Facilitating Migration of Cloud Infrastructure Services — A Model-Based Approach

Ta'id Holmes
Infrastructure Cloud, Deutsche Telekom Technik GmbH
Darmstadt, Germany
t.holmes@telekom.de

***Abstract*—In cloud computing, modeling can be used to specify service topologies. Following a model-driven approach provisioning can be automated resulting in a significant reduction of time and costs. Yet, a forward engineering approach is limited to initial setups. That is, changes introduced posterior to a generation are not addressed *per se* when a complete regeneration is to be avoided. For dealing with differential changes of infrastructure service models, a model-based round-trip engineering approach is proposed that combines the power of model-driven generation with runtime reflection. For this, a *de facto* runtime model is reverse-engineered and model differences are calculated and operated on for enforcing a *de jure* model. While showcasing how to evolve a forward engineering approach accordingly, a particular implication of this contribution is the possibility to facilitate migration of cloud infrastructure services.**

***Index Terms*—cloud service, IaaS, migration, model-based, model difference, OpenStack, provisioning, service topology**

I. Introduction

Model-driven engineering (MDE) enables stakeholders to directly participate in engineering processes by relating to constructive models that abstract from technologies. In the context of cloud computing, service topologies for cloud-based solutions are modeled by architects. For instance, a Web application may comprise a load balancer, web servers, application servers, and a database system. Using modeling techniques, the comprised services such as the infrastructure services are specified in terms of computing power, memory, storage, network, and security rules (cf. [1]).

The provisioning of cloud services in terms of infrastructure is realized by one or multiple infrastructure as a service (IaaS) providers. By applying a model-driven approach provisioning can be automated (cf. [2]). Not only does modeling permit to directly incorporate architects, also the resulting saving of time is significant. Yet, a forward engineering approach is limited to initial setups. That is, a change to a service topology that shall be conducted posterior to an initial provisioning generally would require a complete regeneration.

For example, a cloud architect could decide to add another server instance such as a virtual private network (VPN) gateway to the service topology of the solution. This would not affect already provisioned cloud services, but require the instantiation of a new virtual machine and the deployment of particular security rules associated with the VPN service. Given a forward engineering approach it is only intended to provision an entire service topology. In this case, however, it would be more desirable to only consider the changes to the original model and perform required actions. In other cases it might be more constructive to compare a model describing the target configuration against reality.

For overcoming limitations of forward-engineering and for considering existing infrastructure services a model-based round-trip engineering approach is proposed showcasing how a former approach can be evolved accordingly. It incorporates runtime reflection for applying differential changes to an existing service topology. Comparing two models, such as a target *de jure* model with a current *de facto* runtime model, the work can be applied for enforcing conformance. Moreover, it can facilitate alignment and migration of infrastructure services between cloud deployments.

This paper is structured as follows: The approach is depicted in Section II and Section III describes technical implementation details. Different types of application scenarios – in particular service alignment and migration – are explained in Section IV. Experiences from a case study are presented in Section V followed by a discussion in Section VI. Finally, Section VII compares to related work and Section VIII concludes.

II. From a Model-Driven Forward Engineering to a Model-Based Round-Trip Engineering Approach

For the provisioning of cloud infrastructure services, a model-driven approach can be followed as proposed in [3] using textual domain-specific languages (DSLs). For this, a metamodel comprises IaaS concepts and provisioning is realized through model transformations. As such an approach is limited to forward engineering as outlined in the introduction, the model-based round-trip engineering approach is presented in this section as a superior alternative that does not only incorporate models from design time but also runtime models through runtime reflection.

Figure 1 gives an overview of the round-trip engineering approach. Initially, an architect specifies a cloud infrastructure service topology in terms of a *de jure* model. It is compared with another model originating from the runtime. For generating such a *de facto* model, a reflection service interacts with the interfaces of the IaaS provider. From the two IaaS models a model comprising model differences (referred to in this paper as a *diff-model*) is calculated which is processed by the execution engine. Based on the differences and depending on the kind of difference, the affected model element, and its content a model

