Regression Test Suite Reduction for Cloud Systems

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

Mohamed Aymen Saied
Computer Science and Software Engineering
Laval University
Quebec, Canada
mohamed-aymen.saied@ift.ulaval.ca

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

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

***Abstract*—Cloud providers offer a wide variety of services to their tenants. Providers share large scale infrastructures to host their services and use configurable software customized with configurations to meet different tenant requirements. These configurations are often the main source of errors. Moreover, they undergo frequent changes, therefore, systems' compliance to requirements needs to be re-evaluated frequently using regression testing. The problem of regression test case selection has been extensively addressed in the literature, however, existing approaches do not tackle the problem from the configuration perspective. In this paper, we propose a configuration-based method for regression test suite reduction for cloud systems. Our method targets a set of faults summarized in a fault model, and it relies on a classification of configuration parameters based on their relation to the deployment environment. Our idea is that the relation of the configuration parameters to the environment can be explored to reduce the regression test suite.**

Keywords — cloud systems, configurable systems, configurations, regression testing, test suite reduction.

I. Introduction

Nowadays, service providers offer a wide variety of services to meet their tenants' needs. The software commonly used to provide such services is configurable as configurability helps in meeting with the same software different requirements for different tenants and for different environments, i.e. customizing the software, and in speeding up the time to market of services. Configurable software is customized into a configured instance through a configuration which meets specific tenant's requirements for a specific environment. From software perspective a cloud system can be seen as a set of configured instances and the cloud configuration in this case consists of the set of configurations of the configured instances. These configurations can be the main source of faults and cause of non-compliance with the tenants' requirements.

Cloud systems evolve and undergo frequent reconfigurations [1]. Reconfigurations include additions/removals of configured instances, or changes in the values of configuration parameters to fix a bug or fine tune the performance of a configured instance or the system. One or more configured instances may be impacted by a reconfiguration, e.g. in a single reconfiguration the admin may change the values of configuration parameters for different configured instances. After a reconfiguration, the cloud system should undergo regression testing to check that these changes have the desired effects and did not cause undesired ones. The literature proposes several strategies for regression test case selection. These strategies are either: 1) based on impact analysis [2] such as change based test case selection methods, 2) the operational profile such as test case prioritization [15, 16], 3) test case minimization methods, or 4) none of these methods and retest all.

The impact of a change performed on a given service can go beyond the changed service especially in the cloud. In fact, other services that either depend on the changed service or share resources with it can be impacted by the change as well. Performing regression testing on the premises of the service providers to cover all these bases is hard, costly, and sometimes unfeasible. On the other hand, duplicating the production environment to perform regression tests off the premises of the service provider is resource and time consuming. In addition, the changed service may depend on other external services that cannot be duplicated nor accessed from other environments. Thus, regression testing of cloud systems can only be performed properly when done in the production environment. Therefore, reducing further the regression test suite is important to limit the impact on the system in production.

In this paper we propose a method for reducing the regression test suite for cloud systems by focusing the testing efforts on finding configuration faults. The rationale behind this focus on configuration faults is that: 1) the configuration is the last artifact that was created, so it is the obvious starting point for finding errors; and, 2) the configuration is the artifact that is often accessible to the service provider unlike the code of the configurable software. Our method is based on a classification of configuration parameters into environment dependent and environment agnostic configuration parameters [14]. Using this classification and the relationships between the configuration parameters and the requirements we select (from the regression test suite) the test cases to run after a reconfiguration to detect configuration faults, which we characterize and summarize in a configuration fault model. Our method is impact analysis agnostic and can be combined with any impact analysis method or regression test case selection method. We provide a semi-formal proof of our method to demonstrate its capability to detect any configuration/reconfiguration fault in the proposed fault model.

The rest of this paper is organized as follows. In Section 2 we provide some background on configurable systems and cloud. In Section 3 we describe in details our method before illustrating it with an example in Section 4. We present a semi-formal proof for its configuration fault detection power in

Section 5. We review related work in Section 6 before concluding.

II. BACKGROUND

In this section we provide some background knowledge about configurable systems, configurations and cloud systems. For illustration, we will use the example of Fig. 1.

