained using the method we have proposed for configured instance evaluation.

As we have shown in this thesis, our test method selection approach does not always yield the optimal solution. Given the importance of this activity in test planning, a potential extension of this work would be to propose better solutions that outperform our algorithm in this activity. This activity depends heavily on the environment coverage criterion used to generate test configurations. Therefore, one can also propose other environment criteria that may seem more efficient in their use of the testing effort, i.e. help detect as many errors with a reduced set of test configurations. Test planning can also be improved by adapting the flow of the overall approach to new ways of running a test plan. In fact, in our approach we reasoned on executing one test plan at a time, i.e. our solution did not target having multiple test sessions at the same time. However, test objectives may change during the execution of a test plan, and some events may be more time sensitive than others as they should be handled within a time window. Therefore, one can adapt the execution semantics to deal with the challenges related to running multiple test plans simultaneously. Another extension that flows in the same direction would be to adapt the test plan generation approach to be able to take in new events and change the test plan on the fly as it is being executed.

When devising our test methods, we have focused only on operations performed at the level of virtual or logical resources. Snapshot, clone, and service relocation, are all operations performed on entities directly linked to a single service instance. Investigating the possibility of using operations on other kinds of resources is a track that can be pursued to improve existing test isolation techniques as well as come up with new ones. Such improvements can broaden the applicability of the test isolation techniques and open the door for new ways of designing test plans. One can also investigate how the use of the test isolation techniques presented in this thesis, and others, can serve other software engineering activities such as monitoring, maintenance, model inference, etc.

The configured instance evaluation method, although it has the potential to confirm the presence of test interferences, the search it performs is done in a brute force manner. As a result, it may take the generation of many scenarios to find a test interference revealing scenario. The search for scenarios can be guided using all kinds of techniques including the use of heuristics or evolutionary algorithms. Such improvements can reduce the number of scenarios needed to find a test interference revealing scenario, thus reducing the cost of this activity.

The solution we propose in this thesis for regression test suite reduction assumes the classification of configuration parameters is given as well as the mapping between the requirements and the configuration parameters involved in their realization. We did not provide any solution to automate the creation of such artifacts. In contexts in which configuration generation is automated, one can use the traces of such generation to create these artifacts. Methods related to trace analysis and machine learning based approaches have the potential to solve this problem.

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. In the proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference, EDOC'06, 2006, pp. 175-184.

[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. In the proceedings of the 30th Annual International Computer Software and Applications Conference, COMPSAC'06, 2006, pp. 171-176.

[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. In the proceedings of the 6th International Workshop on Software Engineering and Middleware, SEM'06, 2006, 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, Volume 122, pp. 1-28.

[6] Yardstick. https://wiki.opnfv.org/display/yardstick/Yardstick, Last visited, August 2022.

[7] Fortio operator. https://github.com/verfio/fortio-operator, Last visited, August 2022.

[8] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl. Holistic Configuration Management at Facebook. In the proceedings of the 25th Symposium on Operating Systems Principles, SOSP'15, 2015, pp. 328-343.

[9] X. Qu, M. Acharya, B. Robinson. Impact Analysis of Configuration Changes for Test Case Selection. In the proceedings of the IEEE 22nd International Symposium on Software Reliability Engineering, ISSRE'11, 2011, pp. 140-149.

[10] Z. Dong, A. Andrzejak, K. Shao. Practical and accurate pinpointing of configuration errors using static analysis. In the proceedings of IEEE International Conference on Software Maintenance and Evolution, ICSME'15, 2015, pp. 171-180.

[11] S. Zhang, M. D. Ernest. Automated diagnosis of software configuration errors. In the proceedings of the 35th International Conference on Software Engineering, ICSE'13, 2013, pp. 312-321.

[12] B. Robinson, L. White. Testing of User-Configurable Software Systems Using Firewalls. In the proceedings of the IEEE 19th International Symposium on Software Reliability Engineering, ISSRE'08, 2008, pp. 177-186.

[13] J. Lee, S. Kang, D. Lee. A survey on software product line testing. In the proceedings of the 16th International Software Product Line Conference, SPLC'12, 2012, pp. 31-40.

[14] V. Stricker, A. Metzger, K. Pohl. Avoiding Redundant Testing in Application Engineering. In the proceedings of the 14th International Software Product Line Conference, SPLC'10, 2010, pp. 226-240.

