Comparison and Runtime Adaptation of Cloud Application Topologies based on OCCI

Johannes Erbel, Fabian Korte and Jens Grabowski
University of Goettingen, Institute of Computer Science, Germany

Keywords: Open Cloud Computing Interface, OCCI, Runtime Adaptation, Models at Runtime, Model Comparison.

Abstract: To tackle the cloud provider lock-in, multiple standards have emerged to enable the uniform management of cloud resources across different providers. One of them is the Open Cloud Computing Interface (OCCI) which defines, in addition to a REST API, a metamodel that enables the modelling of cloud resources on different service layers. Even though the standard defines how to manage single cloud resources, no process exists that allows for the automated provisioning of full application topologies and their adaptation at runtime. Therefore, we propose a model-based approach to adapt running cloud application infrastructures, allowing a management on a high abstraction level. Hereby, we check the differences between the runtime and target state of the topology using a model comparison, matching their resources. Based on this match, we mark each resource indicating required management calls that are systematically executed by an adaptation engine. To show the feasibility of our approach, we evaluate the comparison, as well as the adaptation process on a set of example infrastructures.

1 INTRODUCTION

High demand for computing as a utility led to multiple companies providing cloud computing services. However, each provider defines its own framework and tools to access its services. This resulted in the *provider lock-in* problem, i.e., binding a customer to a provider once a cloud application has been build. To tackle this issue multiple standards have been developed to grant a uniform way to manage cloud resources. One of these standards is the *Open Cloud Computing Interface (OCCI)* (OGF, 2016a), developed by the *Open Grid Forum (OGF)*, which specifies a *Representational State Transfer (REST) Application Programming Interface (API)* to manage cloud resources. For this purpose, OCCI defines an extensible metamodel allowing to model cloud resources on the different service layers. Even though OCCI specifies how to manage single cloud resources, no framework exists that allows to adapt running cloud application topologies using OCCI models. To fill this gap, we propose an adaptation process which takes an OCCI model as input, analyzes required adaptation steps, and transforms the running cloud application topology into the desired state. Thus, we provide an engine allowing to adapt running topologies on a high abstraction level that does not require deep knowledge about different cloud provider infrastructures. To achieve this goal, we identified three fundamental steps that need to be executed: (1) The extraction of a runtime model from the current cloud application topology, (2) its comparison to the desired state, and (3) its actual adaptation via systematic management calls. In this paper, we present a process that implements these three fundamental steps and evaluate its feasibility on different cloud application topologies.

The remainder of this paper is structured as follows. Section 2 gives a brief overview of the OCCI standard. After that, Section 3 discusses the problems that need to be addressed by the adaptation process. We define the adaptation process in Section 4 and demonstrate its feasibility in Section 5. Finally, we discuss related work in Section 6, followed by a conclusion and an outlook on future work in Section 7.

2 OPEN CLOUD COMPUTING INTERFACE

OCCI is a cloud standard, developed by the OGF, that specifies a REST API (Fielding, 2000) and a metamodel for a universal management of cloud resources (OGF, 2016a). The OCCI Core model serves

Erbel, J., Korte, F. and Grabowski, J.
Comparison and Runtime Adaptation of Cloud Application Topologies based on OCCI.
DOI: 10.5220/0006774405170525
In *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018)*, pages 517-525
ISBN: 978-989-758-295-0
Copyright © 2019 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

as an extensible metamodel that allows to model all kinds of cloud resources. Hereby, each element in an OCCI model represents a single cloud. Figure 1 provides an overview of the OCCI Core model (OGF, 2016a) and the OCCI infrastructure extension (OGF, 2016c).

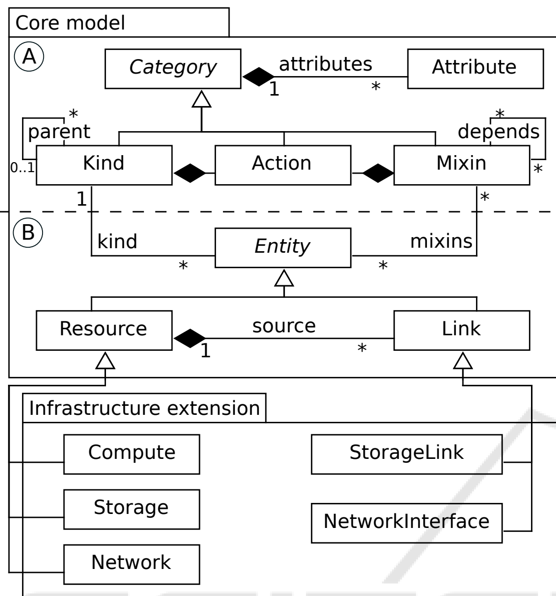


Figure 1: OCCI Core model (adapted from (OGF, 2016a)).