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**. To meet each tenant requirement the cloud system realizes it as a **service instance** resulting in as many **service instances** as there are

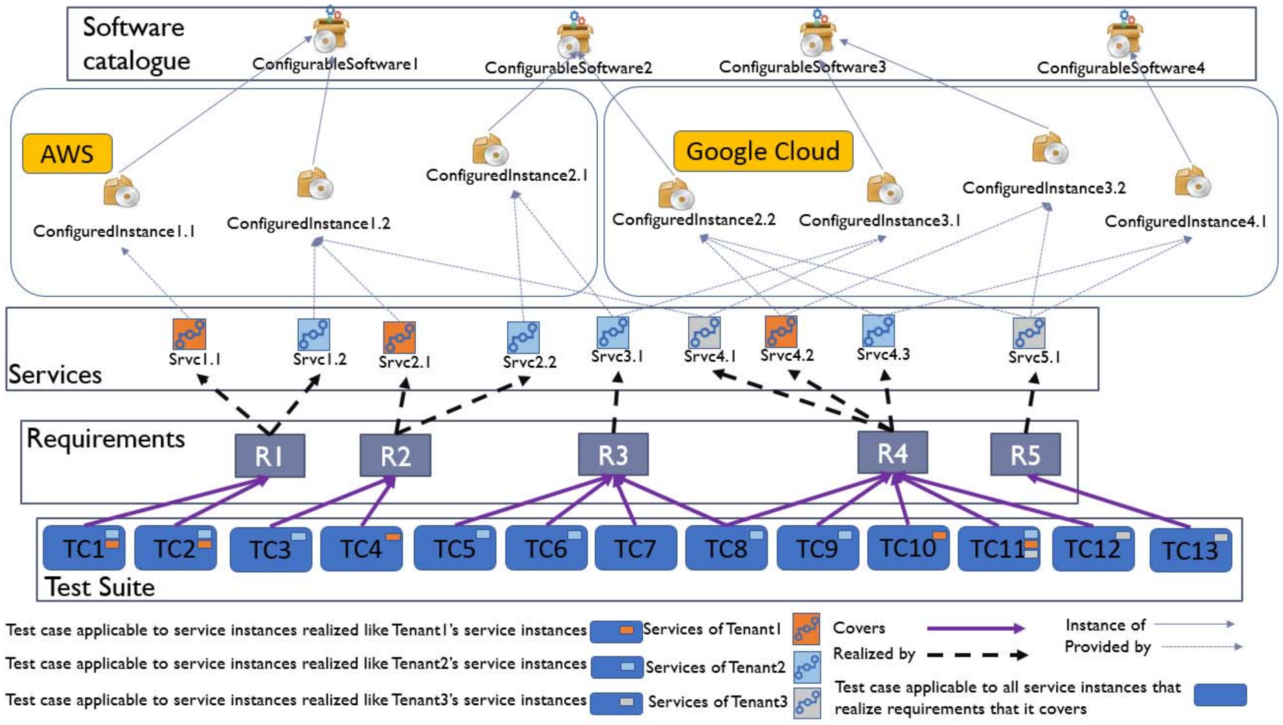

Fig. 1. Example of cloud system

A. *Configurable systems and configurations*

To accommodate various tenants' requirements, service providers use **configurable software**. A **configurable software** is a software that cannot be used without a **configuration**. A **configuration** is a set of (key, value) pairs in which the keys represent parameters of the software's functions. These parameters have values, which 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**.

When a new tenant requests a functionality (functional requirement) with certain characteristics (non-functional requirements), the service provider may create new **configured instances** (e.g. in single tenancy architectures as Configured instance 1.1 and Configured instance 1.2 in Fig. 1.) to satisfy the new requirements or may use an already deployed **configured instance** (e.g. in multi tenancy architectures as Configured instance 2.2 is shared between tenants in Fig. 1.). A **configured instance** is a system resulting from configuring a **configurable software**. A **configured instance** is used to satisfy a refined subset of the requirements (reflecting the purpose of the **configured instance**) towards the **configurable software**. Unlike the **configurable software**, a **configured instance** is ready for use. The set of **configured instances** that a service

tenant requirements (e.g. in Fig. 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 Fig. 1.) or more **configured instances** (e.g. Srvc2.1 of Fig. 1.). Moreover, **service instances** that realize the same requirements for different tenants, may or may not be provided using **configured instances** of the same **configurable software**. In other words, a tenant may have his **service instance** for requirement R provided using a **configured instance** of **configurable software** CS1, while another tenant has his **service instance**, which realizes the same requirement R using a **configured instance** of **configurable software** CS2 (Requirement R3 of Fig. 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 environments.