[15] A. Reuys, S. Reis, E. Kamsties, K. Pohl. The ScenTED Method for Testing Software Product Lines. In Software Product Lines, Springer, 2006, pp: 479-520.

[16] A P. M. S. Neto, I. C. Machado, Y. C. Cavalcanti. A Regression Testing Approach for Software Product Lines Architectures. In the proceedings of the 4th Brazilian Symposium on Software Components, Architectures and Reuse, 2010, pp. 41-50

[17] L. Keller, P. Upadhyaya, G. Candea. ConfErr: A Tool for Assessing Resilience to Human Configuration Errors. In the proceedings of the IEEE International Conference on Dependable Systems and Networks, DSN'08, 2008, pp. 157-166.

[18] B. Robinson. A firewall model for testing user-configurable software systems. PhD thesis. Case Western Reserve University. 2008.

[19] A. Andrzejak, G. Friedrich, F. Wotawa. Software Configuration Diagnosis – A Survey of Existing Methods and Open Challenges. In the proceedings of the 20th Configuration Workshop, CONFWS'18, 2018, pp. 85-92.

[20] T. Wang, X. Liu, S. Li, X. Liao, W. Li. Q. Liao. MisconfDoctor: Diagnosing Misconfiguration via Log-based Configuration Testing. In the proceedings of the IEEE International Conference on Software Quality, Reliability and Security, QRS'18, 2018, pp. 1-12.

[21] O. Jebbar, M. A. Saied, F. Khendek, M. Toeroe. Poster: Re-Testing Configured Instances in the Production Environment - A Method for Reducing the Test Suite. In the proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification, ICST'19, 2019, pp. 367-370.

[22] A. Bertolino, B. Miranda, R. Pietrantuono, S. Russo. Adaptive coverage and operational profile-based testing for reliability improvement. In the proceedings of the 39th International Conference on Software Engineering, ICSE'17, 2017, pp. 541-551.

[23] M. Ozawa, T. Dohi, H. Okamura. How Do Software Metrics Affect Test Case Prioritization? In the proceedings of the IEEE 42nd Annual Computer Software & Applications Conference, COMPSAC'18, 2018, pp. 245-250.

[24] E. Engström, P. Runeson. A Qualitative Survey of Regression Testing Practices. In the proceedings of Product-Focused Software Process Improvement, PROFES'10, 2010, LNCS, vol. 6156, pp. 3-16.

[25] M. J. Harrold, R. Gupta, M. L. Soffa, A methodology for controlling the size of a test suite, ACM Transactions on Software Engineering and Methodology, 1993, pp. 270-285.

[26] G. Rothermel, M. J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In the proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA'94, 1994, pp. 169-184.

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

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

[29] Ansible. https://www.ansible.com, Last visited, August 2022.

[30] Puppet. https://www.puppet.com, Last visited, August 2022.

[31] Chef. https://www.chef.io, Last visited, August 2022.

[32] Docker. https://www.docker.com, Last visited, August 2022.

[33] Kubernetes. https://www.kubernetes.io, Last visited, August 2022.

[34] E. M. Fredericks, B. H. C. Cheng, Automated Generation of Adaptive Test Plans for Self-Adaptive Systems, In the proceedings of the IEEE/ACM 10th International Symposium on

Software Engineering for Adaptive and Self-Managing Systems, SEAMS'1, 2015, pp. 157-167.

[35] 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'11, 2011, pp. 582-587.

[36] T. Cao, P. Felix, R. Castanet, I. Berrada, Online Testing Framework for Web Services, In the proceedings of the 3rd IEEE Conference on Software Testing, Validation and Verification, ICST'10, 2010, pp. 363-372.

[37] K. Lee, L. Sha, A Dependable Online Testing and Upgrade Architecture for Real-Time Embedded Systems, in the proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA'05, 2005, pp. 160-165.

[38] 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'03, 2003, pp. 177-184.

[39] 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'10, 2010, pp. 36-42.

[40] X. Bai, D. Xu, G. Dai, Dynamic Reconfigurable Testing of Service-Oriented Architecture, in the proceedings of the 31st Annual International Computer Software and Applications Conference, COMPSAC'07, 2007, pp. 368-378.

[41] M. Elqortobi, J. Bentahar, R. Dssouli, Framework for Dynamic Web Services Composition Guided by Live Testing, in the proceedings of the International Conference on Emerging Technologies for Developing Countries, AFRICATEK'17, 2017, LNICST, vol. 206, pp. 129-139.

