Model-Driven Elasticity for Cloud Resources

Hayet Brabra[1,2(✉)], Achraf Mtibaa[3], Walid Gaaloul[1], and Boualem Benatallah[4]

[1] Telecom SudParis, UMR 5157 Samovar, Universite Paris-Saclay, Paris, France
hayet.brabra@telecom-sudparis.eu
[2] FSEG, Miracl Laboratory, Universiy of Sfax, Sfax, Tunisia
[3] ENETCOM, Miracl Laboratory, Universiy of Sfax, Sfax, Tunisia
[4] UNSW, Sydney, Australia

Abstract. Elasticity is a key distinguishing feature of cloud services. It represents the power to dynamically reconfigure resources to adapt to varying resource requirements. However, the implementation of such feature has reached a level of complexity since various and non standard interfaces are provided to deal with cloud resources. To alleviate this, we believe that elasticity features should be provided at resource description level. In this paper, we propose a Cloud Resource Description Model (cRDM) based on State Machine formalism. This novel abstraction allows representing cloud resources while considering their elasticity behavior over the time. Our prototype implementation shows the feasibly and experiments illustrate the productivity and expressiveness of our cRDM model in comparison to traditional solutions.

Keywords: Elasticity · Cloud resources · State machine
Orchestration

1 Introduction

Elasticity is one of the main assets characterizing the cloud services. It is achieved through invocation of reconfiguration actions that run as a result of events, allowing a controller to automatically (re)configure cloud resources. However, exploiting such feature poses a great complexity due to the proliferation of tools that offer heterogeneous resource orchestrations and elasticity services [6]. Existing cloud orchestration and elasticity solutions rely on procedural programing languages (e.g., most of them are based on low-level scripting) to support the elasticity of cloud resources [9,15,17,18]. Prominent examples include: Puppet, Docker, Cloudify, AWS AutoScaling and IBM AutoScale [4,5,18]. This diversity implies that cloud programmers are forced to be aware of different low-level cloud service APIs, command line syntax and procedural programming constructs, to describe and control the elasticity of cloud services. Moreover, this problem worsens as the variety of cloud services and the variations of application resource requirements and constraints increase [15,17]. Not only that, the emergence of federated cloud makes this problem too unreasonably complicated.

A more effective solution should allow users to specify their resources and elasticity features regardless the technical specifications of any cloud provider or orchestration tool. We believe that elasticity features should be provided at resource description level. Instead of relying on low-level scripting mechanisms or provider-specific rule engines, we argue that models and languages for describing cloud resources should be endowed with intuitive constructs that can be used to specify a range of flexible elasticity mechanisms.

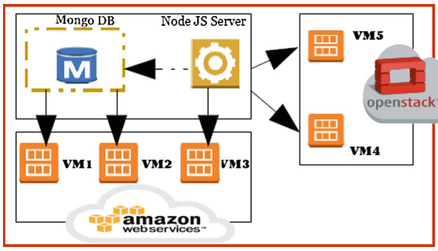
Motivated by these considerations, we propose, in this paper, a cloud Resource Description Model (cRDM) to describe cloud resources and their elasticity. Our model is based on a new abstraction that we call Cloud resource requirement State Machine (C-SM), which allows representing cloud resources while considering their elasticity behaviour. We adopt state-machine formalism as it provides refreshing graphical notations in contrast with existing largely text-based solutions. Indeed, state-based model has been commonly used to model the behaviour of systems due to the fact that it is simple and intuitive. In this model, states will be used to characterize application specific resource requirements, when they are needed while transitions between states are triggered when certain conditions are satisfied. Transitions automatically trigger controller actions to perform the desired resource (re-)configurations to satisfy the requirements of target states. Our model has been prototypically implemented and evaluated using experiments with real cloud use cases illustrating its productivity and expressiveness in comparison to traditional solutions.

The paper is structured as follows: Sect. 2 articulates the motivations of our model. In Sects. 3 and 4, we present our model. Section 5 describes our implementation and evaluation. In Sects. 6 and 7, we present related work and conclusion.

2 Limitations in Current Cloud Description and Elasticity Solutions

In this section, we investigate through a motivating example, specific limitations among existing cloud resources description and elasticity solutions.

Motivating Scenario. Consider a cloud user wants to specify resource requirements and constraints for deploying an e-commerce application, which consists of a Mongo database, a NodeJS server and a Java script application (Fig. 1(a)). The user selects Amazon web services (AWS) to deploy this application and specifies that she needs 5 virtual machine (VM) instances to be hosted in for one year from Monday, 1st of January 2018. Each instance has 8 GB RAM and 4 GHz CPU. As QoS Constraint, the user would like that the availability for each instance must be at least 99%. Moreover, when the average CPU usage for 5 min is greater than or equal 80%, she wants 5 more instances to be added from AWS. In contrast, these 5 instances should be removed whenever the average CPU usage is less than 20%. However, during the business spikes every weekend, whenever



Tool	Interfaces and RDMs Available
OpenStack	SDKs, REST API , Heat template
Amazon cloud services (AWS)	AWS CLI, SDKs, REST, AWS console, AWS Auto Scaling, CloudFormarion
Rackspace	Rackspace CLI, SDKs, REST API, Rackspace Auto Scale
Docker	Docker CLI, REST API, Docker files
Cloudify	Cloudify CLI, REST API, TOSCA

Fig. 1. (a) Cloud Resources for deploying an E-commerce application; (b) List of available cloud description and orchestration Interfaces

the application reaches 10 instances in AWS, she wants to horizontally scale out into another cloud like Openstack by adding 10 instances. Accordingly, the user first needs to describe her application, required resources and dependencies between them. Secondly, she needs to specify elasticity policies that control the application at runtime. Finally, if she wants to use cloud resources from other providers such Rackspace or Openstack, she requires to carry out some adaptations in the application and elasticity policies to make it compatible with the target management interfaces. Additionally, she can use an orchestration tool such Cloudify or Docker to support the multi-providers deployment. However, as shown in Fig. 1(b), AWS, Openstack, Rackspace, Docker and Cloudify provide heterogeneous resource description models (RDMs) and management interfaces and rely on low-level script-based APIs.

