A Model-Driven Framework for Interoperable Cloud Resources Management

Denis Weerasiri[1] Boualem Benatallah[1] Moshe Chai Barukh[1] Cao Jian[2]

[1] University of New South Wales, Australia
{denisw, boualem, mosheb}@cse.unsw.edu.au
[2] Shanghai Jiaotong University, Shanghai, China
cao-jian@cs.sjtu.edu.cn

Technical Report
UNSW-CSE-TR-201514
November 2015

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

The proliferation of tools for different aspects of cloud resource Configuration and Management (C&M) processes encourages DevOps to design end-to-end and automated C&M tasks that span across a selection of best-of-breed tools. But heterogeneities among resource description models and management capabilities of such C&M tools pose fundamental limitations when managing complex and dynamic cloud resources. We propose *Domain-specific Model*s, a model-driven approach for describing elementary and federated cloud resources as reusable knowledge artifacts over existing C&M tools. We also propose a pluggable architecture to translate these artifacts into resource descriptions and management rules that can be interpreted by external C&M tools like Juju and Docker. The paper describes concepts, techniques and current implementation of the proposed system. Experiments on a real-world federated cloud resource show significant improvements in productivity and usability achieved by our approach compared to traditional techniques.

1 Introduction

Cloud computing is evolving in both public and private cloud networks [12]. A third option involves a *hybrid* or *federated* cloud [2, 13], drawing resources from both public and/or private clouds. The many benefits of cloud computing, include enabling virtualisation capabilities and outsourcing strategies. It is estimated that nearly half of all large enterprises will comprise hybrid cloud service deployments by end of 2017 [6].

However, exploiting cloud services poses great complexity. As development becomes increasingly distributed across multiple heterogeneous, and evolving networks, it becomes increasingly difficult to manage interoperable and portable cloud resource solutions. Moreover, cloud applications may possess varying resource requirements during different phases of their life-cycle [12]. Consequently, designing effective cloud resource C&M techniques that cope with both heterogeneous and dynamic environments remains a deeply challenging problem.

Existing cloud C&M techniques typically rely on procedural programming (general-purpose or scripting) languages [12]. Modern systems such as: *Puppet*, *Juju*, *Docker* and *Amazon OpsWorks* provide script-based languages for managing resource configurations over cloud services [5]. Thus even DevOps (i.e., software engineers and system engineers who are collectively involved in designing, developing, deploying and managing cloud applications) would be forced to understand the different low-level cloud service APIs, command line syntax, Web interfaces, and procedural programming constructs - in order to create and maintain complex cloud configurations. Moreover, the problem intensifies with the increasing variety of cloud services, together with different resource requirements and constraints for each application. This inevitably leads to an inflexible and costly environment which adds considerable complexity, demands extensive programming effort, requires multiple and continuous patches, and perpetuates closed cloud solutions.

Drawing analogies from techniques in service composition domain, such as Business Process Execution Language (BPEL), we are encouraged to likewise support the *orchestration* of cloud resources by devising rich abstractions to reason about cloud resource requirements and their constraints. In this paper we therefore investigate how to effectively represent, organise and manipulate otherwise low-level, complex, cross-layer cloud resource descriptions into meaningful and higher-level segments. We believe this would greatly simplify the representation, manipulation as well as reuse of heterogeneous cloud resources. To enable this, we propose a methodology to support the automated translation of high-level resource requirements to underlying provider-specific resource and service calls. More specifically, this paper makes the following main contributions:

Domain-Specific Models for the representation of *Cloud Resource Management Entities (CRME)*: DevOps publicly share *CRME*s in forms of customizable configuration and/or management scripts, as well as elasticity rules. These *CRME*s can be classified into different *domains* such as tool-specific (e.g., *Juju Charm Store*[1]) and task-specific *CRME*s (e.g., e-Commerce software[2]). To better reason about this Configuration and Management (C&M) knowledge, we propose *Domain-specific Model*s and a declarative language for

[1]https://jujucharms.com/
[2]https://bitnami.com/stacks/e-commerce

describing and querying *CRME*s. Given that we architect this layer over existing systems, this significantly enhances the potential for knowledge re-use, since we can better harness interoperability capabilities. The proposed model features: a vocabulary and set of constructs for describing both elementary and task-specific cloud resources (e.g., VMs, database services, load balancer services), federated cloud resources (e.g., packaged virtual appliances), and the relationships amongst resources (e.g., configuration parameters, resource constraints and dependencies). Moreover, our model enables cloud resources to be combined to create higher-level virtual entities, called *Federated CRME*s and *Task-specific CRME*s which shield from complexity and heterogeneity of underlying cloud services. *Domain-specific Model*s also enable the incremental creation, organisation, curation and collective re-use of domain-specific knowledge artifacts to enhance productivity.

Automated Generation of C&M Rules: In order to combat the large number and variety of cloud resources Configuration and Management (C&M) languages (e.g. procedural, activity based and declarative), as well as the various heterogeneous tools/APIs involved to manage resources in different environments (i.e., public, private and federated) - we propose the automated generation of a parameterized resource C&M rule-model. This rule-model will deploy or automatically re-configure the appropriate SaaS, PaaS or IaaS resources with respect to objects in *Domain-specific Model*s. To enable this, we layer the notion of *Connectors* over our *Domain-specific Model*, which act to transform resources represented in the higher-level model into native *CRME*s. DevOps are thus empowered to write automated management tasks over events and actions exposed by the *Connectors* by defining Event-Condition-Action (ECA) rules. Behind the scenes the exposed actions transform *Domain-specific Model*s into low-level resource management rules, which thereby abstracts the complexity of the low-level interfaces (and communication protocols) of the native cloud C&M tools.