A **configuration** is used to realize the 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 may need more than one **configuration** to realize the requirements.

In a single tenant architecture, the **application configuration** is used to realize the requirements of the tenant. In a multi-tenant architecture, one **configured instance** may be shared between multiple tenants. Therefore, we end up with an **application configuration** which is processed by the **configured instance** and shared between all the tenants that are using this instance; and, a **tenant configuration** which is managed by the **configured instance** and specific to each tenant.

Table I illustrates various requirement refinements for an e-commerce application and the **configuration** settings to satisfy these requirements. In the first case, the **configuration** controls the presence/absence of features of the **configurable software** in the **configured instance**. The second and third cases are concretizations of generic features of the **configurable software**. 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).

TABLE I. EXAMPLES OF REFINEMENT OF CONFIGURABLE SOFTWARE'S REQUIREMENTS AND THEIR MANIFESTATION IN THE CONFIGURATION [14]

	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)

The settings in a **configuration** may be straightforward (e.g. the function refinement in the second case of Table I), 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). Restarting the **configured instances** is often required to put **application configurations** into effect. The same does not usually apply to **tenant configurations** as they need to change dynamically.

A **configuration** can also be a **deployment configuration** managed by a third-party software for other purposes such as load balancing (HA Proxy), traffic routing (Ingress), failover (Kubernetes), etc. This type of **configuration** is often dynamic, i.e. it can be changed without a restart of the software itself or the third-party software processing them. The different **configurations** (tenant, application, and deployment configurations) yield different **configured instances** even if the same **configurable software** is used to realize the different requirements.

Configurations undergo changes for different reasons such as to maintain an existing requirement, which has already been satisfied, or to satisfy a new requirement. Since a **service instance** is provided using one (e.g. Srvc 1.1 in Fig. 1.) or more (e.g. Srvc 3.1 in Fig. 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**. In the rest of the paper, we will say that a **service instance** is **“reconfigured”** to refer to a change in any of the **configurations** involved in its provisioning. In addition, we will say that **two service instances are 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 to the requirements of the **service instances**, service providers also design test cases to perform acceptance tests on their cloud systems. The test cases in the acceptance test suite cover all the requirements that should be fulfilled by the **cloud system**. A given test case 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 can be provided using **configured instances** of different **configurable software**, the service provider may have different test cases 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 Fig. 1 a test case may be applicable to a **service instance** only if provided using **configured instances** of specific **configurable software** (e.g. in Fig. 1. TC7 and TC8). A test case may also be applicable to all **service instances** realizing a requirement the test case covers, such as TC5 in Fig. 1.. Such test cases may be test cases used to ensure the compliance of the behavior or interface of the **service instance** to a given standard. A test case may cover one or many requirements. TC1 for instance covers one requirement R1, and TC8 covers multiple requirements R2 and R3.

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 acceptance test suite that is used to validate the cloud system. Selecting the regression test cases can be done using different strategies:

- Retest all [17]: in this case all the test cases in the acceptance test suite will be in the regression test suite.
- Change based test case selection [19]: in this case only impacted configured instances will be subject to tests. Several impact analysis methods were proposed in the literature.
- Test case prioritization [15, 16]: in this case the whole acceptance test suite 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 [18]: in this case the same subset of test cases is used after each reconfiguration.

B. *Configuration Parameters Classification*

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 software** need to be refined in certain ways. The differences between **configurations** in this case are a result of deployments in different environments (operating systems, libraries, network, etc.) on one hand as mentioned in [12]; 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 in consideration, in a previous work [14] we proposed the following classification for **configuration parameters**:

- **Environment agnostic configuration parameter**: a configuration parameter which is independent from the 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.
- **Environment dependent configuration parameter**: a configuration parameter for which the value, to satisfy a requirement, depends on the environment of the configured instance.