[42] G. Schermann, D. Schoni, P. Leitner, H. C. Gall, Bifrost – Supporting Continuous Deployment with Automated Enactment of Multi-Phase Live Testing Strategies, in the proceedings of the 17th International Middleware Conference, Middleware'16, 2016, pp. 1-14.

[43] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, V. Sekar, Gremlin: Systematic Resilience Testing of Microservices, in the proceedings of the IEEE 36th International Conference on Distributed Computing Systems, ICDCS'16, 2016, pp. 57-66.

[44] Netflix – Simian Army, https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey. Last visited, August 2022.

[45] D. G. Feitelson, E. Frachtenberg, K. L. Beck. Development and Deployment at Facebook. IEEE Internet Computing, 2013, vol. 17, no. 4, pp. 8-17.

[46] Blue-green. https://docs.cloudfoundry.org/devguide/deploy-apps/blue-green.html. Last visited, August 2022.

[47] 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, 2014, vol. 47, no. 2, pp. 23-29.

[48] T. Xu, O. Legunsen, Configuration Testing: Testing Configuration Values as code and With Code. arXiv:1905.12195

[49] O. Tuncer, A. Byrne, N. Bila, S. Duri, C. Iski, A. K. Cuskun, ConfEx: A Framework for Automating Text-based Software Configuration Analysis in the Cloud. arXiv:2008.08656

[50] M. Leotta, D. Clerissi, L. Franceschini, D. Olianas, D. Ancona, F. Ricca, M. Ribaudo, Comparing Testing and Runtime Verification of IoT Systems: A Preliminary Evaluation based on a Case Study. In Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE'19, 2019, pp. 434-441.

[51] K. Fogen, H. Lichter, Combinatorial Robustness Testing with Negative Test Cases, in the Proceedings of the IEEE 19th International Conference on Software Quality, Reliability, and Security, QRS'19, 2019, pp. 34-45.

[52] V. Costa, G. Girardon, M. Bernardino, R. Machado, G. Legramante, A. Neto, F. P. Basso, E. M. Rodrigues, Taxonomy of Performance Testing Tools: a Systematic Literature Review, in the proceedings of the 35th Annual ACM Symposium on Applied Computing, SAC'20, 2020, pp. 1997-2004.

[53] Containerd. https://containerd.io/. Last visited August, 2022.

[54] T. Chiba, R. Nakazawa, H. Horii, S. Suneja, S. Seelam, ConfAdvisor: A Performance-centric Configuration Tuning Framework for Containers on Kubernetes. In the Proceedings of IEEE International Conference on Cloud Engineering, IC2E'19, 2019, pp. 168-178.

[55] L. Bulej, V. Horky, P. Tuma, Initial Experiments with Duet Benchmarking: Performance Testing Interference in the Cloud. In the Proceedings of the IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS'19, 2019, pp. 249-255.

[56] C. Xiang, H. Huang, A. Yoo, Y. Zhou, S. Pasupathy, PracExtractor: Extracting Configuration Good Practices from Manuals to Detect Server Misconfigurations. In the Proceedings of the USENIX Annual Technical Conference, USENIX ATC'20, 2020, pp. 265-280.

[57] R. Fonseca, M. J. Freedman, G. Porter, Experiences with tracing causality in networked services. Internet Network Management Workshop/ Workshop on Research on Enterprise Networking, INM/WREN'10, 2010.

[58] J. Mace, P. Bodik, R. Fonseca, M. Musuvathi, Retro: Targeted Resource Management in Multi-tenant Distributed Systems, in the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI'15, 2015, pp. 589-603.

[59] J. Mace, R. Roelke, R. Fonseca, Pivot tracing: dynamic causal monitoring for distributed systems, in the Proceedings of the 25th Symposium on Operating Systems Principles, SOSP'15, 2015, pp. 378-393.

[60] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, C. Shanbhag, Dapper, a Large-Scale Distributed Systems Tracing Infrastructure, Google Technical Report dapper-2010-1, 2010

[61] H. Liu, J. Zhang, H. Shan, M. Li, Y. Chen, X. He, X. Li, JCallGraph: Tracing Microservices in Very Large Scale Container Cloud Platforms, in the Proceedings of the International Conference on Cloud Computing, CLOUD'19, LNCS, vol. 11513, pp. 287-302.