Based on these observations, we concluded that existing cloud resources description and elasticity solutions (1) are rarely transparent and adaptive to support the management of resources across various providers; (2) oblige users to acquire new expertise in multiple RDMs and different elasticity mechanisms; and (3) lead to a costly environments and potential vendor lock-in as exploiting resources from a new provider demands extensive programming effort [9].

3 Identifying Basic Cloud Resources Abstractions

We identify the required modeling abstractions that allow users to specify their cloud resources and their elasticity policies. To do so, we first performed a domain analysis on cloud resources from several providers, including AWS, Openstack, OCCI and TOSCA. We selected these services as they represent the range of different types of cloud services available: commercial offer, open source implementation, and open standard. Then, we analyzed the different RDMs used by the cloud orchestration tools, including Cloudify and Docker and the elasticity solutions used in these tools and cloud solutions cited above. Intentionally, we want a model that is very simple, so that it could help us start from a minimal base and progressively extend it as needed. In this section, we focus on the basic abstractions to be included when using cloud resources from one provider.

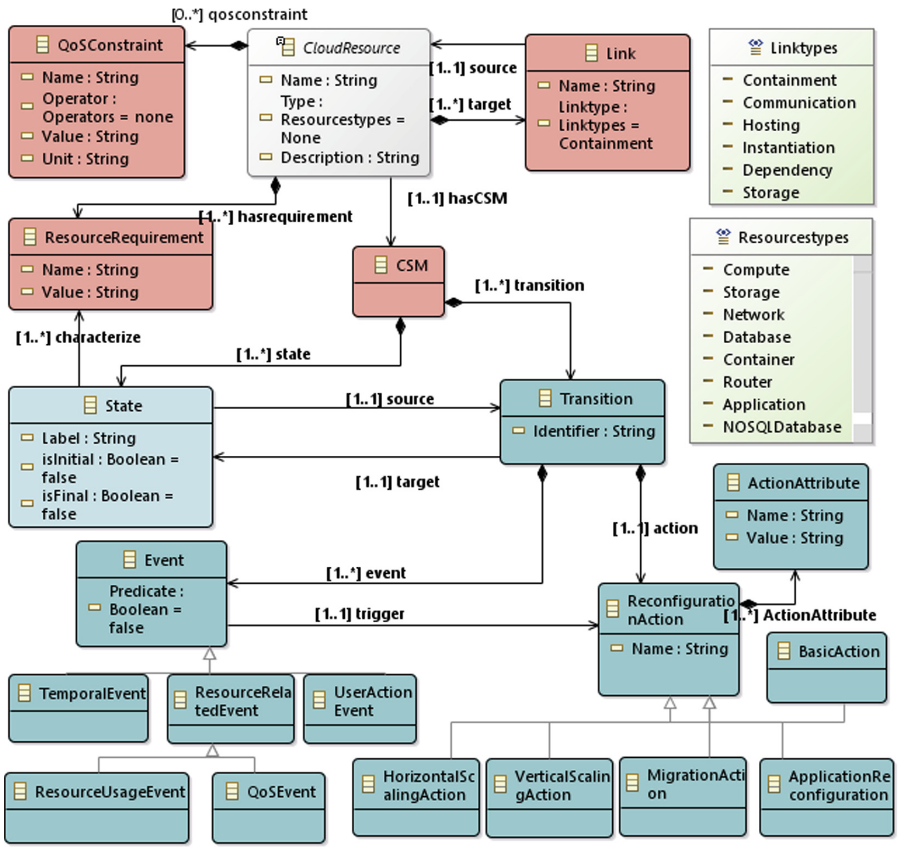


Fig. 2. UML class diagram for the Cloud Resource Description Model

3.1 Abstractions Overview

Figure 2 represents the conceptual UML model illustrating the main modeling abstractions of our cRDM. For illustration purpose, we rely on the motivating example previously described, which consists in deploying an E-commerce application both on AWS and OpenStack. As shown in Fig. 2, CloudResource is the root concept, which is described by a Name that indicates a resource name and Type that indicates its type (e.g. Compute, Database, etc.) and a set of other optional attributes (e.g. Description). Moreover, it is associated by four main concepts. ***Resource Requirement*** includes the name-value pairs describing the requirement attributes associated with the resource, such as CPU, RAM, etc. ***Link*** represents relationship between cloud resources. It is defined by a Name (e.g., hosted-in), Linktype (e.g. Containment, Communication, Hosting, etc.) and source and target participants resources. ***QoS Constraint*** represents a constraint on a particular QoS metric. It is defined through Name that indicates the QoS metric, Operator that indicates the comparison operator, Value that indicates the metric value, and Unit that indicates the measurement unit. ***C-SM*** is a Cloud Requirement State Machine that aims at capturing the elasticity behavior of a cloud resource over its life cycle.