The Core model is composed of two parts: the *classification and identification mechanisms* (**A**), and the *core base types* (**B**). Core base types define `Resources` and `Links`, which both inherit from the abstract `Entity` element. These describe cloud resources and their connections, e.g., a Virtual Machine (VM) and its connection to a network. Each `Entity` is bound to one `Kind` and can be linked to multiple `Mixins`. `Kind` and `Mixin`, inherit from the abstract `Category` element and are part of the classification and identification mechanisms. A `Kind` defines the specific type of an `Entity`, whereas `Mixins` add additional capabilities to it at runtime for which dependencies can be modeled. Each `Category` instance can define any amount of `Attributes` representing client readable attributes to be filled by a corresponding `Entity` instance, e.g., a `Kind` defines an `Attribute` describing the size of a storage which is filled with a value in the `Resource` instantiation. Furthermore, `Category` instances may define `Actions` that can be performed on related `Entities`. Typical examples are `Actions` to start or stop VMs. Each `Kind` can inherit from another `Kind`, whereby it has to be related to one of the three core `Kinds`. These core `Kinds` ensure basic functionalities and are equally named to the core base types `Entity`, `Resource`, and `Link`.

To enhance the Core model's capabilities, OCCI specifies multiple extensions, e.g., the OCCI infrastructure extension (OGF, 2016c) or the OCCI platform extension (OGF, 2016d). These extensions add `Entity` types to the metamodel specializing either the `Resource` or the `Link` element. As an example, Figure 1 includes the infrastructure extension. `Compute` represents an abstraction of a generic processing resource, for example, a VM or container (OGF, 2016c), `Network` describes networking entities, and `Storage` data storage devices. Connections between VMs and networks or storages are modelled via `NetworkInterface` links or `StorageLinks`.

For the scope of this paper, we refer to the metamodel elements of OCCI (depicted as classes in Figure 1) as if they were instances and stick to the introduced font-style to highlight the model representation of the cloud resources. Hereby, we address `Entity` to consider `Resources` as well as `Links`.

3 PROBLEM STATEMENT

Setting up a cloud application is still a complex task, especially due to the heterogeneity of cloud services offered by different cloud providers. To assist during this process, a universal way to manage and adapt cloud application topologies on a high abstraction level would be beneficial. To reach this goal, we identify the following required processes and formulate corresponding questions that need to be addressed.

The derivation of adaptive steps requires the system's runtime information. Therefore, an extraction process is needed to access the current state of the topology. This information can be encoded in an OCCI model which can be subsequently compared to the OCCI model of the desired state. Furthermore, a comparison between the current and target state of the topology is required. To calculate the differences, a procedure is needed that matches the resources from the runtime model to the target model. Executing the identified management calls requires a systematic process resolving the dependencies of the requests. Summed up, we define the following questions:

- **Q1:** How to extract information from the running system into an OCCI model?
- **Q2:** How to recognize whether two elements from different cloud application topologies match?
- **Q3:** What is a suitable order for management calls to adapt a running cloud application topology?

In the following section, our approach to address these questions is introduced.

4 APPROACH

To adapt a running cloud application topology, we propose a models at runtime approach as depicted in Figure 2. The runtime model represents the extracted cloud application topology, whereas the target model encodes the state to be reached. Based on the results of the comparison, we indicate what kind of adaptation steps are required followed by an execution of corresponding REST requests. In the following, we describe briefly these different steps, whereas in detail information can be found in (Erbel, 2017a).

Figure 2: Overview depicting the adaptation process.

4.1 Extraction

An extraction process is required to build a model representation of the current state of the running cloud application. In general, GET requests are send to the OCCI API to gather all provisioned `Category` and `Entity` instances currently running. Based on that, the elements are divided into the different `Entity` and `Category` sub-types: `Resource`, `Link`, `Mixin`, `Kind`, and `Action`. This allows us to rebuild and serialize the cloud application topology into a model conforming to the OCCI metamodel.

