
Proceedings of the 2nd International Workshop on
Model-Driven Engineering on and for the Cloud

CloudMDE 2014

Richard Paige
Jordi Cabot
Marco Brambilla
Louis Rose
James H. Hill

Copyright © 2014 for the individual papers by the papers' authors. Copying permitted only for private and academic purposes. This volume is published and copyrighted by its editors.

Preface

The second workshop on Model-Driven Engineering (MDE) for and in the Cloud was held on 30 September 2014 at UP Valencia, Spain co-located with the ACM/IEEE 17th International Conference on Model-Driven Engineering Languages and Systems. Model-Driven Engineering (MDE) elevates models to first class artefacts of the software development process. MDE principles, practices and tools are also becoming more widely used in industrial scenarios. Many of these scenarios are traditional IT development and emphasis on novel or evolving deployment platforms has yet to be seen. Cloud computing is a computational model in which applications, data, and IT resources are provided as services to users over the Internet. Cloud computing exploits distributed computers to provide on-demand resources and services over a network (usually the Internet) with the scale and reliability of a data centre.

Cloud computing is enormously promising in terms of providing scalable and elastic infrastructure for applications; MDE is enormously promising in terms of automating tedious or error prone parts of systems engineering. There is potential in identifying synergies between MDE and cloud computing. The workshop aimed to bring together researchers and practitioners working in MDE or cloud computing, who were interested in identifying, developing or building on existing synergies. The workshop focused on identifying opportunities for using MDE to support the development of cloud-based applications (MDE for the cloud), as well as opportunities for using cloud infrastructure to enable MDE in new and novel ways (MDE in the cloud).

Attendees were also interested in novel results of adoption of MDE in cloud-related domains that provided insight into early adoption of MDE for building cloud-based applications, or in terms of deploying MDE tools and infrastructure on ‘the cloud’.

The workshop received 12 paper submissions (technical papers, position papers and work-in-progress papers), from which it accepted 8 for presentation at the workshop. Each paper was reviewed by 2-3 members of the program committee, and was selected based on its suitability for the workshop, novelty, likelihood of sparking discussion, and general quality. The workshop also featured a keynote presentation by Daniel Varro of the Budapest University of Technology, Hungary. The organisers thank all authors for submitting papers, our keynote speaker Daniel Varro, the workshop participants, the MoDELS local organisation team, the workshop chairs Alfonso Pierantonio and Gabi Taentzer, and the program committee for their support.

30th September, 2014
Valencia, Spain

Richard Paige
Jordi Cabot
Marco Brambilla
Louis Rose
James H. Hill

Organisation

CloudeMDE 2014 was organised by the Department of Information Systems and Computation (DSIC) at Universidad Politécnica de Valencia.

Program Committee

Muhammad Ali Babar	IT University of Copenhagen, Denmark
Marco Brambilla	Politecnico di Milano, Italy
Jordi Cabot	École des Mines de Nantes, France
Radu Calinescu	University of York, UK
Giuliano Casale	Imperial College London, UK
Marcos Didonet Del Fabro	Universidade Federal do Paraná, Brazil
Aniruddha Gokhale	Vanderbilt University, USA
Esther Guerra	Universidad Autónoma de Madrid, Spain
James H. Hill	Indiana University-Purdue University Indianapolis, USA
Frank Leymann	University of Stuttgart, Germany
Sebastien Mosser	University Nice-Sophia Antipolis, France
Ileana Ober	Université Paul Sabatier, Toulouse, France
Richard Paige	University of York, UK
Dana Petcu	West University of Timisoara, Romania
Istvan Rath	Budapest University of Technology and Economics, Hungary
Louis Rose	University of York, UK
Manuel Wimmer	Vienna University of Technology, Austria

Additional Reviewers

Alexander Bergmayr	Vienna University of Technology, Austria
Gabriel Costa Silva	University of York, UK
Ákos Horváth	Budapest University of Technology and Economics, Hungary

Table of Contents

MDE Opportunities in Multi-Tenant Cloud Applications	1
Mohammad Abu-Matar and Jon Whittle	
Towards Cloud-Based Software Process Modelling and Enactment	6
Sami Alajrami, Alexander Romanovsky, Paul Watson and Andreas Roth	
Towards Pattern-Based Optimization of Cloud Applications	16
Martin Fleck, Javier Troya, Philip Langer and Manuel Wimmer	
Modeling Cloud Messaging with a Domain-Specific Modeling Language	26
Gábor Kövesdán, Márk Asztalos and Laszlo Lengyel	
Cloud DSL: A Language for Supporting Cloud Portability by Describing Cloud Entities	36
Gabriel Costa Silva, Louis M Rose and Radu Calinescu	
Automated Provisioning of Customized Cloud Service Stacks using Domain-Specific Languages	46
Ta'Id Holmes	
UML-based Cloud Application Modeling with Libraries, Profiles, and Templates	56
Alexander Bergmayr, Javier Troya, Patrick Neubauer, Manuel Wimmer and Gerti Kappel	
MDEForge: an Extensible Web-based Modeling Platform	66
Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Amleto Di Salle, Ludovico Iovino and Alfonso Pierantonio	

MDE Opportunities in Multi-Tenant Cloud Applications

Mohammad Abu Matar¹ and Jon Whittle²

¹ Etisalat British Telecom Innovation Center
Khalifa University of Science, Technology and Research
Abu Dhabi, United Arab Emirates
mohammad.abu-matar@kustar.ac.ae

² School of Computing and Communications
InfoLab21, Lancaster University
Lancaster LA1 4WA United Kingdom
j.n.whittle@lancaster.ac.uk

Abstract. Cloud computing promotes economies of scale by sharing software and hardware resources across multiple tenants. To date, there has been relatively little research on how MDE can best support multi-tenant cloud applications, where there is a need to separate the logic and data of multiple tenants. In this position paper, we sketch out five key research opportunities for applying MDE to multi-tenant cloud applications.

1 Introduction

Cloud computing is a widespread model for sharing computing resources that promotes economies of scale by hosting software applications on a network of remote servers shared across multiple customers [1]. Generally speaking, there are two models of cloud computing: single-tenant and multi-tenant. In the single-tenant model, each customer buys a separate instance of a software application which runs on a logically isolated hardware environment. In the multi-tenant model, all customers use the same instance of the software and hardware infrastructure. In this case, the cloud provider must take care only to reveal part of an application appropriate to each customer. Whilst the single-tenant model is simpler both technically and conceptually, the multi-tenant model is preferred because it allows cloud providers to minimize resource requirements as the number of customers increases. By maintaining a single instance for multiple tenants, the provider can significantly reduce the costs of hardware provision, software licenses and software maintenance.

As has been noted elsewhere (e.g., [3]), cloud computing is an appealing application area for model-driven engineering (MDE). Cloud computing and MDE can be related in two ways [3]: (1) MDE for the Cloud, where MDE technologies are used to develop cloud applications; and (2) MDE in the Cloud, where the cloud is used to offer modeling technologies as a service (also referred to as Modeling as a Service or MaaS [3]). Multi-tenant applications bring additional challenges when compared to single-tenant applications: although a multi-tenant approach is financially advantageous for a cloud provider, maintaining separation of logic and data from different

clients is complex. In this paper, we argue that this complexity of multi-tenant applications makes them a good target for MDE.

This position paper presents five key research opportunities for applying ‘MDE for the Cloud’ specifically in the multi-tenant case: (1) MDE as a way to deal with “everywhere variability”; (2) Runtime modification of multi-tenant cloud applications; (3) A Domain-Specific Language (DSL) for the cloud; (4) MDE to support the economics of the cloud; and (5) MDE for enabling new business opportunities in the cloud.

2 “Everywhere Variability” in Multi-tenant Cloud Applications

Multi-tenant cloud applications exhibit a huge number of variation points at many different levels. As one example, data segmentation between tenants can be implemented in different ways: a dedicated database for each tenant, a single database for all tenants but with a separate schema per tenant, or a single database and schema/tables for all tenants [6]. As another example, each tenant may demand a customized business process workflow.

Indeed, the need for multiple tenants to share resources introduces potential variabilities at all levels of the cloud computing stack – that is, at the so-called Software, Platform, and Infrastructure-as-a-Service levels [1]. Different tenants may be offered a different hardware configuration, may have access to different APIs from the cloud provider, and will demand different configurations of an application. The “pay as you go” model of cloud computing actively encourages tenants to pay only for what they need; but this necessitates the maintenance of potentially many thousands of different configurations sharing the same underlying resources.

We categorize the different levels of variability in cloud systems as follows: (1) *Application Variability* – Software as a Service (SaaS) tenants have varying functional requirements [2]; (2) *Data Variability*; (3) *Business Process Variability* where the business workflow may vary; (4) *Platform Variability* – Operating systems, Programming languages, Frameworks, and solution stacks, i.e. Platform as a Service (PaaS); (5) *Provisioning Variability* – Hardware, Networking, Virtual Machines, and Elasticity, i.e. Infrastructure as a Service (IaaS); (6) *Deployment Variability* – Public, Private, Community, and Hybrid clouds [1]; (7) *Provider Variability*, e.g., Amazon Web Services (AWS), Google Application Engine (GAE), or Salesforce.

One of the potential strengths of MDE is the ability to abstract and manage variability; this has been demonstrated in many papers that integrate MDE and software product lines (SPLs) [8,11]. Hence, a combination of MDE and SPLs is an obvious potential solution. Schmid and Rummel [10] discussed how to exploit software product line (SPL) techniques for runtime customization of cloud-based systems. Similarly, a case for applying SPLs to cloud-based development is made in [11] where feature models are adapted to cater for specific cloud computing concerns like provisioning, customization, and price calculation. Schroeter et al. [12] propose to extend the component model (CCM) with tenant configuration and constraints. However, many of these works are at an early stage and there remain many challenges in applying MDE/SPLs to variability in cloud-based systems. A key point to note is that variabil-

ity management in the cloud is fundamentally different even than dynamic SPL approaches [10]: dynamic SPLs adapt a configuration at runtime but there is only a single instance of the product family running at any given time; in cloud-based systems, there will be multiple variations running, one for each tenant.

3 Runtime Modification of Multi-tenant Cloud Applications

A cornerstone of cloud computing is dynamic provisioning and resource allocation to achieve the desired performance and reliability as systems scale. This is especially true of multi-tenant applications where, for example, data access has to be carefully managed so that one tenant's requests do not dominate over others. In the worst case, a cloud application may have to be re-architected at runtime to, for example, move a greedy tenant to a separate server. In current practice, these runtime reconfigurations are handled by dashboards that allow performance monitoring and APIs that allow reconfigurations. However, MDE potentially supports more sophisticated dynamic reconfigurations resulting in radical architectural changes: these might be needed, for example, if an initial estimate on the number of tenants turns out to be wildly off, or if the security requirements of new tenants differ radically from the initial set of tenants.

There have been some attempts to manage dynamic cloud reconfigurations using advanced software development techniques. Jegadeesan and Balasubramanian [4] employ aspect-oriented programming to design service variations for multi-tenant SaaS systems. Almorsy et al. [5] apply MDE to generate security aspects for different tenants and then inject them into tenants' code. Abu-Matar et al. [8] show that MDE can play a central role in the runtime modification of multi-tenant systems by using a shared feature model to manage different tenant requirements at runtime.

Clearly, however, more research is needed in this area. For example, a model based runtime environment could be developed to manipulate tenant configurations by having a representation of the tenants' models at runtime. Another opportunity could be the development of a model-driven platform-as-a-service (PaaS) where multitenant applications can be deployed automatically.

4 Domain-Specific Languages (DSLs) for the Cloud

Multi-tenant cloud applications are specialist applications with their own set of concerns, such as partitioning for different tenants, extensibility to support new tenants, provisioning, testability of a single code base used by multiple tenants, and customization of a single code base for multiple tenants with different requirements [7]. In essence, these concerns define a domain of interest, which could be encapsulated into a DSL for generating and/or maintaining cloud implementations.

There has been some initial work in this direction. For example, CloudML [9] is a DSL to model the provisioning of multi-cloud applications, that is, applications that could run on multiple cloud providers. CloudML models the provisioning of these applications on IaaS clouds and it provides runtime support to deploy the modeled apps. Additionally, the DSL has an associated IDE, namely MODACloudML [9].

However, CloudML does not provide support for multi-tenant cloud software applications, nor it does provide support for tenants' applications evolution at runtime.

A DSL for multitenant cloud applications would include concepts like: Tenant, Tenant Configuration, Cloud Provider, Tenant Database, Tenant Schema, Tenant Table, and Deployment Type. Our previous work on SPL support for cloud architectures included a service oriented cloud meta-model that incorporates some of these concepts [8]. Transformation rules could then be used to model the evolution of multitenant applications like: Tenant Onboarding and Tenant Customization.

5 MDE to Support the Economics of the Cloud

Service providers using the cloud must think carefully about how they charge their customers. Since cloud applications typically run on a "pay as you go" model, service providers must decide if their customers will be charged on a subscription basis or per transaction. In both cases, the service provider must reconcile these charges with the amounts they pay to the cloud provider. Hence, the issue of pricing in multi-tenant cloud applications can be complex, especially when coupled with software maintenance costs which vary widely depending on the architecture chosen. MDE could be used in a novel way to manage these economics. For example, there could be a tool that lets service providers model their applications where the financials are evaluated on the model; hence, service providers can continuously refine their applications based on the predicted financials of the model.

Most current cloud pricing models are based on the usage and/or lease of virtual infrastructure resources where consumers pay for computing, storage, and network resources either on subscription or on-demand basis. For SaaS tenant-based consumption, pricing has to be based on which software features a tenant uses. This would necessitate that SaaS applications are built in a modular manner where features are associated with pricing units. Thus, MDE could be used to generate different metered versions of the same cloud application to suit consumers' needs. MDE could also be used as an enabler for self-service pricing decision support systems.

6 Enabling New Business Opportunities in the Cloud

Whittle et al. [14] argue that MDE is more likely to be a success if it enables new business opportunities rather than simply bringing productivity gains to existing business opportunities. There is a strong case that MDE can bring such new business opportunities to the cloud. In multi-tenant applications, customizations for each tenant are required; however, current approaches typically only allow somewhat simple customizations because of the software maintenance costs that would be incurred by allowing more complex customizations. For example Salesforce [13], the largest SaaS provider, handles tenants' customizations through metadata that allows customers to modify mainly the database.

However, cloud providers typically stop short of allowing radical customizations, such as wildly different business logic or architecture. This is because of the costly

maintenance and evolution costs that would be associated with this. MDE, however, potentially allows these maintenance costs to be reduced, hence enabling service providers to open up completely new business avenues by allowing tenants to add new business functionality that distinguish them from other tenants in contrast to mere database or simple interface customization.