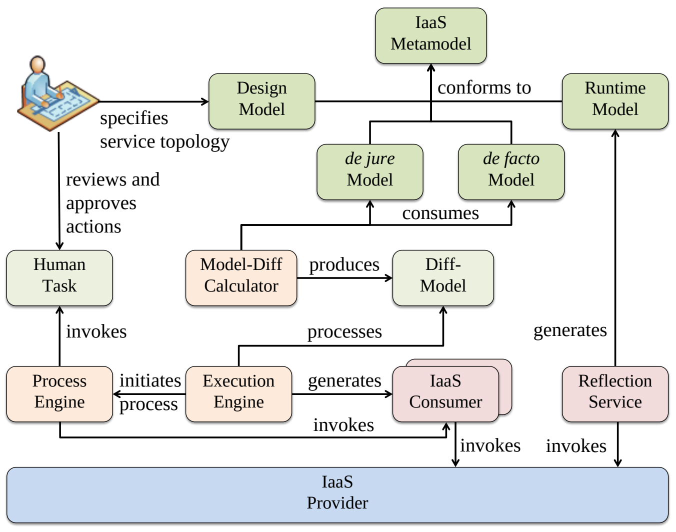

Figure 1. Overview of the Model-Based Round-Trip Engineering Approach

transformation takes place. Appropriate IaaS consumers are generated that – when executed – enforce the *de jure* model. An auxiliary generated business process helps to orchestrate these. In case differences are discovered that shall be subject to a review – e.g., when provisioned services are to be terminated – the business process also involves a human task.

III. Technical Realization

In this section, some technical details regarding the prototypical implementation are explained. Eclipse Xtext (Xtext) [1] was used to define a metamodel using a textual grammar. Diff-models are calculated using EMF Compare [2]. For model transformation, realizing the execution engine, Eclipse Xtend (Xtend) [3] was used. OpenStack [4] served as an IaaS solution.

A. IaaS Metamodel

Figure 2 depicts a metamodel comprising concepts from OpenStack Compute (Nova), a pendant to Amazon Elastic Compute Cloud (EC2) [5]. An `IaaS project` represents a tenant comprising cloud infrastructure services such as `server` instances, `volumes`, and `security groups`. Firewall rules (`FWRules`) translate to EC2 security rules and specify permitted `protocols` and/or open `ports`.

Models conforming to this metamodel were directly transformed to IaaS consumers in the forward engineering approach (cf. [3]). While they originated from the design time, the novel model-based round-trip engineering approach also considers models derived from the runtime. Thus, the metamodel both serves for expressing and for capturing IaaS concepts. For this, the metamodel was enriched with a few runtime concepts

[1]http://eclipse.org/Xtext

[2]http://wiki.eclipse.org/EMF_Compare

[3]http://eclipse.org/xtend

[4]http://openstack.org

[5]http://awsdocs.s3.amazonaws.com/EC2/latest/ec2-api.pdf

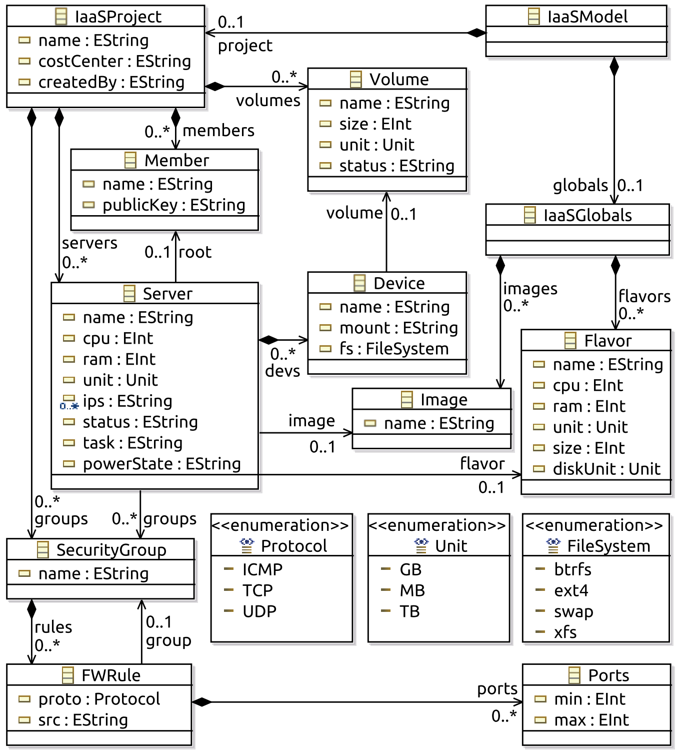

Figure 2. A Common Metamodel for Expressing and Capturing IaaS Concepts

such as assigned Internet Protocol (IP) addresses and different states. As a result, the `Server` and `Volume` classes have been extended with additional attributes (e.g., `ips`, `status`, and `powerState`). In the context of this work, the single purpose of the metamodel remains focused on driving the provisioning.