III. REGRESISON TEST SUITE REDUCTION METHOD

A. *Fault Model*

Configuration designers may make various types of faults. Misconfigurations for instance are the type of faults which is addressed the most in the literature [3, 4, 10, 13]. 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), 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 work we assume that the configuration is valid, and we question the compliance of the cloud system to its 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, 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. cannot 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.

B. 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 environment agnostic or environment dependent. In this classification the configuration parameters are referenced as IdOfProduct-NameOfParam (e.g. configurableSoftware1-confParam1, configurableSoftware 1-confParam2, configurableSoftware2-confParam1). 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. Note that this classification needs to be established only the first time the configurable software is used; and, it can be reused whenever it is needed as it does not depend on the test suite nor the requirements but only on the configurable software itself.

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), 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. In other words, it 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.

C. Classification of Requirements and Service Instances

From the relationships between the requirements and the configuration parameters (c), and the classification of configuration parameters (b), we classify the requirements as proposed in [14]:

- Environment agnostic requirements: they map to environment agnostic configuration parameters only, if any,

- Environment dependent requirements: they map to environment dependent configuration parameters only, and they are about system interfaces for interacting with the environment.

- Composite requirements: they map to environment agnostic as well as 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.

D. 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:
    - o 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
    - o 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:
    - o After applying the first two rules, it has no applicable test cases; and
    - o 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).

IV. ILLUSTRATIVE EXAMPLE

To illustrate our approach, we will use the system shown in Fig. 1. (as system configuration input (a)).

TABLE II. APPLICABILITY MATRIX OF THE ACCEPTANCE TEST SUITE IN FIG. 1.

	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9	TC10	TC11	TC12	TC13
Srvc1.1	X	X											
Srvc1.2	X	X											
Srvc2.1				X									
Srvc2.2			X										
Srvc3.1					X	X	X	X					
Srvc4.1											X	X	
Srvc4.2										X	X		
Srvc4.3								X	X		X		
Srcv5.1													X

Table II represents the applicability matrix (input (f)) that maps each test case to the service instances to which it applies. Table III represents the requirement-service matrix (input (e)) associated with this system.

For illustration purpose we use the following scenario:

- Configured instance1.1 and Configured instance2.2 were reconfigured (input (g)).
- The regression test case selection method used by the administrator consists of selecting test cases that cover at least one requirement realized using a service instance which was reconfigured; and, run these test cases on all the service instances that realize the requirement and to which they are applicable.
- Requirement R1 is environment agnostic, R5 and R3 are composite requirements, and R4 is environment dependent requirement (input (b)).

TABLE III. REQUIREMENT-SERVICE MATRIX OF THE SYSTEM IN FIG. 1.

	Srvc1.1	Srvc1.2	Srvc2.1	Srvc2.2	Srvc3.1	Srvc4.1	Srvc4.2	Srvc4.3	Srvc5.1
R1	X	X							
R2			X	X					
R3					X				
R4						X	X	X	
R5									X

- After the reconfiguration of Configured instance 1.1, configurations owned-by/associated-with Srvc1.1 and Srvc1.2 have the same values for configuration parameters that impact R1; and, TC1 and TC2 already passed for Srvc1.2 under its current configuration (relevant part of the history input (h)).
- Srvc4.3 and Srvc5.1 share configuration parameters which impact R4 and R5 respectively (summary of impact matrix, input (c)).
- For this scenario, the regression test suite will be composed of {TC1, TC2, TC8, TC9, TC10, TC11, TC12, TC13} (input (d)). Moreover, since the administrator runs the tests on each service instance to which they apply, the total number of runs would be 12 runs.

Taking into consideration this regression test suite, we can classify service instances as follows:

- Srvc1.1, Srvc4.2, Srvc4.3, and Srvc5.1 are directly impacted by the change.
- Srvc1.2, Srvc4.1, and Srvc3.1 are indirectly impacted by the change.
- The rest of the service instances are unimpacted by the change.