7 Conclusion

Cloud computing is rapidly becoming the favored computing paradigm for the IT industry. New technological innovations are needed to make multi-tenant cloud computing a sustainable mainstream. To that end, we believe that MDE has the potential to create new disruptive opportunities for multi-tenant applications. In this paper, we have highlighted some of these opportunities and provided samples of ongoing research in this field.

8 References

1. P. Mell and T. Grance, "The NIST Definition of Cloud Computing." National Institute of Standards and Technology, Special Publication 800-145, Bethesda, Maryland, September 2011.
2. K. Schmid and A. Rummler, "Cloud-based software product lines," in Proceedings of the 16th International Software Product Line Conference, 164–170. New York, NY, 2012,
3. H. Brunelière, et al, "Combining Model-Driven Engineering and Cloud Computing," in 4th Workshop on Modeling, Design, and Analysis for the Service Cloud, Paris, France, June 2010.
4. H. Jegadeesan and S. Balasubramaniam, "A Method to Support Variability of Enterprise Services on the Cloud," IEEE International Conference on Cloud Computing, 117-124, 2009.
5. M. Almorsy, J. Grundy, and A. S. Ibrahim, "Adaptable, model-driven security engineering for SaaS cloud-based applications," Automated Software Eng. Journal, vol. 29, 2013, 1–38.
6. D. Betts et al., "Developing Multi-tenant Applications for the Cloud on Windows Azure," Microsoft Patterns & Practices, March 2013
7. S. Walraven, et al, "Efficient Customization of Multi-tenant Software-as-a-Service Applications with Service Lines", The Journal of Systems & Software, vol. 91, 48-62, 2014.
8. M. Abu-Matar, et al, "Towards Software Product Lines Based Cloud Architectures," Proceedings of the IEEE International Conference on Cloud Engineering (IC2E), 2013.
9. Nicolas Ferry, et al, "Towards Bridging the Gap Between Scalability and Elasticity," 4th International Conference on Cloud Computing and Services Science, 746-751, 2014.
10. K. Schmid and A. Rummler, "Cloud-based Software Product Lines," Proceedings of 16th International Software Product Line Conference (SPLC), 164-170, 2012.
11. E. Cavalcante, et al, "Exploiting Software Product Lines to Develop Cloud Computing Applications," the 16th International Software Product Line Conference, 179-187, 2012.
12. J. Schroeter, et al., "Towards Modeling a Variable Architecture for Multi-Tenant SaaS-Applications," VaMoS 2012, 111-120, ACM, New York, NY.
13. https://developer.salesforce.com/page/Multi_Tenant_Architecture, June 8, 2014.
14. J. Whittle, et al, "The State of Practice in Model-Driven Engineering." IEEE Software 31(3), 79-85, 2014.

Towards Cloud-Based Software Process Modelling and Enactment

Sami Alajrami¹, Alexander Romanovsky¹, Paul Watson¹, and Andreas Roth²

¹ School of Computing Science, Newcastle University, Newcastle upon Tyne, UK
{s.h.alajrami,alexander.romanovsky,paul.watson}@ncl.ac.uk

² SAP SE, Karlsruhe, Germany andreas.roth@sap.com

Abstract. Model Driven Engineering (MDE) considers models as a key artifact in software processes, and focus on the creation of models and transformations between them in order to (semi) automatically generate code. In this paper, we step back and consider the software process model itself as a key artifact that can be enacted and semi automated. We support our vision by proposing an architecture for a cloud-based software processes modelling and enactment environment which integrates software development tools and maintains repositories of modelling artifacts and the history of development.

Keywords: Software Process Modelling, Process Enactment, Software Engineering, Software Workflows, Cloud computing.

1 Introduction

As software systems are becoming more pervasive, the complexity of these systems has been increasing and the notion of systems of systems has been adopted. This complexity makes producing software systems a difficult task due to the increasing gap between the problem and the software implementation domains [13]. Model-Driven Engineering (MDE) was introduced as an approach to bridge this gap. MDE is driven by models which are used along with model transformation techniques to (semi) automate the code generation.

Models are used in most engineering domains to provide abstraction from the real world. In software systems, models are used for different purposes such as: documentation, testing, static analysis, and code generation. The use of models helps in representing the problem in a systematic way and displays the right amount of details for different perspectives and at different stages of development.

At the same time, the cloud computing paradigm has evolved to simplify organizational IT management and maintenance and cut both operational and expenditure costs. Cloud offers computing resources on demand using different service models (infrastructures, platforms, and software) and different deployment models (public, private, community, and hybrid) [19].

As the cloud is being widely adopted by both research and industry, researchers have started investigating the potential of using it for some software development phases (especially the computing intensive ones e.g. testing) [21], [5], [22]. In general, there are two perspectives to realize the potential collaboration between cloud and software engineering; (a) the use of cloud to support the software development process, (b) advancing software development methodologies to suite developing software for the cloud. The work presented in this paper fits in the first perspective.

MDE is centred around the creation of models and their relevant transformation techniques in order to automatically generate parts of models or code from other models. Typically, the focus is on modelling of individual phases of the software development process. In [18], authors developed an enactment environment for MDA processes, while authors in [4] investigated the potential of combining MDE and cloud, and proposed the notion of Modelling as a Service (MaaS). In this paper, we focus on the software development process models as a key software artifact, these models can be enacted and the non-interactive or repetitive tasks can be automated. We propose an architecture for software workflows enactment environment in the cloud.

The rest of the paper is structured as follows: a background discussing the use of cloud for software development and software process modelling is established in section 2. Section 3 describes the general architecture for our cloud-based software development platform. The paper concludes with a brief summary of future work.

2 Motivation and Background

2.1 Software Engineering in the Cloud

Today, Global Software Development (GSD) has become a popular development model where teams are distributed (sometimes across continents) and use different sets of tools to support and manage the development process. Developers have their own computers and need to have the tools that they need installed and configured. In addition, each team needs access to shared repositories and collaboration tools. The distribution in GSD brings multiple challenges to software development processes such as: restricted communication, less shared project awareness, and inconsistent builds [7]. Provisioning of software development environments in the cloud should prove beneficial as moving the development process to the cloud not only can reduce the amount of resources (time, money, and manpower) spent on the set up and configuration for each software project, but also can address some of the GSD challenges as shown in [14].

Cloud's accessibility facilitates distributed development by providing a shared development environment (artifacts and tools). Furthermore, cloud can bridge the gap between development and deployment environments. Having a virtually unlimited pool of resources in the cloud helps in allocating sufficient resources to certain heavy computing software development tasks (e.g. model checking or

testing). Eventually, using the cloud to support software development processes will help software teams to focus their efforts on the core problem rather than on setting up and maintaining development environments. There are some commercial cloud-based tools that support different phases of the software development process such as: IDEs (e.g. codenvy ³), testing (e.g. BlazeMeter⁴), issue tracking (e.g. JIRA ⁵). However, these tools are dedicated to support one or more phases of the software development process but not the entire process.

2.2 Software Process Modelling

Despite the current trend of embedding high level abstractions in programming languages to avoid code generation bottlenecks, Several approaches to MDE have been introduced: Model-Driven Architecture (MDA) ⁶, Model-Driven Software Development (MDS) [25], and Domain Specific Modelling (DSM) [17]. However, the focus has always been on modelling individual phases of the software development process rather than the process itself. Modelling software processes has been investigated since late 80s. There are many motivations which led these investigations including: a) improving the understanding for different perspectives, by visualizing the relevant components for each perspective. b) facilitating communication among team members, and c) supporting project management through reasoning in order to improve the process. Furthermore, the models can be partially automated (e.g. repetitive and non-interactive tasks). Several approaches for software process modelling have been introduced over time, they are categorized into four categories [3]:

1. Rules based (e.g. MARVEL [16])
2. Petri net based (e.g. SPADE [1])
3. Programming languages based (e.g. SPELL [9])
4. UML based (e.g. SPEM ⁷)

The first three did not receive industrial take up due to their complexity and inflexibility [15]. The UML approach was based on utilizing the wide adoption and acceptance of Unified Modelling Language (UML) for modelling software processes. Several implementations of this approach have been proposed each with different strengths and weaknesses. Authors in [3], compared six UML-based modelling approaches based on a set of software process modelling requirements. The authors also admit that executability and formality are major weaknesses of UML in the context of software process modelling.

Among the previous approaches, SPEM (Software Process Engineering Meta-model) has become an OMG standard for software process modelling. SPEM is based on the concept of interaction between *Roles* that perform *Activities* which

³ <https://www.codenvy.com/>

⁴ <http://www.blazemeter.com/>

⁵ <https://www.atlassian.com/software/jira>

⁶ <http://www.omg.org/mda/>

⁷ <http://www.omg.org/spec/SPEM/2.0/>

consume (and produce) *Work Products* [8]. However, a major criticism of SPEM in literature is its lack of support for process enactment. As a result, several researchers have proposed different approaches and extensions to support process enactment in SPEM. In [12], authors propose mapping rules to map SPEM models into XML Process Description Language (XPDL) which then can be enacted. In [23], authors propose xSPIDER_ML (a software process enactment language based on SPEM 2.0 concepts). Although xSPIDER_ML is supported with modelling tool and enactment environment, the notion of enactment is limited to process monitoring since developers are supposed to perform their tasks off-line and report their progress to the enactment environment. Authors in [10] introduce eSPEM which is a SPEM extension to allow describing fine-grained behaviour models that facilitate process enactment. They implement a distributed process execution environment [11] based on the FUML standard with emphasis on supporting the ability to share process state on different nodes, suspend and resume process execution, interact with humans, and adapt to different organizations. However, the notion of process enactment in that execution environment also assumes that developers carry out their tasks outside the execution environment and return control back to it once they finish.

Following an enforced formal process modelling can be useful in some cases (e.g. for certifying safety-critical systems). However, in practice, it can be restrictive for the creativity of team members. Organizations have been moving to agile methods to gain more dynamicity and to increase productivity. Therefore, we propose in the next section a less formal, more flexible and adhoc modelling notation than the previously mentioned approaches, with emphasis on the enactment of process models with support of an integrated tool set in the cloud.

3 Proposed Architecture for Cloud-Based Software Process Enactment

As mentioned in the previous section, SPEM lacks support for process enactment. In addition, neither the extensions that are offered by researchers for enactment support are standardized nor widely adopted outside academia. These approaches do not have a proper tool support and do not consider integrating software development tools within the enactment environment. The understanding of software process enactment in most of these approaches seemed to be limited to the concept of process simulation/monitoring. Although this notion of enactment can be useful for project management and monitoring, we think of enactment in a much broader way. Hence, we describe software process enactment as the process of performing software development activities by different actors within an environment that provides enactment support through the integration of development tools and automatic passing of control and data between activities. This means that unlike the approaches mentioned in the previous section, the entire development process execution takes place within one environment where tools and artifacts are available.

3.1 Software Engineering Workflows

Software engineering process can be described as a sequence of operations (*activities*) performed by development team members including customers and managers (*actors*) where activities produce *artifacts* which are used as inputs for other activities. This complies with The Workflow Management Coalition (WfMC) definition of workflow [24] as "the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules". Therefore software process can naturally be seen as a workflow. The idea of using workflow technology for software processes is not new, several researchers have investigated it [2], [20], [6].

Based on our description of enactment and software workflows, we propose an architecture to support software process enactment in the cloud. The benefits that software development can gain from the cloud has been discussed in section 2.1. In addition, provisioning of software development environments in the cloud with elastic resources will direct organization's resources towards solving the actual problem. Allowing third parties to integrate their tools makes it possible to try different tool vendors and different versions interchangeably without a huge adjustment effort. The workflow enactment environment passes the execution control between activities as prescribed in the process model. Non-interactive and repetitive tasks can be automated and benefit from the elasticity of the cloud (e.g. run a distributed model checker on two nodes initially and add more nodes if necessary). The artifacts generated from each activity can be accessed by team members (as it is stored on a repository in the cloud) and will be passed to the next relevant activity as an input. Often, software processes can be reused which adds another advantage for software development firms.

The three logical layers of the proposed architecture are illustrated in Figure 1. The top layer is for process modelling where a project manager or a software developer can create/edit models for either higher level abstract processes (e.g. project plan) or for daily tasks processes (e.g. implementation). The workflow management layer is where the enactment of the processes takes place while the cloud management layer handles the underlying cloud infrastructure issues (e.g. QoS and multiple cloud providers/models).

The process model contains a description of the activities involved in the process and the data and control flow information which guides the process enactment. Activities are performed by human actors and they are categorized in two types: concrete and abstract. Concrete activities can be either local or external. The local activities can be either a self-contained executable code or interactive (to input decisions or data), while the external activities are web services maintained by third parties. High level abstraction can be provided by abstract activities which are non executable activities and by default will be representing a sub-process. Activities are available in activities pool and can be either created by third party or by the development team. In general, each activity has zero or more input ports and zero or more output ports. The type of input and output artifacts that a port can accept/generate is defined at the

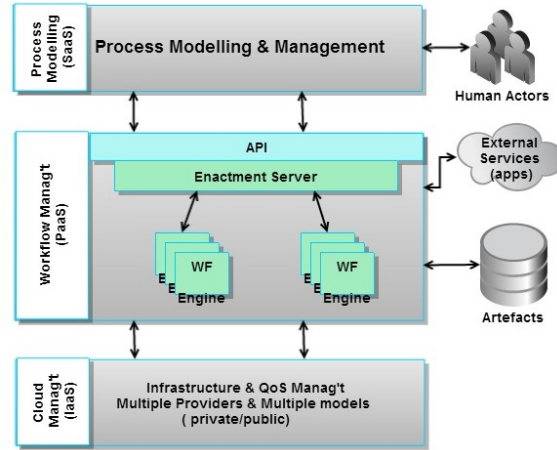


Fig. 1. Proposed architecture to support SW process enactment in the cloud

time of creation of the activity. This guarantees that activities can only be linked to each other when their input/output artifacts are compliant. Activities are executed independently provided that the input needed for them to execute is available. This decoupling allows distributing the execution of the process across several workflow engines (deployed potentially on different virtual machines on the cloud). Activities have configurable parameters to control how the activity will be deployed and executed on the cloud.

3.2 Process Modelling and Definition

Software processes are dynamic and unpredictable. In addition, organizations tailor process models such as waterfall or spiral differently to meet their needs. Hence, a flexible modelling notation is required. This notation needs to be (a) expressive (to express the process and its cloud execution settings), (b) executable, and (c) understandable and easy to use. The notation needs to support a combination of on the fly creation/modification of activities for the purposes of capturing the short term/everyday development and of the longer term activities at the organizational level. Based on that we defined the basic constructs for the software development process model. The process will be represented using a simple graphical notation that can be translated to XML which will then be used to enact the process. The graphical notation can be useful for understanding the process and training new team members. An XML schema has been defined to map the semantics of the graphical notation. These constructs are described in table 1.

