Automated Provisioning of Customized Cloud Service Stacks using Domain-Specific Languages

Ta'id Holmes

Products & Innovation, Deutsche Telekom AG
Darmstadt, Germany
t.holmes@telekom.de

Abstract Cloud computing gave birth to a paradigm in which infrastructure can be requested, provisioned, and used almost instantly in a service-oriented manner. Infrastructure as a service, however, is only the first step in cloud adoption. In fact, cloud computing introduces various distinct service models constituting a cloud service stack. Each of the models abstracts from lower-level cloud services and comprises only a limited set of new concepts. In situations where entire cloud stacks are to be provisioned, the overall complexity needs to be managed. For mastering complexity, model-based approaches have proved beneficial. Equally important, they realize automation while capturing valuable expert knowledge. For this reason, a model-driven approach comprising tailored domain-specific languages for the provisioning of customized cloud stacks has been adopted.

Keywords: cloud, DSL, IaaS, MDE, model-based, provisioning, SaaS

The paradigm of cloud computing was born out of the spirit of service computing. As a result, a stack comprising infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS) offers different functionalities for the provisioning and management of cloud services. Latter services abstract from underlying services introducing the roles of respective service providers and service consumers. This abstraction realizes transparency in terms of hardware, operating system, and possibly also network, location, and employed technologies.

At the same time, the abstraction established by the service models naturally constrains service consumers to some degree as properties of lower cloud services are aggregated. For example, the file system, its redundancy, and distribution usually cannot be controlled by an SaaS provider as it falls into the responsibility of the IaaS provider. For providers of higher-level cloud services, a certain configuration of distinctive underlying cloud service properties may be a key requirement, however.

Thus, in an industrial context, the lack of control over lower-level cloud service properties may hinder the adoption of cloud-based development and operation. Moreover, building SaaS solutions on top of a PaaS usually requires a homogeneous technology stack. While a PaaS may offer additional value, simplifying development and deployment, it may also be perceived as inflexible. Someone having a background on system administration might prefer to build on an IaaS for provisioning and exposing higher cloud services. From a security perspective a setup in which distinct tenants separate the data on a lower-level of the cloud stack may be preferred. That is, while it would be

possible to work with a PaaS using a (multi-tenant) database, a requirement (e.g., from management) may demand that multi-tenancy takes place at an IaaS level so that data is physically separated.

In such cases, where individual setups are to be provisioned on top of IaaS deployments and for keeping the spirit of service-orientation, automation is key. For mastering the complexity it also becomes necessary to elevate concepts of cloud computing from technical terms to higher levels of abstraction. Both challenges can be addressed following a model-based approach. This paper reports on the industrial adoption of such a model-based approach. Domain-specific languages (DSLs) for describing customized cloud stacks are presented together with respective model transformations and services.

The remainder of this paper is structured as follows: Section 1 presents a motivating example. The approach and the DSLs for configuring customized cloud stacks is presented in Section 2. Next, Section 3 revisits the case study by illustrating the applicability of the approach and Section 4 compares to related work. Finally, Section 5 presents lessons learned and discusses on the benefits, risks, and limitations of the approach and Section 6 concludes the paper.

1 Customized Cloud Stacks — A Motivating Example

In an enterprise, internal users can profit from IaaS services by requesting resources using a self-service. This greatly reduces administrative overhead, waiting time, and costs. This is especially true for innovation projects and prototyping as requirements may change over time and iteration cycles need to be kept short. Besides project portals supporting an agile methodology with capabilities such as version control, wiki, and bug-tracking there is often a need to also simply provide innovation projects with a "playground" of readily available server infrastructure. This way, someone familiar with system administration can profit from the full flexibility of the systems. Yet, software and services need to be installed, configured, and deployed. In order to reduce this work – which is an overhead to the project, it would be interesting to automate the latter without putting restrictions on the subsequent use and project-specific customization. Ideally, it would be easy and prompt to demand a complicated server landscape.

For this, let us consider a machine-to-machine (M2M) scenario in which a proof of concept (PoC) has been developed. A multitude of M2M devices gathers data through sensors and emits events that are processed by a backend in a publish–subscribe manner. Finally, a dashboard provides a monitoring view for web clients. The PoC correlates data from the devices with user data within the backend in near real-time. For demonstrating the PoC it suffices to integrate it within a simplified setup. For provisioning a demonstrator for a PoC, an IaaS platform can provide the on-demand infrastructure. Yet, software and services need to be installed, configured, and integrated. That is, higher-level cloud services need to be provisioned as well. The resulting overall cloud service stack is rather particular to the demonstrator (i.e., the PoC) or its context (M2M in this case). Thus, it is referred to in this paper as a *customized cloud stack*.