Using this classification, we classify the requirements as follows:

- R1, R4, and R5 are directly impacted by the change.
- R3 is indirectly impacted by the change.
- R2 is not impacted by the change.

Applying Rule #1, R2 will be removed from the requirement-service matrix. Applying Rule #2 on R1, no test case will be applicable on Srvc1.1 and Srvc1.2. Applying Rule #2 on R4, no test cases will be applicable on Srvc4.1, Srvc4.2, and Srvc4.3. R5 will remain as is since it is a composite requirement.

Rule #3 applies to Srvc1.1 and Srvc1.2, these two service instances will be removed from the applicability matrix.

Rule #4 applies to Srvc4.3, as a result this service instance will be removed from the applicability matrix.

Rule #5 will lead to Srvc4.2 and Srvc4.1 getting their applicability links back.

Finally, Srvc2.1 and Srvc2.2 will be removed from the applicability matrix as R2 was removed previously from the requirement-service matrix as a result of applying Rule #1.

After applying these rules, the applicability of the test cases in the regression test suite goes as follows:

- TC1 and TC2 are not applicable to any service instances as Srvc1.1 and Srvc1.2 were removed from the applicability matrix.
- TC9 is not applicable to any service instance as Srvc4.3 was removed from the applicability matrix.
- TC8 is applicable to Srvc3.1.
- TC10 is applicable to Srvc4.2, TC11 is applicable to Srvc4.1 and Srvc4.2, TC12 is applicable to Srvc4.1; Since Srvc4.1 and Srvc4.2 got their applicability links back as a result of applying Rule #5.
- TC13 is applicable to Srvc5.1.

The resulting reduced regression test suite will be composed of {TC8, TC10, TC11, TC12, TC13}, applicable only to Srvc5.1, Srvc3.1, Srvc4.1, and Srvc4.2. As a result, the total number of runs or tests to be performed is reduced to six.

TABLE IV. CLASSES OF REQUIREMENTS AND TYPES OF FAULTS THAT MAY CAUSE THEIR VIOLATION.

	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

V. SEMI-FORMAL PROOF

As a first step, we established, as shown in Table IV, the link between the classes of requirements and 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 environment dependent configuration parameters and therefore can impact either environment dependent or composite requirements. Environment dependent requirements are by definition impacted by 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. Dimensioning faults can only lead to a violation of composite requirements, because dimensioning faults often involve environment agnostic as well as environment dependent configuration parameters. For instance, a configured instance that is not providing the required availability may have been instantiated with the wrong profile or flavor (which is environment agnostic), or may have the wrong value for a heartbeat (which is environment dependent).

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.

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 passes 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.

VI. RELATED WORK

Misconfigurations are 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 how the configurable system responds to a misconfiguration rather than linking a misconfiguration to a requirement violation. ConfErr [10] for instance focuses on syntactic misconfigurations that humans can make. The authors build a psychological model of the mistakes a configuration designer may make; and uses to generate misconfigurations and evaluate how the configurable system responds to them. MisconfDoctor [13] on the other hand, generates misconfiguration to build a comprehensive set of logs that can be used to diagnose misconfigurations. The authors address both simple syntactic misconfigurations, i.e. mistakes on a single configuration parameter; 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.

The work in [12] presents a survey of configuration fault localization methods. It pinpoints the popularity of white box approaches in testing configurable systems and configured instances. Authors in [2] propose the use of coverage criteria for regression test case selection. After a reconfiguration, the authors propose 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 authors in [3, 4] propose the use of dynamic and static slices of configurable systems and explore their similarities to localize crashing errors and non-crashing errors [4]. Non-crashing errors can be implicitly considered as requirement violations.

The work in [5, 11] 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. Our earlier work in [14] proposes an approach to select test cases (from the acceptance test suite) that will be used to retest a configured instance after it is deployed in the production environment. The approach proposed in this paper is also based on the same classification of configuration parameters as in [14], but it considers the whole cloud system and not only two configured instances in isolation. Moreover, the approach in [14] assumes that the configuration used in the development environment and the one used in the production environment have the same values for all environment agnostic configuration parameters, and also assumes that both configured instances are used to satisfy the same set of requirements. This limits the applicability of the approach in [14], as opposed to the approach proposed in this paper which can apply to any change scenario whether it is a reconfiguration or a change in the environment, etc., as it considers the whole cloud system.

