Domain Model Optimized Deployment and Execution of Cloud Applications with TOSCA

Fabian Glaser

Institute of Computer Science, University of Göttingen, Germany
fglaser@cs.uni-goettingen.de
http://swe.informatik.uni-goettingen.de

Abstract. Cloud computing promises to provide computing power as a utility and the adaptability to application requirements is one of its key benefits. However, using cloud infrastructures still requires a lot of technical expertise, which becomes a burden especially for non-computer scientists. Therefore, using model-driven approaches seems promising and can help to lower this burden by raising the level of abstraction. To achieve the correct scale of the cloud resources, a mechanism is required to map the computational requirements of the users domain model to parameters of the cloud infrastructure. In this paper, we present a framework, which scales the required infrastructure according to the demands of the users domain model. The framework utilizes a metamodel based on the Topology and Orchestration Specification for Cloud Applications (TOSCA) for modelling the cloud applications. Additionally, we introduce a domain-specific language to define a mapping between domain model parameters and parameters of the cloud infrastructure to achieve an appropriate scale.

Keywords: model driven engineering, cloud orchestration, TOSCA

1 Introduction

Due to its elasticity and on-demand self-service characteristics, cloud computing [1] is a great solution for users with varying computational requirements. However, setting up, running and scaling applications and the required infrastructure in the cloud is a cumbersome and error-prone task. Therefore, methods and tools are needed that simplify the process and lower the entry-barrier especially for non computer-scientists. With help of *model driven engineering* (MDE), the level of abstraction is raised and *domain specific language*s (DSLs) help to simplify tasks by focusing on the vocabulary of a certain domain. With MDE also the problem of API-heterogeneity of different cloud providers, often called *cloud-provider lock-in*, can be tackled [2], and graphical tools for modelling cloud infrastructures can be provided [3], [4]. In combination with the templates and scripts used by cloud orchestration and configuration management tools, fully automated, model driven deployment of cloud applications becomes possible.

These methods can be used for example to provide preconfigured computational resources to simulations scientists on demand. However, the appropriate scale of the infrastructure largely depends on what the scientist wants to compute. For example, an algorithm might require a certain amount of RAM in the deployed *virtual machine*s (VMs), or the number of entities in a simulation might require a certain number of cores to be computed efficiently. These parameters are encoded in the *domain model* of the scientist, which comprise all digital artefacts, the scientist created to solve a certain research problem. We argue that the scale of the provided infrastructure should be able to adapt to the computational requirements of the domain model of the scientist automatically.
To tackle this problem, we defined a framework [5] to be able to scale the cloud infrastructure with the help of parametrized *deployment models* of the users application based on the *Topology and Orchestration Specification for Cloud Applications* (TOSCA) [6]. In this paper, we introduce a DSL for the mapping between the domain model of the user and the deployment model to define its correct scale.

The remainder of this paper is structured as follows. After providing the foundations of this work in Section 2, we provide a driving example in Section 3. We discuss our framework in Section 4 and introduce the DSL for the mapping in Section 5. We evaluate the approach with help of a case study on the driving example in Section 6. Related work is given in Section 7. Finally, we draw our conclusions and give an outlook on future work in Section 8.

2 Automated Cloud Application Deployment

To define cloud computing, we refer to the definition given by the *National Institute of Standards and Technology* (NIST) [1]: "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." Thereby, NIST defines three service models for cloud computing, which operate on different levels of abstraction. On the highest level of abstraction is *Software-as-a-Service* (SaaS), where fully fledged applications are delivered to the user e.g., via a web-browser. Below that, *Platform-as-a-Service* (PaaS) offers programming environments or platforms such as pre-configured databases as services. On the lowest level of abstraction the user is able to directly acquire computing resources (e.g., virtual machines, virtual network, and virtual storage) on demand via *Infrastructure-as-a-Service* (IaaS). To offer higher level services such as PaaS and SaaS on top of IaaS, cloud providers can rely on automation achieved with *cloud orchestration* and *configuration management* tools, which we will discuss in the following.

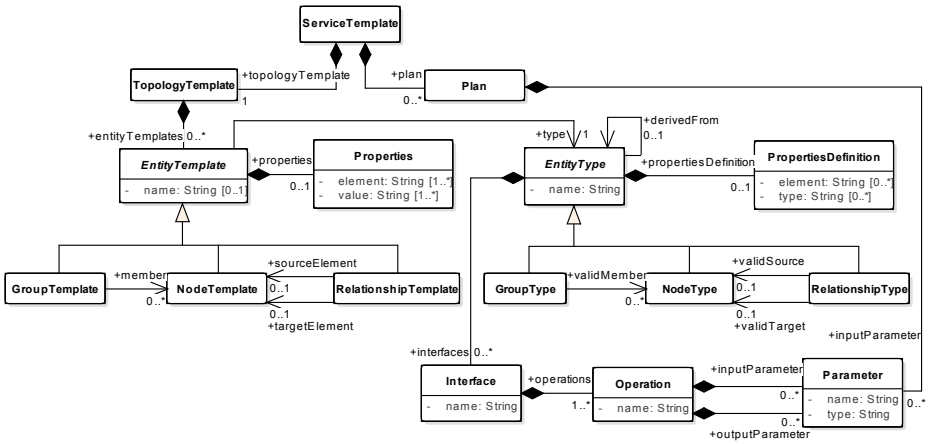

Fig. 1. TOSCA metamodel (adapted from Bergmayr et al. [4]).