[62] H. Lu, A. Srivastava, Y. Sun, ShadeNF: Testing Online Network Functions, in the Proceedings of IEEE International Conference on Cloud Engineering, IC2E'19, 2019, pp. 128-138.

[63] https://status.cloud.google.com/incident/compute/16012. Last accessed June, 18th, 2020.

[64] https://status.cloud.google.com/incident/compute/15046. Last accessed June, 18th, 2020.

[65] https://status.cloud.google.com/incident/appengine/19001. Last accessed June, 18th, 2020.

[66] https://status.cloud.google.com/incident/appengine/16002. Last accessed June, 18th, 2020.

[67] https://status.cloud.google.com/incident/cloud-networking/18012. Last accessed June, 18th, 2020.

[68] November, 20th, 2019 incident. https://status.azure.com/en-us/status/history/. Last accessed June, 18th, 2020.

[69] LXC. https://linuxcontainers.org/. Last visited, August 2022.

[70] Openstack. https://www.openstack.org/. Last visited, August 2022.

[71] CRIU. https://criu.org/Main_Page. Last visited, August 2022.

[72] O. Jebbar, M. A. Saied, F. Khendek, M. Toeroe, Regression Test Suite Reduction for Cloud Systems. In Proceedings of the IEEE 13th IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW'20, 2020, pp. 477-486.

[73] O. Jebbar, F. Khendek, M. Toeroe, Methods for Live Testing of Cloud Services, In Proceedings of the IFIP International Conference on Testing Software and Systems, ICTSS'20, 2020, LNCS, vol. 12543, pp. 201-216.

[74] O. Jebbar, F. Khendek, M. Toeroe, Architecture for the Automation of Live Testing of Cloud Systems, in the Proceedings of the IEEE 20th International Conference on Software Quality, Reliability and Security, QRS'20, 2020, pp. 142-151.

[75] S. Dathathraya, M. Stevens, System and method for generating automatic test plans, U.S Patents: US20050172270A1.

[76] E. B. Lewis, Technique for automatically generating a software test plan, U.S patents: US6546506B1.

[77] J. G. Becker, K. L. McClamroch, V. Raghavan, P. Sun, Automated test execution plan generation, U.S patents: US8423962B2.

[78] M. Jibbe, Method and system for generating a global test plan and identifying test requirements in a storage system environment, U.S patents: US20070079189A1.

[79] ISO/IEC/IEEE 29119-1, Software and systems engineering – Software testing – Part 1: Concepts and definitions, First edition, 2013.

[80] ISO/IEC/IEEE 29119-2, Software and systems engineering – Software testing – Part 2: Test processes, First edition, 2013.

[81] Weave scope. https://www.weave.works/oss/scope/. Last visited, August 2022.

[82] P. Alvaro, K. Kingsbury, Elle: inferring isolation anomalies from experimental observations, arXiv:2003.10554

[83] B. Danglot, P. Preux, B. Baudry, M. Monperrus, Correctness attraction: a study of stability of software behavior under runtime perturbation, Empir Software Eng 23, 2086–2119 (2018).

[84] L. Morell, B. Murrill, and R. Rand. Perturbation analysis of computer programs. In the proceedings of the 12th Annual Conference on Computer Assurance, COMPASS'97, 1997, pp. 77-87.

[85] S. Tallam, C. Tian, R. Gupta, and X. Zhang. Avoiding program failures through safe execution perturbations. In the proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference, COMPSAC '08, 2008, pp. 152–159.

[86] N. Wang, M. Fertig, and S. Patel. Y-branches: when you come to a fork in the road, take it. In the proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques, PACT'03, 2003, pp. 56–66.

[87] Gym. https://github.com/intrig-unicamp/gym. Last visited, November 2021.

[88] Tsung. http://tsung.erlang-projects.org/. Last visited, November 2021.

[89] WapIML. https://github.com/opendata-for-all/wapiml. Last visited, November 2021.

[90] Kubernetes with pod migration. https://github.com/SSU-DCN/podmigration-operator. Last visited, August 2022.

[91] O. Jebbar, F. Khendek, M. Toeroe, A Method for Component Evaluation for Live Testing of Cloud systems. In the proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops, ISSREW'22, 2022, pp. 87-92.

[92] O. Jebbar, F. Khendek, M. Toeroe, Test Plan Generation for Live Testing. Submitted to the Journal of Systems & Software.