The core benefit of our contributions is to build up an ecological knowledge community of C&M tools by extracting resource C&M models of tools and represent them in a linked data model (i.e., Domain-specific models) such that common or related entities among different tools can be exploited. For example, by identifying common concepts among different tools who have different granularities of C&M features, we can seamlessly merge those features for end-to-end C&M via our system. For instance, a VM, which is deployed by a particular tool, can be modified by another tool with fine-grained configuration tasks (e.g., installing software within the VM) which are not supported by the initial tool.

The rest of this paper is organised as follows: In Section 2 we further elucidate sharing and re-use capabilities amongst existing cloud resource C&M techniques, highlighting their limitations. In Section 3 we present the overall system architecture of our proposed platform. In Section 4 we demonstrate our methodology via realistic scenarios. While in Section 5 we present our implementation and evaluation, and conclude in Section 6 with an examination of related work, and discussion of future work in Section 7.

2 Resource Sharing and Re-use for DevOps

As mentioned earlier, we observe that DevOps often share valuable knowledge regarding the C&M of cloud resources (e.g. resource configuration templates, elasticity rule templates). This knowledge may then be utilised, as DevOps may potentially discover knowledge artifacts (e.g. configuration scripts, documentations, forums, binary installers, and portable packages) when defining cloud management processes. For example, the user community of Ubuntu Juju (respectively, Docker) share Charms[1] (respectively, Dockerfiles[1]). Charms and Dockerfiles are a collection of configuration attributes and executable scripts that configure, install and start an application. Some other tools share non-textual packaging formats (e.g., Open Virtualization Format[2] (OVF) and Docker Images[3]) as resource artifacts. Other type of communities (e.g., Snaps in terminal.com[4]) share already deployed cloud resources. DevOps reuse those already deployed resources for purposes of monitoring and controlling them. In the the rest of this section, we examine the current ***limitations*** of existing approaches:

2.1 Heterogeneity in Cloud Management Languages and Tools

Amongst current cloud Configuration and Management (C&M) tools, DevOps often bear the burden of mapping application requirements to scripts that implement the underlying C&M tasks. Typically done using procedures over low-level resource control APIs. For instance, consider the description of a Java based Web application stack using Docker: First we would need to identify the required component resources (i.e., application engine and database) and their relationships; then, implement or reuse C&M scripts for each component resource (e.g., Apache Tomcat as application engine and MySQL as database). In Docker community these scripts are known as *Dockerfiles.* Docker provides a Command Line Interface[5] and a RESTful interface[6] which interpret *Dockerfiles*, build and deploy necessary resources known as *Containers* on a given Virtual Machine (VM).

As explained, this inevitably entails great complexity when exploiting cloud services, and the problems extenuates in distributed environment across multiple heterogeneous, autonomous, and evolving cloud services. More specifically, with existing cloud delivery models, developing a new cloud-based solution generally leads to uncontrollable fragmentation using different C&M languages and tools (e.g., Puppet, Chef, Juju, Docker, AWS OpsWorks) [4, 5, 8]. This makes it very difficult to develop interoperable and portable cloud solutions. It also degrades performance as applications or workloads cannot be partitioned or migrated easily and arbitrarily to another cloud when demand cycles increase.

[1]https://docs.docker.com/reference/builder/
[2]http://www.dmtf.org/standards/ovf
[3]https://docs.docker.com/userguide/dockerimages/
[4]https://terminal.com/explore
[5]https://docs.docker.com/reference/commandline/cli/
[6]https://docs.docker.com/reference/api/docker_remote_api/

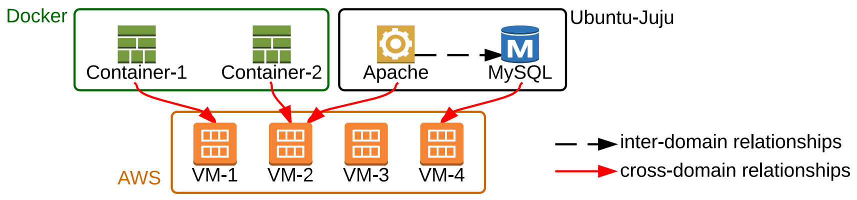

Figure 2.1: Cross vs. Inter domain relationships within a Federated Cloud Resource

2.2 Federated Cloud Resource Relationships

DevOps typically employ multiple C&M tools to automate end-to-end management tasks (e.g., deploying Docker Containers in VMs, managed by AWS (see Fig. 2.1)). As every C&M tool has tool-specific resource description models and management capabilities, DevOps are required to implement ad-hoc scripts to coordinate C&M tasks among these different tools. Consequently, these ad-hoc scripts introduce hard-coded relationships among resources that are orchestrated by different tools. Reusing knowledge artifacts, which include such ad-hoc scripts, is not scalable as DevOps require to manually analyze those knowledge artifacts to realise cross-domain relationships among resources within a federated cloud.