Each of the activities can be configured to specify how it will be executed; parameters include (but not limited to): the specified tool support, the cloud








Element Name	Description	Graphical Notation
Abstract Activity	An abstract activity does not execute anything itself, but represents a high level abstraction of one or more activities. Often, it will represent a sub-process.	
Local Activity	A local activity can be an executable code block or a tool that is deployed within the enactment service.	
External Activity (web service)	A tool or service that is deployed and maintained outside the enactment service.	
Interactive Activity	An activity that involves an interaction point where the human actor is asked to provide some input data (e.g. configuration parameters).	
Decision Point	An interaction point where the human actor is asked to decide what to do next.	
Data flow Dependency	A link between two activities A and B, where B cannot start before A provides an input to it.	
Control flow Dependency	A link between two activities A and B, where none of the two is depending on the other. The link here just represents the order of occurrence.	

Table 1. Basic Software Process Modelling Elements

execution requirement (e.g. on public/private cloud), and accepted and generated artifacts. For the sake of simplicity, the notation does not explicitly support modelling of actors at this stage.

Process Examples: Agile methods are widely adopted in industry as they increase the throughput. SCRUM is one of the agile methods which defines a project management framework. This framework defines a set of roles and a set of meetings with different purposes, attendees, and frequencies. Figure 2 illustrates the high level representation of a scrum sprint. This abstraction can be useful from a management perspective. However, it does not specify any details of how developers are going to implement the process. In reality, most software developers use an IDE, an issue tracking tool (e.g. JIRA), a continuous integration framework (e.g. JENKINS), and a version control system (e.g. GIT). These tools are used on daily basis to write, test, store, and integrate code. A model of the daily development process (representing the implementation sub-process) using the notation defined in this subsection is illustrated in Figure 3. The control flow dependency between the "edit issue tracking" and "edit code" activities sets the order of execution, however, since no data dependency is included here, it allows us to perform any of the activities independently. The decision point allows to create a loop based on the decision of the software developer whether to commit his code or to change it or even to edit the issue tracking. Another example is the parallel model checking process (Figure 4) where the model checking activity can be deployed on multiple nodes to utilize the cloud elasticity in order to improve the model checking performance.

3.3 Workflow (Process) Enactment Service

Once the process is defined using the graphical notation described in the previous subsection (which is translated to XML), it will be validated against the process definition schema to make sure the XML file is valid. Next the enactment service should interpret the XML representation of the process and schedule the

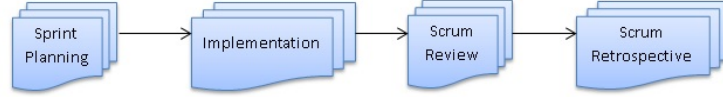


Fig. 2. Scrum high level abstraction

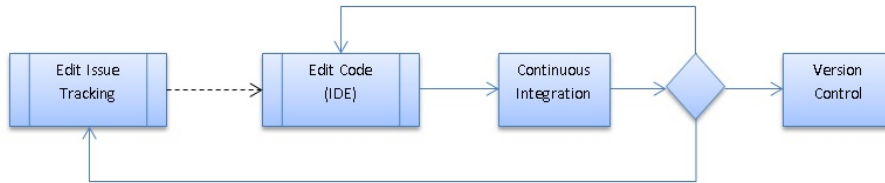


Fig. 3. Daily technical task by a scrum developer

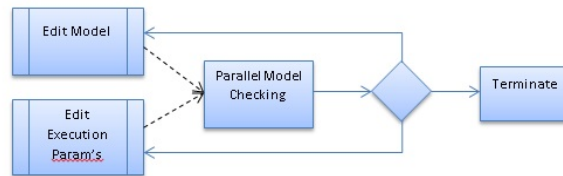


Fig. 4. Parallel model checking process

execution on as many distributed workflow engines as required. The enactment service consists of an enactment server and multiple workflow engines. The service itself is provided as a web service which accepts requests from any type of clients (desktop, web, mobile/tablet). The enactment service consists of:

- Enactment Server: responsible for managing workflow engines and provide smart scheduling algorithms to optimize cost and performance when the workflow execution is distributed across multiple clouds (if necessary). It is also responsible for monitoring the workflow execution and handling exceptions.
- Workflow Engines: responsible for loading the needed tools and artifacts for executing an activity. It also reports back the execution progress to the enactment server.
- Enactment Service API: provides a standard access to the enactment service.

Using the cloud for executing workflows requires addressing certain issues, such as: portability and QoS of the cloud resources. Authors in [7] identified seven needed quality attributes for a cloud infrastructure to provide tools as a service. These attributes will be used as a guidance for the enactment service implementation.

4 Conclusion and Future Work

In this paper, we proposed a cloud-based software process modelling and enactment environment which harnesses both of cloud and workflows benefits. We considered software process model as a main artifact in the software development process. The process model can be (partially) automated and supported by development tools which are integrated within the enactment environment. We proposed the core of a simple modelling notation for modelling different parts of the software process. Currently, a prototype of the enactment service is being implemented to run on a single cloud initially which will be extended to run on different clouds later. The future work includes: assessing the modelling notation after applying it to more case studies, applying provenance to provide reasoning about the software process, and investigating possible support for interoperability between different tools.

References

1. Bandinelli, S., Fuggetta, A., Ghezzi, C.: Software process model evolution in the spade environment. *Software Engineering, IEEE Transactions on* 19(12), 1128–1144 (1993)
2. Barnes, A., Gray, J.: Cots, workflow, and software process management: an exploration of software engineering tool development. In: *Software Engineering Conference, 2000. Proceedings. 2000 Australian*. pp. 221–232 (2000)
3. Bendraou, R., Jezequel, J., Gervais, M.P., Blanc, X.: A comparison of six uml-based languages for software process modeling. *Software Engineering, IEEE Transactions on* 36(5), 662–675 (Sept 2010)
4. Brunelière, H., Cabot, J., Jouault, F.: Combining Model-Driven Engineering and Cloud Computing. In: *Modeling, Design, and Analysis for the Service Cloud - MDA4ServiceCloud'10: Workshop's 4th edition (co-located with the 6th European Conference on Modelling Foundations and Applications - ECMFA 2010)* (2010)
5. Bucur, S., Ureche, V., Zamfir, C., Candea, G.: Parallel symbolic execution for automated real-world software testing. In: *Proceedings of the Sixth Conference on Computer Systems*. pp. 183–198. *EuroSys '11, ACM* (2011)
6. Chan, D., Leung, K.: Software development as a workflow process. In: *Software Engineering Conference, 1997. Asia Pacific ... and International Computer Science Conference 1997. APSEC '97 and ICSC '97. Proceedings*. pp. 282–291 (1997)
7. Chauhan, M.A., Babar, M.A.: Cloud infrastructure for providing tools as a service: Quality attributes and potential solutions. In: *Proceedings of the WICSA/ECSA 2012 Companion Volume*. pp. 5–13. *WICSA/ECSA '12* (2012)
8. Combemale, B., Crgut, X., Caplain, A., Coulette, B.: Towards a rigorous process modeling with spem. In: *ICEIS (3)*. pp. 530–533 (2006)
9. Conradi, R., Jaccheri, M.L., Mazzi, C., Nguyen, M.N., Aarsten, A.: Design, use and implementation of spell, a language for software process modelling and evolution. In: *Proceedings of the Second European Workshop on Software Process Technology*. pp. 167–177. *EWSPPT '92* (1992)
10. Ellner, R., Al-Hilank, S., Drexler, J., Jung, M., Kips, D., Philippsen, M.: espem a spem extension for enactable behavior modeling. In: *Modelling Foundations and Applications, Lecture Notes in Computer Science*, vol. 6138, pp. 116–131 (2010)

11. Ellner, R., Al-Hilank, S., Drexler, J., Jung, M., Kips, D., Philippsen, M.: A fuml-based distributed execution machine for enacting software process models. In: Modelling Foundations and Applications, Lecture Notes in Computer Science, vol. 6698, pp. 19–34. Springer Berlin Heidelberg (2011)
12. Feng, Y., Mingshu, L., Zhigang, W.: Spem2xpdl: Towards spem model enactment
13. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: 2007 Future of Software Engineering. pp. 37–54. FOSE '07, IEEE Computer Society, Washington, DC, USA (2007)
14. Hashmi, S., Clerc, V., Razavian, M., Manteli, C., Tamburri, D., Lago, P., Di Nitto, E., Richardson, I.: Using the cloud to facilitate global software development challenges. In: Global Software Engineering Workshop (ICGSEW), 2011 Sixth IEEE International Conference on. pp. 70–77 (2011)
15. Henderson-Sellers, B., Gonzalez-Perez, C.: A comparison of four process meta-models and the creation of a new generic standard. *Information and Software Technology* 47(1), 49 – 65 (2005)
16. Kaiser, G., Barghouti, N., Sokolsky, M.: Preliminary experience with process modeling in the marvel software development environment kernel. In: System Sciences, 1990., Proceedings of the Twenty-Third Annual Hawaii International Conference on. vol. ii, pp. 131–140 vol.2 (1990)
17. Kelly, S., Tolvanen, J.P.: Domain-specific modeling: enabling full code generation. John Wiley & Sons (2008)
18. Maciel, R., da Silva, B., Magalhaes, P., Rosa, N.: An integrated approach for model driven process modeling and enactment. In: Software Engineering, 2009. SBES '09. XXIII Brazilian Symposium on. pp. 104–114 (Oct 2009)
19. Mell, P., Grance, T.: The nist definition of cloud computing. *National Institute of Standards and Technology* 53(6), 50 (2009)
20. Oberweis, A.: Workflow management in software engineering projects. In: Proceedings of the 2nd International Conference on Concurrent Engineering and Electronic Design Automation. pp. 55–60 (1994)
21. Oriol, M., Ullah, F.: Yeti on the cloud. In: Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on. pp. 434–437
22. Pakhira, A., Andras, P.: Leveraging the Cloud for Large-Scale Software Testing A Case Study: Google Chrome on Amazon, chap. Hershey, PA, USA, pp. 252–279. IGI Global (2013)
23. Portela, C., Vasconcelos, A., Silva, A., Silva, E., Gomes, M., Ronny, M., Lira, W., Oliveira, S.: xspider_ml: Proposal of a software processes enactment language compliant with spem 2.0. *Journal of Software Engineering and Applications* 5(6), 375 – 384 (2012)
24. Specification, W.M.C.: Workflow Management Coalition, Terminology & Glossary (Document No. WFMC-TC-1011). Workflow Management Coalition Specification (1999), <http://www.wfmc.org/Download-document/WFMC-TC-1011-Ver-3-Terminology-and-Glossary-English.html>
25. Voelter, M., Groher, I.: Product line implementation using aspect-oriented and model-driven software development. In: Software Product Line Conference, 2007. SPLC 2007. 11th International. pp. 233–242. IEEE (2007)

Towards Pattern-Based Optimization of Cloud Applications^{*}

Martin Fleck, Javier Troya, Philip Langer, and Manuel Wimmer

Vienna University of Technology, Business Informatics Group, Austria
{lastname}@big.tuwien.ac.at

Abstract. With the promise of seemingly unlimited resources and the flexible pay-as-you-go business model, more and more applications are moving to the cloud. However, to fully utilize the features offered by cloud providers, the existing applications need to be adapted accordingly. To support the developer in this task, different cloud computing patterns have been proposed. Nevertheless, selecting the most appropriate patterns and their configuration is still a major challenge. This is further complicated by the costs usually associated with deploying and testing an application in the cloud.

In this paper, we encode the pattern selection problem as a model-based optimization problem to automatically compute good solutions of configured pattern applications. Particularly, we propose a two-phased approach, which is guided by user-defined constraints on the non-functional properties of the application. In the first phase, a preliminary set of promising solutions is computed using a genetic algorithm. In the second phase, this set of solutions is evaluated in more detail using model simulation. We demonstrate the proposed approach and show its feasibility by an initial case study.

Keywords: Cloud Computing, Goal Modeling, Model Simulation, Genetic Algorithm, Cloud Computing Patterns

1 Introduction

The seemingly unlimited resource offerings and the flexible pay-as-you-go business model are, amongst others, the main driver of the adoption of the cloud computing paradigm. As a result, many different cloud providers have emerged. This has also sparked a major interest in the migration of existing applications to the cloud [17]. Besides the cloud provider selection, adapting the application to make the best out of the cloud provider offerings is often very challenging. Cloud computing patterns [6, 13, 18] have been introduced as cloud provider-independent solutions to reoccurring problems in cloud computing. Developers can use these patterns in their design decisions and operationalize them in the context of a specific cloud provider. This step, however, requires detailed insight of the software architecture, the cloud computing paradigm, the offerings of specific cloud providers, and the usage of the given application. Furthermore, the developers have to deal with a possibly infinite search space of pattern applications and a solution has to satisfy multiple, probably conflicting, objectives [11].

^{*} This work is co-funded by the European Commission under the ICT Policy Support Programme, grant no. 317859.

In this paper, we present a model-based approach aimed to support developers in selecting the most appropriate cloud patterns and their configurations. Particularly, the approach consists of two phases and is guided by user-defined constraints on the non-functional properties of the application. In the first phase, a preliminary set of promising solutions is computed using a multi-objective genetic algorithm which uses estimates to determine the fitness of a solution due to the huge search space. In the second phase, this set of solutions is evaluated in more detail using model simulation to better support the final decision by the user, i.e., selecting the most appropriate solution.

The rest of the paper is organized as follows. In Section 2, we describe our proposed approach as well as the necessary input from the stakeholders. Section 3 showcases the applicability of our approach in a case study, while Section 4 discusses related work. The paper concludes in Section 5 with an outlook on future work.

2 Approach

The central aim of our approach is to find a configuration of patterns that best satisfies the needs of the application stakeholder, i.e., the reason why the application is moved to the cloud in the first place. We therefore provide the stakeholder with a goal modeling language that is capable to express these needs in terms of non-functional properties (NFPs). Based on these goals, we approach the pattern selection problem with two subsequent steps, as shown in Figure 1. In the first step, a multi-objective evolutionary algorithm is used to calculate a preliminary set of good solutions. A solution is a set of configured cloud optimization patterns and evaluated based on estimates on how certain patterns impact properties of the applications. In the second step, each solution returned by the evolutionary algorithm is additionally ranked based on the more detailed analysis performed by model simulation. The resulting ranked set of solutions together with their approximated success to fulfill the goals is then presented to the stakeholders.