4.2 Comparison

During the comparison of two OCCI models, we check whether two `Entities` from the source and target model match. As `Resources` aggregate `Links`, we define a common structure for the comparison process which mainly relies on a matching of `Resources`. The structure is depicted and exemplified in Figure 3. After the calculation of a *resource match*, the `Links` from the matched `Resources` are compared to investigate whether their connections have changed. Hereby, we match `Links` of the same `Kind`, their source, and their target. In a final step, we mark each `Entity` assigning the required adaptation step. Hereby, we categorize them from the target model's point of view, marking them as *old* , *new*, *missing*, or *updated element*. Non matched `Entities` are marked as new or missing, depending on whether they are from the source or target model. Matched `Entities` are compared based on their `Attributes` to categorize them as updated, when they differ, or old element, if they are equivalent. These marked elements then serve as input for the adaptation steps. In the following, we examine multiple techniques to calculate a suitable resource match.

Figure 3: Comparison procedure for OCCI models.

4.2.1 Resource Matching

Model elements can be matched considering two types of information: their attributes and their structural position. Thus, we investigate matching strategies based on these distinct levels and a combination of both. For the attribute match, we only consider unchangeable ones, as changeable attributes do not grant enough information for a suitable match based on attribute similarities, e.g., multiple VMs with the same (virtual) hardware configuration may exist at one point in time. In case of OCCI, unchangeable attributes are the `Kind` and id of each element. Such an attribute matching can be seen as a *static identity-based* matching (Kolovos et al., 2009), as only discrete element identifiers are considered. However, the id of each element is not always available, as they may be assigned by the cloud itself. Even though the id assigned by the cloud can be stored and mapped to the model id, changes made outside of the framework can not be considered by this matching.

Thus, we utilized the *Similarity Flooding Algorithm*, a structure based graph matching algorithm (Melnik et al., 2002). This *similarity-based* algorithm iteratively calculates how well two nodes out of two different graphs match, using structural information. Therefore, we first transformed our runtime and target OCCI model into a *Pairwise connectivity graph (PCG)* (Melnik et al., 2002) indicating possible map pairs. These map pairs contain two `Resources`, one from each model, whereby both posses at least

one Link of the same Kind. Also, a map pair is created containing the target Resources of these Links, due to the evidence of a similar connection. This PCG is then transformed into an *Induced propagation graph (IPG)* (Melnik et al., 2002) that adds edges with weights to the graph, which are used to iteratively calculate a map pair's *fixpoint value*, indicating how well the two Resources of it match.

To utilize the information about already known ids, we propose an *mixed matching* approach depicted in Figure 4. Here, the deterministic results of a static identity-based matching is used to adjust the possible map pairs taken into consideration by the similarity-based approach. Thus, not only correctly identified matches are addressed, but also wrong candidates are excluded increasing the accuracy and performance of the overall comparison procedure. As the Similarity Flooding Algorithm calculates possible matchings based on their structure, we added a filter that compares Resources based on their attributes when they consider the same element as match and have equivalent structural values. Currently, this filter chooses the Resource with the most attribute values in common. This filter will be enhanced by weighting attributes in future work. Furthermore, we add a post-processing that matches Resources without any Links based on their attributes, due to missing structural information. In the following, the adaptation steps utilizing the comparison results are discussed.

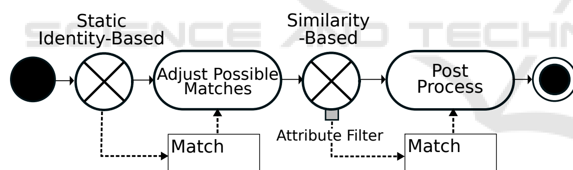


Figure 4: Mixed matching approach.

4.3 Adaptation Steps

Being RESTful, the OCCI API offers four requests to manage resources: GET, POST, PUT and DELETE. We map these operations onto the different adaptation steps required to transfer a cloud application topology from one state to another. After the extraction (GET), every Entity marked as missing is deleted (DELETE) reducing the amount of parallel running resources to a minimum to avoid quota conflicts. Subsequently, every Entity marked as updated gets adjusted (PUT), which brings the Entities in the required state for the rest of the cloud application topology to be provisioned. This is required, as for example, an inactive VM needs to be started in order to allow for additional operations to be performed on it. Finally, the Entities marked as new are provisioned (POST) to finalize the adaptation process. Hereby, a provisioning order must be identified that considers the already running resources and resolves the dependencies of the single resources to be created. In the following these adaptation steps are examined in more detail.

4.3.1 Deprovisioning

The deprovisioning process systematically deletes resources no longer needed in the running topology. Hereby, we separate the missing elements into Links and Resources. First the Links are deleted, followed by the deprovisioning of the Resources. Thus, we decouple a cloud resource before it is deleted increasing the robustness of the process.