In such situations, in which entire cloud service stacks are to be provisioned in a service-oriented manner, automation is required. Beyond that and for supporting the on-demand provisioning of customized cloud stacks, complexity needs to be mastered.

2 Demanding Cloud Stacks using Domain-Specific Languages

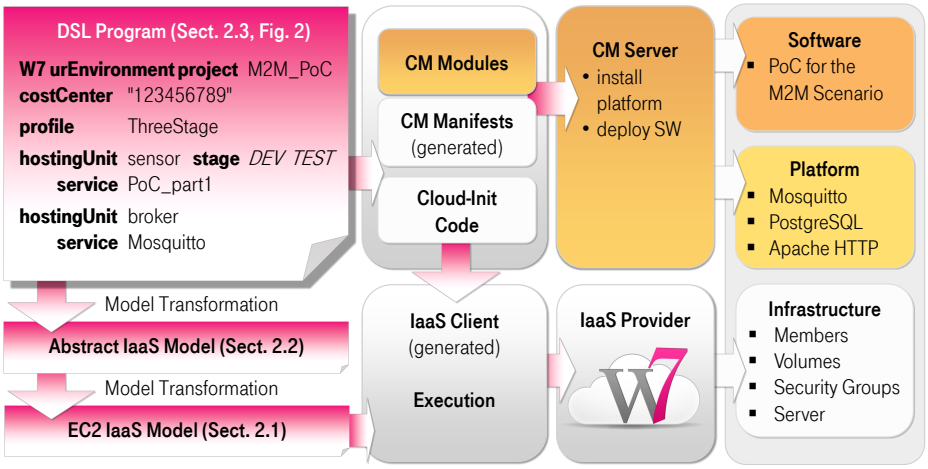

Figure 1: Overview of the Model-based Approach for the Automated Provisioning

Using the motivating example, Figure 1 gives an overview of the approach. On the right hand side the customized cloud stack is depicted and the left hand side illustrates its model-based specification and transformation. Provisioning is realized by executing a generated IaaS client and optionally and in addition by relying on configuration management (CM) software as shown in the middle of the figure.

A bottom-up approach was chosen for the engineering of the DSLs (cf. [6]) and its transformations. For this reason the first DSL, presented in Section 2.1, simply reflects IaaS concepts and is closely related to the respective application programming interfaces (APIs) through code generators. In terms of the original model-driven architecture (MDA) proposal [1], an instance of the abstract DSL, i.e., an instance of the metamodel as defined by the grammar, corresponds to a platform-specific model (PSM).

The second DSL still focuses on infrastructure but abstracts from some concepts and is outlined in Section 2.2. It builds on conventions and leaves out some details. As a result, it eases the specification of IaaS. Compared to the first DSL, when used, this DSL produces more compact code (referred to as *DSL programs*). As a consequence, it also reduces the chance for errors; i.e., some validators, that need to check PSMs, are not required because of conventions the DSL is based on. A DSL program is parsed and mapped through model-to-model transformation to a PSM. Finally, code generators produce the respective service consumers for the provisioning of the demanded infrastructure as specified in the DSL programs.

[1] http://omg.org/cgi-bin/doc?omg/03-06-01

Eventually, a third, high-level DSL permits the specification of customized cloud stacks and is explained in Section 2.3. A model is first mapped to a general IaaS model and then transformed via a PSM to the respective IaaS client that realizes the provisioning of the customized cloud stack.

2.1 A Domain-Specific Language for IaaS APIs

An IaaS called Wolke 7 (W7) is deployed internally at Deutsche Telekom. Based on OpenStack [2] it enables self-service through a dashboard and exposes Amazon Web Services (AWS) and other APIs for management. When starting to adapt a model-based approach for the overall goal, the initial step was to build a metamodel for Amazon Elastic Compute Cloud (EC2) [3]. This was realized by defining a grammar of a concrete DSL using Eclipse Xtext (Xtext) [4].