2.1 Cloud Orchestration

To be able to manage and reuse configured resources in the cloud, *cloud orchestration* tools have emerged. Since many different cloud-provider dependent definitions of the term exist and it lacks of widely accepted definition, we will use the following definition in the scope of this paper:

> *Cloud Orchestration* refers to the automated launch and life-cycle management of resources e.g., VMs, virtual storage, or virtual networks in the cloud. It also assigns software configurations to the defined resources, without defining the installation process or the configuration of the software itself. Cloud orchestration tools often provide additional functionality for automatic (event-based) scaling of the deployed infrastructure.

Cloud Orchestration tools use template languages that allow to define the topology and also the life-cycle operations on the topology in a reusable manner. Examples include the language of Amazons CloudFormation [7] and the *Heat Orchestration Template* (HOT) language of OpenStacks Heat orchestrator. The *Organization for the Advancement of Structured Information Standards* (OASIS) aims to standardize such a template language with the *Topology and Orchestration Specification for Cloud Applications* (TOSCA) [6]. The first version of the standard based on the *Extensible Markup Language* (XML) was originally published in 2013, while a draft of a simplified rendering based on *YAML Ain't Markup Language* (YAML) [8] was first published in 2015 and is still under development.
A simplified metamodel of TOSCA is depicted in Figure 1. A *ServiceTemplate* captures the structure and the life-cycle operations of the application. It consists of a *TopologyTemplate* and a *Plan*. Plans define how the cloud application is managed and deployed. TopologyTemplates contain *EntityTemplates*, which are

either *NodeTemplates* that define e.g., the virtual machines or application components, *RelationshipTemplates* that encode the relationships between the NodeTemplates, e.g., that a certain application component is deployed on a certain virtual machine, or *GroupTemplates*[1] that allow to define groups of NodeTemplates, which e.g. should be scaled together. EntityTemplates have *Properties*, e.g., the IP address of a virtual machine, and a certain *type* that references an *EntityType*. The *EntityType* defines the allowed Properties through *PropertyDefintions*, and have *Interfaces*, which define the *Operations* that can be executed on the type, e.g., the termination of a certain application component, or the restart of a virtual machine. Operations have *Parameters* that define their input and output. In addition to parameters for operations, TOSCA also allows to define input parameters for Plans. These parameters can be used to parameterize the deployment workflow of the model and can e.g., include the virtual machine type to use or the number of instances of a certain type to launch.

2.2 Configuration Management

To enforce a certain software configuration on the resources defined above, *Configuration Management* tools are used. We use the following definition of the term in scope of this paper:

> *Configuration Management* refers to the automated and reusable enforcement of a certain software configuration on several machines. It comprises the configuration of the operating system, the installation and configuration of software and the configuration, launch, and termination of services.

Configuration Management tools became popular with the rise of the DevOps movement [9] in recent years. They use declarative domain specific languages to define the desired software configuration in a reusable manner. Examples are the language used in Puppets [10] *manifests* or the language that defines Ansibles [11] *playbooks*. In Ansible, playbooks are based on YAML and define *tasks* that should be executed on a group of hosts. The following listing shows the definition of the software configuration for a webserver with Ansible:

```
1 - hosts: webservers
2   vars:
3     http_port: 80
4  tasks:
5  - name: ensure apache is at the latest version
6     yum: name=httpd state=latest
7  - name: write the apache config file
8     template: src=/srv/httpd.j2 dest=/etc/httpd.conf
```

[1] GroupTemplates and GroupTypes are currently part of the TOSCA YAML specification, but not part of the TOSCA XML specification. We included them in the metamodel, because we need their functionality to for modeling scalability in our deployments.

The configuration is enforced on the hosts in the group `webservers` (Line 1), the http port is set with help of a variable (Line 3). In the first task (Line 5), an Apache webserver is installed with help of the package manager `yum`, and in the second task (Line 7) a template for the server configuration is copied to the hosts. Additionally, tasks can be encapsulated to form *roles*, and several roles can be assigned to a host or a group of hosts.