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 are similar to configurable systems as they enable the definition of a core reference architecture (configurable system) that can be concretized into custom architectures called products (configured instances) that realize specific requirements. [6] presents an extensive survey on approaches that explore the similarities between a SPL and its products as well as similarities between products of the same SPL to reuse test cases and eliminate redundant one. The work in [7] 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 in [8] extends the work in [7] to include dataflow analysis to eliminate redundant test cases. The authors in [9] use the differences between the architectures to select test cases that only target the parts of the architecture that were not tested previously; thus, reducing the number of test cases necessary to test a new product based on its similarities with previous products. They follow a white box testing approach that has no consideration for the requirements for which the SPL or the product were made.

VII. Conclusion

Configuration faults are the main source of tenants' requirements violation in cloud systems. This problem is aggravated as cloud systems undergo reconfiguration very frequently. Performing regression testing on systems in production is necessary but this needs to be done carefully in order to limit the impact. Reducing the size of the regression test suite is a first step towards the limitation of this impact. The method proposed in this paper reduces the size of the regression test suite while preserving the fault detection power according to the proposed fault model. The proposed approach can be used with any impact analysis or regression test case selection method. The proposed method does not depend on the types of configurations involved as it can be used with single stack configurations or cross stack configurations provided that some requirements are realized using more than one configured instance. To the best of our knowledge, this is the first configuration fault model that does not treat configuration faults as syntactic errors or t-way interactions between configurations parameters, but as causes of tenants' requirements violations. As of future work we intend to conduct an empirical validation of our method to see how it applies to real world applications.

Acknowledgment

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

References

[1] 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 SOSP, 2015, pp. 328-343.

[2] X. Qu, M. Acharya, B. Robinson. Impact Analysis of Configuration Changes for Test Case Selection. In the proceedings of the 22nd IEEE ISSRE, 2011, pp. 140-149.

[3] Z. Dong, A. Andrzejak, K. Shao. Practical and accurate pinpointing of configuration errors using static analysis. In the proceedings of IEEE ICSME, 2015, pp. 171-180.

[4] S. Zhang, M. D. Ernest. Automated diagnosis of software configuration errors. In the proceedings of the 35th ICSE, 2013, pp. 312-321.

[5] B. Robinson, L. White. Testing of User-Configurable Software Systems Using Firewalls. In the proc. of the 19th IEEE ISSRE, 2008, pp. 177-186.

[6] J. Lee, S. Kang, D. Lee. A survey on software product line testing. In the proceedings of the 16th SPLC, 2012, pp. 31-40.

[7] V. Stricker, A. Metzger, K. Pohl. Avoiding Redundant Testing in Application Engineering. In the proc. of the 14th SPLC, 2010, pp. 226-240.

[8] A. Reuys, S. Reis, E. Kamsties, K. Pohl. The ScenTED Method for Testing Software Product Lines. In Software Product Lines: Research Issues in Engineering and Management, 2006, pp: 479-520.

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

[10] L. Keller, P. Upadhyaya, G. Candea. ConfErr: A Tool for Assessing Resilience to Human Configuration Errors. In the proc. of the IEEE International Conference on Dependable Systems and Networks (DSN), Anchorage, AK, 2008, pp. 157-166.

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

[12] A. Andrzejak, G. Friedrich, F. Wotawa. Software Configuration Diagnosis – A Survey of Existing Methods and Open Challenges. In the proceedings of CONFWS'18.

[13] 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), Lisbon, 2018, pp. 1-12.

[14] 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 proc. of the 12th IEEE ICST, April 2019.

[15] 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, 2017, pp. 541-551.

[16] M. Ozawa, T. Dohi, H. Okamura. How Do Software Metrics Affect Test Case Prioritization? In the proceedings of the 42nd IEEE International Conference on Computer Software & Applications, 2018, pp. 245-250.

[17] E. Engström, P. Runeson. A Qualitative Survey of Regression Testing Practices. In the proceedings of Product-Focused Software Process Improvement, 2010.

[18] 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.

[19] 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, 1994, pp. 169-184.