Besides a project identifier and an optional description, an IaaS project states a cost center for internal service charging and the creator of the project, and enumerates its members. Finally, security groups, volumes, and servers are defined. The grammar rule for a security group comprises firewall rules (`FWRule`) that state the protocol, the source (`src`), and one or more destination (`dst`) ports or port ranges. For the source either another security group needs to be referenced or a network address has to be specified. Grammar rules for `volumes` and `servers` are defined similarly. They comprise further rules and capture concepts such as `images`, `flavors`, `cpu`, `ram`, and `disk`.

The resulting DSL closely reflects EC2 concepts, is, consequently, rather platform-specific, and does not realize much of an added value apart from the fact that these concepts are now available to the modeling. In particular, the abstract DSL constitutes a target metamodel for the higher-level DSLs. Clients using the IaaS APIs naturally form the target of the execution engine [5]. Because the DSL is tightly bound to these, an IaaS client can easily be generated through a model-to-text transformation. Besides a shell script using Euca2ools [6] also a shell script using OpenStack Compute (Nova) [7] client has been developed.

2.2 A Simplifying, Abstracting Language for IaaS

Abstracting from the IaaS DSL, security groups comprise respective firewall rules and aggregate servers. On the one hand, the aggregation of servers in security groups is a constraint compared to the EC2 model where servers are associated with one or more security groups. On the other hand, it simplifies configuration for DSL users, presumed that a server only needs to "reside" within a security group. A project can specify `defaults` that apply to the server definitions such as the default `flavor` or `image`. These concepts are directly used from the lower-level DSL through language referencing

[2] http://openstack.org
[3] http://awsdocs.s3.amazonaws.com/EC2/latest/ec2-api.pdf
[4] http://eclipse.org/Xtext
[5] The execution engine interprets the DSL programs or transforms the models (cf. [6]).
[6] http://eucalyptus.com/docs/euca2ools/3.0/euca2ools-guide-3.0.2.pdf
[7] http://nova.openstack.org

(cf. [6, p. 119]) using Xtext's `import` statement. A server may overwrite these defaults by locally specifying respective values. In contrast to the EC2 model, volumes do not have to be defined explicitly and attached to a server. Here they are defined implicitly using `mount` statements. An advantage is that the block device can be formatted and mounted into the filesystem during provisioning. In addition, an offsite backup strategy may be specified using duplicity [8] behind the scenes. Such features can be activated with a few DSL keywords and parameters and made effective due to the realized automation while following best practices. They already present added value to the plain EC2 IaaS.

2.3 Specifying Customized Cloud Stacks

Having abstracted from EC2 previously, the third DSL focuses on specifying customized cloud stacks. The main idea is to not only state infrastructure but also software and services. That is, an entire cloud service stack can be specified using a DSL. In order to build on established CM solutions as well as not to pollute the DSL with technical aspects of the deployment the latter are weaved into the model-driven approach. Yet, the DSL is complete so that modeling is not blocked by the other activities lowering the barrier to obtain at least some cloud services such as the infrastructure. Also, security groups with all their technical details are abstracted from as much as possible. In addition, the various stages of the engineering lifecycle such as development, test, and production are considered. That is, infrastructure is provisioned similarly for each of the stages. This way it can be ensured that a cloud stack for testing or preproduction is provisioned equally as for production. Exceptions to such replications are possible; e.g., a repository may only be required for development.

The overall toolchain comprises the following parts: besides the parsing of the DSL programs and their subsequent transformation, cloud-init files may be weaved into a userdata that is passed when launching servers. This takes place when a cloud-init file is available for a `service` as specified in the project. For (further) service provisioning Puppet [9] can be used. Indeed, Puppet is preferred over cloud-init for the CM and provisioning making it only necessary to supply a single cloud-init file with a Puppet directive for configuring the Puppet agent when launching a server instance. Similarly to cloud-init files, Puppet modules are included into manifest files of respective servers when the name of a `service` matches. While the DSL is complete (cf. [6, p. 109]), the approach currently relies on Puppet experts for providing respective modules realizing separation of concerns (SoC). That is, the conceptual part can be expressed using the DSL and the technical details for the provisioning are supplied separately. In particular, Puppet modules can be developed prior or subsequently to the DSL programs and made available to other projects through a common repository.