4.3.2 Updating

To update an Entity, OCCI suggests using either a POST request for a partial update or a PUT request for an entire update (OGF, 2016b). Additionally, POST(Action) requests can be used to update an Entity, as they change the state of specific Attributes. To evaluate what kind of request has to be used to adapt a single Entity, we investigate the differences between the Entity and its match. To allow for such a separation, we check how the change of specific Attributes can be achieved. To check whether the execution of Actions is suitable, it must be known which Action affects which Attribute and how. Therefore, we define a list of OCCI specified Actions and their behavior. As OCCI does not exactly define the differences between the partial updates of POST requests and the full updates over PUT requests, the request to use depends on the OCCI implementation.

4.3.3 Provisioning

The provisioning of new Entities is the most crucial part of the adaptation process, as dependencies between the Entities must be resolved and the already deployed Entities have to be considered. To resolve these dependencies, we utilized the approach described in (Breitenbücher et al., 2014) in which *provisioning plans* are generated out of *Topology and Orchestration Specification for Cloud Applications (TOSCA)* (OASIS, 2013) models. We adapted this process to not only perform on OCCI models, but also provide a suitable provisioning plan interpreter which executes the required REST requests in the described order.

In (Erbel, 2017b), we already discussed this process considering initial deployments ignoring runtime information. In short, we performed a *model-*

to-model transformation (M2M) on the OCCI model generating a *Provisioning Order Graph (POG)*, which is a directed acyclic graph describing the dependencies of the cloud resources. Based on this graph, a provisioning plan is generated, which is represented by a workflow model. This plan does not only indicate the order in which the different `Entities` have to be provisioned, but also which of these requests can be send in parallel. To utilize the gathered runtime information, we adjusted the generation of the provisioning plan by removing updated and old elements from the generated POG with all edges connected to it. Thus, all dependencies requiring these resources to be running are resolved and no creation task for these elements is created within the provisioning plan.

5 EVALUATION

To show the feasibility of our approach, we evaluate the comparison process by investigating the accuracy of the different resource matching strategies in Section 5.1, followed by examining the complete adaptation process using a sample topology in Section 5.2.

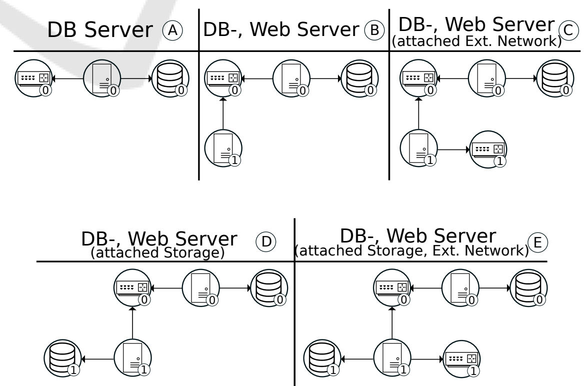
As visualizations of complete OCCI application topologies are too large for a depiction of our example scenarios, we compress the OCCI models in a graph based manner. Hereby, circles represent `Resources` and edges represent `Links`. The different images of these resources represent the `Kind` of the `Resource`, i.e. `Compute`, `Network`, and `Storage` nodes. The subscripts on the bottom right of each node represents the id for the corresponding `Resource`, which makes them uniquely identifiable throughout the `Resources` of their `Kind`. As the standard provides only one `Link` type to connect two different `Resources`, the annotation of their corresponding `Kind` is omitted. To highlight specific `Attributes`, we add additional information about the `Resource` on the bottom of the node.

For the evaluation we used an OpenStack[1,2] cloud with the *OpenStack OCCI Interface (OOI)*[3] installed, which implements the OCCI API for OpenStack addressing the infrastructure extension. For modelling the OCCI models, we use the Ecore metamodel described in (Merle et al., 2015). To extract the running cloud application topologies, we utilized the *jOCCI-API* (Kimle et al., 2015), which is a java library for performing OCCI-conformant REST-queries. Hereby, all OCCI related tools conform to OCCI-version 1.2. To implement typical model-driven techniques such as model transformations we use Eclipse *Epsilon* (Paige et al., 2009).