Moreover understanding and visualizing component resources and their relationships are always useful due to several reasons. For example, managing cloud resources without properly understanding or ignoring the available relationships leads to Service-Level-Agreement (SLA) violations. A team of DevOps may deploy multiple *MySQL* database servers in a VM for different applications without reusing an *existing* database server due to little awareness about resources already deployed within the VM. In general, limited awareness of global and local view of component resources and relationships hinders optimal resource C&M processes that span across different tools. Consequences of such situations can be catastrophic by disrupting the complete resource infrastructure[7].

3 Federated Cloud Resources Management Architecture: An overview

To overcome the limitations described in Section 2, we propose a layered architecture that enables: (a) *Domain-specific Models*; linked-data-model based Configuration and Management (C&M) of cloud resources; and (b) *Connectors*; automated translations of these high-level *Domain-specific Models* into low level resource descriptions and management rules. This architecture thereby shields consumers from heterogeneity and complexity of underlying cloud services. Furthermore, curators (i.e., experts in cloud resource C&M tools) in our framework incrementally and collectively: (i) derive *Domain-specific Models*; and (ii) implement transformation rules. DevOps communities are thus able to reuse these models and transformation rules to describe and manage elementary, federated and task-specific cloud resources.

[7]http://aws.amazon.com/message/65648/

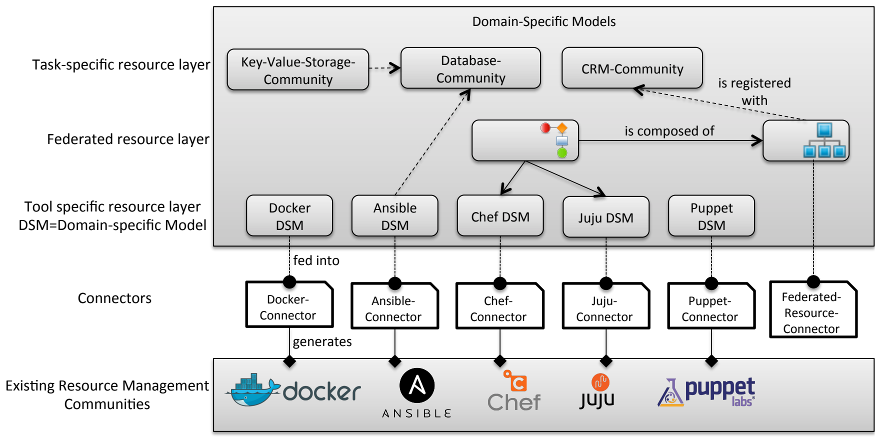

Figure 3.1: System Overview

Fig. 3.1 illustrates the system design and interactions of main layers in our proposed approach; which are elucidated as follows:

Existing Resource Management Communities represent tools and APIs available for cloud resource C&M. We discussed about cloud resource C&M communities and their knowledge reuse aspect extensively in Section 2.

Domain-specific Models layer consists of three sub layers: (i) Tool-specific resource layer; (ii) Federated resource layer; and (iii) Task-specific resource layer. All sub-layers consist of a collection of *Domain-specific Model*s. Tool-specific resource layer includes *Domain-specific Model*s, each of which represents cloud resource management entities (CRME) (e.g., resource description, deployment and management events and actions) and relationships among those entities of a particular *Existing Resource Management Community*. For example, *Docker DSM* (see Fig. 3.1) includes linked CRMEs, which are provided by Docker engine, to describe, configure and manage cloud resources. Tool-specific *CRME*s can also be combined to create higher-level *Domain-specific Model*s to represent and manage federated cloud resources, which are to be managed by a set of *Existing Resource Management Communities*. Federated resource layer represents such higher-level *Domain-specific Model*s. Task-specific resource layer represents categorisations of *Domain-specific Models* for specific categories of cloud applications. For example, *Database Community* (see Fig. 3.1) may include a set of *Domain-specific Model*s, each of which facilitates particularly for Configuration and Management (C&M) of databases such as key-value storages, relational databases and graph databases. The goals of *Domain-specific Model*s are to (a) capture essential C&M entities: resource description model, management action model and event model offered by a particular community by abstracting out heterogeneous notations (e.g., Docker Image, Juju Charm, OVF[1]); and (b) query and analyse (e.g., dependency analysis) the underlying cloud resources. We further elucidate on *Domain-specific Models* in Section 4.

Connectors layer includes rules that transform *Domain-specific Model* based resource descriptions into native resource description artifacts. *Connector*s also include rules that transform event and action entities of the *Domain-specific Model* into event descriptions and actions exposed by the *Existing Resource*

[1]http://www.dmtf.org/standards/ovf

Management Community. DevOps implement management tasks over events and actions exposed by *Connector*s. For example, DevOps may implement ECA rules such as: when *event* patterns (e.g., a user changes a configuration attribute of a cloud application) are matched and their *conditions* (e.g., application is started) are satisfied, the specified C&M *actions* (e.g., deploy, delete, re-configure, start and stop) are fired. Underlying implementations of the specified actions transform *Domain-specific Model*s into low-level resource descriptions and manipulation rules. A further goal of *Connector*s is to abstract out the complex and low-level interfaces and communication protocols of C&M tools from users. We further exemplify and illustrate technical details of Connectors in Section 4.3 using Docker as an example.