The rule for the project resembles the definition from the lower-lever DSLs, i.e., (meta)data such as the cost center or members are listed. Differently, it comprises a `profile` and `hostingUnits` with `services`. At some places it uses references to separately defined entities, i.e., the `profile` and `serviceTypes` can be defined globally or individually for the project. The `profile` defines `stages` where each

[8] http://duplicity.nongnu.org
[9] http://puppetlabs.com

stage can be bound to a dedicated cloud. This way, the production environment can be located at a different cloud region than where development takes place. A hostingUnit corresponds to a server if no particular scale parameters are passed. Otherwise multiple server instances will be created for constituting a cluster. If not explicitly bound to one or more stages, the servers of a hostingUnit will be instantiated in all the stages. Similarly, a service of a hostingUnit can further refine its own instantiation, i.e., it can specify stages out of the subset of its hostingUnit. If a service shall not be exposed externally, it can be declared as internal. In this case no allowing security rule will be generated for those ports, which the serviceType may be associated with. Finally, a serviceType may imply other services. This permits to define transitive dependencies amongst serviceTypes.

3 Revisiting and Resolving the Case Study

A motivating example has been described in Section 1 in which a demonstrator for a PoC in the context of M2M is wanted and is to be developed within a customized cloud service stack. Expressed in the DSL presented in Section 2.3, Figure 2 depicts the programs for describing the respective cloud service stack. The project (see Figure 2a) comprises a hostingUnit simulating a sensor during the stages of development (DEV) and test (TEST) while in production real M2M devices generate the data. The sensor hosts the first part of the cloud-based PoC (PoC_part1). A broker is realized by the Mosquitto [10] software. As it is a MQ Telemetry Transport (MQTT) [11] broker, it implies the rather abstract serviceType MQTT (cf. Figure 2b) with the default ports 1883 and 8883 for Transport Layer Security [12]. Other hostingUnits similarly host other services such as PostgreSQL [13] or the other parts of the PoC. Note that dependencies are specified for all parts of the PoC discretely. This simplifies the task of defining the cloud service stack and moves responsibility to defining the respective serviceTypes which can be realized by a different information worker or even role at a different place. Please also note that definitions can often be reused and, as a best practice, can be moved to a standard library. In this self-containing example it would have sufficed to only define the project specific serviceTypes for the different parts of the PoC while the other definitions (including the profile) would have been contributed to a standard library from which they would be available.

The services listed in the hosting units are deployed together with their transitive dependencies on the respective server instances using the underlying CM software. Also their ports are considered for the IaaS security rules. From the DSL programs – their generated CM files and IaaS clients – and the supplied Puppet modules, the entire cloud service stack is built automatically and without further user interaction. An additional management server must be present, however, that acts as the Puppet master for the servers as defined in the DSL program. Its hostname and certificate are injected into the Puppet agent configuration of cloud-init (see Section 2.3).

[10] http://mosquitto.org
[11] http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/MQTT_V3.1_Protocol_Specific.pdf
[12] http://ietf.org/rfc/rfc4346.txt
[13] http://postgresql.org

```
W7 urEnvironment project
M2M_PoC
costCenter "123456789"
members {
    "t.holmes@telekom.de"
    "r.schwegler@telekom.de"
}
createdBy "t.holmes@telekom.de"

profile ThreeStage

hostingUnit sensor flavor S
                    stage DEV TEST
    service PoC_part1

hostingUnit broker flavor S
    service Mosquitto

hostingUnit converter flavor S
    service PoC_part2

hostingUnit analytics flavor S
    service PoC_part3

hostingUnit db
    service PostgreSQL

hostingUnit www
    service ApacheWSGI
    service PoC_part4
```

(a) A Customized Cloud Stack

```
W7 urEnvironment globals
profile ThreeStage
    stages DEV ("development")
           TEST ("test")
           PROD ("production")
serviceType Apache implies
    service Web
serviceType ApacheWSGI implies
    service Apache
serviceType Mosquitto implies
    service MQTT
serviceType MosquittoClient implies
    service PyXB
serviceType MQTT
    ports TCP 1883,8883
serviceType PostgreSQL
    ports TCP 5432
serviceType PoC_part1 implies
    service MosquittoClient
serviceType PoC_part2 implies
    service MosquittoClient
serviceType PoC_part3 implies
    service MosquittoClient
    service SQLAlchemy
serviceType PoC_part4 implies
    service SQLAlchemy
serviceType PyXB
serviceType SQLAlchemy
serviceType Web
    ports TCP 80,443
```

(b) Profile and Service Type Definitions

Figure 2: DSL Programs for the Machine-to-Machine Scenario

The approach permitted to successfully setup a customized cloud stack for the development of a PoC within the M2M context. Once the PoC was implemented, the entire stack for demonstration purposes could be provisioned within the dimension of minutes. Accessing the live demonstrator, finally, is as simple as opening the assigned floating IP address of the web server with the dashboard in a web browser.