B. Reverse-Engineering a Runtime Model

The reflection service, as required for evolving the forward engineering approach, was realized as a representational state transfer (REST) service. Associated with a deployed IaaS solution, it consumes a tenant identifier and returns a *de facto* model reflecting the current provisioned cloud infrastructure services of the tenant. As mentioned, such a model conforms to the same metamodel as used for design time.

The service invokes the IaaS provider's application programming interfaces (APIs) for performing the runtime reflection. Programmatically, first, a token is retrieved after an initial authentication that is used in subsequent requests. Next, a list of security groups is requested. For each of the security groups, the security rules are requested then. Next, a list of server instances is retrieved. Iterating over all the instances, the various associated properties are looked up such as the image, the flavor, the assigned security groups, the security key, and associated IP addresses. Finally, volumes and associated instances are looked up before the token is invalidated. Meanwhile, the model is populated with the obtained information. Finally, the *de facto* model is returned to the caller.

Table I
MODEL TRANSFORMATION EXAMPLES: RELATING MODEL DIFFERENCES TO IAAS API CALLS

Model Element	**API**	**Kind**	**REST Operation** (simplified paths by omitting version)	**Description**
IaaSProject.members	Keystone	addition	POST users	creates a new user
			PUT tenants/{tenantId}/users/{userId}/roles/OS-KSADM/{roleId}	associates a user using a role
Member.publicKey	Keystone	addition	POST keypairs	sets/updates publicKey of a user
IaaSProject.groups	Nova	addition	POST {tenant_id}/os-security-groups	creates a new security group
IaaSProject.servers	Nova	addition	POST {tenant_id}/servers	creates a new server instance
		deletion	DELETE {tenant_id}/servers/{server_id}	terminates a server instance
IaaSProject.volumes	Nova	addition	POST {tenant_id}/os-volumes	creates a new volume
Server.devs	Nova	addition	POST {tenant_id}/servers/{server_id}/os-volume_attachments	attaches a volume to a server
Server.ips	Nova	addition	POST os-floating-ips	allocates a floating IP
			POST {tenant_id}/servers/{server_id}/action	assigns a floating IP
		deletion	DELETE os-floating-ips/{id}	deallocates a floating IP
			POST {tenant_id}/os-fixed-ips/{fixed_ip}/action	releases a floating IP

C. Calculating a Diff-Model

From two models (e.g., one model specified by an architect and one model generated by the reflection service as outlined in the previous section) a diff-model is calculated describing the discrepancy between a *de jure* and a *de facto* model. For calculating a diff-model a two-way comparison is performed using EMF Compare with the matching strategy being based on the object content ignoring identifiers. A resulting kind of difference categorizes as an addition, a deletion, a change, or a move. Former differences that merely depict extensions to the service topology can be resolved by provisioning the additional cloud services. Other differences may require review and approval prior to application. For this reason, different IaaS consumers are generated containing the respective service calls as described next.

D. Model Transformation

Next, the calculated diff-model is processed by the execution engine. Depending on the kind of difference and the concerned model element respective IaaS API calls are generated. Note that the IaaS metamodel is closely related to IaaS APIs (i.e., OpenStack Identity Service (Keystone) and Nova). For this reason transformation rules can easily be expressed programmatically by MDE developers. Similarly to the forward engineering approach, Xtend extension methods have been implemented in the round-trip engineering approach: Besides the object (e.g., `Server`) also the container (e.g., `IaaSProject`) and the name (e.g., `servers`) of the model element are taken as parameters into account as well as the kind of difference (e.g., `addition`). This way, it is similarly possible to generate respective actions although a diff-model is processed. Evolving a forward engineering approach, thus, requires respective adaptation of the execution engine.