5.1 Comparison

To evaluate the comparison process we evaluate adaptation scenarios created from common cloud application topologies, depicted in Table 1. These are used in order to check which resource matching strategy is suitable for basic adaptation tasks, e.g., the creation or deletion of cloud resources. Due to space constraints only this subset of basic adaptation scenarios is provided, whereas scenarios representing corner cases of the matching strategies are presented in (Erbel, 2017a). The topologies, depicted in Table 1, describe a suite of Database Server (DBS) and Webserver (WS) related cloud application topologies in form of OCCI models. For example, the DB Server topology is represented by a `Compute` node, representing a VM which is connected to a `Network` node, as well as a `Storage` node. From here on, we expanded this example topology, considering an attached WS, a connection to an external network, and a WS with an attached storage device. For each of these topologies, we calculated a resource match using the following strategies: a static identity-based matching, a matching based on the Similarity Flooding Algorithm, and a mixed approach. Herewith, we evaluate whether the comparison strategies are able to handle multiple additions and removals of different cloud resources from several DBS andWS based topologies.

Table 1: Adaptation scenarios.



The mixed matching approach utilizes both benefits of the static identity-based and similarity-based matching approaches. In our case, `Resources` are first compared based on their id, followed by a comparison of `Resources` not already matched based on their position in the topology's structure. Hereby, `Resources` of equivalent structure are compared on

[1] All URLs have been last retrieved on 01/22/2018.
[2] https://releases.openstack.org/newton/
[3] http://ooi.readthedocs.io/en/1.2.0/

their attribute level. Due to the usage of the static identity-based matching as first comparison strategy, the mixed approach is able to correctly match every adaptation scenario. To provide a sufficient evaluation of the similarity-based part of the mixed matching approach, we evaluated each case with every `Resource` having different ids in the source and target model. The results are depicted in Table 2.

Table 2: Adaptation scenario results with different ids.

Source \ Target	(A)	(B)	(C)	(D)	(E)
DB Server (A)	Similarity Flooding	Similarity Flooding	Similarity Flooding	Attribute Filter	One VM Matched
DB-, Web Server (B)	Similarity Flooding	Similarity Flooding	Similarity Flooding	Attribute Filter	One VM Matched
DB-, Web Server (attached Ext. Network) (C)	Similarity Flooding	Similarity Flooding	Similarity Flooding	Attribute Filter	One VM Matched
DB-, Web Server (attached Storage) (D)	Attribute Filter	Attribute Filter	Attribute Filter	Attribute Filter	Attribute Filter
DB-, Web Server (attached Storage, Ext. Network) (E)	One VM Matched	One VM Matched	One VM Matched	Attribute Filter	Similarity Flooding

Legend: Similarity Flooding (hatched); Attribute Filter (dots); One VM Matched (empty)

Using the Similarity Flooding Algorithm the adaptation scenarios can only be partially solved: As soon as the second storage is within the source or target model, the structural matching approach is not able to correctly match the `Resources`, as symmetric structures result in non-deterministic solutions. For example, when comparing case A to D, it can not clearly be identified whether the topology of A represents the DBS or the WS within the topology of D. These cases, can be correctly matched using the attribute filter that compares two `Resources` of identical structural values on the attribute level.

However, the filter is not enough to correctly match the cases addressing topology E. Here, the structural values between the VM of the DBS and WS differ too much to trigger the attribute filter. This problem can be bypassed when at least one of the VMs is matched during the static identity-based phase of the mixed matching approach. It should be noted, that also a proper parameter study to adjust the attribute filter may help to overcome this issue. Another approach to solve this issue may be an iterative comparison utilizing multiple steps of similarity-based algorithms only choosing the best candidates, which is part of future work. When case E is compared to itself, the Similarity Flooding Algorithm implemented as part of the mixed matching approach is able to solely calculate a correct match, as the information about the additional connection to the external network is enough to identify the WS within the topology.

5.2 Example Adaptation

To discuss the complete adaptation process, we examine the procedure of the different adaptation steps based on a minimal example, depicted in Figure 5. The target model is composed of a `Compute` node, `VM1`, representing a VM. This VM is connected to a network `NW0` and a storage `Stor0`. To cover all subprocesses of the adaptation procedure in one minimal example, we created an artificial runtime state resulting in a cloud application topology that is composed of one active and one inactive VM, which are both connected to a network. We omit a detailed description of the extraction process, as it utilizes an already existing interface to generate the OCCI runtime model. Furthermore, the comparison step is not discussed in detail, as it is already evaluated within the previous section. Thus, we assume each id to be known in this example resulting in the marking of the `Entities` depicted in Figure 5. These, indicate a deletion of `VM0` and its connection to `VM0`, the update of `VM1`, and the addition of `Stor0` and its attachment to `VM0`.