3 Driving Example

Our driving example originates from material science and uses the *Open Field Operation and Manipulation* (OpenFOAM) [12] software package. It exemplifies a case where the domain model of the scientist has an influence on the required scale of the cloud resources. OpenFOAM is a extensible C++ toolbox for solving systems of numerical equations, primary from the domain of computational fluid dynamics on a predefined environment. It has a large user base both from industry and academia and can be executed across large-scale *High Performance Computing* (HPC) clusters using the *Message Passing Interface* (MPI). Typically, using OpenFOAM involves three steps. In the *Pre-processing* step, the mathematical model, the description of the domain, on which the model should be solved, and a mesh, which describes the decomposition of the domain for computation is defined. In the *Solving* step, user-defined or predefined solvers are used to solve the mathematical model on the domain, and in the *Post-Processing* step additional tools can be used to visualize and analyse the created solutions. We will refer to the artefacts created by the scientist as the *domain model*, which in case of OpenFOAM consists of the following parts:

1. The *geometry* of the domain on which the mathematical model should be solved and how the mesh on this domain is created.
2. The *initial and boundary conditions* for the problem for each parameter.
3. The *physical properties* for the system of partial differential equations (PDEs) to be solved.
4. The *control* of the simulation, such as the simulation time and the reading and writing of the solution.
5. A *domain decomposition* that describes how the domain should be decomposed for parallel computation.

Regarding a suitable scale of the infrastructure, information on how many worker nodes can be utilized are encoded in the domain decomposition. Information on how much storage is needed is influenced by the total length of the simulation and the frequency with witch simulation data is written do disk.

We use OpenFOAM to exemplify the usage of our framework, which uses a combination of MDE, cloud orchestration and configuration management to automatically provision and scale the cloud infrastructure.

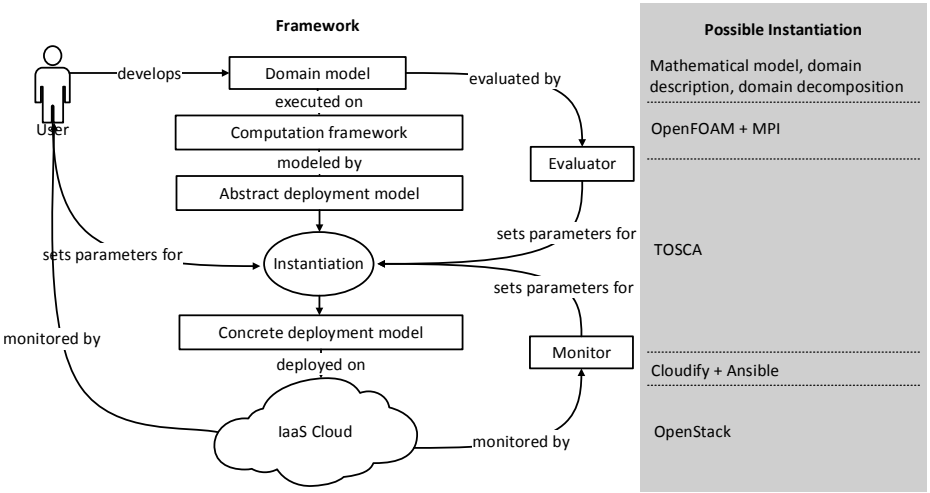

Fig. 2. A framework for adapting application deployments according to domain model demands.

4 The Framework

An overview of our framework and a possible instantiation for our driving example is depicted in Figure 2. We will discuss its components in the following. In this paper we focus on the role of the evaluator and the static evaluation of the domain model.

4.1 Domain Model

Domain models come in very different formats, they even might consist of code that is later on compiled and linked to external libraries. In most cases, no formal metamodel for the domain is available. In case of OpenFOAM, the parts of the domain model described in Section 3 are encoded in text files that have a OpenFOAM specific format.

4.2 Computation Framework

The *computation framework* (CF) represents the required software for executing the domain model and its dependencies. It is the most restrictive component for the cloud deployment, since it encodes how the computational load is distributed on the underlying infrastructure and defines the needed soft- and hardware configuration. In the driving example, OpenFOAM and its dependencies represent the CF. OpenFOAM uses MPI for distributed computation, hence it requires a MPI cluster to run and distribute the computational load.

4.3 Deployment Models with TOSCA

With the given template/type of mechanism TOSCA and its ability to define input parameters, we can distinguish between two types of deployment models: We call a model with unset parameters *abstract* and a model with instantiated parameters *concrete*. The transformation from an abstract model into a concrete model is called *instantiation*. The appropriate setting of the parameters for the deployment during the instantiation process, is done with the help of three sources: the user, static information from the domain model, and runtime information from the CF.

The *deployment model* for the CF comprises three elements: a cloud orchestration template, configuration management scripts and a description on how the domain model parameters are mapped to parameters of the infrastructure used by the *evaluator*. We introduce the language for the mapping in Section 5.