4 Extracting Domain-specific Models from Tool-specific Resource Artifacts

In this section we illustrate our methodology of analysing existing Configuration and Management (C&M) tools to derive *Domain-specific Model*s. Using a real-world example, we demonstrate how we derive their key entities (i.e., resource description entities, management actions and events) which thereby constitute the *Domain-specific Model*s.

We built *Domain-specific Model*s for a diverse range of tools and languages: *Docker*, *Juju* and *TOSCA*. For each, we first analyzed existing knowledge sources (e.g., C&M language specifications, user documentations, forums and resource description repositories) to understand and extract key entities for describing cloud resources. Next, we extracted relationships between the entities by understanding how entities are associated when describing composite cloud resources. These entities and relationships constitute the resource description model of the respective *Domain-specific Model*. Likewise, to model management capabilities of the *Domain-specific Model*, we again analyzed knowledge sources and extracted what actions and events are provided by these tools, such as for manipulating the given resource. Finally we integrated the extracted events and actions as two sets of entities to the *Domain-specific Model*.

4.1 An Embryonic Cloud Resource Configuration & Management Model

During this kind of reverse engineering analysis, we need a language that captures characteristics of *Domain-specific Model*s. It should be noted that we use conventional Entity-Relationship (ER) models to represent *Domain-specific Model*s. In this manner, ER-constructs can capture the high-level design of *CRME*s as entities, and likewise explicitly represent relationships between *CRME*s. Additionally, ER-model based resource descriptions act as documentations that explicitly describe the resource, and the relationships amongst other resources. This may then allow DevOps to browse, visualise and comprehensively analyse a global view of cloud resources across multiple C&M tools. Alternatively with existing script-based approaches, complex cloud resources are often just documented separately in forms of ad-hoc Wikis that outdate quickly unless continuously maintained. ER based *Domain-specific Model*s also support a machinereadable syntax, which is consumed by software like *Connector*s to automatically generate cloud resource descriptions, deployment scripts and management scripts of C&M tools. Our embryonic data model consists of two aspects.

1. Resource Description Model: It describes language constructs provided for representing cloud resources in a C&M tool, in terms of relevant entities and relationships. Entities and relationships include attributes that characterise them. For example, an entity that represents a VM may include CPU, memory and storage as attributes.

2. Resource Management Model: It expresses language constructs, provided to configure, deploy, monitor and control cloud resources by a C&M tool. Resource Management Model consists of two sub-models.

   (a) Action Model: It specifies available actions (e.g., deploy, configure, migrate), which manage cloud resources, as a set of entities with relevant attributes that express required input and output parameters.

   (b) Event Model: It expresses events related to the life cycle of cloud resources in terms of entities with necessary attributes that describe events [12]. It should be noted that, the issues of event detection while important, they are complementary to research issues addressed in our work and outside the scope of this paper.

In the following section we explain how we leverage our embryonic model to define a *Domain-specific Model*, using *Docker* as a real-world example. We chose *Docker* as it is open-source and emerging industry standard, which is vastly praised by DevOps communities.

4.2 *Docker*-based Domain-specific Model

Docker comprises *CRME*s that required for application deployment over software *containers*. Docker is a Container-based virtualization technique, which offers a lightweight and portable resource isolation alternative to VMs. Container-based virtualization techniques have been emerged to simplify and accelerate the modeling and deployment of cloud resources. More specifically, for composite service-based cloud resources, which depend on multiple service platform for their operations, container-based virtualization techniques enable accelerated and efficient modeling and deployment of optimally configured, scalable and lightweight platform instances. By analyzing the Docker language specification[1] we identified four key resource description entities: (1) `Container`, (2) `Image`, (3) `Registry` and (4) `Hosting-Machine` (see Fig. 4.1).

The central entity: `Container` represents a virtualised software container where DevOps deploy an application or a component of an application (e.g., an *Apache Web-Server* installed on *Ubuntu OS* with dependent libraries). Deployment knowledge of the application and its dependencies (or application components) is represented via the entity, `Image`. Such knowledge is either represented using one monolithic `Image` instance or a set of `Image` instances that each represents deployment knowledge of an application component. In other words, the `Image` possesses deployment knowledge required to instantiate