After the comparison, the deprovisioning of the missing elements takes place. At first, the `Links` are separated from the `Resources` followed by their deprovisioning. Thus, first the `Link`, `VM0` $\rightarrow$ `NW0` is deleted, detaching `VM0` from the network, followed by a deprovisioning of `VM0` itself. It should be noted that, when a `Resource` is marked as missing, each `Link` contained within it is also marked as missing, i.e., each `Resource` is always completely detached before it is deleted. As `Entities` to be deleted are missing in the target model, the information about these elements is extracted from the runtime model. Thereafter, the updated elements are handled, changing the state of `VM1` to `active`. In the proposed example, we intentionally chose an element adaptation that can be performed by `Actions`, as OOI does not provide any implementation for PUT or POST update requests. To bring the `Attribute` "state" from inactive to active, the start `Action` is executed on `VM1`.

During the provisioning process, the POG is created according to the target model from which we remove the old and updated elements, as shown in Figure 6. Additionally, this figure depicts the provisioning plan for the original POG to highlight the differences. As this process is already discussed in (Erbel, 2017b), a shortened version is given in the following. As each provisioning of an infrastructure `Resource` is independent from each other, they represent vertexes without incoming edges. Therefore, a workflow is created which starts with a parallel provisioning of each `Resource` followed by the provi-

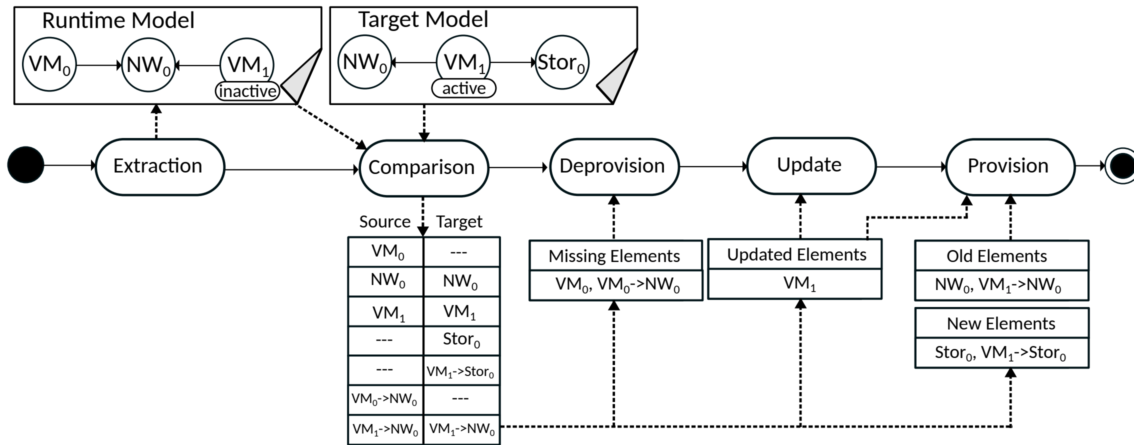


Figure 5: Minimal adaptation example.

sioning of `Links` connecting them. As we adapt the POG by deleting the old and updated elements, we remove `VM0`, `VM0` → `NW0`, and `NW0`. Thus, the POG is transformed into a workflow that provisions `Stor0` followed by the creation of the `Link` `VM0` → `Stor0`. After the creation of these `Resources` the runtime topology is adapted to the topology depicted in the target model, finishing the adaptation process.

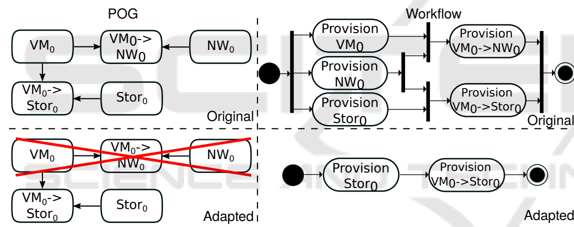


Figure 6: Workflow adaptation.

5.3 Discussion

In the former evaluation, we showed that the proposed process is capable of detecting the differences between two states of a cloud application topology and of performing corresponding steps to adapt it to the desired state. In this section, the results of this evaluation are used in order to answer the questions posed in Section 3.

- **Q1:** How to extract information from the running system into an OCCI model?

To extract the runtime information into an OCCI model, GET requests are send to the OCCI API gathering information about all running cloud resources. As we currently only consider application topologies, information about the running infrastructure is sufficient. The extraction of software states into OCCI models is part of future work.

- **Q2:** How to recognize whether two elements from different cloud application topologies match?