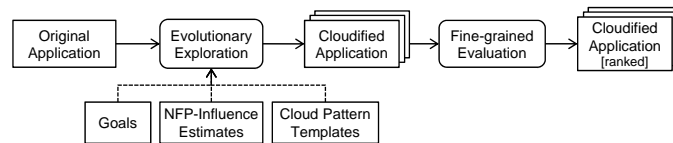


Fig. 1. Approach Overview

2.1 Goal Modeling

Goal modeling originally stems from early phases of requirements engineering, where a goal is an objective for the system from the perspective of a stakeholder. In the goal modeling language we provide, the goals are based on a set of (non-functional) properties. More concretely, a goal defines a target value or target range for a specific property in the context of the software application, e.g., the response time of a request or the utilization of a specific component. These target values must be set in the range of the property under consideration, e.g., utilization can only take floating point values between zero and one. Each goal must be set into the context of a specific workload or usage scenario, as it is not feasible to show that a goal holds in all possible cases. Furthermore, the importance of a goal is given by a numeric priority, whereby a smaller

Caching	Horizontal Scaling	Auto-Scaling													
<i>Problem:</i> The same entities are retrieved multiple times from the database.	<i>Problem:</i> Not all day-to-day user requests can be handled due to a lack of resources.	<i>Problem:</i> Not all user requests can be handled due to a lack of resources. However, the resource demand changes often resulting in low times and high peaks.	<i>Template:</i>												
<i>Effect:</i> The frequently-accessed entities are stored in a Cache, improving the retrieval of data (reads).	<i>Effect:</i> Deploy multiple instances of one node/service to provide more resources.														
<i>Template:</i>	<i>Template:</i>	<i>Effect:</i> Start with a certain number of nodes and dynamically adjust the number depending on certain monitored properties, thus providing more resources only if necessary.													
<table border="1"> <thead> <tr> <th>Cache</th> </tr> </thead> <tbody> <tr> <td>Application: Entity</td> </tr> </tbody> </table>	Cache	Application: Entity	<table border="1"> <thead> <tr> <th>Horizontal Scaling</th> </tr> </thead> <tbody> <tr> <td>Application: Service</td> </tr> <tr> <td>NrInstances: Int[2, ∞]</td> </tr> </tbody> </table>	Horizontal Scaling	Application: Service	NrInstances: Int[2, ∞]	<table border="1"> <thead> <tr> <th>Auto-Scaling</th> </tr> </thead> <tbody> <tr> <td>Application: Service</td> </tr> <tr> <td>MinInstances: Int[1, ∞]</td> </tr> <tr> <td>MaxInstances: Int[1, ∞]</td> </tr> <tr> <td>ScalingVariable: Variable[*]</td> </tr> <tr> <td>ScaleInThreshold: Real[-∞, ∞]</td> </tr> <tr> <td>ScaleOutThreshold: Real[-∞, ∞]</td> </tr> </tbody> </table>	Auto-Scaling	Application: Service	MinInstances: Int[1, ∞]	MaxInstances: Int[1, ∞]	ScalingVariable: Variable[*]	ScaleInThreshold: Real[-∞, ∞]	ScaleOutThreshold: Real[-∞, ∞]	
Cache															
Application: Entity															
Horizontal Scaling															
Application: Service															
NrInstances: Int[2, ∞]															
Auto-Scaling															
Application: Service															
MinInstances: Int[1, ∞]															
MaxInstances: Int[1, ∞]															
ScalingVariable: Variable[*]															
ScaleInThreshold: Real[-∞, ∞]															
ScaleOutThreshold: Real[-∞, ∞]															

Fig. 2. Example Patterns and their Pattern Templates in a UML class-like notation

number indicates a higher priority. Summarizing, we consider goals to be Boolean conditions concerning NFPs in the context of a software system under a specific workload with a user-defined priority.

Example: The most important objective (priority 1) is that the average response time of a log in-request is less than 2 seconds when ten users log in at the same time.

2.2 Cloud Computing Patterns

Cloud computing patterns provide a generic solution to a reoccurring problem in a specific context in the cloud computing domain and need to be concretized by the developer when used. In the ARTIST project [1] we have collected over 30 of these cloud computing patterns from different sources [6, 13, 18]. In this work we focus on patterns that are applied in order to *optimize* the properties of an application that is to be deployed on the cloud. We therefore assume that the base architecture of the application is already suitable for the cloud and no major architectural refactorings need to be done. To use the informally described patterns in our approach, we translate them into so-called *pattern templates*, which specify where the pattern can be applied and how it can be configured. Figure 2 shows a small excerpt of the collected patterns and the resulting pattern templates.

Caching can be applied on any entity class that is persisted in a datastore, while scaling can be applied on any service class. In horizontal scaling, the number of instances of a service is fixed from the beginning and can range from two instances to a theoretically unlimited number of instances – in practice this number is limited by the specific cloud provider. By contrast, auto-scaling provides a lower and upper bound on the number of instances, and the actual number is adapted during the application runtime based on the value of the *ScalingVariable* and the two variable-specific scaling thresholds. If the value of the variable is less or equal than the specified *ScaleInThreshold*, one service instance is removed; if the variable value is greater or equal than the *ScaleOutThreshold*, an additional instance is created. Any numerical variable which can be evaluated during runtime can serve as auto-scaling variable, e.g., utilization.

When applying a cloud computing pattern in a concrete use case, we create an instance of the respective pattern template, i.e., we provide concrete values for all the parameters defined in the template. The set of the concrete values for a pattern is called a *pattern configuration*. Each applied pattern configuration has an impact on the (non-functional) properties of the system. This impact is usually specific to the software system. Estimations about the gained impact on the properties may be gained from more detailed pattern descriptions, experience, and cloud benchmarking services. An example can be found in Table 1.

2.3 Evolutionary Algorithms

The aim of our approach is to select a sequence of pattern applications that satisfies the goals modeled by the stakeholder. The pattern selection problem consists of a possibly infinite search space of configurations and a solution has to satisfy multiple, probably conflicting, objectives [11]. We therefore categorize our problem as a multi-objective combinatorial optimization (MOCO) problem, for which several methods have been discussed in the literature (cf. [5]). For our approach, we choose an evolutionary algorithm for the pattern selection problem, namely the nondominated sorting genetic algorithm II (NSGA-II) [4], guided by the estimated impact of a pattern on the NFPs.

Search Space. The search space consists of all possible patterns configurations as defined by the pattern templates and may be infinite, e.g., when considering floating point values. Therefore it is not possible to produce the complete search space in advance, but rather generate new random configurations based on the templates, if necessary.

Solution Space. A genetic algorithm maintains a set of solutions, called a *population*, and deploys selection, re-combination, and mutation operators to improve the quality of the solutions in the population in each iteration. In our approach, a (candidate) solution is a selected sequence of pattern configurations. To ensure the validity of candidate solutions, *solution constraints* requiring domain knowledge about the different patterns can be used to specify how configurations can be combined. As an example, it makes no sense to apply both, horizontal scaling and auto-scaling, on the same service, thus a constraint classifying such a solution as invalid may be specified. One drawback when using NSGA-II is that the length of the solution (n) must be fixed in advance, i.e., the number of pattern configurations appearing in a solution. To allow the calculation of solutions with less or equal than n pattern configurations, we introduce a pattern configuration *placeholder*, which may take one or more places in the solution, but has no influence on any of the NFPs.

Objective Space. To evaluate the quality (*fitness*) of a solution, the solution needs to be mapped to the objective space. In multi-objective optimization, this objective space consists of multiple dimensions, each dimension referring to one objective. Usually these objectives are competing, so that no single point in the objective space exists that dominates all other points, resulting in a set of optimal solutions. In our approach, the objective space is not pre-defined, but specified by the stakeholder implicitly by defining the goals. Each property that has a goal specified upon is one dimension in the objective space that needs to be evaluated. The evaluation of a solution candidate for each of these dimensions in the objective space is done by a so-called *fitness function*. This fitness function guides the algorithm into good areas of the solution space.

Fitness. We define the fitness of a solution in a specific dimension to be the sum of the weighted, relative distance between the property value resulting from applying the solution and the target value or target range set by the user for each goal of this property. The relative distance of a goal is the difference between the resulting property value and the user-defined target value or target range in relation to the target value or range. For target ranges, the mean of the range is taken as target value, however a fulfilled goal always results in a relative distance of zero. An additional penalty (weight) for each goal that has not been achieved is calculated by multiplying the relative distance with the proportional goal priority, resulting in a higher penalty for higher priority goals. The goal of the algorithm is to find a solution that minimizes the fitness values.

2.4 Model Simulation

Running NSGA-II gives us a set of solutions which form the Pareto front from the previously infinite solution space. These solutions can be evaluated in more detail using the more execution expensive, but also more precise, model simulation. For this, we build on our previous work [16] that is based on graph transformations supported by the *e-Motions* framework [15]. By using *e-Motions*, we run the modeled system and perform a more detailed evaluation also considering additional properties such as the contention of resources. The results from the model simulation are used to rank the solution set calculated by NSGA-II. The ranked solution set together with the approximate success of each solution to fulfill the goals is then presented to the stakeholders for the final decision about which configurations of the cloud computing patterns should be applied.

3 The Petstore Case Study

In this paper, we show the applicability and feasibility of our approach based on the Petstore case study. The case study is executed with the Java prototype we have developed using the NSGA-II implementation provided by the MOEA Framework¹. The Petstore is a small web application allowing potential customers to create an account, log into this account, and order pets from a pre-defined pet catalogue. Previously the Petstore has been running on the local web server of the company, however now the company wants to move the Petstore application to the cloud to improve scalability and reduce cost. The Petstore architecture is realized with three entity classes and five services.

Entity Classes. The Petstore application maintains three entity classes, namely *Item*, *Customer*, and *Order*. All products in the Petstore are stored in the form of an item entity. Customers can create an account at the Petstore, log in, search for items, and place orders. An order consists of the registered contact information of a customer as well as the items and the quantity the customer has put into the shopping cart. Available service functionality is depicted in Figure 3, as explained later.

Services. Internally the Petstore uses different services to provide the necessary functionality to customers. The *Application Service* is the only service that a customer directly interacts with. It uses the *Customer Service*, *Catalog Service* and *Order Service* to handle the customer data, item data, and order data, respectively. All of these three services use the *Entity Service* to handle the persistence and the retrieval of data from a permanent data store.

3.1 Setup

Patterns. For this case study, we select the three patterns already introduced in Section 2.2: *Caching*, *Horizontal Scaling*, and *Auto-Scaling*. Considering the application conditions, caching can be applied on any of the three entity classes, while scaling can be applied on any of the five service classes. We assume that both scaling patterns improve performance (the more instances, the faster they process data) and worsen cost (each instance is billed by the cloud provider). Estimations about the gained speedup or utilization can be partially retrieved from a more detailed pattern description, but can also be gained from experience or dedicated cloud benchmarking services. Pricing information can be gathered from the website of the specific cloud provider. The resulting estimated impact for each pattern is summarized in Table 1.

¹ MOEA Framework, Version 2.1: <http://moeaframework.org/>

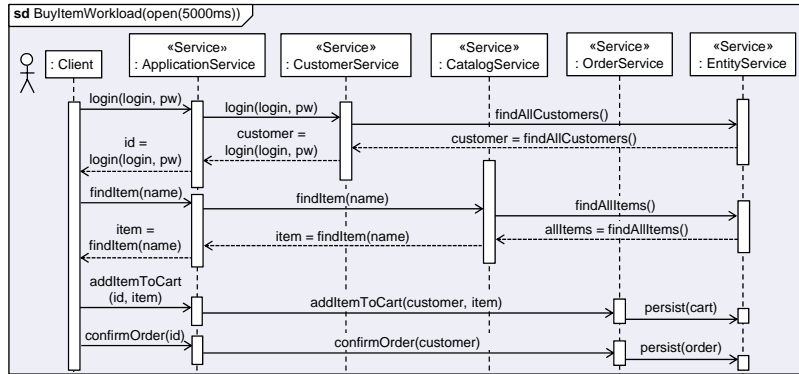


Fig. 3. The Petstore Scenario Workload

Furthermore, we define application constraints to guarantee that a pattern is not applied on the same entity or service multiple times and that the two scaling patterns are not applied on the same service at the same time.

Table 1. Estimated impact of considered patterns

Caching Impact				Scaling Impact	
Price per TimeUnit	SpeedUp Item	SpeedUp Customer	SpeedUp Order	Price per TimeUnit and service instances	SpeedUp n Instances
0.0015	5	3	1	0.0010	n

Goals. As mentioned in the previous section, all goals must be set in the context of a specific workload, as it is not feasible to show that certain goals hold in all possible use cases. For this case study, we consider a scenario where a single user connects to the Petstore application, logs into his or her account, searches for a specific item by name and then places an order on this item. Ten requests arrive in the application (modeling the connection of ten users) with an exponential distribution of five seconds (5000 time units). The scenario is summarized in Figure 3 with a sequence diagram. The main reason for moving the Petstore application to the cloud is to reduce cost and improve scalability, or more precisely, to reduce the overall *cost* and improve the *response time* of customer requests and the *utilization* of different services. Cost and response time are both properties which can have values in the range of $[0.0, \infty]$, with a lower value being considered better than a higher value. Utilization has a value range of $[0.0, 1.0]$ with neither lower values nor higher values being clearly better, making utilization suitable for a target range instead of a single target value. Too low utilization can suggest an idle resource, which produces cost and brings no benefit. Too high utilization can indicate an overloaded resource, resulting in a slower performance or a situation where consumers of the application are not served.

In this case study we assume that the following goals should be fulfilled within the context of the Petstore scenario. The application of a property is indicated by the property name and the applied element in parenthesis, an asterisk (*) marks the whole application. The priority of a goal is given in square brackets after the condition.

- Goal 1:** $Cost(*) \leq 900$ [3]
- Goal 2:** $ResponseTimePerRequest(*) \leq 30000$ [2]
- Goal 3:** $0.15 \leq Utilization(EntityService) \leq 0.25$ [1]
- Goal 4:** $0.15 \leq Utilization(CustomerService) \leq 0.25$ [3]

NSGA-II Configuration. As mentioned in the previous section, genetic algorithms use selection, re-combination, and mutation operators to evolve the population into a good area of the solution and thus objective space. For selecting candidate solutions, we use a so-called *tournament selection strategy*, which takes n random candidate solutions from the population and allows the best one to be considered for re-combination (in our case, $n = 4$). Two candidate solutions are re-combined into two new candidate solutions by means of a single point crossover operator. This operator splits each solution at a random point into two parts and merges the first part of the first solution with the second part of the second solution and vice versa. After re-combination the validity of the resulting solutions is checked and mutation can take place. Invalid solutions are given the worst possible fitness and should eventually be removed from the population. Mutation occurs at a low rate (1.5%) in a solution and changes one of the pattern configurations concrete values slightly. In our case, this means that each parameter of a pattern configuration has a slight chance of being modified, e.g., the number of instances for horizontal scaling. Furthermore, we define a solution length of eight, as there are only eight classes on which at most one pattern can be applied. The algorithm should maintain 200 solutions per population and continue for at most 1000 iterations.