The abstract deployment model for the OpenFOAM cluster is shown in Figure 3. Since there is no standardized graphical syntax for TOSCA available, we use the following notation: NodeTemplates are depicted by boxes with solid lines, RelationshipTemplates are visualized by connections between the boxes, and GroupTemplates are depicted with boxes with dashed lines. For the NodeTemplates and the Groups we additionally list the type and a subset of the Properties. One virtual machine serves as a gateway node. This node gets a public IP address (*floating IP*) assigned and is reachable from the outside of the cloud. The gateway node is connected to an extra volume which provides the storage for the simulation data. An arbitrary number of virtual machines is deployed to serve as worker nodes in the cluster to do the calculations. The gateway node exports its volume via a *Network File System* (NFS), which is then mounted and shared by the worker nodes. The software configuration for the gateway and mpiworker nodes is modeled with help of a NodeTemplate of type ansible.nodes.Application. With help of these NodeTemplates the corresponding Ansible roles for the software configuration are associated to the host in which the NodeTemplate is contained. Since the software configuration for the worker nodes is dependent on the software configuration of the gateway, we use an additional *depence_on* relationship between the Ansible nodes. In the abstract deployment model, several parameters can be adjusted to provide an appropriate scale for the required computational power. Hence, the following parameters are kept as input parameters of the model:

P1: The size S of the NFS.

P2: The virtual machine type T of the gateway and worker nodes. The virtual machine *type* or virtual machine *flavor* is the common way of IaaS providers to encode the hardware configuration of a virtual machine. This includes RAM size, number of compute cores, and disk space.

P3: The number of worker nodes N. MPI can be used to distribute computation across a single machine with multiple cores, or across multiple machines.

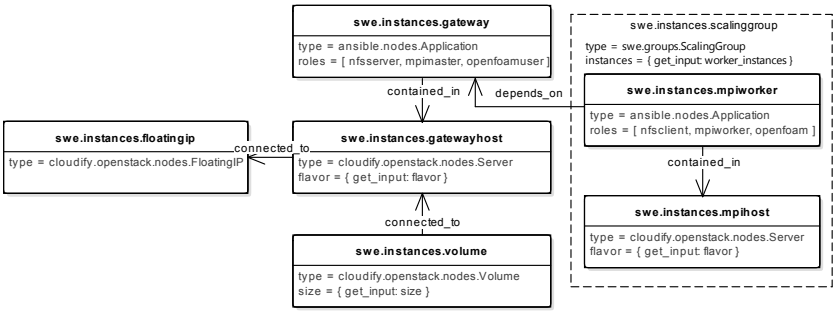

Fig. 3. Abstract Deployment Model for OpenFOAM.

4.4 Evaluator and Monitor

To find suitable parameter settings that match the requirements of a domain model, we distinguish between *static evaluation*, whereby the CF is not executed in the cloud, and *dynamic evaluation*, whereby the CF is executed and monitored. The static evaluation is performed before the CF is deployed on the cloud infrastructure and to derive its initial appropriate scale. We implement the static evaluation with help of the *evaluator*. This evaluator maps values of parameters of the domain model to suitable parameter settings for the concrete deployment model. This mapping is domain specific and needs to be defined for each domain separately. For this purpose, we defined a small DSL, which will be presented in Section 5.

Dynamic evaluations are done by monitoring the execution of the CF with help of a *monitor*. According to the outcome of the deployment evaluation, the parameters that have been used for the initial deployment are readjusted and a new instantiation of the abstract deployment model is initiated. Hereby, either a new concrete deployment model is created and deployed, or the existing concrete deployment model and its instantiation is adjusted. Dynamic evaluation and adjustments of a deployed infrastructure is nowadays implemented by many cloud orchestrators with their ability to process scaling policies that define under which conditions certain actions are automatically triggered on the infrastructure. For example, if the number of accesses on a webserver exceeds a certain threshold (condition), deploy an additional webserver (action). While it is worthwhile to investigate, if parts of these scaling policies can be derived from the domain model and the abstract deployment model of the CF, we focus on the determination of the appropriate scale for the infrastructure before the CF is deployed in scope of this paper.

4.5 Automated Deployment

The deployment of the CF is fully automated to avoid manual interaction with the cloud and enable transparent deployment of the CF for the user. A cloud orchestration framework is used for the orchestrated launch of the infrastructure

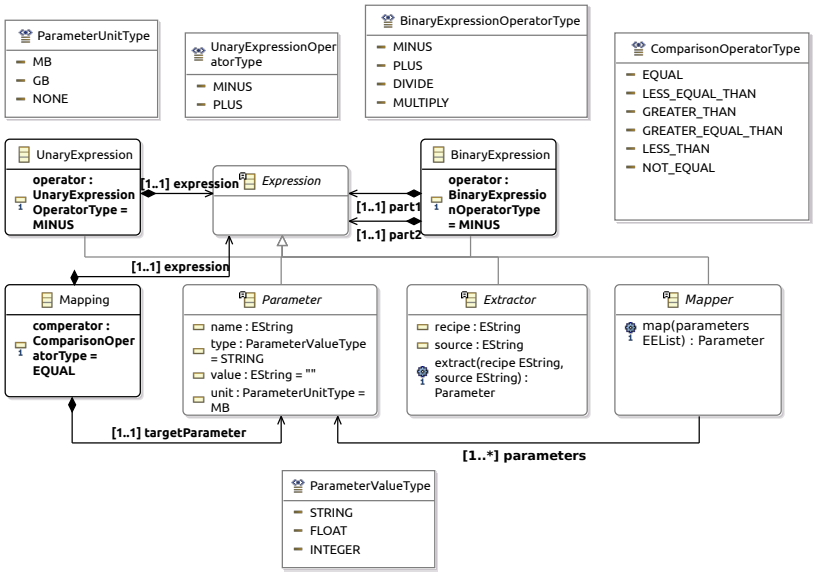

Fig. 4. Metamodel of the mapping language.

and a configuration management tool is utilized to automatically configure the launched infrastructure. As depicted on the right hand side in Figure 2, we use the cloud orchestrator Cloudify and the configuration management tool Ansible to automatically deploy the CF on a private OpenStack [13] cloud.