To answer this question, we stated several techniques to compare two topologies and match their elements. Hereby, we showed that a reasonable matching can be calculated based on the information contained within the elements attributes and structure. During the evaluation of the comparison process, we showed that the combination of a static identity and similarity-based matching improves the accuracy of the results in case not all elements can be clearly identified. As a similarity-based matching strategy, we utilized the Similarity Flooding Algorithm which calculates matching resources based on their structure within the topology. However, as symmetric structures represent the major drawback of such matching strategies, we added an attribute filter supporting the decision-making for multiple match candidates.

Furthermore, we proposed a modular process that can utilize any resource matching strategy in order to derive further information about equivalent resource connections, and corresponding adaptation steps to be performed. As shown in the evaluation of the comparison process, cases exists that are indistinguishable for the information provided requiring for a manual inspection.

- **Q3:** What is a suitable order for management calls to adapt a running cloud application topology?

As cloud tenants may be limited by an amount of cloud resources that can be acquired simultaneously, we start with the deprovisioning of cloud resources not needed anymore. Thereafter, we adapt each element requiring for an update to bring it into the desired state. This is needed for the provisioning process, as the target model depicts a working state of the cloud application and thus inconsistencies in the runtime have to be avoided. For the provisioning process itself, we enhanced the approach presented in (Erbel, 2017b) by utilizing the gathered runtime information.

Nevertheless, when further OCCI extensions are utilized, this overall design may result in problems, especially as the application layer adds a multitude of dependencies between the different resources requiring a more fine-grained resolution.

5.4 Threads to Validity

Even though we presented a feasible approach, circumstances exist that threaten its validity. One of the biggest impacts is the utilization of additional OCCI extensions, which may result in different requirements and more dependencies that need to be considered. However, these could not be considered due to the lack of compatible OCCI implementations. Additionally, major changes in OCCI's structure may influence parts of our processes.

Moreover, we did not test how the comparison as well as the adaptation process perform for larger topologies. Nevertheless, the provided cases were sufficient to investigate the advantages and drawbacks of the different comparison strategies and show the feasibility of our approach. Furthermore, an evaluation of the proposed adaptation process on multi-cloud environments is missing which may result in requirements not discovered yet.

6 RELATED WORK

The authors in (Breitenbücher et al., 2014) describe the generation of provisioning plans from TOSCA models. Nevertheless, this approach does not consider already provisioned resources. Similar to our work, the approaches proposed in (Holmes, 2015) and (Ferry et al., 2014) extract and compare models to adapt cloud application topologies. However, in these approaches metamodels were used that are not standard conform. Unfortunately, also no evaluation of their comparison processes are given, making a comparison to our results impossible.

To create an autonomic model for the management of cloud services, the authors in (Lejeune et al., 2017) utilize a directed acyclic graph as a metamodel. While their metamodel directly describe dependencies, we generate them allowing to utilize a more mature metamodel and standardized API. Additionally, we performed an evaluation of the identified steps in a real test environment instead of a simulation.

In (Kolovos et al., 2009) multiple types of model comparison approaches are presented. One of them is EMF Compare (Eclipse, 2011), which adds the capability to compare models to the Eclipse Modeling Framework (EMF) framework. Even though it is highly customizable, its build-in matching strategy mainly operates on the attribute level, evaluating the similarity of elements based on their features. In contrast, our approach also considers the structural information of the topology.

OCCIware (Parpaillon et al., 2015; Zalila et al., 2017) is a framework capable of modeling, and managing any kind of resources using OCCI. Thereby, it does not allow the comparison and adaptation of an already running cloud application topology to a target model and our approach is hence complementary.

7 CONCLUSION

We presented a model-based approach to adapt cloud application topologies at runtime, using the OCCI standard. To perform this adaptation, we identified three fundamental steps to be performed: the extraction of a runtime model, the comparison to the target model, and the execution of identified adaptation steps. For the comparison process we presented a generic process calculating required adaptation steps that mainly relies on the identification of matching resources. Therefore, we combined static identity-based with similarity-based matching techniques to utilize already known information in order to increase the accuracy of the overall comparison process. For the adaptation process, we identified and evaluated the sequence of deprovisioining, updating, and provisioning requests. For each of these tasks, we investigated different requirements. To deprovision a resource, we decoupled it from the rest of the topology. To update a resource, we checked whether it can be handled with help of OCCI actions. For the provisioning process, we identified dependencies of the single resources from which we generated a provisioning plan. Overall, we presented an approach capable of adapting a cloud application topology only requiring the desired state of the cloud application topology as input. Thus, we reduce the need for any human intervention and tackle the provider lock-in by extracting standard-conform REST calls directly from the model.