[1]https://docs.docker.com/reference/builder/

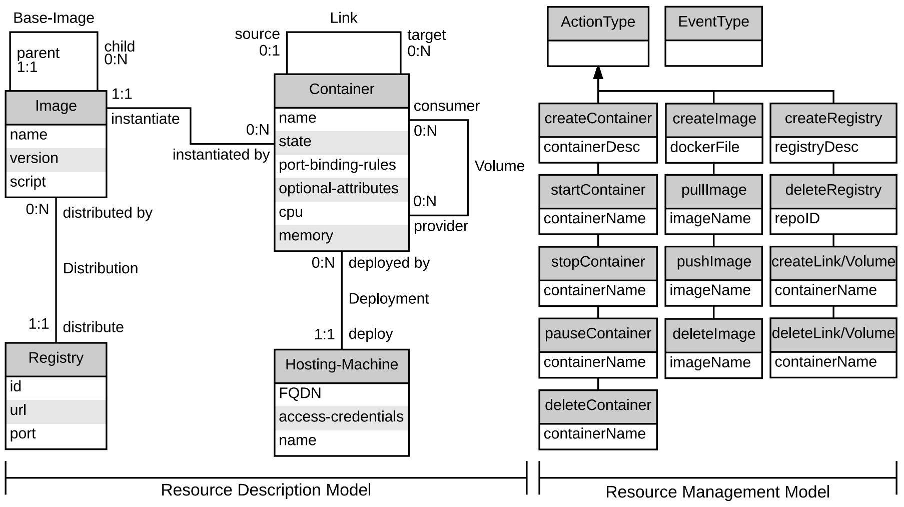

Figure 4.1: *Domain-specific Model* for Docker

a `Container`. The entity `Registry` represents a repository of `Image`s where DevOps organise, curate and share resource deployment knowledge. Likewise, the entity `Hosting-Machine` represents the location where a Container is hosted (e.g., VM or physical machine).

By further analyzing into the main entities, we derive attributes that characterize each entity. For example, a `Hosting-Machine` in Docker is identified using a FQDN (Fully Qualified Domain Name) of VM or physical machine.

Finally we extract and integrate events and actions, offered by the tool (see Fig. 4.1). For example, Docker exposes actions like create, start, stop, pause and delete to manipulate `Containers`. Similarly Docker offers actions to manipulate other entities and relationships, but does not support any events.

4.3 Implementing Connectors

Once a *Domain-specific Model* is described, curators may then implement a *Connector* that serve to bridge the *Domain-specific Model* with the interface of the particular Configuration and Management (C&M) tool. In our framework, we provide a mechanism for DevOps to contribute *Connector*s. The interface of our *Connector* has one mandatory operation called `init`. DevOps must implement this operation such that `init`: (i) accepts ER-model based resource descriptions; (ii) generates native resource descriptions; and (iii) return a unique id that represents the generated resource description. DevOps refer this unique-id for subsequent management operations on that particular resource description. Additionally a *Connector* can have any number of operations that include rules to perform management operations supported by the native tool. For example, the *Connector* for Docker has a method called `createContainer` which: (a) accepts a unique id; (b) prepare the `Hosting-Machine` to deploy a `Container`; and (c) invoke `docker run` command in the Docker CLI[2] along with an `Image`.

[2]https://docs.docker.com/reference/commandline/cli/

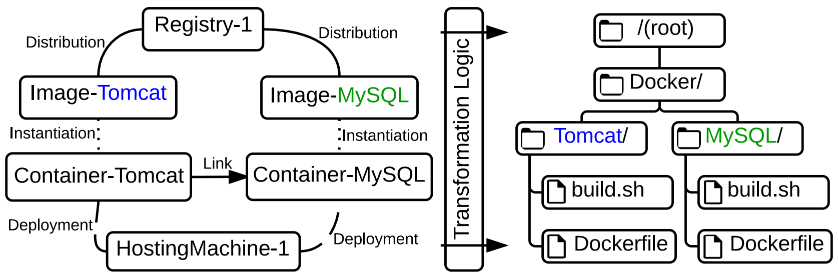

Figure 4.2: Technical transformation of *Domain-specific Models* (per each `Image` instance)

Therefore, once both the *Domain-specific Model* and *Connector* are registered in our framework, DevOps are then able to create cloud resources and moreover, implement management processes based on the *Domain-specific Model*.

Finally, although we demonstrate our methodology for a specific framework chosen above, the same could equally be applied to derive the *Domain-specific Models* and *Connectors* for any cloud resource C&M tool in our framework. In Appendix **??**, we illustrate the *Domain-specific Model* for Ubuntu juju.

Technical transformation of Docker-based Domain-specific Models:

The transformation logic to generate the native resource descriptions from instances of the *Domain-specific Models* are heavily linked to the particular *Existing Resource Management Communities*. We describe the approach below to perform a transformation from instances of Docker-based *Domain-specific Model*. The right side of Fig. 4.2 shows two files, named *Dockerfile* and *build.sh*, are generated for each `Image` instance, named *Image-Tomcat* and *Image-MySQL*, in the left side of Fig. 4.2. *Dockerfile* is a script that includes configuration rules that are required to generate a concrete Image in Docker runtime. These configuration rules are stored within the `script` attribute of an `Image` instance (see Fig. 4.1).

The file, *build.sh* is generated based on a sequence of commands which (1) read the *Dockerfile*, (2) generate a concrete Image, (3) upload the generated concrete Image to a specified `Registry` (i.e., *Registry-1* in Fig. 4.2), and (4) create a concrete Container from the concrete Image in a specified `Hosting-Machine` (i.e., *HostingMachine-1* in Fig. 4.2). In addition, *build.sh* may include commands to instantiate relationships (i.e., `Links` and `Volumes` in Fig. 4.1) between dependent concrete Containers. The transformation logic extracts required input data for these commands from attributes specified within the instances of Docker-based *Domain-specific Model*.

When it is required to *reuse* an existing Image from Docker Registry, instead of constructing one from scratch, `Image` instances are modeled without a `script` attribute. In such situations, the transformation logic does not generate a *Dockerfile*, but include additional commands within *build.sh* to reuse an existing concrete Image to generate a concrete Container.