Fitness Function. To evaluate the quality of the solutions produced by the NSGA-II, we need to provide values for *response time*, *cost* and *utilization* by incorporating the impact estimations. As the fitness function is executed many times, we use a very simple model analysis technique, which may not be very precise, but is very fast to execute. First, we retrieve the configured number of instances for each of the services. Then we execute the scenario for all requests and services and calculate the runtime of each service by summing up the reduced execution times (original execution time divided by number of instances) of each operation call that has been made to this service during the execution. The sum of all operation executions is the total runtime of the application. Each request is seen as independent and no contention of resources is considered. Based on the runtime, we calculate both the utilization and the cost for each service using the provided pricing and speedup information. The resulting response time for each request is the total runtime divided by the number of requests.

3.2 Results

After running the NSGA-II algorithm, we are faced with 3 solutions, one of which is depicted in Figure 4. On this set of solutions, we run the model simulation as described in Section 2.4 to gain more detailed information about how close the solutions are to fulfilling the goals set by the user. For this, we need to define a metamodel and behavioral in-place rules that model the system at runtime. For each solution, an instance of this metamodel containing the applied patterns must be created and executed. The result of the model simulation is shown in Table 2. The first line presents the original configuration (no patterns applied), while the other three have some patterns applied. The left-hand side of the table shows the values for the NFPs of interest, while the middle part shows the distance to each goal, and the right-hand side displays the overall distance to the goals and the rank of the solutions.

Regarding the solutions, (1) and (2) use four patterns, while (3) uses three. Solution (1) auto-scales the *Entity Service* and the *Application Service* depending on the queue length. The first service ranges between 3 and 7 instances, while the second one does

«Entity» «Cache» Item	«Service» «HorizontalScaling» { NrInstances = 4 } EntityService	«Service» «AutoScaling» { MinInstances = 2, MaxInstances = 4, ScalingVariable = QueueLength, ScaleInThreshold = 3, ScaleOutThreshold = 7 } CustomerService	«Placeholder»	...
-----------------------------	--	---	---------------	-----

Fig. 4. Solution (3) with pattern configurations and placeholders

between 1 and 4. Solution (1) also has horizontal scaling for *Customer Service* and *Order Service*, with two instances for each one. Solution (2) auto-scales the *Order Service* depending on the queue length between 1 and 4 instances, and it also applies horizontal scaling in the *Entity Service* and *Customer Service*, with 4 and 3 instances, respectively. Caching on *Item* is applied as well. Finally, Solution (3), also depicted in Figure 4, applies caching on *Item* and horizontal scaling for *Entity Service* with 4 instances. It auto-scales the *Customer Service* between 2 and 4 instances depending on the queue length.

Table 2. The Petstore Scenario Workload Results

#	Cost	RespT	Util ES	Util CS	G1	G2	G3	G4	Sum	Rank
B	921	134500	0.95	0.217	0.024	5.225	7.5	0	12.749	-
(1)	935	29018	0.176	0.257	0.039	0	0	0.284	0.322	3
(2)	992	31698	0.215	0.168	0.102	0.085	0	0	0.187	1
(3)	948	32845	0.243	0.187	0.053	0.142	0	0	0.196	2

While we have a clear ranking according to the model simulation and the calculated distances, we still provide the user with all possible solutions and their detailed evaluation values to allow additional human reasoning. A user could still decide to apply solution (1) instead of the other solutions if she wanted the utilization of the *Entity Service* to be closer to the smallest target value or she could also decide to apply solution (3) instead of solution (2), because cost may still be the driving factor of the migration. Despite the ranking, we can note that none of the solutions is surprising and they probably could have been found by an expert using the estimated impact on the patterns and the knowledge about the system execution. However, we assume that with a more complex application and a higher number of goals and/or patterns, the manual derivation of solutions becomes harder.

4 Related Work

In software engineering, patterns are important ingredients to document knowledge on how to solve recurring problems since the well-known book by the Gang of Four [9] describing patterns in the context of object-oriented design. With the appearance of the cloud computing paradigm, the community has already started working on cloud computing patterns [6, 13, 18]. For our approach, we studied different pattern descriptions, created pattern templates, and estimated the effect of each pattern on the different NFPs.

Optimization techniques are used to solve a variety of different problems [3]. Research in metaheuristics for combinatorial optimization problems aims to optimize the techniques applied in evolutionary algorithms [19]. At the same time, the focus of research has shifted from being rather algorithm-oriented to being more problem-oriented [2]. This is also reflected in the emerging search-based software engineering

paradigm [10, 12], which considers cloud computing as one of its application fields to tackle several multi-objective optimization problems [11]. Furthermore, the combination of model-driven engineering with search-based techniques is also investigated in several studies [14]. Following this path, we have applied a specific genetic algorithm to our optimization problem. To the best of our knowledge, there is only one prior work that applies optimization techniques to come up with an optimal configuration of a cloud application. In [8], the authors also use a combination of multi-objective search and simulation for finding an optimal deployment strategy for a given set of components of an application. In our approach, we go one step further and aim to optimize not only the deployment of the components, but also the usage of cloud computing patterns that are applicable on class-level granularity, what is of major interest when moving to PaaS providers.

An orthogonal optimization of cloud applications is targeted in the MODAClouds² and Passage³ projects, where the multi-cloud deployment of applications is studied by the application of the models@runtime notion [7]. Our approach currently does not foresee any support for the multi-cloud deployments, but may be extended by additional patterns supporting such scenarios as well in the future.

5 Conclusions and Future Work

In this paper we have introduced a pattern-based optimization approach for cloud applications. We follow a model-based approach to select configurations of cloud optimization patterns that satisfy some restrictions in terms of non-functional properties, and we determine the best configuration using model simulation.

Currently, our approach faces some limitations, some of which we want to address in the future. First of all, we have assumed that the base architecture is suitable for the cloud. This might not be the case for all applications and additional architectural refactoring patterns may be applied before our approach. Also, the simulation of the results through *e-Motions* is not straightforward as we need to create a new meta-model for each system the approach is applied upon. Furthermore, *e-Motions* presents some scalability issues when the models to be simulated grow in size. Other simulation tools might not have these drawbacks and might be more easy to use. For now, we have not evaluated the scalability of our approach in detail. More use cases, also industrial-sized use cases, need to be evaluated to experiment with more complex patterns as well as a larger number of patterns, goals, and trade-offs involved. Regarding the input, we need initial estimates on the impact a pattern has on an application. It may prove difficult to get these estimates manually from experts. Automation support based on benchmarks, partial application execution or log analysis could be integrated to support the user in collecting the estimates.

In the paper we have presented a proof-of-concept of our approach, from which we will address several future lines of work next. Firstly, we will apply benchmarks to measure the improvement associated with optimization patterns in large-scale applications provided as use cases in the ARTIST project. Secondly, we also plan to consider more optimization patterns from our catalogue, as well as study their influence after the ap-

² MODAClouds: <http://www.modaclouds.eu/>

³ Passage: <http://www.paasage.eu>

plication is deployed on the cloud. This would allow us to evaluate the feasibility and scalability of our approach in a more realistic setting. Thirdly, we plan to extend our goal modeling language to represent NFPs that are not taken into account in the current version, such as security properties. Finally, we plan to further study the application of different evolutionary algorithms for selecting the best configuration of optimization patterns.

References

1. Bergmayr, A., Brunelière, H., Canovas Izquierdo, J.L., Gorrionogitia, J., Kousiouris, G., Kyriazis, D., Langer, P., Menychtas, A., Orue-Echevarria, L., Pezuela, C., Wimmer, M.: Migrating Legacy Software to the Cloud with ARTIST. In: Proc. of CSMR. pp. 465–468 (2013)
2. Blum, C., Puchinger, J., Raidl, G.R., Roli, A.: Hybrid metaheuristics in combinatorial optimization: A survey. *Applied Soft Computing* 11(6), 4135–4151 (2011)
3. Coello, C.A.C.: A Comprehensive Survey of Evolutionary-Based Multiobjective Optimization Techniques. *Knowl. Inf. Syst.* 1(3), 129–156 (1999)
4. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *Trans. Evol. Comp* 6(2), 182–197 (2002)
5. Ehrgott, M., Gandibleux, X.: A survey and annotated bibliography of multiobjective combinatorial optimization. *OR-Spektrum* 22(4), 425–460 (2000)
6. Fehling, C., Leymann, F., Retter, R., Schupeck, W., Arbitter, P.: *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer (2014)
7. Ferry, N., Rossini, A., Chauvel, F., Morin, B., Solberg, A.: Towards Model-Driven Provisioning, Deployment, Monitoring, and Adaptation of Multi-cloud Systems. In: Proc. of CLOUD. pp. 887–894 (2013)
8. Frey, S., Fittkau, F., Hasselbring, W.: Search-based genetic optimization for deployment and reconfiguration of software in the cloud. In: Proc. of ICSE. pp. 512–521 (2013)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edn. (1994)
10. Harman, M.: The current state and future of search based software engineering. In: Proc. of ICSE. pp. 342–357 (2007)
11. Harman, M., Lakhota, K., Singer, J., White, D.R., Yoo, S.: Cloud engineering is search based software engineering too. *Journal of Systems and Software* 86(9), 2225–2241 (2013)
12. Harman, M., Mansouri, S.A., Zhang, Y.: Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.* 45(1), 11:1–11:61 (2012)
13. Homer, A., Sharp, J., Brader, L., Narumoto, M., T., S.: *Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications*. Microsoft Patterns & Practices (2014)
14. Kessentini, M., Langer, P., Wimmer, M.: Searching models, modeling search: On the synergies of SBSE and MDE. In: Proc. of CMSBSE@ICSE. pp. 51–54 (2013)
15. Rivera, J., Duran, F., Vallecillo, A.: A graphical approach for modeling time-dependent behavior of DSLs. In: Proc. of VL/HCC. pp. 51–55 (2009)
16. Troya, J., Vallecillo, A., Duran, F., Zschaler, S.: Model-driven performance analysis of rule-based domain specific visual models. *Inf. and Soft. Technology* 55(1), 88–110 (2013)
17. West, D.M.: *Saving Money Through Cloud Computing*. Brookings Institution (2010)
18. Wilder, B.: *Cloud Architecture Patterns*. O’Reilly (2012)
19. Zitzler, E., Deb, K., Thiele, L.: Comparison of Multiobjective Evolutionary Algorithms: Empirical Results. *Evolutionary Computation* 8, 173–195 (2000)

Modeling Cloud Messaging with a Domain-Specific Modeling Language

Gábor Kövesdán, Márk Asztalos and László Lengyel

Budapest University of Technology and Economics, Budapest,
Hungary
{gabor.kovesdan, asztalos, lengyel}@aut.bme.hu

Abstract. This paper introduces a domain-specific modeling language (DSL) for modeling application-level network protocols. Application-level messages may be expressed in object-oriented general-purpose programming languages as classes. Instances of these classes can be sent through the network with the help of a customized serialization process. However, protocols have several special characteristics that do not fit easily into this abstraction, for example, bitfields or specially encoded lists. Furthermore, the limitations of generic serialization frameworks inhibit using them for this purpose. These factors suggest creating a DSL that more easily expresses these protocols and allows for code generation to support application-level messaging. Application-level messaging is a crucial part of cloud services that follow the Software as a Service (SaaS) paradigm and it must be implemented at both clients and servers. A DSL that allows for efficient modeling of the messages and generating implementation code significantly simplifies the development of cloud applications.

Keywords: Modeling • Domain-Specific Languages • Code Generation • Protocols • Cloud

1 Introduction

Nowadays, there are several high-level communication standards that allow for network communication between two pieces of software. One group of these technologies consists of object-oriented remoting standards, like *Common Object Request Broker Architecture (CORBA)* [1] or Java's *Remote Method Invocation (RMI)* [2]. The other kind of commonly used technologies includes variants of Web Services, namely, the *Simple Object Access Protocol (SOAP)* [3] and *RESTful Web Services* [4]. Despite the availability of these mechanisms, still numerous software vendors decide to develop a lightweight binary application-level protocol that has a lower network footprint and does not require depending on resource-intensive libraries and application servers. However, when it comes to developing such a protocol, developers do not get too much help. The *Specification and Description Language (SDL)* defined by *ITU-T Z.100* [5] allows for describing system behavior in a stimulus/response fashion and thus, it can also be used to specify network protocols. However, SDL has a wide scope and focuses on the stimulus/response relations and does not capture message structure. In theory,

code generators can be developed for SDL to generate implementation but the generated code will cover only the stimulus/response relations. The *Protocol Implementation Generator (PiG)* [6] is a domain-specific modeling language [7] [8] that is designed for code generation but this solution also focuses on interactions. We have not found a domain-specific language with code generator that allowed for the modeling of message structure. In cloud services, especially in those that follow the Software as a Service (SaaS) paradigm, the message structure has more importance than communication states and interactions. First, these systems do not maintain a permanent connection and their messaging is often limited to notifications and request-response messages. Secondly, lower level protocols hide the establishment and the closing of connections, which in turn, involves communication states and interactions. Because of these factors, the development of SaaS messaging primarily consists of determining the message structure and developing the supporting code. Using binary messaging is more challenging to implement than relying on commonly supported formats, such as XML or JSON, that have extensive support in third-party libraries. However, this is the most concise form and thus it generates less network footprint and it is faster to parse. This suggests investing in a DSL and code generator to facilitate this development task. The messaging logic needs to be developed for both the client application and the cloud server. If they do not run on the same platform, the supporting code cannot be shared and has to be developed twice. A DSL and code generation techniques can remedy these difficulties. A code generator can be constructed that uses the model of the message structure and generates the supporting classes and the boilerplate code, even for multiple platforms, if necessary. Such a tool facilitates development and can ensure that the implementations in different languages are consistent.

In this paper such a DSL is presented. Our solution, *ProtoKit*¹, is a lightweight framework that focuses on modeling message structure and generating code to manipulate messages. It encompasses a metamodel that can be used to describe a wide variety of features that can be encountered in application-level messaging. The DSL syntax is similar to Java class definitions because the message structure shares some commonalities with them. The tool targets the use of binary messages. This format is the most concise and helps to save bandwidth, although it does not support well versioning and maintaining backward compatibility. As mentioned before, *ProtoKit* focuses on message structure since it is the most important factor in cloud messaging. It does not deal with modeling interaction: that is simply left for the application developers. We believe that if individual messages can be handled easily, dealing with simple interaction scenarios from handwritten code is easy. However, we may decide to implement modeling interactions in later versions of *ProtoKit*. The tool primarily targets cloud service providers because they always have to implement messaging at the server side and they often also provide the client software or the client library. In this way, they facilitate the use of their service to the consumers and they do not have to publish the protocol specifications. In this scenario, cloud service consumer indirectly benefit from *ProtoKit* as well. Additionally, if there is no appropriate client software or library but the protocol

¹ See <http://gaborbsd.github.io/ProtoKit/>.

specification is available, the service consumers may also use the tool to develop their own solution.