5 Mapping Domain Model Parameters to Infrastructure Parameters

Since the domain models come in very heterogeneous formats and in most cases lack of a formal metamodel, it is not possible to define a formal model transformation from the domain model to a model that is executable on the infrastructure. Instead, our approach is to provide a mapping mechanism, that is able to describe how parameters of the deployment model can be computed from extracted parameters of the domain model.
To be able to define the mapping for the evaluator, we developed a DSL which we will discuss in the following. The metamodel for the mapping language is depicted in Figure 4. A *Mapping* consists of a *TargetParameter* and an *Expression*. Hereby, the *TargetParameter* represents a parameter of the abstract deployment model and the *Expression* describes how the value for this parameter can be extracted from the domain model. The *Comperator* describes the relationship between the *TargetParameter* and the *Expression*. It can be e.g., of type *EQUAL*, to define that the *TargetParameter* must match the outcome of the *Expression*. *Expressions* can be unary, binary, simple parameters, of type

Extractor, or *Mapper*. *Extractors* encode how parameters are extracted from *Sources* of the domain model, which can either be files or folders. *Extractors* contain a *recipe*, which define additional information on how a parameter is extracted from a *source*. *Mappers* define how the extracted values are mapped to target parameters. They implement logic, where parameters of the domain model require a setting of the deployment model, which can not be derived automatically from the extracted value, e.g., a certain number of cores might be required for computation, but the deployment model does only allow to set the flavor of the VMs, and not the number of cores directly.

For the time being, we defined basic Extractors, that extract information from the structure of the domain model e.g., the number of files the model consists of, and their size. We also defined Extractors that extract file content, e.g., the number of lines in a file, and the ability to extract information with help of a regular expression (*FileContentExtractor*). Since our target IaaS system OpenStack does not allow to determine a fitting VM flavor for a given number of cores automatically, we defined a Mapper that maps compute cores to the VM flavor (*FlavorMapper*) and the other way around (*CoreMapper*). Additional domain-specific Extractors and Mappers can be defined and implemented that inherit from the corresponding base classes. We exemplify the usage of the language in Section 6.3.

6 Evaluation

We prototypically implemented the evaluator based on the language introduced above and the instantiation process to evaluate the framework. We now investigate if we are able to derive an appropriate scale of the deployed infrastructure with the introduced framework. To evaluate the mapping mechanism on our driving example, we require different domain models for OpenFOAM. The tutorial data for OpenFOAM 2.4.0[2] comprise around 200 domain models. For the evaluation of our framework and the mapping we picked six domain models with different computational requirements. The selected domain models are given in Table 1. Even if these tutorial domain models are small in comparison with realistic OpenFOAM simulations that require large-scale HPC clusters to be computed, they are suitable to test our framework.

6.1 Implementation

We prototypically implemented our framework with help of the *Eclipse Modeling Framework* (EMF) [14]. We used the *XML Schema Definition* (XSD) of TOSCA to generate an Ecore-metamodel. This metamodel served as a basis for code generation for the implementation of the evaluator and the instantiation process.

[2] Available online at `https://github.com/OpenFOAM/OpenFOAM-2.4.x/tree/master/tutorials`.

Table 1. Selected cases from OpenFOAM tutorial data.

	Domain Model	**Required Cores**
1	multiphase/twoPhaseEulerFoam/laminar/mixerVessel2D/	1
2	heatTransfer/buoyantBoussinesqSimpleFoam/iglooWithFridges/	2
3	multiphase/multiphaseInterFoam/laminar/damBreak4phaseFine/	4
4	combustion/fireFoam/les/oppositeBurningPanels/	6
5	multiphase/interDyMFoam/ras/DTCHull/	8
6	multiphase/interDyMFoam/ras/testTubeMixer/	16

EMF was also used for the definition and implementation of the mapping language. The utilized cloud orchestrator Cloudify currently supports only a subset of the functionality of TOSCA and is additionally not completely compliant with the standard. We used the Eclipse *Epsilon Generation Language* (EGL) [15] to generate Cloudify compliant YAML templates from our TOSCA metamodel. TOSCA is still subject to change and the development of the TOSCA YAML version is a little ahead of the TOSCA XML version. Since the TOSCA YAML version introduces some new features that are not yet reflected in the TOSCA XML schema, we added the desired features to our generated Ecore-metamodel manually. Hereby, we added *GroupTypes* and *GroupTemplates*, that allow to group NodeTemplates and the ability to set concrete values for Parameters.