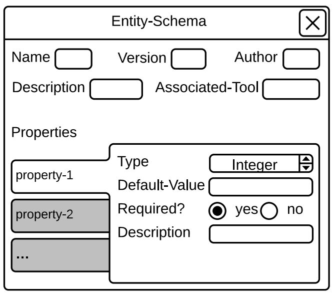

Figure 4.3: Entity-Schema

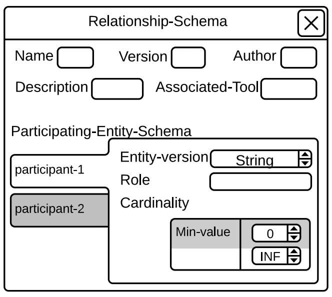

Figure 4.4: Relationship-Schema

4.4 Incremental & Collective Evolution of Domain-specific Models

Analyzing a typically script-based C&M tool, in order to extract its Cloud Resource Management Entities (CRME), is inherently a manual and tedious process. It also entails DevOps' expertise of the particular C&M tool. To facilitate this manual process, our proposed framework allows DevOps to incrementally and collectively create and curate *Domain-specific Models*. In this way, distributed curators may build upon the efforts of others, and thus provides a more reliable solution.

As *Domain-specific Model*s are based on ER model, our framework provides DevOps two schemas: Entity-Schema and Relationship-Schema (see Fig. 4.3 and 4.4) respectively, showing how we would define the structure of entities and relationships among entities of *Domain-specific Model*s. DevOps leverage these two schemas and create schema objects to describe *CRME* of C&M tools. In our current implementation, we reuse JSON-Schema specification [7] to define Entity-Schema and Relationship-Schema. Accordingly, the derived *Domain-specific Model*s employing the ER-schemas for a particular C&M tool result the models as previously illustrated (see Fig. 4.1).

The Entity-Schema and Relationship-Schema enforce curators to provide mandatory attributes (i.e., name, version, author, associated tool) for book-keeping and curation tasks of *Domain-specific Model*s. The Entity-Schema allows defining any number of arbitrary attributes under Properties section. The Relationship-Schema enforces curators to specify two participating entities (e.g., the relationship between `Container` and `Image` in Fig. 4.1), roles of the entities and cardinality constraints (i.e., minimum and maximum number of entity objects that may join the relationship).

5 Implementation and Evaluation

For evaluation purposes, we built: (i) A Java-based Proof-Of-Concept (POC) prototype of our framework; and (ii) *Domain-specific Model*s for *Docker* and *Juju*. We conducted two experiments analysing overall usability (i.e., learnability and efficiency) and productivity (i.e., total number of lines-of-code (LOC) to produce a solution) of our approach.

Implementation:

Our POC implementation includes a Git repository where *Domain-specific Model*s and their objects reside. We also implemented a Command-Line-Interface (CLI) that create and manage model objects in the repository. *Domain-specific Model*s in the repository are encoded using JSON-Schema[1] such that existing tools[2] are reused to: (i) generate template resource descriptions as Javascript object Notation (JSON) objects; (ii) verify; and (iii) visualise those objects. To generate templates of resource descriptions and instantiate them, we integrated a JSON Editor[3] into our POC implementation, which generates resource description templates for a given *Domain-specific Model*. *Connector*s are implemented as RESTful services which are exposed via ServiceBus API (based on our previous work) [3]. To execute ECA rules over *Domain-specific Model*s, we implemented a Java-based rule engine. Based on the methodology explained in Section 4, we derived two *Domain-specific Model*s from *Docker* and *Juju* (see Fig. 4.1) and registered in the Git repository. *Docker* and *Juju* were chosen due to their large community-bases. Furthermore, *Docker* and *Juju* specialise on two different concerns (i.e., application deployment inside software containers vs. application deployment across VMs and software containers), which facilitate to evaluate the adaptability of our approach to cater different aspects of cloud resource C&M.

Use-Case Scenario:

We demonstrate our implementation based on a use-case scenario. Consider we would like to model and deploy a software development and distribution platform. This platform is intended for software engineers who want to manage the entire lifecycle of a project. Multiple projects can leverage this platform by just cloning the deployment multiple times. This platform requires an AWS-EC2 VM where a Docker Container resides in. The Docker Container includes Redmine[4], a project management service, and a Git client[5]. The Redmine service is intended to (1) extract commits from a specified source repository in GitHub via the Git client and (2) link them with relevant bug reports. In addition AWS-S3 bucket (i.e., a key-value storage), which acts as a software distribution repository, is required.

To model the above resource configurations, we first use the JSON Editor[15] to create a graph of objects from Juju-based *Domain-specific Model*. Such that, the object graph represents an AWS-EC2 VM, where Docker engine is installed. We then deploy the object graph via our CLI. We invoke the init action of Juju Connector via the CLI with the object graph as an input. The init action (1) commits the object graph into the Git repository, (2) generates a native resource description, and (3) returns a unique reference id back to the user. We then invoke deployCharm action with the returned reference id. The successful execution of deployCharm action (1) deploys the AWS-EC2 VM, and (2) commits a new version of the object graph into the Git repository. The new version

[1]http://json-schema.org/latest/json-schema-core.html
[2]http://json-schema.org/implementations.html
[3]https://github.com/jdorn/json-editor
[4]http://www.redmine.org/
[5]http://git-scm.com/

represents the deployed VM. From the new version, we extract the IP address and access credentials, which are required when deploying the Docker Container.

Next, we model an object graph from Docker-based *Domain-specific Model*, such that the object graph represents resources and dependencies required to deploy Redmine and Git client as a Docker Container. The `Hosting-Machine` object of the object graph includes attributes whose values are initialized with the IP address and the access credentials of the previously deployed AWS-EC2 VM by Juju Connector. Similar to how we used CLI with Juju connector, we execute init and deployContainer actions to deploy the Docker Container and commit the object graph in the Git repository.

Once resources are deployed, they generate events. Relevant Connectors are responsible to capture those events and publish them through defined event types in the relevant *Domain-specific Model.* ECA rules can be deployed in our rule engine or employ CloudBase (based on our previous work) [14] to subscribe to particular events and implement management processes by exploiting actions offered by the relevant *Domain-specific Model.*

Evaluation:

To evaluate our approach, we measured the overall productivity and usability gained by *fourteen* DevOps (*eight* system administrators and *six* software engineers) from a cloud-based software development company. Prior to the experiments, we introduced our system through presentations and hands-on sessions. We then provided each participant a deployment specification of the software development and distribution platform, which we introduced under the *Use-Case Scenario* in Section 5. The deployment specification only illustrated the high-level requirements and participants were supposed to understand high-level requirements and implement those requirements using our system. For quantitative comparison purposes, we implemented the same deployment specification in three other cloud resource management approaches; (a) Docker, (b) Juju; and (c) Shell scripts. The main reason to choose Shell scripts was to estimate an upper bound of the result set.

We measured (i) the total number of LOC (*actual* LOC written and how many generated by our approach), excluding white spaces and comments; (ii) number of external dependencies/libraries required to describe and deploy the software development and distribution platform; and (iii) time taken to complete the modeling task. We measured the correctness of the modeling tasks by deploying each resource description and checking whether the resultant deployment complied with the initial deployment specification.

5.1 Analysis and Discussion

Results of the experiment (see Table 5.1) show that lines-of-code (LOC), number of external dependencies, and time-to-modeling are improved when using our framework over other prevalent resource management techniques. More specifically, the time-to-modeling is reduced by 26.7% compared to the average of other approaches. We argue that graphical modeling support and resource management over explicit event and action models improve the time-to-modeling. Comparing with proprietary and script-based management languages like *Docker* and *Juju*, we argue that providing an entity-relationship (ER) model

Table 5.1: Results of the experiment

Parameters	**Shell Scripts**	**Docker**	**Juju**	***Our approach***
average time-to-modeling (min)	103	95	72	66
#lines-of-code (LOC)	107	116	127	82 (manual) 839 (generated)
knowledge shareable from different C&M tools?	no	no	no	yes

based abstraction for describing cloud resources further improves the time-to-modeling for users. Moreover, our high-level modeling approach enables knowledge reuse which much easily facilitates implementing management processes spanning across multiple vendors. Pure shell-scripts, *Docker* and *Juju* based solutions required 116 LOC on average to model the deployment plan. Whereas in contrast, our framework based solution only required 82 manual LOC and generated 839 LOC - due to the assistance of the JSON-based resource description templates and Java-based *Connector*s which are more verbose compared to shell script based approaches. The study thus confirms the overall improved usability and productivity by employing the *Domain-specific Model* methodology. Moreover, by embracing a knowledge-sharing paradigm (inspired from industry[6]), the benefits of our approach are further improved: Given the fact users in this scenario would not require the efforts of development, registration and maintenance of the resource descriptions - since this could be pre-done once and re-used multiple times for the benefit of many.

6 Related Work

In this section we briefly explore the technological landscape and survey cloud resource Configuration and Management (C&M) languages; as well as interoperability concerns amongst cloud resource C&M tools. We compare and contrast our proposed approach with these related work.

Cloud resource C&M tools (e.g., Puppet, Chef, Juju, Docker, AWS OpsWorks), and research initiatives provide domain specific languages to represent and manage resources in a cloud environment [4, 5, 8, 17]. These languages are either template-based or model-driven [9]. Template-based approaches (e.g., Open Virtualization Format) aggregate resources from a lower level of the cloud stack and expose the package, along with some configurability options, to a higher layer. Model-driven approaches (e.g., TOSCA [11]) define various models of the application at different levels of the cloud stack, and aim to automate the C&M of abstract pre-defined composite solutions on cloud infrastructure [10]. Our approach proposes *Domain-specific Model*s, a methodology to extract cloud resource management entities from such model-driven and template-based C&M languages. These *Domain-specific Model*s provide a vocabulary to build elementary and federated cloud resources, as an *abstract*-layer over these multiple and diverse languages.

To build federated cloud resource management solutions across heterogeneous C&M tools, we need a middleware that either: (a) defines a unified cloud

[6]https://jujucharms.com/

resource C&M language (e.g., TOSCA, MODAClouds [11, 1]), which is conformed by every tool; or (b) provide a pluggable architecture that accepts and interprets different resource C&M models, offered by any tool. The former method is unfeasible as it would require existing tools undergo major architectural changes or complex model transformations to conform to a new language provided by the middleware. We thus believe the latter approach provides a more pragmatic and adaptive solution that can be integrated amongst a set of already existing and prevalent tools.

TOSCA is an open standard for unified representation and orchestration of cloud resources [11]. Wettinger et al. propose a model transformation technique that generates TOSCA based resource descriptions from resource descriptions in Chef and Juju [16]. Wettinger et al. and our work both focus on addressing drawbacks of heterogeneity among different Configuration and Management (C&M) tools. But our main goal is to build up a knowledge ecosystem by extracting resource C&M models of tools and represent them in a linked data model (i.e., Domain-specific models) such that common or related concepts across different tools can be exploited. For example, in Docker, the entity named `Hosting-Machine` (see Fig. 4.1) represents a VM where `Containers` are deployed, albeit Docker run-time cannot itself provision VMs. JuJu on the other hand focuses on managing a set of VMs, and can thus provision VMs. If Domain-specific models for both of tools are linked together, we can automate end-to-end deployment of Docker Containers on VMs which are provisioned by Juju.

In the domain of multi-cloud application development, wrapping heterogeneous cloud resources has been researched [10] and implemented as language libraries (e.g., Apache jclouds[1]). However, the fact that providers furnish different offerings and change them frequently often complicates these approaches.

7 Conclusion and Future Work

In this paper, we have presented a *Domain-specific Model* with declarative language to describe and query reusable resource management entities of complex cloud environments. We further propose a pluggable architecture, which translates these entities into deployment and/or management scripts, as well as elasticity rules of existing C&M tools. To evaluate the usability and productivity of our approach, we implemented a proof-of-concept prototype. Our approach yields significantly promising results, a 26.7% reduction of modeling time compared to traditional C&M techniques. We deduce the improved productivity and usability of *Domain-specific Model* based cloud resource C&M. As future work, we plan to provide (1) a visual language over *Domain-specific Model*s for interactive exploration and comprehension of cloud resource configurations; (2) a cloud resource recommender system based on a C&M knowledge acquisition technique (by extending our previous work [15]); and (3) resource migration support across different *Domain-specific Model*s.

[1]http://jclouds.apache.org

Bibliography

[1] D. Ardagna and et al. Modaclouds: A model-driven approach for the design and execution of applications on multiple clouds. In *MISE, 2012 ICSE Workshop on*, pages 50–56, June 2012.

[2] A. Bahga and V. K. Madisetti. Rapid prototyping of multitier cloud-based services and systems. *Computer*, 46(11):76–83, 2013.

[3] M. C. Barukh and B. Benatallah. Servicebase: A programming knowledge-base for service oriented development. In *DASFAA*, pages 123–138. Springer, 2013.

[4] T. C. Chieu and at al. Solution-based deployment of complex application services on a cloud. In *SOLI, 2010 IEEE International Conference on*, pages 282–287. IEEE, 2010.

[5] T. Delaet, W. Joosen, and B. Vanbrabant. A survey of system configuration tools. In *Proceedings of the 24th International Conference on LISA*, pages 1–8. USENIX Association, 2010.

[6] Gartner says cloud computing will become the bulk of new it spend by 2016. `http://www.gartner.com/newsroom/id/2613015`. Accessed: 07/12/2014.

[7] Json schema. `http://json-schema.org/latest/json-schema-core.html`. Accessed: 6/12/2014.

[8] A. V. Konstantinou and et al. An architecture for virtual solution composition and deployment in infrastructure clouds. In *Proceedings of the 3rd International Workshop on VTDC*, pages 9–18. ACM, 2009.

[9] V. Misic and et al. Guest editors' introduction: Special issue on cloud computing. *IEEE Transactions on Parallel and Distributed Systems*, 24(6):1062–1065, 2013.

[10] F. Moscato and et al. An analysis of mosaic ontology for cloud resources annotation. In *FedCSIS, 2011*, pages 973–980. IEEE, 2011.

[11] OASIS. *Topology and Orchestration Specification for Cloud Applications (TOSCA), Version 1.0*, 2013.

[12] R. Ranjan and B. Benatallah. Programming cloud resource orchestration framework: Operations and research challenges. *CoRR*, abs/1204.2204, 2012.

[13] B. Veeravalli and M. Parashar. Guest editors' introduction: Special issue on cloud of clouds. *IEEE Transactions on Computers*, 63(1):1–2, 2014.

[14] D. Weerasiri, B. Benatallah, and M. C. Barukh. Process-driven configuration of federated cloud resources. In *Database Systems for Advanced Applications*, pages 334–350. Springer, 2015.

[15] D. Weerasiri, B. Benatallah, and J. Yang. Unified representation and reuse of federated cloud resources configuration knowledge. Technical Report UNSW-CSE-TR-201411, Department of CSE, University of New South Wales, 2014.

[16] J. Wettinger, U. Breitenbucher, and F. Leymann. Standards-based devops automation and integration using tosca. In *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*, pages 59–68, Dec 2014.

[17] M. S. Wilson. Constructing and managing appliances for cloud deployments from repositories of reusable components. In *Proceedings of the 2009 Conference on HotCloud'09.* USENIX Association, 2009.