7.1 Future Work

In future work, we will extend the proposed process to support the platform extension (OGF, 2016d) defining elements to model complete cloud applications. We assume, that this additional information especially supports the comparison process, as the cloud application's structure gets more distinguishable. Moreover, we will test further candi-

dates for pre-comparison steps such as iteratively using similarity-based strategies in which only the most suitable matches are considered. Also, the attribute filter can be enhanced by evaluating which attribute changes fit to the structure of a resource. Finally, the adaptation process can be enhanced by state machines describing how attributes are affected by different management actions to reduce the need for manual configurations.

ACKNOWLEDGEMENTS

We thank the Simulationswissenschaftliches Zentrum Clausthal-Goettingen (SWZ) for financial support.

REFERENCES

Breitenbücher, U., Binz, T., Képes, K., Kopp, O., Leymann, F., and Wettinger, J. (2014). Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 87–96. IEEE.

Eclipse (2011). EMF Compare. Available online: `http://www.eclipse.org/emf/compare/`, last retrieved: 01/22/2018.

Erbel, J. (2017a). Comparison And Adaptation Of Cloud Application Topologies Using Models At Runtime. Master's thesis, Institute of Computer Science, University of Goettingen, Germany.

Erbel, J. (2017b). Declarative Cloud Resource Provisioning Using OCCI Models. In *Informatik 2017, 47. Jahrestagung der Gesellschaft für Informatik*.

Ferry, N., Brataas, G., Rossini, A., Chauvel, F., and Solberg, A. (2014). Towards bridging the gap between scalability and elasticity. In *Proceedings of the 4th International Conference on Cloud Computing and Services Science - Volume 1: (CLOSER 2014)*, pages 746–751. INSTICC, SciTePress.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine.

Holmes, T. (2015). Facilitating Migration of Cloud Infrastructure Services: A Model-Based Approach. In *CloudMDE@MoDELS*, pages 7–12.

Kimle, M., Parák, B., and Šustr, Z. (2015). jOCCI–General-Purpose OCCI Client Library in Java. In *International Symposium on Grids and Clouds (ISGC)*, volume 15.

Kolovos, D. S., Di Ruscio, D., Pierantonio, A., and Paige, R. F. (2009). Different Models for Model Matching: An analysis of approaches to support model differencing. In *Comparison and Versioning of Software Models, 2009. CVSM'09. ICSE Workshop on*, pages 1–6. IEEE.

Lejeune, J., Alvares, F., and Ledoux, T. (2017). Towards a generic autonomic model to manage cloud services. In *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*, pages 147–158. INSTICC, SciTePress.

Melnik, S., Garcia-Molina, H., and Rahm, E. (2002). Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 117–128. IEEE.

Merle, P., Barais, O., Parpaillon, J., Plouzeau, N., and Tata, S. (2015). A Precise Metamodel for Open Cloud Computing Interface. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 852–859. IEEE.

OASIS (2013). Topology and Orchestration Specification for Cloud Applications. Available online: `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca`, last retrieved: 01/22/2018.

OGF (2016a). Open Cloud Computing Interface - Core. Available online: `https://www.ogf.org/documents/GFD.221.pdf`, last retrieved: 01/22/2018.

OGF (2016b). Open Cloud Computing Interface - HTTP Protocol. Available online: `https://www.ogf.org/documents/GFD.223.pdf`, last retrieved: 01/22/2018.

OGF (2016c). Open Cloud Computing Interface - Infrastructure. Available online: `https://www.ogf.org/documents/GFD.224.pdf`, last retrieved: 01/22/2018.

OGF (2016d). Open Cloud Computing Interface - Platform. Available online: `https://www.ogf.org/documents/GFD.227.pdf`, last retrieved: 01/22/2018.

Paige, R. F., Kolovos, D. S., Rose, L. M., Drivalos, N., and Polack, F. A. (2009). The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering. In *Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on*, pages 162–171. IEEE.

Parpaillon, J., Merle, P., Barais, O., Dutoo, M., and Paraiso, F. (2015). OCCIware-A Formal and Tooled Framework for Managing Everything as a Service. In *Projects Showcase@ STAF'15*, volume 1400, pages 18–25.

Zalila, F., Challita, S., and Merle, P. (2017). A model-driven tool chain for OCCI. In *On the Move to Meaningful Internet Systems. OTM 2017 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2017, Rhodes, Greece, October 23-27, 2017, Proceedings, Part I*, pages 389–409.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS