A DSL-based Approach for Elasticity Testing of Cloud Systems

Michel Albonico
AtlanMod team
(Inria, Mines Nantes, Lina)
UTFPR, Brazil and
Mines Nantes, France
michelalbonico@utfpr.edu.br

Amine Benelallam
AtlanMod team
(Inria, Mines Nantes, Lina)
amine.benelallam@inria.fr

Jean-Marie Mottu and
Gerson Sunyé
AtlanMod team (Inria, Mines
Nantes, Lina) - University of Nantes
{jean-marie.mottu,
gerson.sunye}@inria.fr

Abstract

Elastic cloud systems automatically respond to workload changes by (de-)allocating resources according to a configuration specification. Testing such systems requires effort from the tester. In particular, the tester specifies the sequence of resource variations he/she is willing to test, and then, drive the system through this specific sequence of variations. In this paper, we propose a Domain-Specific Language (DSL) aiming at reducing the tester's effort to write and execute elasticity testing. Our DSL abstracts test case specification from the cloud provider's libraries, making our language portable. The DSL compiles into executable code. In our preliminary results, we show that our approach reduces the amount of words to specify test cases w.r.t. dedicated libraries. It also shows how much this improvement scales when running a test on multiple cloud providers.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/ Specifications—Languages, Tools; C.2.4 [*Distributed Systems*]: Distributed applications

Keywords Cloud computing, Elasticity testing, Model-Based Testing, Domain-Specific Language (DSL)

1. Introduction

One of the core design principles in Cloud-Based Systems (CBS) is resource auto-scaling, also known as *elasticity*. It is defined as the ability of a cloud infrastructure to vary its resource configuration according to demand (in particular allocating or deallocating resources) (1; 4; 5; 10). However, due to dynamic resource allocation, new issues may arise and cause the cloud system to fail (3). In response to this matter, several work (3; 7; 8; 11) have been proposed.

Gambi et al. (8) propose a conceptual framework for testing elastic CBS that helps tester to manage four activities: test case generation, test execution, data analysis, and test evolution. In this paper, we focus on the first two ones.

These two activities require tester's effort to design and implement test cases, and to deploy and configure CBS repeatedly to run each test case. In test cases, tester specifies (a) how the system should be set up before running the test, (b) the test scenario (e.g. which request(s) is sent to verify a specific behavior), (c) how the system is supposed to behave correctly or not. That specification requires information dedicated to the elasticity. This includes the elasticity workflow, i. e., through which elasticity configuration the CBS is led (e. g., sequence of resource allocation and deallocation). To manage the elasticity workflow, workload must be properly generated. The deployment and (re-)configuration of the CBS also includes load generators setup, setting auto-scaling on cloud provider, (re-)starting CBS components, etc.

Considering elasticity in test cases is complicated since it requires managing many parameters. Cloud providers, such as Amazon Elastic Compute Cloud (EC2)[1] and Google Cloud Compute Engine (CE)[2], usually provide Command-line Interfaces (CLI). CLIs help the tester to setup elasticity when testing since they abstract CBS deployment, management, including elasticity's parametrization. However, they admit wide variability of CBS configuration, more than necessary for elasticity testing. Furthermore, each cloud provider has its own CLI, which preclude portable commands (that execute over any cloud provider).

In this paper, we propose a generic Domain-Specific Language (DSL) for elasticity setup when testing CBS. Our approach alleviates test case generation by centralizing the elasticity setup in this DSL, whereas the other part of a test case can be implemented as usual (e.g. in JUnit). This DSL helps tester to set up CBS and its dependencies, auto-scaling, desired elasticity workflow, and test execution schedule. Then, it is compiled to a code that automatically executes elasticity testing, without further tester's interaction. In preliminary results, our DSL reduces the number of words to set up elasticity testing, and requires lower tester's effort to adapt a test case to be executed on several cloud providers.

Here is the paper structure. Section 2 introduces some background. Section 3 describes the DSLs. Section 4 describes the implementation. Section 5 discusses preliminary results. Section 6 reports related work. Section 7 concludes.

[1] `https://aws.amazon.com/ec2/`

[2] `https://cloud.google.com/compute/`

2. Background

In this section, we describe major aspects of cloud computing elasticity. We also remain with Thiery et al. (14) DSL for setting up deployment of CBS.