4 Related Work

The OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) [14] standard aims at portability of cloud services. It permits the self-contained description of entire cloud services stacks. As such, it enables the control of lower-level cloud service properties. Building on top of a heavyweight technology stack it requires – without further tool support – experts to bundle cloud applications. Moreover, it relies on a TOSCA container such as OpenTOSCA [2]. In contrast, the approach presented in this paper is lightweight, directly operates on an IaaS provider, and aims at reaching end-users facilitating self-service. For instance, DSL programs can be written also by

[14] http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html

developers that are not familiar with, e.g., (process-driven) service-oriented architecture (SOA) technologies. While both of the approaches have a common goal in describing service stacks, they have different focuses: As TOSCA's main objective is the technical portability of cloud services (cf. [3]) it does not need to simplify the process of specifying a customized cloud stack in the first place which is a focus of the work presented. Nevertheless, there is work under way to address the usability of TOSCA: Winery [15] [5] enables users to graphically model service topologies.

AWS CloudFormation [16] facilitates the instantiation of a collection of cloud services through templates. The higher-level DSLs also aim at instantiating cloud services using an IaaS provider, yet follow a different approach. Compared to the DSL programs the templates are not intended for end-users, i.e., they must be written by experts. Once available, however, they can be interpreted by a web-based management console where a user can specify parameters. Beyond the instantiation of a collection of cloud services, the presented work also considers the engineering lifecycle and supports the instantiation in multiple cloud regions. Again, the work presented may be combined with other technologies following a model-driven approach. That is, from the DSL programs respective templates could be generated. Please note that while possible the intended usage pattern is different in this case, however: a template – relatively expensive in its creation – is expected to be instantiated often, whereas using the DSLs rather new, different, or modified programs are transformed promptly as needed.

Configuration management software such as Puppet or Chef [17] (both internal DSLs) generally are too low-level compared to what this approach aims for, i.e., enable non-experts to specify customized cloud stacks. Yet, overall complexity cannot be reduced and the functionality of CM software is welcomed, required, and thus incorporated into the approach. While accessing CM software and offering experts the possibility to integrate into and contribute to the overall toolchain, the DSLs presented reach for a wider audience and leverage the added value of incorporated technologies and IaaS providers. As external DSLs they are more tailored and leave out features of a general purpose host language. An approach that started to follow the vision of incorporating end-users is JuJu [18]: a graphical user interface permits cloud users to graphically design deployments. Besides the abstraction from community-contributed scripts called Charms, further support for different roles as common in model-driven engineering (MDE) approaches, such as for conducting multi-step configuration, may be desirable.

5 Discussion and Lessons Learned

While the previous section compared to the state of the art by discussing the various approaches, this section reflects on the presented work describing applicability, benefits, risks, and limitations. Finally, some lessons learned are presented.

The descriptive DSL programs are easy to write, compact, and intuitive. Differences between versions of a program can easily be recognized by users when using a version

[15] http://projects.eclipse.org/projects/soa.winery
[16] http://aws.amazon.com/cloudformation
[17] http://getchef.com
[18] http://juju.ubuntu.com

control system. This is because, besides some references, the textual DSL does not comprise concepts that are scattered across multiple places. The approach realizes SoC, i.e., technical details are realized by CM experts, e.g., developers, while a high-level description of a cloud stack is specified by, e.g., a cloud architect. Thus, common to MDE approaches, different roles are incorporated – each working within a defined level of abstraction. This lowers entry barriers for each of the roles and results in more efficiency.

The presented work can build on different IaaS providers and can be applied in various contexts. The former applies as EC2 is a de facto standard supported by a variety of IaaS providers, but in other cases an adapted code generator would make use of the respective APIs or clients. While an M2M scenario was used as a case study, it is not limited to this context. Other DSL programs for describing cloud stacks may differ significantly and thus the DSL can be used for diverse scenarios having different contexts. The work is not only interesting when developing a PoC as pictured in the motivating example. Besides innovation projects, the work can be applied also in (evolutionary) prototyping scenarios and when testing a minimum viable product.

Furthermore, it can help to analyze cloud stacks and support the substitution of services with either mockups or implementations. For example, in the motivating example, certain components such as the M2M devices may be simulated in the beginning. While these simulators may continue to be deployed in development and testing, real M2M devices would take over in production. Another example would be the substitution of the MQTT broker: Mosquitto could be replaced by RabbitMQ [19]. Supporting such substitutions can accelerate evaluation of services; particularly when combined with automatic performance tests.

The work has been conducted using GNU/Linux-based operating system images for the server instances. Yet, as Puppet is a cross-platform CM, the approach does not come with such a restriction per se. A current presumption is that any dependencies between `hostingUnits` and/or within Puppet modules are taken care by Puppet experts. As mentioned in the previous section there is a risk that TOSCA obsoletes parts of the work presented in this paper. As TOSCA is a standard by now and further development and tool support is to be expected from the community, it could be contemplated to support TOSCA as a future work. This would be beneficial for rapid application development and would provide a way to make TOSCA and related technologies accessible. That is, TOSCA would form the target language for the DSLs as presented in this paper. It is expected that the model-based approach proves flexible enough to undertake migration of cloud stacks to TOSCA if desired.

Once the overall toolchain was automated and it was possible to provision entire service stacks, soon a new use case emerged. Not only should it be possible to describe and provision a customized cloud stack but it would be interesting to also support changes. That is, while a (new or modified) stack can always be (re)provisioned, it would be nice to only consider changes in case of an existing, previously built stack. For supporting this use case, the change impact needs to be analyzed and dealt with. In simple scenarios, the stack would merely be extended making it necessary to solely deploy the new cloud services. In order to realize the use case, a differential approach was adapted. That is, a service consuming and parsing the DSL programs forming a target

[19] http://rabbitmq.com

model, invokes a service that reflects on the current state using the IaaS provider and that generates a runtime model (cf. [4]). For this, the IaaS metamodel has been enriched with runtime aspects. From the target and the current runtime model an Eclipse Modeling Framework (EMF) DiffModel [20] is calculated which is interpreted for executing the changes. Further work needs to be undertaken in this regard; e.g., to involve users for approving particular changes.

6 Conclusion

When a customized stack of cloud services is preferable over a uniform stack, comprehensive provisions need to take place for realizing the entire service stack on top of an IaaS. For mastering complexity and for realizing automation, it is feasible – both from a technical and a practical point of view – to apply model-based technologies for the provisioning of customized cloud stacks. For this, DSLs can be engineered – as shown in this paper – that are well suited for the specification of such stacks. Abstracting from technical details the approach simplifies specification while realizing platform independence. Accessing well-established software for realizing the low-level configuration, the work presented combines the best of two worlds: i.e., CM – backed and driven by a strong community – and modeling that advances engineering to higher levels while reaching and incorporating end-users. Because of the modeling dimension of the approach, it is believed that the presented work – focusing on a textual, descriptive DSL interface for customized on-demand stacks – can easily be adopted, adjusted, and even combined with other work that aims at easing the provisioning of service topologies.

Acknowledgments The author would like to thank Robert Schwegler and Bernard Tsai for providing valuable feedback regarding the design of the DSLs, peer reviewers for their estimated service, Tassilo Huch for his support in preparing Figure 1, and Mike Machado for proofreading.

References

1. Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.): Service-Oriented Computing - 11th International Conference, ICSOC 2013, Berlin, Germany, December 2-5, 2013, Proceedings, Lecture Notes in Computer Science, vol. 8274. Springer (2013)
2. Binz, T., Breitenbücher, U., Haupt, F., Kopp, O., Leymann, F., Nowak, A., Wagner, S.: OpenTOSCA - A Runtime for TOSCA-Based Cloud Applications. In: Basu et al. [1], pp. 692–695
3. Binz, T., Breitenbücher, U., Kopp, O., Leymann, F.: TOSCA: Portable Automated Deployment and Management of Cloud Applications. In: Bouguettaya, A., Sheng, Q.Z., Daniel, F. (eds.) Advanced Web Services, pp. 527–549. Springer (2014)
4. Blair, G.S., Bencomo, N., France, R.B.: Models@ run.time. IEEE Computer 42(10), 22–27 (2009)
5. Kopp, O., Binz, T., Breitenbücher, U., Leymann, F.: Winery - A Modeling Tool for TOSCA-Based Cloud Applications. In: Basu et al. [1], pp. 700–704
6. Völter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L.C.L., Visser, E., Wachsmuth, G.: DSL Engineering - Designing, Implementing and Using Domain-Specific Languages. dslbook.org (2013)

[20] EMF Compare: http://wiki.eclipse.org/EMF_Compare