Table I displays some examples of how various differences are related to IaaS API calls and considered by the model transformation. For example, in case the diff-model contains an `addition` of a server instance (`IaaSProject.servers`), a REST consumer is generated that when executed calls the `POST` operation `{tenant_id}/servers`. While the Uniform Resource Locator contains a path parameter for the tenant identifier other data has to be passed in the body using the JavaScript Object Notation such as a reference to the desired image or the to be associated security groups. If a difference does not comprise all required information for code generation, additional sources (e.g., a list of not yet considered differences) need to be consulted by transformation rules that are thus accessible within an instance of the execution engine.

Although the levels of abstraction between different elements of the metamodel and API operations are often alike, this is not always the case. Therefore some differences do not result in a direct mapping. That is, sometimes multiple actions have to be generated for a calculated difference. For instance, a user may have to be created prior to assigning her to a tenant. Also, for assigning a floating IP it may first be necessary to allocate a floating IP from a pool that in a second step can be assigned to a server instance. Such a technical detail is abstracted from in the metamodel and therefore has to be incorporated during code generation. Another cause that may complicate implementation of a model transformation is the design of the target language, i.e., the OpenStack APIs in this work. While most REST operations are similar between the kinds of difference, there are exceptions to a potential naming convention. For instance, not all operations that are related to deletions use the Hypertext Transfer Protocol option `DELETE`: For releasing a floating IP a `POST` request has to be issued and the keyword `unreserve` has to be passed in the body.

Processing order is crucial because of composition relationships. For example, if a difference consists of an existing server instance referencing a new volume group (another difference), the volume group has to be created first and can then be attached to the server instance. Contrary, if the volume group is to be deleted again it has to be detached from the server instance first and can then be terminated. Thus, the algorithm needs to apply differences respectively in order, e.g., by traversing the metamodel, by following composition relationships, and by applying matching differences only once.

Finally, differences that do not categorize as additions or deletions may be transcribed or may need to be handled accordingly. Sometimes there is no disparity in state if a difference such as a change or a move is realized using a deletion and addition in which case these operations of the API can be resorted. If this is not the case the respective

API needs to support these kinds of difference by providing additional operations to the minimum set.

The generated actions (i.e., API calls) are aggregated in two IaaS consumers: non-critical alignments, i.e., actions resulting from additions, are placed in the first consumer. Actions originating from other differences that require review are stored in a different consumer. In addition, a business process is generated for their orchestration and for incorporating a human review.

E. Model-Based Enforcement

Once the execution engine has processed the diff-model the auxiliary business process is initialized that invokes the consumer comprising the non-critical alignments. In case of critical alignments, e.g., differences that result in the termination of infrastructure services, a review is to be undertaken. For this the generated business process contains approval tasks relating to the critical differences. When approved the respective IaaS consumer is invoked by the business process.

IV. Application Scenarios

In this section the applicability of the approach is illustrated by discussing different types of scenarios. That is, for all of the scenarios the same approach is applied as presented in Section II using a *de jure* and a *de facto* model for determining respective provisioning actions.

A. Initial Provisioning of Infrastructure Services

Given a model describing a cloud infrastructure service topology, how can these services be provisioned?

The first type of usage scenario equals simply to what has been covered by the functionality of the forward engineering approach. That is, from a service topology, infrastructure services are to be provisioned in a greenfield scenario. As no cloud infrastructure services are existing in the respective tenant, the reflection service yields an empty *de facto* model. This is compared against the user supplied *de jure* model comprising the to be provisioned infrastructure services. As a consequence, the diff-model comprises additions only. Each addition is transformed by the execution engine into code that realizes the provisioning of the respective service. Thus, in contrast to the forward engineering approach the execution engine of the round-trip engineering approach processes a diff-model. In both approaches, a complete IaaS consumer for provisioning all of the services is generated using model transformation: By comparing a model against an empty model, the round-trip engineering approach yields the same actions as the forward engineering approach that processes the model. In fact, an IaaS consumer that results from transforming the diff-model instantiates all the infrastructure services as present in the model. As a consequence, the IaaS consumers generated by the different approaches are equal in terms of their behavior.

This way, functionality of the forward engineering approach is covered and as required by this scenario. In the following some further application scenarios are described proving the round-trip engineering approach to be a superior alternative to the formerly presented forward engineering approach.

B. Design Time Modification and Runtime Adaptation

Given previously provisioned cloud infrastructure services in a tenant, how can changes to an original model be propagated and be enforced for adopting the respective tenant?

The second type of usage scenario derives from the initial motivation described in the introduction. In this scenario an initial provisioning as previously described has taken place. The original model was modified then. Instead of rebuilding the entire service topology from scratch, changes are identified first and applied to the existing IaaS provider.

While in this scenario it would be possible to compare two models from design time, i.e., a revised version with an original describing the current topology in terms of infrastructure services the latter model is reverse-engineered using a reflection service. A reason for this is that the resulting model effectively constitutes a *de facto* model for the given point in time. If the former version of a model is to be used as the basis for comparison it would require that it reflects reality and that no changes must be undertaken apart from the model. Relying on the reflection service, this restriction is not necessary. Indeed, services can manually be provisioned without the risk to jeopardize the round-trip engineering approach.

Thus, in this scenario the *de jure* model is a revised version of a model and the *de facto* model is obtained using the reflection service. As in the previous scenario, the differences may only comprise additions. In this case they are applied directly. However it is also possible that differences are of other types. In such cases a review is scheduled.

C. Alignment of Cloud Infrastructure Services

Given two different cloud tenants, how can the cloud infrastructure services be aligned between them so that one tenant comprises the same infrastructure services as the other?

With the approach it is equally possible to compare two runtime models, e.g., for aligning provisioned cloud infrastructure services of two tenants in different clouds. For this the respective reflection services yield corresponding models. According to what model is elected as the *de jure* model the tenant of the other model is aligned subsequently. This scenario is similar to the previously described scenario in a sense that the diff-model may contain any kind of difference (i.e., not only additions as in the initial provisioning) and that the model transformation is alike. In fact, the only distinction is the selection of two runtime models. When comparing two tenants using the reflection service with the target being empty, all cloud infrastructure services from the source tenant will be provisioned at the target tenant. This is exploited for facilitating the migration of infrastructure services as described next.

D. Facilitating Migration of Infrastructure Services

Given two IaaS deployments, how can the infrastructure services from one cloud be migrated to the other?

By realizing the alignment of cloud infrastructure services as previously described, migration can be facilitated. First, the tenants are created at the new deployment. Next, they need to be migrated. For this, two runtime models are compared; one originating from the source cloud comprising the provisioned services to be migrated and one empty model from the target. Next, the execution engine aligns the target with the source for facilitating the migration of the cloud services. Finally, the original cloud services are terminated once the migration has been affirmed. While this does not realize yet complete migration of cloud services (e.g., the data stored in volumes which is not in scope of this paper), it does automate some basic steps and as a consequence facilitates migration of cloud infrastructure services. After the alignment of the infrastructure services has taken place, the business process schedules a task for manually completing and verifying the migration. When the migration is affirmed to have succeeded the IaaS consumer is invoked for terminating the original cloud services completing the migration.

V. Experiences from a Case Study

In a case study, a migration from OpenStack 2012.1 (Essex) to OpenStack 2013.2 (Havana) was attempted. For this, the setup and procedure as described in the previous paragraph was chosen. That is, two OpenStack deployments were available in parallel, with one constituting the target cloud. While the reflection service operated on the former version, the execution engine generated IaaS consumers for the later version. While realizing a proof of concept, the prototype was not deployed to undertake a factual migration of real IaaS tenants as further problems need to be addressed such as how to migrate (assigned) floating IP addresses. Also the migration of data, e.g., in volumes, was not yet tackled. Finally, the monitoring and reporting of the overall migration has to be worked out.

As some code in the reflection service and the execution engine is specific – not only to the IaaS solution but also IaaS version – dedicated plugins hold respective parts. In fact, when conducting the case study with these two OpenStack deployments an incompatibility of APIs was detected. While libraries and projects (e.g., DeltaCloud [6]) exist that try to abstract and unify various IaaS interfaces there is no established standard for IaaS provider services adequately implemented by IaaS solutions. EC2, widely adopted in industry, could be a candidate, but because of some limitations and convenience (e.g., names of server instances) the (OpenStack native) Nova API was chosen. During prototyping it was discovered, however, that between these two versions the API underwent some changes that required adaptation of the code. With the possibility to provide plugins for each specific IaaS (version) the prototype is extensible and adaptable. This way, further IaaS solutions can be supported for applying the approach.

VI. Discussion

In contrast to a naïve model-driven, and forward-only approach, the presented model-based round-trip engineering approach enables alignment of existing cloud infrastructure services. Incorporating runtime reflection for reverse-engineering a runtime model of infrastructure services the approach paves the way for facilitating their migration.

[6]http://deltacloud.apache.org

A. Comparing Forward and Round-Trip Approaches

In both approaches the same metamodel is used for expressing and for capturing concepts from IaaS provisioning (see Figure 2). In addition to an IaaS metamodel the round-trip engineering approach relies on a diff-metamodel. The diff-metamodel provides the means to describe changes between (any) two models that conform to the same metamodel. In particular it comprises addition and deletion of model elements as concepts. This diff-metamodel does not need to contain domain specific concepts but can be a generic metamodel. That is, for any metamodel it can describe changes between conforming models (cf. [4]). If a generic metamodel is used the execution engine needs to consider the domain specific type of an affected model element in addition to the (generic) kind of change for deriving appropriate actions.

Although the same IaaS APIs can be used in both approaches, the forward engineering approach does not perform runtime reflection as in case of the round-trip engineering approach. For this reason the models used in the forward engineering approach are limited to design time models whereas the round-trip engineering approach also incorporates reverse engineered runtime models. In contrast to the forward engineering approach, the execution engine of the round-trip engineering approach does not operate on a model conforming to the IaaS metamodel. Instead, a diff-model is processed. Both approaches can be used for an initial provisioning. In addition, the round-trip engineering approach is able to apply incremental changes posterior. Finally, the forward engineering approach operates on a single cloud whereas the round-trip engineering approach can be applied in a multi-cloud deployment, i.e., to compare, align, and migrate infrastructure services between two clouds.

B. Round-Trip Engineering Approach

The generated *de facto* model is not only useful for (partial) provisioning (e.g., in an iterative practice) but potentially also valuable to stakeholders for representation purposes (cf. model-based reporting [5]). Moreover a design time model can be validated against runtime by calculating mismatches. Such results can be fed into monitoring for alerting.

Note that certain adaptation scenarios can already be achieved by providing only some transformation rules. For example, if the diff-model contains only an addition of a new server, it suffices to call the transformation rule for generating the respective Nova API operation as shown in Table I. Thus, not all kinds of differences need to be supported for covering a wide range of adaptation scenarios. Yet, for the transformation to be complete and fully automated any model element of the metamodel must be mapped depending on the kind of difference to appropriate API calls.

As mentioned, the work serves as a first step towards the migration of cloud services. For completely migrating cloud infrastructure services further work has to be carried out. For example, there are remaining challenges such as how to migrate (assigned) floating IP addresses.

VII. Related Work

In this section this work is compared and related to other work, approaches, and efforts from academia and industry. First, the topic of model differences is discussed briefly. Next, some related work on runtime models and model-based adaptation is presented. Finally, the cloud computing context is looked at.

Cicchetti et al. [4] present a metamodel independent approach for describing model differences. In the realization of this work this is applied using the diff-metamodel from EMF Compare. Langer et al. [6] present a contribution that permits to identify composite operations of a revised model such as refactorings. Equally building on EMF technology and utilizing the EMF Compare project the article also gives a well explained and very detailed background on diff-models, their use and metamodel. For the migration of infrastructure services no composite operations need to be considered but when alignment between two cloud deployments is to be retained it could be beneficial to identify composite changes and apply them as such.

The idea of comparing a target model against reality as applied in this work is the base of many studies. Bencomo et al. [7] evaluate runtime systems against valid states restricted by requirements. Building on the idea of models@run.time (cf. [8]) Ferry et al. [2] propose a framework based on CloudML [7] for the provisioning, deployment, monitoring, and adaptation of multi-cloud systems. Although related in various aspects, it does not address migration of cloud (infrastructure) services which is the focus of this work. It presumes a causal connected system whereas this work showcases how to introduce runtime models, e.g., when moving from a forward to a round-trip engineering approach. Further distinctions are the reverse engineering of runtime models, the differentiation of adaptations, and the integration of human reviews. Last but not least, the approach presented for calculating and processing a diff-model is agnostic to the metamodel. In contrast CloudML manually implemented specific comparisons related to its metamodel and could thus profit from utilizing a library such as EMF Compare.

The OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) standard [8] is not limited to infrastructure services but covers the complete service stack and management of cloud applications (cf. [9]). Aiming at portability, current advancements are driven by design models and there is currently no work that permits to analyze a system and reverse-engineer a model. Although this work was realized using a EC2 metamodel, the TOSCA metamodel can be used equally for expressing and capturing infrastructure services.

CloudFormation [9] and the OpenStack Orchestration (Heat) [10] pendant permit to describe setups in a file based manner. While not at a modeling level these technologies can be used to provision services in different cloud regions. Also, reverse engineering may be possible, yet, the comparison and resolution for aligning infrastructure services is not addressed.

Besides, there are a couple of related metamodels and DSLs (e.g., CloudML see above) targeting cloud applications: Bergmayr et al. [10], e.g., present an UML-based approach for the modeling of cloud applications. It is believed that a particular metamodel such as the one presented in Figure 2 is not pivotal to the approach. It must comprise or permit to express IaaS concepts though. In case the metamodel is to be substituted (e.g., for adopting TOSCA), the reflection service and the execution engine have to be adapted. If a certain technology shall be used not only for the metamodel but also within model transformation this would require the expression of incremental changes using the technology and that such increments may be applied posteriori to an initial deployment.

VIII. Conclusion

The migration of cloud infrastructure services between IaaS deployments can be facilitated using a model-based round-trip engineering approach combining generation techniques with runtime reflection. By realizing this, this paper showcased how a model-driven forward engineering approach can be evolved to a round-trip engineering approach by enriching a metamodel with runtime aspects, by establishing a reflection service, and by basing the model transformation on processing a diff-model.

Acknowledgments

The author would like to thank peers cordially for valuable feedback, communicative exchange, and for kindly providing information of avail.

References

[1] E. Wittern, A. Lenk, S. Bartenbach, and T. Braeuer, "Feature-Based Configuration of Vendor-Independent Deployments on IaaS," in *18th IEEE International Enterprise Distributed Object Computing Conference*, M. Reichert, S. Rinderle-Ma, and G. Grossmann, Eds. IEEE, 2014, pp. 128–135.

[2] N. Ferry, H. Song, A. Rossini, F. Chauvel, and A. Solberg, "CloudMF: Applying MDE to Tame the Complexity of Managing Multi-cloud Applications," in *IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC)*. IEEE, Dec 2014, pp. 269–277.

[3] T. Holmes, "Automated Provisioning of Customized Cloud Service Stacks using Domain-Specific Languages," in *2nd International Workshop on Model-Driven Engineering on and for the Cloud*, vol. 1242. CEUR-WS.org, Sep. 2014, pp. 46–55.

[4] A. Cicchetti, D. D. Ruscio, and A. Pierantonio, "A metamodel independent approach to difference representation," *Journal of Object Technology*, vol. 6, no. 9, pp. 165–185, 2007.

[5] T. Holmes, "From Business Application Execution to Design through Model-Based Reporting," in *16th IEEE International Enterprise Distributed Object Computing Conference*, C.-H. Chi, D. Gasevic, and W.-J. van den Heuvel, Eds. IEEE, Sep. 2012, pp. 143–153.

[6] P. Langer, M. Wimmer, P. Brosch, M. Herrmannsdörfer, M. Seidl, K. Wieland, and G. Kappel, "A posteriori operation detection in evolving software models," *Journal of Systems and Software*, vol. 86, no. 2, pp. 551–566, 2013.

[7] N. Bencomo, J. Whittle, P. Sawyer, A. Finkelstein, and E. Letier, "Requirements reflection: requirements as runtime entities," in *ICSE (2)*, J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, Eds. ACM, 2010, pp. 199–202.

[8] G. S. Blair, N. Bencomo, and R. B. France, "Models@ run.time," *IEEE Computer*, vol. 42, no. 10, pp. 22–27, 2009.

[9] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, "TOSCA: Portable Automated Deployment and Management of Cloud Applications," in *Advanced Web Services*, A. Bouguettaya, Q. Z. Sheng, and F. Daniel, Eds. Springer, 2014, pp. 527–549.

[10] A. Bergmayr, J. Troya, P. Neubauer, M. Wimmer, and G. Kappel, "UML-based cloud application modeling with libraries, profiles, and templates," in *2nd International Workshop on Model-Driven Engineering on and for the Cloud*, vol. 1242. CEUR-WS.org, Sep. 2014, pp. 56–65.

[7]http://cloudml.org

[8]http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf

[9]http://aws.amazon.com/cloudformation

[10]http://docs.openstack.org/developer/heat/