Example. Figure 3 shows the corresponding cRDM instance of the motivation example. It consists of four cloud resources instantiated from the CloudResource concept: Node-JS, NodeBookshop, Mongo-Database and VM1. For example, VM1 is a virtual machine defined by the name vmcompute and type Compute and represents a host for Node-JS and Mongo-Database resources. This was accomplished by hosted-in links between these resources. Besides, the VM1 is associated with VM1-requirements, VM1-QoS and VM1-CSM. VM1-requirements indicates that VM1 should have 4 GB RAM, 4 GHz CPU, 5 instances and is acquired from AWS provider. VM1-QoS indicates that VM1 availability should be at least 99%. VM1-CSM specifies the elasticity behaviour related to VM1.

3.2 States

A state has a string label and two attributes (isInitial and isFinal) indicating whether the state is a final or initial. It is also associated with the ResourceRequirement concept to characterize the application-specific resource requirements in that state. As illustrated in the Fig. 3, VM1-CSM instance consists of four states S1, S2, S3 and S4 that VM1 may go through during its life cycle where S1 is the initial state and S4 represent the final state. Each state is annotated with resource requirements that should be satisfied under it. For example, the state S1 can only be reached if 5 instances of VM1 from AWS have been started.

3.3 Transitions

We use transitions to express the elasticity behaviour of a cloud resource. Each transition is labeled with a re-configuration policy. A reconfiguration policy aims at automatically triggering a reconfiguration action as result of events to satisfy the requirements of the target state. Conceptually, as shown in Fig. 2, each transition is specified by Identifier that indicates the transition identifier, source that indicates the source state and target that indicates the target state and consists of one or more Event and one Reconfiguration Action. Thus, cloud users can annotate the transitions with those events and reconfiguration actions to model reconfiguration policies of cloud resources without referring to any low-level scripting mechanisms or provider specific policy engines.

3.3.1 Events

We distinguish three event types to trigger a reconfiguration action: Temporal Events, Resource Related Events and User Action Events (see Fig. 2). Each event is specified via a logic predicate expressed using the common grammar used in the traditional state machine [16].

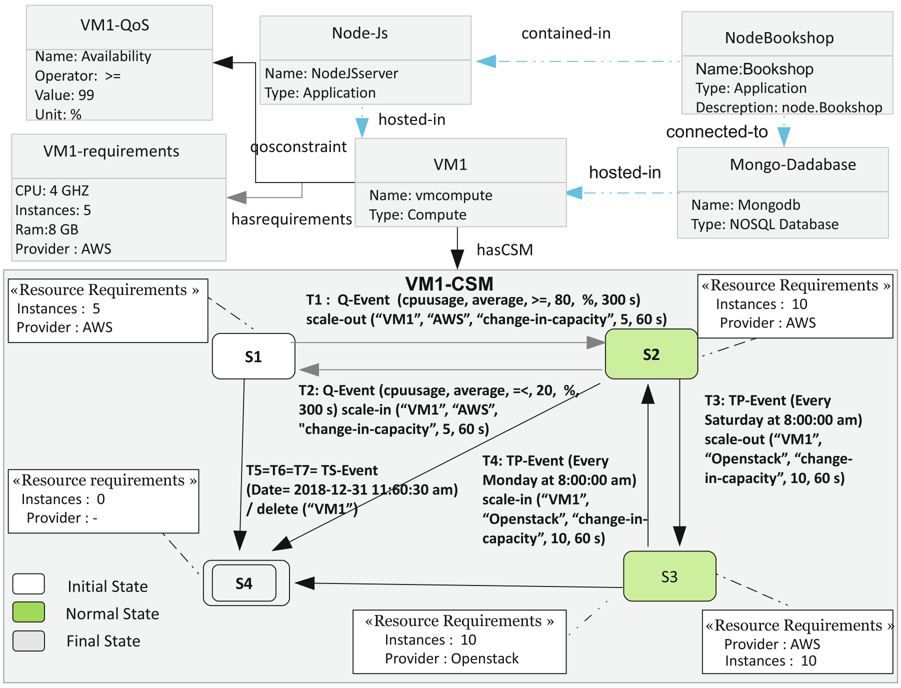

VM1-QoS
Name: Availability
Operator: >=
Value: 99
Unit: %
Node-Js
Name: NodeJSserver
Type: Application
contained-in
NodeBookshop
Name:Bookshop
Type: Application
Descreption: node.Bookshop
hosted-in
connected-to
VM1-requirements
CPU: 4 GHZ
Instances: 5
Ram:8 GB
Provider : AWS
qosconstraint
VM1
Name: vmcompute
Type: Compute
hosted-in
Mongo-Dadabase
Name: Mongodb
Type: NOSQL Database
hasrequirements
hasCSM
VM1-CSM
«Resource Requirements » Instances : 5 Provider : AWS
T1 : Q-Event (cpuusage, average, >=, 80, %, 300 s) scale-out ("VM1", "AWS", "change-in-capacity", 5, 60 s)
«Resource Requirements » Instances : 10 Provider : AWS
S1
S2
T2: Q-Event (cpuusage, average, =<, 20, %, 300 s) scale-in ("VM1", "AWS", "change-in-capacity", 5, 60 s)
T3: TP-Event (Every Saturday at 8:00:00 am) scale-out ("VM1", "Openstack", "change-in-capacity", 10, 60 s)
T5=T6=T7= TS-Event (Date= 2018-12-31 11:60:30 am) / delete ("VM1")
T4: TP-Event (Every Monday at 8:00:00 am) scale-in ("VM1", "Openstack", "change-in-capacity", 10, 60 s)
«Resource requirements » Instances : 0 Provider : -
S4
S3
Initial State
Normal State
Final State
«Resource Requirements » Instances : 10 Provider : Openstack
«Resource Requirements » Provider : AWS Instances : 10

Fig. 3. cRDM instance provided by the cloud user for the use case

Temporal Events. Actions may require that certain temporal events occurred, to be executed. We identify two patterns that these events can take: specified date and periodicity patterns as they are the most used in practice [1,2]. The specified date pattern specifies that an action needs to be executed at a specified date. It defined through a predicate called TS-Event (c) with c defined as **c::= (Date = D)** | **c** ∧ **c** | **c** ∨ **c**, where Date is a clock, and D is an absolute date expressed as yy-mm-dd hh-mm-ss am/pm format. As shown in Fig. 3, TS-Event (Date=2018-12-31 11:60:30 am) within the transition T5, T6 and T7 represents a temporal event expressed using this pattern, which will be triggered on 2018-12-31 at 11:60:30 am. Furthermore the periodicity pattern specifies that a certain action needs to be executed following a certain periodicity rule over time, which is defined using TP-Event (p) predicate with p is defined as **p::=D** | **p** ∧ **p** | **p** ∨ **p**, where D is a date defined following a periodicity rule and expressed using the following EBNF notations.

```
<D>::= 'Every' <weakday>'-'{<weakday>'-'} 'at' <Time>
|'Everyday' 'at' <Time> 'Except' <weakday>'-'{<weakday>'-'}
<weakday>::= Monday| Tuesday |....| Sunday
<Time>::= <Hour>':'<Minute>':'<Second>'am'|'pm'
..........
```

For example, as shown in Fig. 3, TP-Event (Every Saturday at 8:00:00 am) within T3 represents a temporal event, expressed using the above notations, which shall be triggered every Saturday at 08:00:00 am.

Resource Related Events. Actions need to be executed once certain resource metrics reach a predefined threshold. This type of event includes two sub-types of event that we call Resource Usage Events and QoS Events. Both event sub-types are defined using the predicate Q-event(q, fc, op, tr, u, w), where q is a metric, fc precises the metric evaluation way (average, maximum, etc.), op $\in$ $\{=, \neq, <,$ etc.$\}$, tr is a threshold value, u is the used unit, and finally w defines a time window during which the metric could be evaluated. Resource Usage Events are defined using metrics related to the resource usage such as CPU usage, RAM usage, etc. While QoS events are expressed through QoS metrics such as availability, response time, etc. As shown in Fig. 3, the user defines their resource related events in terms of CPU usage, such as Q-Event (cpuusage, average, >=, 80, %, 300 s) within T1, which checks whether the CPU usage average is greater than or equal 80% for 5 min across all VM instances.

User Action Events. Actions are executed at the behest of a cloud user. For instance, a cloud user can demand to set manually the capacity of the VM resource. These events are defined through a predicate called U-Event(c) over a set of messages M, with c defined as **c::=(message=e)** $|$ **c** $\wedge$ **c** $|$ **c** $\vee$ **c**, where message is an incoming message from a user and e $\in$ M. For example, U-Event (message=Stop) will be triggered when we receive from user a stop message.

3.3.2 Reconfiguration Actions

They specify how a cloud resource should behave when certain events occurs. To identify them, we examined the reconfiguration and elasticity mechanisms analyzed on research surveys [4,5] and proposed by cloud providers and orchestration tools. We organize them into five categories: horizontal scaling, vertical scaling, migration, application reconfiguration and basic actions. Each reconfiguration action is defined by a name and a set of attributes defining the required inputs to execute this action. In the following, we choose JSON schema to illustrate each action.

Horizontal Scaling (HS): represents the possibility to scale out and in by adding or removing instances (e.g. VM). As shown in Fig. 4(a), HS action can have scale-in or scale-out name and contains 4 attributes. The resource-target represents the resource name that will be adjusted. The adjustment-type specifies the adjustment way which can be change-in-capacity (Add/Remove the given number of resource instances), exact-capacity (Set the current number of resource instances) or percent-change-in-capacity (Add/Remove a given percentage to the instances number). The adjust specifies the adjustment value. Finally, the cooldown indicates the time period during which no other actions on the same resource will be taken.

Vertical Scaling (VS): aims at scaling up and down of resources such as processing, memory. As shown in Fig. 4(b), VS action can have scale-up or

```
(a) { "HorizontalScalingAction" : {"Name :"{"type":
"enum['scale-in', 'scale-out']"},
   "ActionAttributes":{
         "resource-target":{"type":"string"},
         "adjustment-type":{"type":"enum['exact-capacity',
          'change-in-capacity',  'percent-change-in-
capacity']"},
         "adjust":{"type": "string"},
         "cooldown":{"type": "number"} } }

(c) { "MigrationAction" : {"Name": "migrate"},
   "ActionAttributes":{
         "target":{"type": "string"},
         "host":{"type": "string", "default": "None"},
         "type":{"type": "enum ['Cold', 'Hot']"},
         "cooldown":{"type": "number"} } }
```

```
(b) { "VerticalScalingAction" : {"Name": {"type"  :
"enum ['scale-up', 'scale-down']"},
     "ActionAttributes":{........,
           "attribute-target":{"type": "string"},
         ...., ....., ....., ......... } }

(d) { "ApplicationReconfigurationAction" : {"Name":
"update"},
   "ActionAttributes":{
         "resource-target":{"type": "string"},
         "attribute-target":{"type": "string"}
          "attribute-value":{"type": "string"}  } }

(e)  { "BasicAction" : {"Name": {"type" : "enum ['start',
'stop',  'delete', 'restart']"},
   "ActionAttributes":{
         "resource-target": {"type": "string"} }}
```

Fig. 4. A JSON schema describing the main attributes for each reconfiguration action

scale-down name and has same attributes of HS action. Moreover, we add the attribute-target to indicate the attribute name (e.g. CPU, RAM) to be modified.

Migration: includes two migration types: VM Migration and Application migration. In this category, we identify only one action which is a Migration action. As shown in Fig. 4(c), a migrate action has migrate name and contains a set of attributes: The target represents the VM or application component name that will be migrated. The host which can be filled by the host name (e.g. The node name in case of a VM migration or the VM name in case of an application component migration). In some cases, it can be filled by None, so the controller action must choose the appropriate host automatically. The type indicates the migration type: Cold or Hot [4].

Application Reconfiguration (AR): consist of changing specific application aspects such as DB recovery policy. As shown in Fig. 4(d), this action has as name update and contains a set of attributes: the resource-target represents the application component name; the attribute-target indicates the attribute name to be modified and the attribute-value indicates the new value to be assigned.

Basic Actions: regroup the basic actions applied on a cloud resource including: start, stop, restart and delete. As shown in Fig. 4(e), each basic action should contain a Name that should be start, restart, stop, or delete and resource-target as attribute that indicates the resource to be manipulated.

Example. As shown in Fig. 3, the user used both the HS and basic actions: scale-out (“VM1”, “AWS”, “change-in-capacity”, 5, 60 s) within T1 represents an instance from HS action, which allows to add for VM1 resource 5 instances from AWS provider, where 60 s represent a clowdown period that must be respected after triggering this action. delete (VM1) within T5, T6, T7 represents a basic action that aims at removing permanently the resource VM1.

4 Supporting Multi-providers Abstractions

We now present how the above abstractions can be extended in order to support the multi-provider scenario. More specifically, we need to identify required events leading a user to acquire cloud resources from a new provider and correspondingly extend the reconfiguration actions. In fact, once a service has been deployed in a cloud resource from a specific provider, different situation can occur at runtime: (1) Service can be scaled manually (User Action Events) or dynamically (Resource Related or Temporal Events) from a new provider; (2) Service can be migrated to another provider at the behest of its user (User Action Events); (3) Service can be migrated in case that the current provider does not respect the QoS constraints (Resource Related Events); (4) finally, service can be migrated to another provider when this provider offers better utility than the previous one. Consequently, our basic resource model should be extended to support these new considerations. To support (1), (2) and (3) we have to extend the reconfiguration actions (i.e. horizontal and vertical scaling, migration, etc.) by adding the property provider as an attribute to these actions. For instance, the scale-out ("VM1", "Openstack", "change-in-capacity", 10, 60 s) within T3 shows HS action from a new provider which is OpenStack. Furthermore, to support (4), a new type of events should be defined along with the above events, that we call Market related events. As Market related events depend on QoS and resource properties, we define it as one of the Resource Related Event.

Market Related Events. Events can be triggered whenever there is a cloud resource offer providing QoS or any other property (e.g., price) that better satisfies the user needs. We expressed it using the predicate M-Event (q, r, op, tr, u, p), where q is the QoS or resource property, r is the cloud resource, and op, tr and u have the same definition within Q-event. While p indicates the new provider which can be filled by the provider name such Openstack, or by any, so, the controller action must automatically choose the most appropriate provider.

5 Implementation and Evaluation

5.1 Proof of Concept

We built a proof-of-concept (POC) prototype for cRDM, called *cRDM Core*[1], which supports users to describe and configure their cloud resources by exploiting the underlying cloud orchestration and providers solutions. Figure 5 shows an overview of our prototype architecture. cRDM core is the central part in this architecture, consisting in: *cRDM editor*, *cRDM Validation*, *cRDM Generation* and *Elasticity controller*. The *cRDM editor* implemented using Sirius technology[2] and Java to provide a drag-and-drop interface enabling user to graphically instantiate from cRDM model the corresponding cRDM instance. This instance is then simply serialized as JSON/XMI file. *cRDM Validation* exploits this file to

[1] http://www-inf.it-sudparis.eu/SIMBAD/tools/Cloud-RDM/.
[2] https://www.obeodesigner.com/en/product/sirius.