6.2 Metrics

We aim to produce deployments that are *efficient*. We call a deployment *efficient* if it neither utilizes more nor less resources than actually needed. To measure the efficiency of the deployment, we use the following metrics to detect if too many or to few resources were provisioned:

M1: Average load on the provisioned cluster during the execution of the domain model. This number should be close to the number of provisioned cores in the cluster, indicating that all cores are utilized for computation.

M2: Utilized portion of the NFS [%]. To detect over-provisioning of the storage size, we measure how much of the storage has been actually used to store the resulting data of the simulation.

6.3 Selection and Mapping of Domain Model Parameters

A suitable size for the NFS S depends on the expected size of the simulation outcome. This in turn depends on the total simulation time T_{total} and on the frequency f_{write} with which partial results are written do disk. Both are parameters of the domain model. Given an estimate for the size S_{part} of the partial results, the size for the distributed file system can be calculated as

$$S = \frac{T_{total}}{f_{write}} \times S_{part}. \tag{1}$$

To pick a suitable virtual machine type T for the gateway and worker nodes from the types $TYPES$ offered by the IaaS provider, we need the number of cores N_{core}, we can utilize. The number of cores we can utilize, depends on the number of subdomains N_{sub} of the domain decomposition of the domain model as described in Section 3. Hence, we pick the virtual machine type as

$$T = \min_{t \in TYPES,\ t.cores \leq N_{sub}} |t.cores - N_{sub}|. \tag{2}$$

The suitable number of worker nodes N, we can use for distributed computation depends on the number of subdomains N_{sub} in the domain model, but also on the virtual machine type T we picked in the last step. We can than calculate a suitable number of worker nodes as

$$N = \lceil \frac{N_{sub}}{T.cores} \rceil - 1. \tag{3}$$

We subtract one, since the gateway node is also used for computation.

The following listing shows how the setting of parameter S is defined with help of the XML serialization of the mapping language:

```
 1 <mapping>
 2    <targetParameter xsi:type=‘‘mapping:TargetParameter’’
 3     name=‘‘size’’ type=‘‘INTEGER’’ unit=‘‘GB’’/>
 4    <expression xsi:type=‘‘mapping:BinaryExpression’’
 5     operator=‘‘MULTIPLY’’>
 6     <part1 xsi:type=‘‘mapping:BinaryExpression’’
 7       operator=‘‘DIVIDE’’>
 8       <part1 xsi:type=‘‘mapping:FileContentExtractor’’
 9         recipe=‘‘endTime((\s+)(\d+(.\d+)?))#3’’
10         source=‘‘system/controlDict’’/>
11       <part2 xsi:type=‘‘mapping:FileContentExtractor’’
12         recipe=‘‘writeInterval((\s+)(\d+(.\d+)?))#3’’
13         source=‘‘system/controlDict’’/>
14     </part1>
15     <part2 xsi:type=‘‘mapping:FileContentExtractor’’
16       recipe=‘‘partSize((\s+)(\d+(.\d+)?))#3’’
17       source="system/partSizeDict"/>
18   </expression>
19 </mapping>
```

The TargetParameter size is set with help of a BinaryExpression that implements the multiplication of Equation 1 (Line 4–18). It itself contains a second BinaryExpression (Line 6–14) that implements the division of the equation. The simulation time T_{Total} is extracted from the domain model with help of a *FileContentExtractor* defined in the lines 8–10, and the write interval f_{write} is extracted from the domain model with help of a FileContentExtractor defined in the lines 11–13. The expected size of the partial results can not be automatically

Table 2. Results for the evaluated and deployed OpenFOAM cases.

		Deployed Cluster			Metrics	
	Domain Model	**#VMs**	**#Cores**	**NFS Size**	**M1**	**M2**
1	multiphase/twoPhaseEulerFoam/laminar/mixerVessel2D/	1	1	1 GB	0.94	32%
2	heatTransfer/buoyantBoussinesqSimpleFoam/iglooWithFridges/	1	2	1 GB	1.57	55%
3	multiphase/multiphaseInterFoam/laminar/damBreak4phaseFine/	1	4	3 GB	3.79	92%
4	combustion/fireFoam/les/oppositeBurningPanels/	2	8	9 GB	5.91	84%
5	multiphase/interDyMFoam/ras/DTCHull/	2	8	3 GB	7.88	100%
6	multiphase/interDyMFoam/ras/testTubeMixer/	4	16	1 GB	15.92	26%

derived from the domain model. It is a good example for a parameter that needs to be provided by the user or with help of runtime information from executing the CF. Since we are not able to utilize runtime information yet, we provided the expected size of the partial results as part of the domain model. It is read with a FileContentExtractor from a file defined in the lines 15–17. Together with the domain model itself, the parameters mapping is passed to the evaluator, which evaluates the domain model and returns a list of initialized parameters for the deployment model. These parameters are then used in the instantiation process.

6.4 Results and Discussion

We executed the mapping on the OpenFOAM cases which are provided by Table 1, and deployed the CF and the corresponding infrastructure automatically in a small IaaS cloud based on OpenStack [13]. Then we executed the domain model and collected the metrics defined in Section 6.2 with help of the cluster monitoring tool Ganglia [16].

The results are summarized in Table 2. The number of deployed virtual machines (#VMs), the total number of provisioned cores (#Cores) and the size of the provisioned NFS is given. The total cluster size is automatically adjusted to each domain model. The average load on the cluster (**M1**) indicates that except for domain model 4 all provisioned cores were used for computation. In Case 4 only 6 of the 8 provisioned compute cores were utilized. Since the abstract model of our OpenFOAM cluster only allows the same virtual machine type for all nodes in the cluster, and no virtual machine type with 6 compute cores is available, 2 cores were over-provisioned. The framework was also able to adjust the size of the NFS. Since the size of the provisioned NFS is rounded to full GB and the required size for the partial results is only a rough estimate, in most cases not 100% of the NFS was utilized (**M2**).

While the presented evaluation only considers fairly small tutorial domain models, it shows that we are able to automatically adjust the scale of the provisioned resources to the computational requirements of the domain model. In case of the size for the NFS, additional user input was required to provide an estimate for the size of the partial results. However, this information could also be automatically derived during runtime. We are going to extend our work to use runtime information of the CF in the future.

7 Related Work

Besides TOSCA, other cloud-related standardization attempts exist. In the MDE community, the *Open Cloud Computing Interface* (OCCI) [17] received the most attention. Merle et al. [18] defined a metamodel for OCCI with help of EMF to provide a common basis for the generation and conformance testing of OCCI tools. This metamodel is used by Paraiso et al. [19] to model the deployment of applications with help of containers. Several works extend the *Unified Modeling Language* (UML) to be able to capture cloud-specifics [20], [21], [22]. Bergmayr et al. [4] show how to convert refined UML models to TOSCA templates. Their approach is also based on an Ecore metamodel generated from the TOSCA XSD. With the Cloud Application Management Framework (CAMF) [3], Loulloudes et al. attempt to build a whole IDE to manage cloud applications with the help of TOSCA.

Other approaches developed completely new cloud-specific modelling languages. Brandtzaeg et al. introduce CloudML [23], Silva et al. define the Cloud-DSL[24], and Hamdaqa et al. present the StratusML [25]. All of theses languages are specifiably tailored for the modeling of cloud applications. Bunch et al. define Neptune [26], a domain specific language especially to deploy scientific applications in the cloud. While our approach in modeling the CF is similar to the works introduced above, the definition of the mapping between the domain of the user and the deployment model is new.

Similar to the concept we defined for the dynamic update of the deployment during runtime, Ferry et al. [27] define a Models@Runtime approach for the deployment of cloud applications. We will evaluate the work of Ferry et al. when we extend our implementation to be able to utilize runtime information of the CF.

8 Conclusion and Outlook

Cloud orchestration and configuration management enable fully automated deployment of applications in the cloud. In our work, we combine the two technologies with MDE and a mapping mechanism to bridge the gap between the domain model to be computed and the required cloud infrastructure to enable appropriate scaling according to the domain model demands. The introduced mechanism determines an appropriate scale of the infrastructure before it is deployed in the cloud. In this paper we presented an initial evaluation of the concept with help of a prototypical implementation and an example from the domain of simulation science. Our initial experiences show that it is possible to scale the infrastructure appropriately with information extracted from the domain model. However, some information on the runtime behaviour of the domain model can not be predicted by static evaluations. As future work, we will move towards the automated modification and adaptation of our deployment models during runtime. The evaluation, we presented in this paper only covers an initial case study. To fully show the validity of our approach, we will conduct more case studies with software stacks from different domains.

Acknowledgements

The work of Fabian Glaser is partially funded by the Joint Centre of Simulation Technology (`https://www.simzentrum.de/`) of the University of Göttingen and the Technical University of Clausthal (Project 11.4.1).

References

1. P. Mell and T. Grance, "The NIST Definition of Cloud Computing," National Institute of Standards and Technology (NIST), Gaithersburg, MD, Tech. Rep. 800-145, September 2011. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf
2. D. Ardagna, E. Di Nitto, P. Mohagheghi, S. Mosser, C. Ballagny, F. D'Andria, G. Casale, P. Matthews, C.-S. Nechifor, D. Petcu *et al.*, "Modaclouds: A model-driven approach for the design and execution of applications on multiple clouds," in *2012 ICSE Workshop on Modeling in Software Engineering (MISE)*. IEEE, 2012, pp. 50–56.
3. N. Loulloudes, C. Sofokleous, D. Trihinas, M. D. Dikaiakos, and G. Pallis, "Enabling Interoperable Cloud Application Management through an Open Source Ecosystem," *IEEE Internet Computing*, vol. 19, no. 3, pp. 54–59, May 2015.
4. A. Bergmayr, U. Breitenbücher, O. Kopp, M. Wimmer, G. Kappel, and F. Leymann, "From Architecture Modeling to Application Provisioning for the Cloud by Combining UML and TOSCA," in *6th International Conference on Cloud Computing and Services Science (CLOSER)*, 2016.
5. F. Glaser, "Towards Domain-Model Optimized Deployment and Execution of Scientific Applications in Cloud Environments," in *Doctoral Symposium at the 5th Conference on Cloud Computing and Services Sciences (DCCLOSER 2015), Lisbon, Portugal*, May 2015.
6. OASIS, "Topology and Orchestration Specification for Cloud Applications (TOSCA) 1.0," November 2013, [Available online; http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html fetched on 03/12/2015].
7. Amazon Web Services, AWS CloudFormation - Infrastructure as Code & AWS Resource Provisioning, https://aws.amazon.com/cloudformation last accessed 06/12/2016.
8. OASIS, "TOSCA Simple Profile in YAML Version 1.0," February 2016, [Available online; http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.html fetched on 06/12/2016].
9. M. Hüttermann, *DevOps for developers*. Apress, 2012.
10. Puppet, Puppet - The shortest path to better software, https://puppet.com last accessed 06/12/2016.
11. Red Hat, Ansible is Simple IT Automation, https://www.ansible.com/ last accessed 06/12/2016.
12. OpenCFD, OpenFOAM - The Open Source Computational Fluid Dynamics (CFD) Toolbox, http://www.openfoam.com/ last accessed 06/12/2016.
13. OpenStack Open Source Cloud Computing Software, https://www.openstack.org/ last accessed 06/12/2016.
14. The Eclipse Foundation, Eclipse Modeling Project, https://eclipse.org/modeling/emf/ last accessed 06/12/2016.

15. The Eclipse Foundation, Epsilon Generation Language – Code Generation Language, http://www.eclipse.org/epsilon/doc/egl/ last accessed 06/12/2016.
16. Ganglia Monitoring System, http://ganglia.info/ last accessed 06/12/2016.
17. R. Nyren, A. Edmonds, A. Papaspyrou, and T. Metsch, "Open Cloud Computing Interface - Core," April 2011, [Available online: http://ogf.org/documents/GDF.183.pdf].
18. P. Merle, O. Barais, J. Parpaillon, N. Plouzeau, and S. Tata, "A Precise Metamodel for Open Cloud Computing Interface," in *8th IEEE International Conference on Cloud Computing (CLOUD)*. IEEE, 2015, pp. 852–859.
19. F. Paraiso, S. Challita, Y. Al-Dhuraibi, and P. Merle, "Model-Driven Management of Docker Containers," in *9th IEEE International Conference on Cloud Computing (CLOUD)*, San Francisco, United States, Jun. 2016. [Online]. Available: https://hal.inria.fr/hal-01314827
20. A. Bergmayr, J. Troya, P. Neubauer, M. Wimmer, and G. Kappel, "UML-based Cloud Application Modeling with Libraries, Profiles, and Templates," in *3rd International Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE)*, 2014, pp. 56–65.
21. A. Kamali, S. Mohammadi, and A. A. Barforoush, "UCC: UML profile to cloud computing modeling: Using stereotypes and tag values," in *7th International Symposium on Telecommunications (IST)*. IEEE, 2014, pp. 689–694.
22. J. Guillén, J. Miranda, J. M. Murillo, and C. Canal, "A UML Profile for Modeling Multicloud Applications," in *Service-Oriented and Cloud Computing*. Springer, 2013, pp. 180–187.
23. E. Brandtzæg, S. Mosser, and P. Mohagheghi, "Towards CloudML, a model-based approach to provision resources in the clouds," in *8th European Conference on Modelling Foundations and Applications (ECMFA)*, 2012, pp. 18–27.
24. G. C. Silva, L. M. Rose, and R. Calinescu, "Cloud DSL: A Language for Supporting Cloud Portability by Describing Cloud Entities," *CloudMDE 2014*, p. 36, 2014.
25. M. Hamdaqa and L. Tahvildari, "Stratus ML: A Layered Cloud Modeling Framework," in *2015 IEEE International Conference on Cloud Engineering (IC2E)*, March 2015, pp. 96–105.
26. C. Bunch, B. Drawert, N. Chohan, C. Krintz, L. Petzold, and K. Shams, "Language and runtime support for automatic configuration and deployment of scientific computing software over cloud fabrics," *Journal of Grid Computing*, vol. 10, no. 1, pp. 23–46, 2012.
27. N. Ferry, G. Brataas, A. Rossini, F. Chauvel, and A. Solberg, "Towards Bridging the Gap Between Scalability and Elasticity," in *4th International Conference on Cloud Computing and Services Science (CLOSER)*, 2014, pp. 746–751.