2.1 Cloud Computing Elasticity

Main cloud infrastructures [3] admit by default threshold-based auto-scaling (i. e., elasticity). Figure 1 depicts a typical threshold-based elasticity. In this figure, we see that *resource demand* (continuous line) varies overtime, such variation essentially follows workload variation. For explanatory reasons, we only consider a resource demand that increases from 0 to 1.5, then decreases to 0.

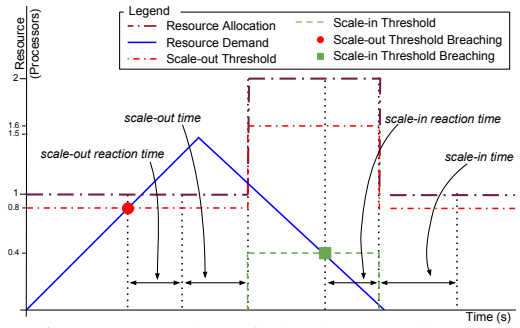

Figure 1: Representation of Cloud Computing Elasticity.

When resource demand exceeds the *scale-out threshold* and remains higher during the *scale-out reaction time*, cloud elasticity controller assigns a new resource. The new resource becomes available after a *scale-out time*, which is the time spent to allocate it. Once the resource is available, the threshold values are updated accordingly. It is similar considering the *scale-in*, respectively. Except that, as soon as the scale-in begins, the threshold values are updated and the resource is no longer available. Nonetheless, the infrastructure needs a *scale-in time* to release the resource.

When CBS are deployed on a cloud infrastructure, workload fluctuations lead to resource variations (elasticity). These variations drive the CBS to new, elasticity-related, states. It starts in a *ready* (*R*) state, then it alternates *scaling-out* (*SO*) and *scaling-in* (*SI*) states with *ready* state (2).

2.2 DSL for Cloud-Based System Deployment

When testing CBS through elasticity, testers first need to deploy the system under test. Thiery et al. (14) propose a DSL for setting up deployment of CBS. Their DSL is divided into two dimensions: deployment bundle, and cloud provider. We just introduce them in that subsection.

2.2.1 Deployment Bundle (DB)

It sets up CBS components and dependencies. We can set up multiple instances of software bundle. Each software bundle may group either CBS or testing tool software components. Software component may have dependencies, such as other software component or external files (i. e., configuration files or executable scripts).

Listing 1[4] shows an example of software bundle setup for a Web application. First of all, we describe software artifacts. In the example, *http* is installed by a package installer (e. g., apt-get, yum, etc.), while *phpapp* is an interpreted PHP application, remotely transferred from local (*src*) to remote (*dest*) path. After, we describe external files, also with local and remote paths. Finally, we describes software bundles. In the example, we first bundles Web application artifacts (*app*), then the testing tool ones (*testingTool*).

Listing 1: Example of software bundle setup.

```
software_bundles {
  software httpd : pkg 'apache2' '2';
  software phpapp : src './app/',
    dest '/var/www/app/';
  source apachecfg : src 'httpd.conf',
    dest '/etc/apache/httpd.conf';
  bundle wsrv :
    app phpapp, dep (httpd, php, mysq),
    src (apachecfg, createdb, addserver),
    provScript (createdb, addserver);
  bundle appbench :
    testingTool app_bench, dep (java),
    src (bench_conf,bench),
    testScript (bench);}
```

2.2.2 Cloud Provider (CP)

It sets up cloud providers' resource used for software bundles deployment. Software bundles are deployed on deployment instances. Each deployment instance starts up an Operational System (OS) on a Virtual Machine (VM) that may have different combinations of computational resource (CPU, memory, etc.). Deployment instances use resource from a cloud provider's geographic zone. Software bundle may require some port configuration, e. g., port 80 for external interactions with Web server.

Listing 2 describes an example of Amazon EC2's resource setup. In this DSL, we start by setting up an VM image (*image*), which refers to existing cloud provider's image. In the example, we set an hypothetical image identifier (*ami-1234*), and describe which Operational System (OS)[5] runs in the image. Then, we list available cloud provider's geographical zones (*zone*). Finally, we set up deployment instances, referring a software bundle, VM image, machine type, port configuration, and geographic zone.

Listing 2: Example of cloud provider's resource setup.

```
resources EC2 {
  image iU704i386 :
    imageId 'ami-1234',
    os 'Ubuntu' '7.04' 'i386';
  zone EUWest :
'eu-west-1a', 'eu-west-1b';
  instance websrv :
    image iU704i386, machineType m3.large, zone EUWest,
    portConfig '80' = '0.0.0.0', bundle wsrv;}
```

[3] http://www.rightscale.com

[4] Syntax has been adapted to the same syntax of our proposal.

[5] We cannot straightforwardly get OS information from cloud provider.

3. Elasticity Testing DSL

Despite Thiery et al. DSL allows a variety of cloud system deployment, it does not address cloud computing elasticity and elasticity testing. In this paper, we propose a DSL that complements Thiery et al. work, adding support to set up elasticity and elasticity testing. Our DSL is three-dimensional: auto-scaling, elasticity workflow, and test method schedule.

3.1 DSL to Set Up Auto-Scaling (AS)

To enable elasticity, we must set up auto-scaling on cloud provider. Threshold-based auto-scaling is a common strategy among major cloud providers.It basically consists of varying a cloud resource when a threshold is breached for a while (see Section 2.1). Figure 2 illustrates the model that represents threshold-based auto-scaling setup.

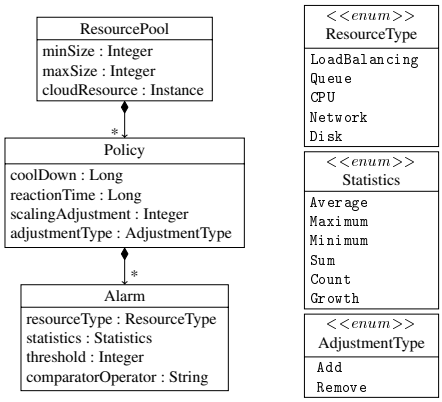

Figure 2: Model of threshold-based auto-scaling setup.

ResourcePool describes the cloud resource (*cloudResource*) that is varied, and restricts its amount (between *minSize* and *maxSize*). Resource variation is regulated according to the *Policy* properties. *Policy* properties state time constraints (i. e., cool down[6] and reaction time periods), and resource variations (e. g., add one resource) performed when the time constraints are satisfied. *Policy* is checked every time an alarm (*Alarm*) is triggered, i. e., threshold is breached. *Alarm* describes a threshold and resource usage breaching it.

Listing 3 describes an example of threshold-based auto-scaling. The resource pool *wsrv_pool* is associated to the *websrv* deployment instance (see Listing 2), and may have from 1 to 10 instances. As policy, we set a cool down period of $60,000\ ms$, and a reaction time with same duration, adding one new resource from *wsrv_pool* resource pool. We set the alarm *highCPU* assuming threshold is breached when the maximum (*statistics=Maximum*) CPU usage (*resourceType=CPU*) is higher than (*comparatorOperator=>*) 60%.

[6] New variation is not allowed during the previous scaling activity.

Listing 3: Example of threshold-based auto-scaling setup.

```
elasticity {
 resourcePool wsrv_pool : minSize 1,
 maxSize 10, cloudResource websrv;
 policy wsrv_policy : resourcePool slaves,
  coolDown 60000, reactionTime 60000,
  scalingAdjustment 1, adjustmentType Add;
 alarm highCPU : resourceType CPU,
  statistics Maximum, threshold 60,
  comparatorOperator '>', policy wsrv_policy;}
```

3.2 DSL to Set Up Elasticity Driving (ED)

For some tests, such as regression testing and bug reproduction, it may be necessary to have a deterministic elasticity, reaching or repeating a strict behavior. In a previous work (2), we address deterministic elasticity generating proper workload that drives CBS through required elasticity behavior, i. e., sequence of elasticity states. Despite our previous work successfully drives cloud applications through the given sequence of elasticity states, it requires much tester's effort. Tester has to write substantial amount of code to set up elasticity driving.

Another contribution of our DSL is to abstract elasticity driving setup. Figure 3 illustrates the DSL model. In the model, we have a resource pool (*pool*), which refers to cloud resource that is driven. It also admits a workload type (*workType*), and either to generate a sequence of elasticity states (*GeneratedFlow*) or to preset one (*PresetFlow*). To generate a sequence, we have to set the number of scaling-out (*scalingOuts*) and scaling-in (*scalingIns*) a sequence must have. Then, elasticity states are distributed in a way all scaling-in and scaling-out happen, respecting *ResourcePool*'s properties. For a preset sequence, we set the elasticity states in the order we want them to occur.

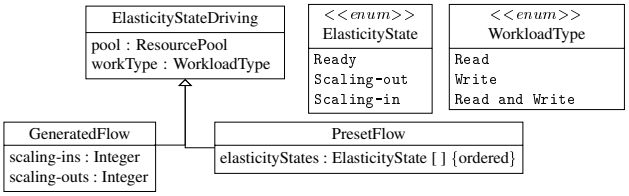

Figure 3: Model of elasticity driving setup.

Listing 4 shows an example of elasticity driving setup. In the example, we set up elasticity driving to drive cloud resource in *wsrv_pool* pool using *Read* workload type. We preset the following sequence of elasticity states: *scaling-out*, *ready*, *scaling-in*, *ready*, *scaling-out*.

Listing 4: Example of elasticity driving setup.

```
driving {
 drive wsrv_drive :
   pool wsrv_pool,  workType Read,
   states set(scaling-out, ready,
              scaling-in, scaling-out, ready);}
```

3.3 DSL to Set Up Test Methods Schedule (TS)

Here, we follow a principle of our previous paper (3), where some of CBS's tasks only occur during certain elasticity states. For instance, massive data replication only occurs when new nodes are already added (ready state). Therefore, it is not necessary to test it beforehand, i. e., while resource is being added (scaling-out state).

Our DSL allows tester to set up the execution of test methods during either specific states or all elasticity states. Figure 4 illustrates the model of setup of test method schedule. The model allows to associate *test methods* (*TestMethod*) to test suites (*TestSuite*). Test suites are associated to elasticity states (*ElasticityState*), their drivings, and execution strategy (*Strategy*), i. e., in parallel, or in sequence.

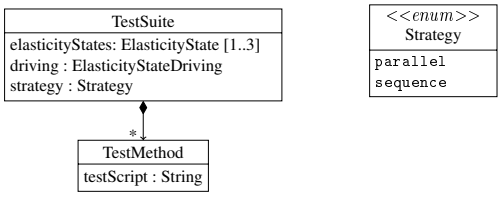

Figure 4: Model of test executions setup.

Test methods have test script (*testScript*) as attribute. A test script can be any executable command, such as a java class (e.g. JUnit) or a shell script. In this way, tester can write generic test methods. Then, he/she associates test methods to elasticity states they must execute along.

Listing 5 shows an example of setup of test method schedule. In the example, we set two test methods (*test.test1* and *test.test2*) from a Java Archive (JAR) file (*test.jar*). Then, they are associated to scaling-out state of elasticity driving *wsrv_drive*, and set to execute in parallel.

Listing 5: Example of setup of test method schedule.

```
tests {
 test t1:
  script 'java -jar ./test.jar test.test1';
  test t2 :
  script 'java -jar ./test.jar test.test2';
 suite s1 : states scaling-out,
  driving wsrv_drive,
  test_method (t1,t2), in parallel; }
```

4. Compiling Specification into Executable Code

In this section, we explain the compilation of elasticity testing setup written in our DSL into executable code. Figure 5 depicts the workflow of this compilation.

First, tester writes test case setup (*TESetup.es*) using our DSL (in the figure, divided by specialization). Then, tester selects a cloud provider's mapping file (*DSL to CLI Mapping*), whereas it is necessary a mapping file for each cloud provider we want to execute the elasticity testing. The mapping file maps elements of our DSL to arguments of cloud

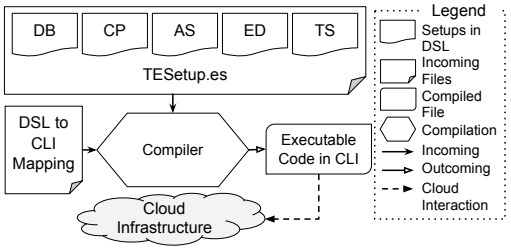

Figure 5: Workflow of compilation of elasticity testing specification into executable code.

provider's CLI. Finally, we compile the test case setup into a list of cloud provider's CLI commands. The compiler is generic, compiling any mapping file written after our model.

In our mapping file model, we first name the cloud provider. Then, we set the CLI commands used by the elasticity testing. Every command receives an unique name within a preset list, the command, and a list of arguments with their values. The command names help on mapping the commands necessary during the elasticity testing. For instance, when it is necessary to add a new instance, we map the command that is named as e. g., *addInstanceCommand*. On the other hand, if we need to set a threshold, we map the command that is named as *setThresholdCommand*.

Listing 6 shows a partial mapping file for Amazon EC2 cloud provider, named as *AWS*. It describes Amazon EC2's command to deploy instances, i. e., *aws ec2 run-instances*, named as *addInstanceCommand*. One of the arguments of this command is the image identifier (*–image-id*), which receives as value the element *imageId* from cloud provider's resource setup (see Listing 2). When compiling, we get the values set using our DSL, and complete the CLI commands. A compiled command to deploy Amazon EC2's instances would be $awsec2run - instances - -image - idiU704i386[...]$,where we replace the element *imageId* by the value in cloud provider's resource setup.

Listing 6: Partial mapping file for Amazon EC2.

```
provider AWS {
 addInstanceCommand 'aws ec2 run-instances' :
  '--image-id' imageId, [...]; }
```

Theoretically, using mapping file makes our approach portable to any cloud provider that allows command line. We only provide mapping files for Amazon EC2, Google CE, and Openstack, though we can write mapping files for other cloud providers. If CLIs of the other cloud providers allow further arguments, which are not covered by our DSL, we can set them with fixed values in the mapping file. Writing mapping files is not a tester's task. A specialist, such as a network manager, can write these files.

5. Preliminary Results

In this section, we measure the impact of using our DSL on writing elasticity testing setup. We consider two case stud-

ies, already used in previous papers, where we test distinct cloud applications through elasticity: 1) a MongoDB replica set (3), and 2) a distributed web application (2).

The main objective of this work is to alleviate tester's effort considering elasticity when generating test cases. Therefore, we so far provide a way to write elasticity part of test cases, then compile it to executable code. The current study do not go further on elasticity testing execution. However, we plan to do it as part of a future work.

5.1 First Case Study (CS1): Testing a MongoDB Replicat Set

The first case study tests MongoDB deployed as a replica set. In this experiment, we drive MongoDB through the following sequence of elasticity states: *R, SO, R, SO, R, SI, R, SO, R, SI, R, SI, R, SI, R*. Here, we consider a test method that tests performance of MongoDB through all elasticity states. The current experiment considers only the elasticity setup of a test case, since testers do not use our DSL to write the rest of the test case (i.e. the test methods referred in TS part of our DSL).

5.2 Second Case Study (CS2): Testing a Distributed Web Application

The second case study tests a distributed web application. Its architecture is made by a centralized database server, a load balancer, and *n* web servers. We drive the web application through 10 scale-out and 10 scale-in in sequence. In our previous paper, we do not test the web application. Here, we consider an hypothetically test case is associated to scaling-out and scaling-in elasticity states. We choose this test method schedule to be different than first case study.

5.3 Results

For each case study, we write the elasticity testing setup in our DSL. These setups are compiled to CLI for three different cloud providers: Amazon EC2, Google CP, and OpenStack. Then, we compare tester's effort on writing elasticity testing in both, our DSL and CLIs. We measure tester's effort in amount of words: *total amount of words*, and *cumulative amount of new words*.

5.3.1 Total Amount of Words

Total amount of words refers to the amount of words in an elasticity testing setup. Table 1 describes the amount of words of elasticity testing setups for the elasticity testing case studies. Setups written in our DSL contain almost the same amount of words for all cloud providers (only 6 words changed as explained next subsection), while setups written with CLIs differ in amount of words according to cloud provider. Furthermore, setups written in our DSL result in fewer words for all cloud providers and case studies. Considering the amount of words as an effort, our DSL reduces considerably the tester's effort: Amazon EC2 ($CS1 \approx -24\%$, $CS2 \approx -22\%$), Google CP ($CS1 \approx -38\%$, and $CS2 \approx -36\%$), and OpenStack ($CS1 \approx -43\%$, and $CS2 \approx -39\%$).

Approach	Cloud Provider	CS1	CS2
Our DSL	*All Cloud Providers*	246	273
CLIs	*Amazon EC2*	326	364
	Google CP	392	433
	OpenStack	430	476

Table 1: Total amount of words for case study setups.

Figure 6 depicts the effort in amount of words to write setups for the case studies. In the figure, the dashed line connects *CS1* efforts, while the solid line connects *CS2* efforts. Furthermore, we see that such lines never cross each other, and the distance between them is almost homogeneous. This is because from CS1 to CS2, the effort varies proportionally (with an approximation between 1.09% and 1.1%) for every setup. This means the difference among setups should be constant for other case studies. An encouraging result, which would result in less effort even when writing future elasticity testing setups in our DSL.

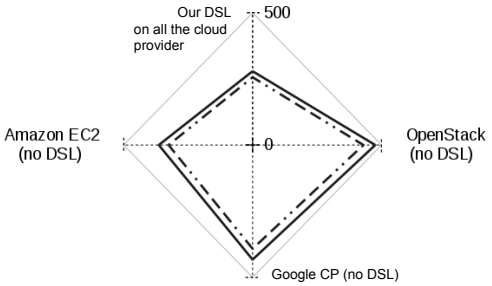

Figure 6: Effort on writing elasticity testing setups.

5.3.2 Cumulative Amount of New Words

Total amount of new words refers to the cumulative amount of words necessary to re-write an existing elasticity testing setup, making it suitable to other cloud provider. For instance, a tester may execute the same elasticity testing over different cloud providers, (re-)writing setups for all of them. We use the following formula to represent: $C_i = C_{i-1} + (S_i \odot S_{i-1})$, where i denotes the sequence the setup is written, and $S_i \odot S_{i-1}$ denotes the amount of new words from previous to current setup (S).

Graph of Figure 7 illustrates the cumulative amount of new words as case studies setup is (re-)written for a given sequence of cloud providers: Amazon EC2, Google CE, and OpenStack. In the figure, continuous lines illustrate the cumulative amount of new words for setups written in our DSL, while dashed lines illustrate the cumulative amount of new words for setups written with cloud providers' CLIs.

In the graph, we cannot see the variation for setups written in our DSL. This is because using our DSL the variation is slight, only 6 words change from one setup to another. These words refer to cloud provider's resource, such as image identifier and zone name, named distinctly among cloud

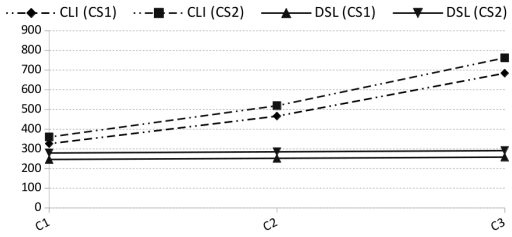

Figure 7: Total amount of new words of elasticity testing setups.

providers. On the other hand, the variation for setups written in cloud providers' CLIs is visible, it more than double from first to last setup ($\approx *2.1$ for both case studies).

6. Related Work

In literature, there are some model/DSL-based approaches for the deployment and provisioning of CBS. However, most of them do not cover elasticity setup. Thiery et al. (14) propose a model-based approach to automate the deployment of CBS. Likewise, Kirschnick et al. (12) propose a DSL that is limited to the provisioning and deployment. Other work propose DSLs to deploy Software-as-a-Service (SaaS) (13) and Platform-as-a-Service (PaaS) (6). but none address Infrastructure-as-a-Service (IaaS). Goncalves et al. (9) propose Cloud Modeling Language (CloudML), which models services, resource profiles, and developer's requirements. However, their work requires the cloud provider to describe services and resources in CloudML, which is unusual. Finally, there are commercial DSL-based orchestration tools, such as Chef [7] and Puppet [8]. These tools allow the deployment and provisioning of CBS, as well as the elasticity setup. However, they are not suitable for elastic testing since they do not support neither to set features related to elasticity states, nor to schedule test methods execution.

7. Conclusion

In this paper, we propose a DSL-based approach to set up elasticity testing. Its major contributions are portability and reduction of tester's effort to write elasticity testing specifications. With a few changes in the setup, elasticity testing is executed over multiple cloud providers. With cloud providers' mapping files, we can easily adapt our approach to execute elasticity testing on any cloud provider. Our approach reduces considerable the amount of words on writing elasticity testing specifications. In future work we will focus on automatic resource discovering. For instance, finding the cheapest resource that fits testing requirements. This makes specification in our DSL completely portable: a single specification executed over multiple cloud providers without any change. We also think in new features, such as test case generation, and elasticity controller.

[7] `https://www.chef.io/chef/`

[8] `https://puppet.com/`

Acknowledgments

Work supported by CAPES Foundation (Science Without Borders p. 9070–13–3), Ministry of Education of Brazil.

References

[1] D. Agrawal, A. El Abbadi, S. Das, and A. J. Elmore. Database scalability, elasticity, and autonomy in the cloud. *DASFAA'11*, pages 2–15, 2011.

[2] M. Albonico, J.-M. Mottu, and G. Sunyé. Controlling the Elasticity of Web Applications on Cloud Computing. In *Proc. of the 31st SAC*, pages 816–819. ACM, 2016.

[3] M. Albonico, J.-M. Mottu, and G. Sunyé. Monitoring-based testing of elastic cloud computing applications. In *Companion of ACM/SPEC ICPE*, pages 3–6. ACM, 2016.

[4] L. Badger, T. Grance, R. Patt-Comer, and J. Voas. *Draft Cloud Computing Synopsis and Recommend-ations*. Nist Special Publication 800-146, 2011.

[5] M. M. Bersani, D. Bianculli, S. Dustdar, A. Gambi, C. Ghezzi, and S. Krstić. Towards the Formalization of Properties of Cloud-based Elastic Systems. In *Proc. of the PESOS*, pages 38–47. ACM, 2014.

[6] R. Boujbel, S. Rottenberg, S. Leriche, C. Taconet, J. P. Arcangeli, and C. Lecocq. MuScADeL: A Deployment DSL Based on a Multiscale Characterization Framework. In *Proc. of the 38th IEEE COMPSACW*, pages 708–715, 2014.

[7] A. Gambi, A. Filieri, and S. Dustdar. Iterative test suites refinement for elastic computing systems. In *Proc. of the 9th ESEC/FSE*, pages 635–638. ACM Press, 2013.

[8] A. Gambi, W. Hummer, H.-L. Truong, and S. Dustdar. Testing Elastic Computing Systems. *IEEE Internet Computing*, 17(6):76–82, 2013.

[9] G. Goncalves, P. Endo, M. Santos, D. Sadok, J. Kelner, B. Melander, and J. E. Mangs. CloudML: An Integrated Language for Resource, Service and Request Description for D-Clouds. In *Proc. of the 3rd IEEE CloudCom*, pages 399–406, 2011.

[10] N. R. Herbst, S. Kounev, and R. Reussner. Elasticity in Cloud Computing: What It Is, and What It Is Not. In *Proc. of the ICAC*, pages 23–27, 2013.

[11] S. Islam, K. Lee, A. Fekete, and A. Liu. How a Consumer Can Measure Elasticity for Cloud Platforms. In *Proc. of the 3rd ACM/SPEC ICPE*, pages 85–96, 2012.

[12] J. Kirschnick, J. Alcaraz Calero, L. Wilcock, and N. Edwards. Toward an architecture for the automated provisioning of cloud services. *IEEE Comm. Mag.*, 48(12):124–131, 2010.

[13] K. Sledziewski, B. Bordbar, and R. Anane. A DSL-Based Approach to Software Development and Deployment on Cloud. In *Proc. of the 24th IEEE AINA*, pages 414–421, 2010.

[14] A. Thiery, T. Cerqueus, C. Thorpe, G. Sunye, and J. Murphy. A DSL for Deployment and Testing in the Cloud. In *Proc. of the IEEE ICSTW*, pages 23–27, 2014.