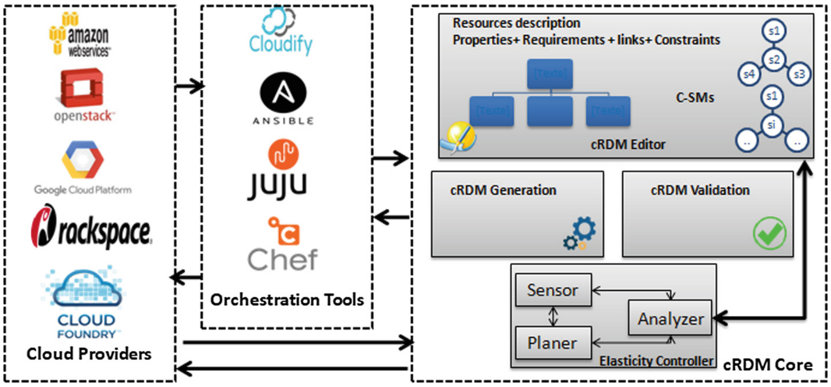


Fig. 5. Architecture overview

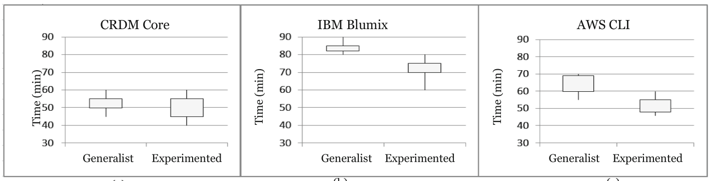
check the user cRDM instance consistency by verifying both its syntax and structure. The *cRDM Generation* proceeds to generate all required files for ensuring the deployment task. This generation is based on a model-driven generation technique and a set of connectors that serve to interpret the high-level descriptions related to cloud resources and identify their low-level scripts and commands required to manage and configure them. While the generation phase is important, it is out of the scope of this paper due to the space limitation. Finally, the *Elasticity Controller* is implemented in Java and allows the execution of elasticity policies. From the defined C-SM machines, the elasticity controller detects the needed information to be monitored and identifies the appropriate reconfiguration actions to adapt the cloud resources at runtime.

5.2 Evaluation

We conducted two experiments using the implemented prototype to evaluate the productivity and expressiveness of our cRDM in comparison with traditional solutions. All experiment objects and results have been published online (see footnote 1).

5.2.1 Experimentation 1

Experimental setup. We evaluated the productivity of our cRDM by conducting a user-study with 18 participants from Master students of the university of Paris-Saclay. We test the productivity in terms of the efficiency and the usefulness of cRDM in describing the cloud resources and their elasticity behaviour. The *efficiency* is measured in terms of the time taken to complete the modeling and deployment tasks. The *usefulness* is determined via a questionnaire that asses the participants feedbacks about our cRDM. The questionnaire (see footnote 1) devised into four main parts: Background, Functionality, Usability, Insights/Improvements. The background questions aim at evaluating the participants familiarity with existing cloud resource description and elasticity tools.



(a) (b) (c)

(d)

	cRDM Core	IBM Bluemix	AWS CLI
Average	50,625	73,5625	61,75
Observations	18	18	18
Variance	59,58	69,8	94,3

Fig. 6. Time to complete the task; (a), (b) and (c) Time grouped by level of expertise; (d) Average time for all the participants.

The functionality questions verify whether the participants correctly understand the main functionalities of cRDM. The usability questions sought to discover whether the key modeling abstraction offered are easy and intuitive. We asked all participants to model and deploy the application described in our motivation scenario only on AWS provider using our cRDM Core. For quantitative comparison purpose, we asked them to do the same scenario with two provider-specific solutions viz. IBM bluemix platform [2] and AWS CLI [1]. For the sake of analysis, we classified a total of 18 participants into 2 main groups: (1) Generalist: who have average knowledge of cloud tools (12 participants) and (2) Experimented (6 participants): who have a sophisticated understanding of cloud tools.

Evaluation Results. Results of the experiment in Fig. 6(a), (b) and (c) show the time taken using cRDM core, IBM bluemix and AWS CLI for the modeling and deployment tasks. As shown, it was pleasantly surprising that even generalist participants demonstrated a significant reduction in time. More precisely, the time taken to complete these tasks was reduced by 17% in comparison to other solutions. On the whole, as shown in Fig. 6(d), the participants took on average 50 min using our tool, 73 min using IBM bluemix and 61 min using AWS CLI. This demonstrates the efficiency of our cRDM model. In fact, we argue that provider independent resource abstractions and graphical model like state machine to describe the elasticity behaviour significantly improve the time-to-modeling. In contrast, proprietary solutions such AWS CLI and IBM bluemix necessarily demand extensive programming and documentation efforts. More specifically, by using AWS CLI to deploy the requested application, participants are inevitably forced to understand firstly AWS CloudFormation model to create the corresponding template for both describing the required resources and their scaling scripts, which also like AWS CLI provides low-level resource descriptions. Likewise, participants are invited to understand diverse DevOps tools such cloud foundry CLI when using IBM bluemix.

Moreover, to evaluate the cRDM usefulness, we use the Usability section of the questionnaire by asking participants to rate the usability for each abstraction (scale 0–5). We examined the basic cRDM abstractions: Cloud Resource,

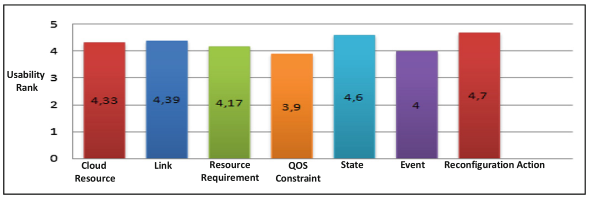


Fig. 7. Usability rate of the main cRDM abstractions

Link, Resource Requirement, QoS Constraint, State, Reconfiguration actions. We observed that the mean score for all abstractions in Fig. 7 is greater than the neutral value of 3 with a noticeable difference. Overall participants reported that our model is a familiar and intuitive, especially C-SM is not far from natural language and allows defining cloud resources elasticity behaviour in a very simple and easy way. Accordingly, giving these observations, we confirm that the key modeling abstractions offered are useful and comprehensible.

5.2.2 Experimentation 2

To evaluate the expressiveness of our cRDM, we used the two well known evaluation techniques in the literature [11]: Comparison of the model with standards or other proposed models; Application of the model to realistic examples or use cases. The former provides a qualitative evaluation, while the latter quantitatively evaluates our model.

Qualitative Evaluation. Herein, we test cRDM expressiveness by evaluating its overall coverage to others RDMs. We choose two RDMs: TOSCA [3] and AWS Cloud Formation [1]. We intentionally choose these solutions as they represent the range of the different types of RDMs available in cloud community: Provider-specific model and open standard. Concretely, we compared the concepts defined in the different models to the ones defined in our cRDM. The results summarized in Table 1(a) show that our cRDM model has a high coverage of TOSCA (80%) concepts related to cloud resources. In contrast, we observe

Table 1. cRDM coverage to (a) RDMs; (b) TOSCA use cases; (c) AWS CF use cases

RDMs	Concepts	Covered	Rate
TOSCA	26	21	80 %
AWS Cloud Formation	9	3	33%
Average			56.5%

(a)

	TOSCA use cases			
	case 1	case 2	case 3	case 4
cRDM instances	25	15	41	125
Generated TOSCA instances	36	18	64	172
TOSCA instances	36	18	64	172
Coverage by the generation	100%	100%	100%	100%

(b)

	AWS CF use cases			
	case 1	case 2	case 3	case 4
cRDM instances	26	36	66	64
Generated AWS CF instances	70	80	102	130
AWS CF instances	70	80	102	130
Coverage by the generation	100%	100%	100%	100%

(c)

a low coverage of AWS Cloud Formation concepts (33%). This is not surprising because we intentionally aimed to hide low-level and technical descriptions related to any provider-dependent solutions. By analyzing AWS Cloud Formation, we reveal that it provides user 9 main concepts to create and configure their cloud resources. However, only three concepts have inevitably to be included in the user template to specify the resources and their properties: Resources, Type and Properties. Therefore, in our cRDM, we support only these three concepts. Others Cloud Formation concepts include Metadata, Packages, Mapping, Transform, are totally omitted in our cRDM with the aim of avoiding any handling with low-level commands and hard coded scripts. Similarly, we only supported TOSCA concepts that are most commonly used in resource descriptions and do not oblige the user to deal with low-level scripting complexity like Interface and Artifact concepts. In the following, we show that the no-support of the low-level scripting at resource description level, does not reduce our cRDM expressiveness.

Quantitative Evaluation. In order to quantitatively evaluate our cRDM expressiveness, we rely on 8 use cases identified in the different chosen models: TOSCA (4 use cases) and AWS Cloud Formation (4 use cases). These use cases are specified as JSON and YAML templates. We use our cRDM core to construct the corresponding cRDM instances for these use cases using our cRDM model. Afterward, we exploit these instances to generate TOSCA and AWS Cloud Formation templates using our generation technique. Finally, we evaluate our cRDM expressiveness in terms of its coverage to use cases, by comparing the concepts instances in generated templates with the ones in use cases templates. Table 1(b) (respectively Table 1(c)) reports, for each TOSCA (respectively AWS CF) use case, the number of obtained instances using cRDM model, the number of obtained instances after the generation, and the number of instances that really exist in TOSCA (respectively AWS CF) use case as well as the coverage percentage of our cRDM after the generation. The obtained results show that we have a complete coverage (100%) of all used use cases, this meaning that all used use cases have been successfully covered when using our cRDM model during the generation. Therefore, we confirm that the no-support of low-level descriptions of cloud resources in our cRDM model does not have any influence on its expressiveness. On the contrary, this leading to a considerable reduction in the modeling complexity. For example, as specified in Table 1(b), to model the first TOSCA use case, we made only 25 instances using our cRDM. While using TOSCA model, this use case is composed of 36 instances since it includes low level and specific descriptions related to each defined cloud resource. To overcome this loss, we exploit the 25 instances created by our cRDM to interpret the required low-level scripts and commands. Therefore, we generate 11 additional instances. This means that the complexity modeling was reduced by 31%

Table 2. MCR rate for TOSCA and AWS CF uses case

	TOSCA use cases				AWS CF use cases			
	Case 1	Case 2	Case 3	Case 4	Case 1	Case 2	Case 3	Case 4
MCR rate	31%	17%	36%	28%	62%	55%	36%	51%

(i.e. 1−(25/36)). Table 2 reports the modeling complexity reduction rate (MCR rate) for each use case using our cRDM model.

5.3 Threats to Validity

Some potential threats to validity exist in our contribution. First, since we relied on cloud orchestration tools and providers-specific solutions, any change in these solutions oblige us to continually update our connectors to ensure a compliant generation. Second, we validated our model productivity through a cloud use case that supports only the horizontal scaling as it represents the most widespread mechanism in the cloud. Third, our use case has been conducted with 18 students and support only providers-specific solutions such AWS CLI and IBM Bleumix. We believe that a larger number of participants, including professionals and a use case supporting complex elasticity scenarios such as VM/Application migration, need to be considered. Fourth, we validated the productivity and expressiveness of our cRDM model. Our work request a further evaluation, including the correctness of our elasticity controller, which represents a good evidence that the defined state machines behave as expected at runtime. Furthermore, the performance evaluation of the whole system as well as a comparison with orchestration tools like Docker and Cloudify will also be considered.

6 Related Work

Various cloud resource and elasticity description languages and models have been proposed in industry and research. Market-leading providers like AWS CloudFormation [1] and CA AppLogic aim to describe and deploy complete application stacks. They propose provider-specific representations while our approach allows the description of cloud resources and their elasticity policies in a provider-independent way. Modern resource orchestration systems like Puppet, Juju, Ansible and Chef provide scripting-based languages for describing resource configurations over cloud services [18]. However, even sophisticated programmers are regularly forced to understand different low-level cloud APIs to create and maintain complex resource configurations and describe their elasticity policies. In contrast to the above, we contribute in providing high-level modeling abstractions that facilitate the description of cloud resources as well as their elasticity without referring to any low-level languages or providers-specific formats.

In research, to the best of our knowledge, cloud resources and their elasticity description have been studied separately. In [13], the authors introduce a description and deployment model called "Virtual Solution Model" that models a resource as a provider-independent resource configuration. However, this model allows users to describe resource configurations from a single provider for a single deployment. Moreover, various research works have concentrated on using semantic-based languages [6] to describe cloud resources. However, the largest amount of researcher's attention have been largely focused on cloud resources discovery and selection. Although there are a few of research work [8,10] that have recently emerged to deal with the orchestration aspects, none of them consider elasticity policies that are required at control runtime. Other related works have only focused on the definition and implementation of the cloud resource elasticity [7,12,14,19]. They have based on domain specific languages to define the elasticity policies for cloud resources. Our model differs from these works in four main points: (i) We based our model on state machine to describe the elasticity behaviour of a cloud resource, the state machine model considerably simplifies the manner of understanding compared to textual notations provided by the most of the previous work; (ii) We support multiple types of event, in contrast to the above solutions, which support only resource-usage based events; (iii) We support multiple type of elasticity actions, while the previous approaches support only vertical and horizontal scaling; (iv) Finally, our model allows to manage elasticity across multiple clouds, in contrast to these works, which define elasticity rules for scaling resources from only one provider.

7 Conclusion

This paper proposed a novel Cloud Resource Description Model based on a state machine, that provides high-level abstractions to describe cloud resources and their elasticity features. The adoption of state machine to describe the elasticity behaviour hides the actual complex implementation within cloud orchestration and providers tools. The model has been implemented and evaluated using experiments showing significantly its productivity and expressiveness. Future work will focus on evaluating first the correctness of our elasticity controller, and then the performance of the whole orchestration system. Besides, we plan to specify the formal semantics of our cRDM model.

References

1. AWS CLI: AWS Cloud Formation. https://aws.amazon.com/documentation/
2. IBM bluemix platform. https://www.ibm.com/cloud-computing/bluemix/
3. TOSCA. https://www.oasis-open.org/committees/tosca/
4. Al-Dhuraibi, Y., et al.: Elasticity in cloud computing: state of the art and research challenges. IEEE Trans. Serv. Comput. (TSC) (2017)
5. Naskos, A., Gounaris, A., Sioutas, S.: Cloud elasticity: a survey. In: Karydis, I., Sioutas, S., Triantafillou, P., Tsoumakos, D. (eds.) ALGOCLOUD 2015. LNCS, vol. 9511, pp. 151–167. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29919-8_12
6. Brabra, H., et al.: Semantic web technologies in cloud computing: a systematic literature review. In: IEEE SCC, pp. 744–751 (2016)
7. Copil, G., Moldovan, D., Truong, H.L., Dustdar, S.: SYBL: An extensible language for controlling elasticity in cloud applications. In: International Symposium on Cluster, Cloud, and Grid Computing, pp. 112–119 (2013)
8. Dastjerdi, A.V., et al.: Cloudpick: a framework for QoS-aware and ontology-based service deployment across clouds. Softw. Pract. Exper. **45**(2), 197–231 (2015)

9. Weerasiri, D., Barukh, M.C., Benatallah, B., Cao, J.: A model-driven framework for interoperable cloud resources management. In: Sheng, Q.Z., Stroulia, E., Tata, S., Bhiri, S. (eds.) ICSOC 2016. LNCS, vol. 9936, pp. 186–201. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46295-0_12
10. Dimitris, G.: A semantic framework to support the management of cloud-based service provision within a global public inclusive infrastructure. Int. J. Electron. Commer. **20**(1), 142–173 (2015)
11. Horkoff, J., Aydemir, F.B., Li, F.-L., Li, T., Mylopoulos, J.: Evaluating modeling languages: an example from the requirements domain. In: Yu, E., Dobbie, G., Jarke, M., Purao, S. (eds.) ER 2014. LNCS, vol. 8824, pp. 260–274. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12206-9_21
12. Jrad, A.B., Bhiri, S., Tata, S.: Description and evaluation of elasticity strategies for business processes in the cloud. In: SCC, pp. 203–210 (2016)
13. Konstantinou, A.V., et al.: An architecture for virtual solution composition and deployment in infrastructure clouds. In: International Workshop on Virtualization Technologies in Distributed Computing, pp. 9–18 (2009)
14. Kritikos, K., et al.: SRL: A scalability rule language for multi-cloud environments. In: International Conference on Cloud Computing Technology and Science (2014)
15. Liu, C., Loo, B.T., Mao, Y.: Declarative automated cloud resource orchestration. In: Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC) (2011)
16. Ponge, J., et al.: Analysis and applications of timed service protocols. ACM Trans. Softw. Eng. Methodol. **19**(4), 1–38 (2010)
17. Ranjan, R., Benatallah, B.: Programming cloud resource orchestration framework: operations and research challenges. CoRR (2012)
18. Thomas, D., Wouter, J., Bart, V.: A survey of system configuration tools. In: International Conference on Large Installation System Administration, LISA (2010)
19. Zabolotnyi, R., et al.: SPEEDL-a declarative event-based language to define the scaling behavior of cloud applications. In: IEEE World Congress on Services (2015)