The rest of this paper is organized as follows. In Section 2 the motivation for creating a protocol modeling language is explained. Section 3 introduces the metamodel that was used to describe the problem domain. Section 4 explains the concrete syntax and show its grammar. Section 5 gives a detailed explanation of how the *ProtoKit* language was implemented. Section 6 presents a case study in which the solution is evaluated and Section 7 concludes. Despite not being a cloud protocol, the *Domain Name System (DNS)* protocol will be used as an example throughout the paper. The DNS protocol is well-known, has a wide variety of features that has to be handled in the metamodel and has similar characteristics as cloud messages: it lacks complex interactions and uses a simple request-response communication model. We are working on other applications that use *ProtoKit* and offer cloud services but they are still in an early phase of the development. Therefore, we have chosen to use the caching DNS server as an example.

2 Motivation

Protocol message types are very similar to classes in object-oriented programming languages: both notions define a complex data type with some properties that hold values. For example, a protocol message type may hold a transaction number of integer type, the identifier of the sender as an integer, an integer count that specifies the length of the payload and last but not least a variable-length payload either as a text or as binary data. Modeling such protocol message types is definitely possible with object-oriented languages but protocol message types have several specific properties that needs further boilerplate code in object-oriented languages. The following list summarizes these:

1. Defining the length of the field is paramount since the fields in a single message will be parsed by calculating the boundaries. In general-purpose programming languages (GPLs), this aspect is handled in a lazier manner. We usually choose from byte, short integer, normal integer and long integer variable types based on our needs of precision. Some languages, like C, do not strictly define the byte sizes of these type, whereas others, like Java, do. Still in the latter case, the byte length of a specific field is not explicitly reflected in the code, that is, the programmer must be conscious that, for example, a long variable takes 64 bits. Protocol message type definitions warrant for a more precise notation that explicitly expresses field lengths.
2. GPLs do not allow easily accessing fields on a per bit basis. Protocols do need such feature so that they can keep the network footprint low and avoid wasting bandwidth. One-bit boolean fields grouped to one or more bytes as flags are frequent. In GPLs, there is no native type that maps to such fields. Although it is possible to manipulate particular bits of a larger integer by using bitwise operators or by helper accessor methods, it requires more coding and makes the code less readable. A protocol message type definition language certainly needs to be able to handle data bit by bit.
3. Protocol messages often use counter fields that describe how many of a particular entry is found in the variable-length payload part of the message. These counters make it possible to properly parse the variable-length part of the message. In protocol

definitions, it would be practical to directly associate counters to the corresponding list of entries.

4. After the protocol message type is modeled, the serialization and deserialization of messages must be implemented so that messages can be transmitted and received over the network. Some GPLs, like Java, offer a standardized way for serialization but it does not fit well serialization of protocol messages. Serialization of protocol messages has several specific characteristics:
 - (a) Usually, it has to be strictly ordered based on the specification order of the fields.
 - (b) Length of fields is strictly specified, possibly on a per-bit basis.
 - (c) Counts of entries must be handled as well.
 - (d) Some fields may be encoded in a specific manner, for example, the *Domain Name System (DNS)* [9] protocol specifies its own encoding for the requested domain names.

When implementing the serialization of protocol messages, these requirements must be properly addressed. The above reasons suggest introducing a DSL for protocol message types that takes into account the above criteria and allow for easy and fast modeling of protocol message types.

Kövesdán et. al published an intent catalog [10] that lists and describes the possible motivating factors behind creating DSLs and their main characteristics. *ProtoKit* uses the following intents:

1. *Specialized Tool*: GPLs are not able to properly express all of the features of network protocols.
2. *Modeling Tool*: from the textual description, a model is constructed. This is used later for code generation.
3. *Domain-Specific Formalism*: since human language is ambiguous, developers may decide to also include the *ProtoKit* description in specifications.
4. *Human-Friendly Notation*: the *ProtoKit* language is much more concise than, for example, a Java class definition, therefore it is easier to read and write for non-programmers.

3 The Metamodel

The metamodel that we used for modeling network protocols is depicted in Figure 1. The elements of the metamodel are the following:

- *DataType*: message type or complex data structure. The class that is used to encapsulate the whole message is not treated in any special way so there is no need for introducing other metamodel element. A complex data structure may as well be embedded into another one.
- *BinaryField*, *IntegerField*, *StringField*, *ListField*, *CountField*, *LengthField*, *BitField*: specific types of fields that are embedded into messages or complex data structures.

- *BitFieldComponent*: *BitField* is further divided into components, which occupy only specific bits of the member. We generate specific getters and setters for these to handle the appropriate bits transparently.
- *Formatter*: some fields are encoded in a specific way. This kind of encoding can be implemented with the support of formatters. For formatters, only a template is generated that has to be filled in by the developer of the application.
- *ProtocolModel*: aggregates the *DataType* and *Formatter* elements into a model. It can be used to traverse the model for code generation.
- *Field*: abstract type of fields used as messages or complex data structure members.

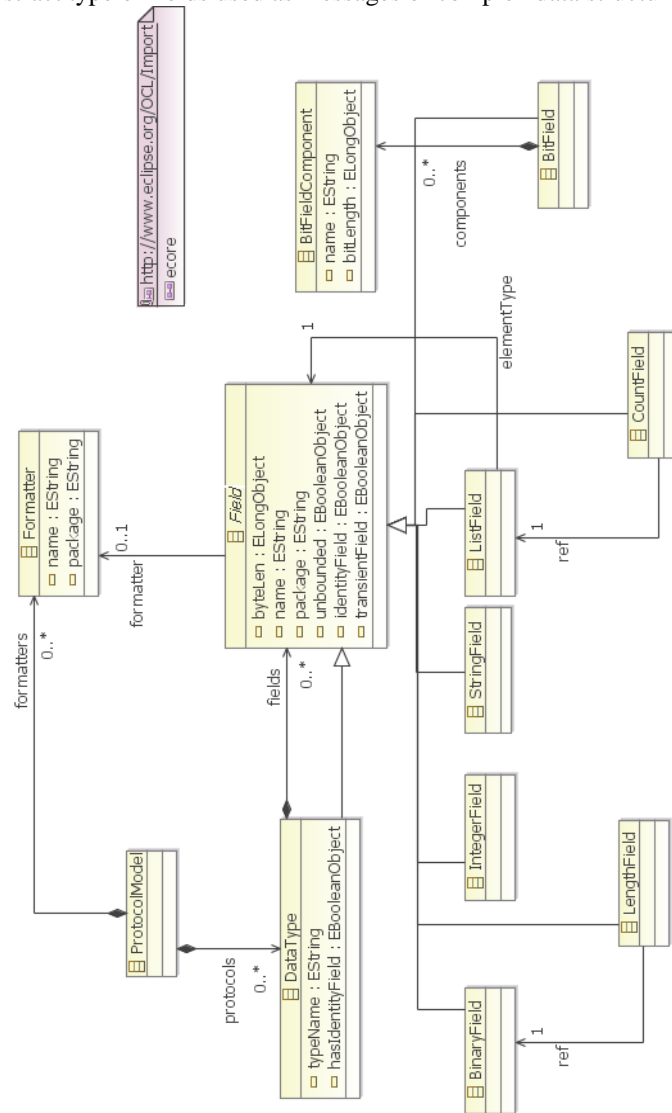


Fig. 1. The metamodel used for the *ProtoKit* language

4 The Concrete Syntax of the Language

In this section, we briefly describe the concrete syntax of *ProtoKit*. The syntax is somewhat similar to class diagrams. This is helpful for the developers since the nature of the models is also similar to class diagrams. A model starts with the *package* keyword and an identifier. These will define the package of the generated Java classes. After this, we can define protocol messages and embedded data types. The former starts with the *protocol* keyword and the latter uses the *datatype* keyword. The definition of messages and data types is given in curly braces. We specify fields by their name and type, separated by a colon. The type may have arguments that refer to the length of the field or to a referred field in case the field is a counter or a length field. After a field of the string type, we can also specify a formatter with its name. Components of *bitfields* are also defined in curly braces and these are always treated as integer types so only their length is specified in bits. The names of normal fields (that are not components of *bitfields*) can be preceded by the *transient* keyword and an asterisk. The former means that the field will not be serialized² and the latter marks the fields that should determine the identity of instances. This is used for generating *equals()* and *hashCode()* methods. The ANTLR grammar of the language is cited in the following code listing.

```
grammar NetworkProtocol;

start: packageDefinition? protocolDefinition+;

packageDefinition: 'package' name = ID;

protocolDefinition: ('protocol'|'datatype') name = ID '{'
variableDefinition+ '}';

variableDefinition: trans='transient'? identityVar='*'? name =
ID ':' ( intType| stringType| binaryType| embeddedType| bitfieldType
| listType| countType| lenType);

intType: type = 'int' ('(' len = NUMBER ')')?;

stringType: type = 'string' ('(' len = (NUMBER|'*') ')') for-
matterDefinition?;

binaryType: type = 'binary' ('(' len = (NUMBER|'*') ')');

embeddedType: type = ID;

bitfieldType: type = 'bitfield' '{' bitfieldDefinition+ '}';

listType: type = 'list' '(' (listElement = ID) ')';
```

² It can be disputed whether this feature should be part of the modeling language since it is not related to the actual model but to the implementation. However, it allows for adding implementation-specific members to the generated classes without having to modify them. This does not make it necessary to deal with manual changes when regenerating classes.

```

countType: type = 'count' '(' len = NUMBER ',' countedList = ID
')';

lenType: type = 'length' '(' len = NUMBER ',' countedField = ID
')';

bitfieldDefinition: name = ID ':' bitLength = NUMBER;

formatterDefinition: 'formatter' name = ID;

ID: [a-zA-Z]+;

NUMBER: [1-9] [0-9]*;

WS: [ \t\r\n]+ -> skip

```

5 Implementation Decisions

In this section, we describe step by step what implementation decisions we considered during the developing of *ProtoKit*. This gives a deeper insight into the development process and allows for understanding how our motivations and the requirements affected the architecture and the development process. By examining these points regarding a new DSL that is being developed, we can also reuse these experiences as a recipe.

1. *Implementing generic functionality in the runtime framework.* It is not a trivial decision at what extent the code should be factored out into a generic runtime framework and what should be generated. Developing generic code is generally more challenging and sometimes may have worse performance than a customized solution. For example, in Java environment, it is often done by using reflection, which has a performance hit. On the other hand, generic code is more reusable and helps to reduce the generated code. It is easier to generate custom code than having a generic framework but the latter is easier to test since it does not depend on the input model. The reusable framework can be covered by extensive unit tests, whereas the generated code is not trivial to test. It may be tested for the complete functionality but it is difficult to think of all of the possible corner cases. Because of these considerations, in *ProtoKit* we have implemented most of the functionality in a reusable runtime framework. The most important example of this is the generic serialization logic. The generated classes only encompass the serialization parameters. It would have been possible to generate the serialization logic and to make it part of the generated class but this would have resulted in a significantly more complex generator.
2. *Separating tree parser logic from code generator.* In theory, it is possible to process the input file in an event-driven approach with a single read since we can store arbitrary amount of details in the state of the parser that will be required in further phases of the processing. However, this technique has several drawbacks. Because of the event-driven nature, the code fragments that are generated are triggered by visiting a certain grammar rule. This does not facilitate generating non-consecutive code fragments from the same node. For example, if the generated code is a Java class, certain nodes can be mapped to Java variables. Java variables are usually declared

with *private* visibility and accompanying getter and setter methods are provided with *public* scope. Although it is not mandatory, the variable declarations are conventionally placed together at the beginning of the class definition and the getter and setter pairs come after all variable definitions. A single variable and its getter and setter methods are generated from the same node so they cannot be generated in the conventional order by a purely event-driven manner. This requires the parser to use its internal state to store some details. This makes the parser more complex and partly leads to building a semantic model in the memory. Secondly, the use of a pure event-driven approach does not allow for validating cross-references in the input file. To address these issues, we have separated the tree parser logic and the actual code generation. The only responsibility of the tree parser is building a semantic model. By using a semantic model, validation and code generation become much easier. Furthermore, the use of a semantic model better facilitates changes in the syntax or building another, possibly visual modeling language on top of the code generator.

3. *Decoupling code generation and formatting.* Although generated code does not necessarily need to be read by humans, readability is definitely a great advantage. For example, it may help debugging or facilitate the comprehension of how the *ProtoKit* tooling works. However, hardcoding formatting, like indentation level, line breaks etc. in the generator significantly deteriorates the readability of the generator itself. The readability of the generator is definitely more important than the readability of the generated code so this is not a viable trade-off. However, collecting the output in a buffer and using a code beautifier before flushing the code to the disk has proven to be a favorable solution. This solution makes the code formatter reusable. In *ProtoKit*, the code formatter provided by *Eclipse JDT* [11] has been used.
4. *Using template language for code generation.* When traditional programming languages are used for code generation, fragments of the generated text must be quoted and concatenated to the variables that are substituted. All of this is written to a buffer with method calls. These method calls, the quotation marks and the concatenations deteriorate the readability and make it hard to see what output will be actually generated. Template languages reverse the logic: everything that is written will be part of the output by default and only variable substitutions and branching need special markup. In Java environment, *Xtend* [12] is a good choice of a template language. It does not compile directly to bytecode but to Java source code so it can be easily used anywhere where Java is used. *ProtoKit* is written mostly in Java but the code generator classes are implemented in *Xtend*. There are some branching statements and substitutions but the readability of the generator is much better than it could be in pure Java. The separation of the code formatter logic and the use of *Xtend* have significantly improved the productivity during the development of *ProtoKit*.
5. *Using a metamodeling framework and decoupling the validator logic.* The validation logic was first coded into the tree parser but as we added more features to the language, it started to deteriorate the readability so we decided to factor it out. At the same time, we introduced an explicit metamodel, described with the *Eclipse Modeling Framework (EMF)*. [13] Using a modeling framework allows for reusing its validation solutions and we wanted to benefit from this. However, associating line and

column numbers from the input text with validation errors is more challenging. Its implementation requires attaching extra information to the model.

6 Evaluating *ProtoKit*

To demonstrate that *ProtoKit* is in fact useful and really helps the development of applications that use network communication, it must be put into practice. For this purpose, we have implemented a caching DNS server that either answers queries from its local cache or forwards queries to the configured resolver. *ProtoKit* was in fact easy to use. We could model DNS messages easily. The generated classes and the generic serialization logic highly simplified the process. Describing the caching DNS server is beyond the scope of this paper but we have summarized some statistics about the generated and handwritten code in Table 1.

Table 1. Statistics in code lines

Language	Generated	Reused	Hand-written	TOTAL
ProtoKit	0	0	39	39
Java	463	328	289	1080
TOTAL	463	328	328	1119

In this case, only about 27% of the Java code had to be manually implemented. If we also take into account the textual model of the protocol, we get about 29% hand-written code, which seems to be a really good proportion.

The caching DNS server is a simple but realistic application so we believe it is a good case study to evaluate *ProtoKit*. More complex applications also contain more application logic that is independent of the networking code. The proportion will be worse in such cases but *ProtoKit* is only meant to facilitate network messaging.

Using SaaS cloud services is always more common in computing, especially in mobile applications. As described above, cloud messaging protocols are similar to the DNS protocol in complexity: they use a request-response communication model and do not encompass complex interactions. Since *ProtoKit* performed well in modeling DNS messages, it will also be very useful for implementing the messaging between clients and the cloud services. The Android platform uses Java as its main programming language – however the class library is slightly different – which means that the generator can also be used for Android applications with no or few modifications. Furthermore, Java is also a popular platform in backend development, so the messaging code can be shared between the client and the cloud server backend. Besides, *ProtoKit* is easy to extend to support other target languages.

7 Conclusion

In this paper we have reported about our progress in developing the *ProtoKit* language and tooling. We have explained our motivations and how we created a DSL to solve

these issues. We have also summarized the decisions that we met during the implementation and why we took these decisions. Finally, we have also reported on a simple but realistic application that we developed with the help of *ProtoKit*. In this case study *ProtoKit* in fact simplified the development. However, the main focus of this solution is developing cloud messaging. Cloud messages use simple interactions but require a message structure that is easy to work with and can be serialized efficiently. This is exactly what *ProtoKit* is meant to be used for. We hope that it will prove to be useful in practice on the long term and that our experiences will help other developers that implement cloud-enabled applications.

Acknowledgments. This work was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TAMOP-4.2.2.C-11/1/KONV-2012-0013) organized by VIKING Zrt. Balatonfüred. This work was partially supported by the Hungarian Government, managed by the National Development Agency, and financed by the Research and Technology Innovation Fund (grant no.: KMR_12-1-2012-0441).

References

1. The Object Management Group: CORBA 3.3 Specification, <http://www.omg.org/spec/CORBA/3.3/>
2. Grosso, W.: Java RMI, O'Reilly Media (2001)
3. World Wide Web Consortium: Simple Object Access Protocol (SOAP) Specification, <http://www.w3.org/TR/soap/>
4. Richardson, L., Ruby, S.: RESTful Web Services, O'Reilly Media (2007)
5. International Telecommunication Union: ITU-T Z.100 Standard. Specification and Description Language (SDL), http://www.itu.int/ITU-T/studygroups/com10/languages/Z.100_1199.pdf
6. Quaresma, J.: A Protocol Implementation Generator, Master Thesis, http://nordsecmob.aalto.fi/en/publications/theses_2010/jose_quaresma.pdf
7. Fowler, M.: Domain-Specific Languages, Addison-Wesley (2010)
8. Kelly, S., Tolvanen, J.: Domain-Specific Modeling: Enabling Full Code Generation, Wiley-IEEE Computer Society Press (2008)
9. The Internet Engineering Task Force: Domain Names – Implementation and Specification, Request for Comments 1035, <http://www.ietf.org/rfc/rfc1035.txt>
10. Kövesdán, G., Asztalos, M., Lengyel, L.: A classification of domain-specific language intents, International Journal of Modeling and Optimization, vol. 1, no. 4, pp. 67–73 (2014)
11. The Eclipse Project: Eclipse Java Development Tools, <http://www.eclipse.org/jdt/>
12. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend, Packt Publishing (2013)
13. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework (2nd Edition), Addison-Wesley Professional (2008)

Cloud DSL: A Language for Supporting Cloud Portability by Describing Cloud Entities

Gabriel Costa Silva, Louis M. Rose, and Radu Calinescu

Department of Computer Science, University of York,
Deramore Lane, York YO10 5GH, UK

`gabriel@cs.york.ac.uk`,
`{louis.rose,radu.calinescu}@york.ac.uk`

Abstract. Different cloud platforms offer similar services with different characteristics, names, and functionalities. Therefore, describing cloud platform entities in such a way that they can be mapped to each other is critical to enable a smooth migration across platforms. In this paper, we present a DSL that uses a common cloud vocabulary for describing cloud entities covering a wide variety of cloud IaaS services. Through analysis of existing cloud DSLs, we advocate that our cloud DSL is more expressive for the purpose of describing different cloud IaaS services. In addition, when used along with TOSCA, our preliminary analysis suggests that our Cloud DSL significantly reduces the workload of creating cloud descriptions in a TOSCA specification.

Keywords: DSL; Cloud computing; Portability; TOSCA

1 Introduction

Costa Coffee, Starbucks, and Caffè Nero are the three largest coffee shop companies in the UK. Although they all sell their coffees in three common sizes – small, medium, and large – they name these sizes differently. This can create a lot of confusion. A Starbucks customer might get frustrated when ordering a *Grande* in Caffè Nero since that, in the Starbucks vocabulary *Grande* means medium, whereas for Caffè Nero it means large. Likewise, those unfamiliar with Starbucks might find contradictory that *Tall* is the smallest size. Furthermore, similar products also have different characteristics. For instance, a medium skimmed latte in Starbucks¹ contains 156% more calories than its Nero² version. Thus, although their products look similar, they are not all the same. Therefore, having a detailed description of each product is critical to prevent misunderstandings.

Like in the coffee-shop market, cloud platforms offer similar services, but with different names, characteristics, and functionalities. For example, consider the Amazon S3 and Dropbox storage services. Overall, they provide the same functionality: file storage, storage elasticity, and interfaces for management. However,

¹ <http://www.starbucks.co.uk/quick-links/nutrition-info>

² <http://www.caffenero.co.uk/Nutrition/hotdrinks.aspx>

a closer look reveals critical differences, including the use of different file systems. Whereas Dropbox has one single file system root, Amazon S3 uses multiple root containers called “buckets”. Furthermore, a bucket has a region, which specifies a geographical location for the content stored within. Like for coffee shops, to support a smooth migration of applications across different cloud platforms, it is critical to describe the *semantics* of cloud entities.

Semantic differences are critical in cloud as they hinder the smooth migration of assets (e.g., data and applications) across providers [21]. For example, the differences between Amazon S3 and Dropbox hinder the migration (transfer) of files from Dropbox to Amazon S3, as it is necessary to create a bucket and assign it to a region. Cloud portability, i.e. the ability to migrate an asset deployed in one cloud to another [18], is one of biggest challenges in cloud computing, and it has been widely addressed by both academia and industry [17].

Providing homogeneous description of cloud platforms is a potential solution to overcome semantic differences and achieve cloud portability in this highly heterogeneous environment [3], [15], [19]. Recent research reveals three means to describe cloud platforms [22]. Although we present them here, discussing their benefits and drawbacks is beyond the scope of this paper.

- (i) A *platform abstraction* solution consist of describing concepts of either the entire platform or its elements, at different levels, e.g. as proposed by MODA-Clouds [3]. This solution can adopt different technologies to achieve their goals, such as ontologies [6] and Model-Driven Architecture [3]. This solution covers both design- and run-time, e.g. as proposed by meta cloud [20];
- (ii) *Standardized references* focus on design-time only through at defining references for cloud platforms [15], APIs [7], [1], [9], or applications [11]. In the context of our paper, a reference is a set of rules, or constraints that a cloud user or provider must follow. However, most of solutions in this type focus on setting up references for cloud APIs. According to Escalera & Chavez, cloud APIs consist of software libraries used by application developers to manage cloud services [7]. Apart from [11], which targeted cloud applications, other solutions in this type target only cloud providers; and
- (iii) *Domain Specific Languages* (DSLs). A DSL is a language tailored for a particular domain or context [5]. Like *standardized references*, DSL-based solutions are for use at design-time. However, some solutions might provide support for run-time mechanisms, such as CloudMF [8]. As DSLs are defined for a particular purpose [16], these solutions cover a wide range of goals, such as automatic generation of mobile-cloud applications [19], and description and comparison of Service Level Agreements (SLAs) [2]. Regardless of the purpose, meta-models are the cornerstone of these solutions. Finally, DSL-based solutions are mainly intended to support cloud users.

In this paper, we present a DSL that uses a common cloud vocabulary for describing cloud platform entities, such as services and resources across a wide variety of cloud platforms. Unlike existing Cloud DSL, this work covers a wide variety of cloud IaaS services, contributes to different phases of cloud portability, facilitates the communication of services and resources to different levels of

stakeholders, and enables the description of different types of clouds, such as federation and inter-clouds. In addition to positioning this work in the related literature (Section 5), we contribute to the advance of the state-of-the-art in both cloud portability and Model-Driven Engineering (MDE) by: (i) supporting cloud portability via a common meta-model for cloud computing (Section 2); (ii) facilitating the visualization and communication of cloud assets amongst different stakeholders by providing a graphical editor (Section 3); and (iii) reducing the effort of describing cloud entities in TOSCA cloud standard [4] (Section 4).

2 Cloud Meta-model

The meta-model, which describes the domain covered by the language, is the cornerstone of a DSL. A DSL consists of abstract and concrete syntax — whereas the abstract syntax defines the constructs of the language, the concrete syntax defines the representation of these constructs [5], [16]. For example, the abstract syntax of the Web Service Description Language (WSDL) defines a set of entities and their properties, such as *ServiceType* and *InterfaceType*, representing, respectively, a service exposed to a client, and its interfaces. To specify *Services* and *Interfaces*, one uses XML statements like `<service />` and `<interface />`. These XML statements are the concrete syntax of the WSDL.

Due to the heterogeneity of cloud platforms and services, creating a cloud meta-model that covers a broad range of cloud services is not a straightforward task. To devise a cloud meta-model, we: (i) analysed four sources of information; (ii) identified correspondences amongst similar entities across these different sources; and (iii) combined entities and their relationships into a new meta-model. Our analysis started with an extensive literature review [22], in which we identified some critical cloud entities. Next, we leveraged the contribution of two important standardization efforts, OGF OCCI [9] and DTMF CIMI [1]. We also investigated relevant research projects focusing on cloud portability, in particular MODAClouds³ and REMICS⁴. Finally, we examined a wide range of cloud IaaS services, including Amazon SQS, Microsoft Azure Compute, and Rackspace Cloud Files. Figure 1 shows our cloud meta-model.

A cloud *Platform* provides *Service*, such as computing and storage. A cloud *Platform* has a name, such as Amazon Web Services, or OpenNebula. A *Service* might be managed through multiple *Management Interfaces*, which are provided by cloud *Platform*. A *Management Interface* has a type, such as RESTful, or Query-based, and properties, such as authentication attributes. The *Platform* is responsible for the *Cloud User* management. A *Cloud User* is identified by its name, and can have several keys to access its *Resources*. A *Service* is identified by its name, such as EC2, or Cloud Servers. A *Service* might be supported by other service. For example, Amazon EBS and S3 supports Amazon EC2, providing persistent storage. Each *Service* might operate in a different *Region*. A *Region* represents a wide geographical location, such as Europe or Asia. In addition to

³ <http://www.modaclouds.eu>

⁴ <http://www.remics.eu>

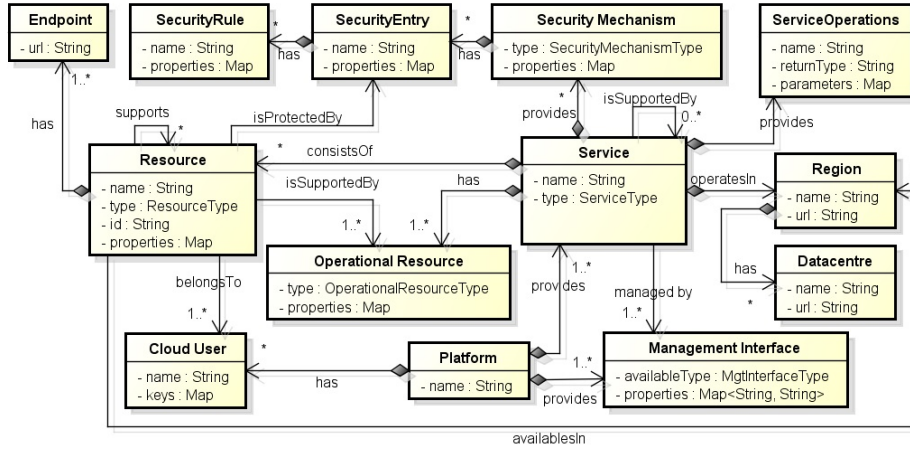


Fig. 1. Abstract syntax of the Cloud DSL

its name, a *Region* might have a particular code assigned by the cloud *Platform*. This code is represented in this meta-model by the *url* attribute. A *Region* might have several *Datacentres*, which are identified by their names and urls.

A *Service* has *ServiceOperations*. A *ServiceOperation* represents those management operations that a *Cloud User* can perform on its *Resource* through a *Management Interface*, such as sending a message to a queue in a message queue service. A *ServiceOperation* is identified by its name, such as *runInstance* or *listImages*, for example. Some operations rely on input parameters, such as the name of image, and region. As these operations are implemented by APIs, they have a single return value, which might be a list of results, for example. A *Service* exists to provide *Resources*, such as VMs and storage containers. Each *Platform* might define different properties and states for *Resources* provided by its *Service*. However, two properties are present in most of *Resources*: *id* and *name*. Once the *Resource* has been created, it is made accessible by one or more *Endpoints*. The *Resource* is available in one of the *Region* supported by the *Service*. One *Resource* might support another. This is the case of storage containers, which are used to support a set of files (*Resource*).

Some *Resources* rely on *OperationalResource*. An *OperationalResource* represents internal resources provided by the *Service*, such as the hardware of a VM, or the engine of a database service. In addition to the *Cloud User* credentials, some services provide further *Security Mechanism*, such as firewall and permissions. A *Security Mechanism* consists of a set of *Security Entry*. For example, a file in a storage service might have different permissions (*SecurityEntry*). Each *SecurityEntry* is assigned to the *Resource* it protects. More sophisticated mechanisms require further elements, represented in the meta-model by *Security Rule*. This is the case of security groups, provided by Amazon EC2.

3 Proof of Concept

To implement our Cloud DSL, we adopted Epsilon⁵, a suite of languages and tools for model management operations, such as model transformation and analysis. The implementation process consisted of four steps: (i) creating the cloud meta-model using Emfatic, a textual notation for creating Ecore models; (ii) annotating the meta-model with EuGENia annotations. EuGENia is a tool that takes advantage of model transformation techniques to mitigate the complexity of GMF and EMF [13]; (iii) generating the graphical editor using the EuGENia tool; and (iv) adjusting graphical components, such as figures used to represent cloud entities. The graphical editor consists of three parts (Figure 2): (i) a canvas, in which cloud entities and their relationships are represented by graphical components; (ii) a palette, which presents the cloud meta-model entities; and (iii) the properties tab, which shows the properties of each selected entity.

To evaluate the expressiveness of our Cloud DSL, we used our Cloud DSL to describe the Amazon Web Application Hosting (AWAH) reference architecture⁶. The reference architecture consists of seven different Amazon services, which provides several resources for a Web application, such as storage and DNS routing. In order to implement this reference architecture, we hosted the Java PetStore⁷ application using the services defined in the reference. Figure 2 shows the description of two services: Amazon CloudFront and Amazon S3. Amazon CloudFront is a content distribution service, which routes requests to the nearest content storage location.

In this implementation of AWAH reference architecture, we have only one distribution configured, which represents a *Resource* for this service (*Distribution.PetPictures*). As Amazon CloudFront does not store the content, it relies on a storage service, in this case, Amazon S3. The figure shows three resources provided by Amazon S3: *PetPictures*, *petstorestaticpages*, and *index.html*. Whereas the first two resources are buckets, the third is a file. The two buckets are protected by a *Security Entry*, defined using the *Security Mechanism* (PERMISSION). The location of *PetPictures* bucket is explicitly defined by a *Region*, represented by *US Standard* in this example.

The concrete syntax of our Cloud DSL represents a *ServiceOperation* as a container inside the *Service*. *ServiceOperation* might have several parameters. In this example, the operations described represent those required to operationalise the AWAH architecture, such as starting an instance. In Figure 2, Amazon S3 operations are hidden in the *Service* entity (note the “+” signal just below the service name). Figure 3 (a), shows the Amazon RDS service and the operations described: *launchDBInstance*, and *terminateDBInstance*. Whereas the first operation has six parameters, the second has only one. Parameters are represented by an “i” signal. This notation is used throughout our Cloud DSL to represent parameters. For example, Figure 3 (b) shows two *OperationalResources* (HARD-

⁵ <http://www.eclipse.org/epsilon/>

⁶ http://aws.amazon.com/architecture/?nc1=f_cc

⁷ <http://www.oracle.com/technetwork/articles/javaaee/petstore-137013.html>

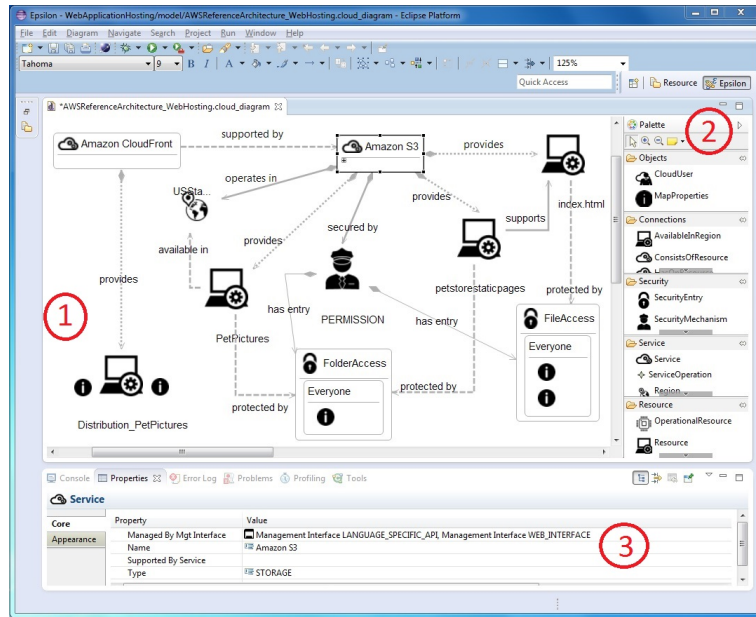


Fig. 2. The concrete syntax of the Cloud DSL implemented by a graphical editor

WARE and VM_IMAGE) which contain particular properties. Figure 3 (c) shows the properties tab for one cloud *Resource* (VM_INSTANCE).

Figure 3 (b) shows the Amazon EC2 service, and its related *Resource*, *DynamicWebSite*. This resource is a VM which hosts the dynamic content of the Java PetStore application. In the properties tab (Figure 3 (c)), it is possible to see the properties of this resource, such as id (generated by the cloud platform), and the *Cloud User* which owns the VM. *DynamicWebSite* is supported by two *OperationalResources*: HARDWARE, and VM_IMAGE. Whereas the former represents the type of instance used, the latter represents the Amazon Machine Image (AMI) used. The AMI contains the operational system as well as all applications required to run the Website. Different from Amazon S3 (Figure 2), Amazon EC2 enables specifying a particular datacentre. It is possible to note it in the Figure 3 (b), just above the globe, which represents a *Region*. Finally, in order to access the VM, an endpoint is made available. The concrete syntax for it is small rings, located just in the right side of *DynamicWebSite* resource.

4 Towards Simplifying Cloud Services and Resources Description in TOSCA

Topology and Orchestration Specification for Cloud Applications (TOSCA) is a standard supported by OASIS, and intended to support application portabil-

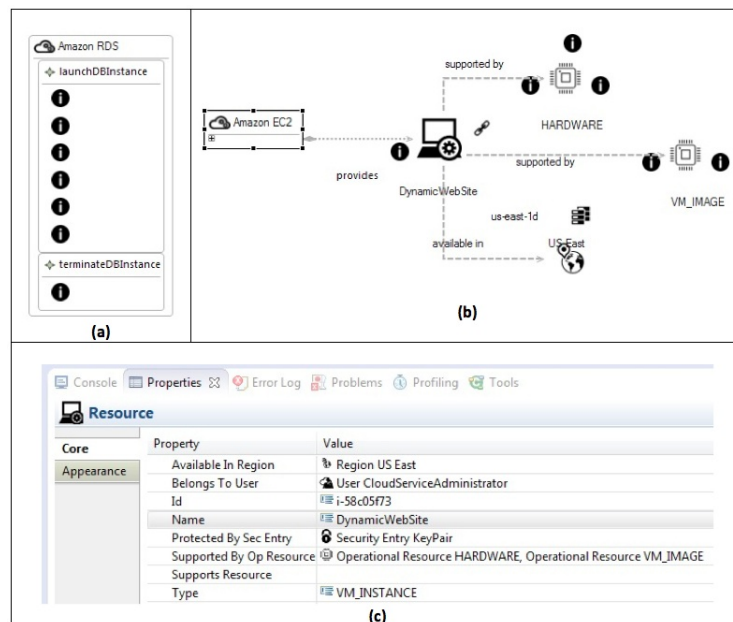


Fig. 3. Our Cloud DSL in action describing the AWAH reference architecture

ity across clouds. TOSCA defines types, that describe applications and cloud services, and templates that represent instances of these types. The TOSCA ecosystem comprises: specification and run-time environments. Whereas the former covers both application topology and activity orchestration, the latter is responsible for processing these specifications [4]. Several companies demonstrated the benefits of using TOSCA migrating an application across cloud platforms⁸.

Despite from the benefits that TOSCA can provide, describing cloud resources in a TOSCA specification is a cumbersome task. TOSCA does not use the typical cloud vocabulary, such as services and resources. Instead, it defines a set of abstract elements, such as nodes, capabilities, and policies. Although this strategy enables the specification of both cloud and application components, it complicates the specification of cloud platform entities using TOSCA elements, specially because TOSCA official documentation does not define how to map cloud entities to TOSCA elements. In addition, as TOSCA specification is defined as a XML document, it is quite hard to have an overview of cloud entities.

Therefore, we proposed using our Cloud DSL to specify cloud services and resources for TOSCA specification. To this end, we are taking advantage of MDE techniques, in particular, model-to-model and model-to-text transformations (MT). As Hermans, Pinzger & van Deursen identified in their study [12], a DSL along with MT techniques contribute to reduce effort and increase productivity by automating repetitive tasks, such as code generation. Indeed, Brambilla,

⁸ <https://www.oasisKopen.org/events/cloud/2013/TOSCAdemo>

Cabot & Wimmer analyse that code generation can save much effort for implementing CRUD operations, which are responsible for 80% of software functionality in data-intensive applications [5]. To achieve these benefits, we have begun work on mapping between our cloud meta-model and TOSCA elements.

However, this cloud-to-TOSCA mapping is an on-going work, which has required substantial intellectual and technical effort. Thus, describing these mappings is beyond the purposes of this paper. Here, we report on what we want to achieve once the mapping is complete. Our preliminary analysis has shown that it is possible to achieve similar results to those reported in [12], in particular, effort reduction. Our hypothesis for such statement is underpinned by the fact that a single cloud entity can be mapped to more than one TOSCA element.

For example, a cloud *Service* is represented in TOSCA as a *TNodeType*. However, as exists dependences between services, they also become a *TCapabilityType* and a *TRequirementType*. For instance, Amazon EC2 relies on Amazon EBS to store persistent data. Therefore, Amazon EBS provides the storage capability whereas Amazon EC2 requires such a capability. In addition, a cloud *Service* carries information used by TOSCA *TNodeTemplate*. Thus, writing a TOSCA specification manually would require that those four entities and all their related information were encoded by a human developer — which is both a time consuming and an error-prone activity.

5 Related Work

In 2010, Gonçalves et al. presented the first DSL devised specifically to describe cloud entities, CloudML [10]. CloudML is underpinned in the D-Cloud context aiming at allowing cloud computing providers to describe both cloud resources and services, and cloud developers, to describe their computing requirements. D-Clouds stands for Distributed Clouds, and authors define it as “*smaller datacentres sharing resources across geographic boundaries.*” CloudML is an XML-based language, and it is based on three requirements: (i) representation of physical and virtual resources as well as their state; (ii) representation of services provided; and (iii) representation of developer’s requirements. In contrast to this DSL, our work is not limited to a particular context. As our Cloud DSL captures essential characteristics of cloud platforms, it can be used to model different types of clouds, such as federation and inter-cloud.

In 2011, Liu & Zic presented Cloud#, a textual DSL that enables cloud providers to describe internal organization of cloud resources [14]. Focused on computing services, their requirements are: (i) to express computation units and different privilege levels of computation; (ii) to allow programmable bidirectional control and data transfer between computation units; and (iii) to model physical resources. The two cornerstone entities in their meta-model are *CUnit*, which represents a cloud, a virtual machine, or an operating system; and *Action*, which defines a computation task. Different from this DSL, which defines a textual language to describe computing services, our Cloud DSL provides a graphical representation of cloud entities, presented in a diagram. Thus, our Cloud DSL

facilitate not only the visualisation of cloud entities, but also the communication of the cloud strategy amongst different levels of stakeholders.

In 2012, Alkandari & Paige reported on-going work towards a DSL for describing and comparing SLAs offered by different cloud providers [2]. Authors came up with two meta-models, one for describing SLAs offered by cloud provider, and another to describe SLAs required by cloud users. In addition, an algorithm was developed to compare models from cloud users to those from cloud providers. However, this DSL is limited to the first phase of cloud portability - analysis. Although our Cloud DSL cannot describe SLAs with the richness of detail as this DSL does, our Cloud DSL contributes to different phases of cloud portability, such as analysis and migration.

Finally, in 2013, Ferry et al. introduced the CloudMF [8]. CloudMF aims at supporting provisioning and deployment of applications in multiple clouds at run- and design-time. To accomplish this objective, CloudMF covers four requirements: (i) separation of concerns; (ii) provider independence; (iii) reusability; and (iv) abstraction. CloudMF consists of two components: (i) CloudML, the modelling environment (DSL); and (ii) Models@run-time, which provides an abstract representation of the running system. However, this DSL is limited to computing and storage services. Our Cloud DSL enables the description of a wide variety of cloud services, such as message queue, scaling, and DNS routing.

6 Conclusion

This paper introduced a Cloud DSL which supports cloud portability by describing cloud platform entities. The wide coverage of cloud IaaS services, and our ongoing work towards the integration of our Cloud DSL and TOSCA, suggest that our Cloud DSL can cover critical aspects of cloud service specification. As next step, we will investigate literature to identify means of evaluating and comparing our Cloud DSL to other cloud-related languages. Finally, in our project, we have been working towards mapping our Cloud DSL to platform-specific offerings by mapping entities of cloud meta-model to platform-specific cloud APIs. Exploiting these capabilities requires the maintenance of these mappings.

Acknowledgments. This work was funded in part by CNPq - Brazil and EU FP7 project OSSMETER (Contract #318736).

References

1. DTMF CIMI, <http://www.dmtf.org/standards/cloud>
2. Alkandari, F., Paige, R.F.: Modelling and comparing cloud computing service level agreements. In: 1st Intl Workshop on Model-Driven Engineering for High Performance and CCloud computing. pp. 1–6. ACM Press, New York, USA (2012)
3. Ardagna, D., Di Nitto, E., Mohagheghi, P., Mosser, S., Ballagny, C., D’Andria, F., Casale, G., Matthews, P., Nechifor, C.S., Petcu, D., Gericke, A., Sheridan,

- C.: MODAClouds: A model-driven approach for the design and execution of applications on multiple Clouds. In: 4th Intl Workshop on Modeling in Software Engineering. pp. 50–56. IEEE, Zurich (Jun 2012)
4. Binz, T., Breitenbücher, U., Kopp, O., Leymann, F.: TOSCA: Portable Automated Deployment and Management of Cloud Applications. In: *Advanced Web Services*, pp. 527–549. Springer, New York (2014)
 5. Brambilla, M., Cabot, J., Wimmer, M.: *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers (2012)
 6. Ejarque, J., Alvarez, J., Sirvent, R., Badia, R.M.: A Rule-based Approach for Infrastructure Providers' Interoperability. In: *IEEE 3rd CloudCom*. pp. 272–279. IEEE, Athens (Nov 2011)
 7. Escalera, M.F.P., Chavez, M.A.L.: UML model of a standard API for cloud computing application development. In: 9th Intl Conf on Electrical Engineering, Computing Science and Automatic Control. pp. 1–8. IEEE, Mexico City (Sep 2012)
 8. Ferry, N., Chauvel, F., Rossini, A., Morin, B., Solberg, A.: Managing multi-cloud systems with CloudMF. In: 2nd Nordic Symposium on Cloud Computing and Internet Technologies. pp. 38–45. ACM, Oslo, Norway (2013)
 9. Forum, O.G.: Open Cloud Computing Interface (OCCI), <http://occi-wg.org/>
 10. Goncalves, G., Endo, P., Santos, M., Sadok, D., Kelner, J., Melander, B., Mangs, J.E.: CloudML: An Integrated Language for Resource, Service and Request Description for D-Clouds. In: *IEEE 3rd CloudCom*. pp. 399–406. IEEE, Athens (Nov 2011)
 11. Hamdaqa, M., Livogiannis, T., Tahvildari, L.: A reference model for developing cloud applications. In: 1st International Conference on Cloud Computing and Services Science. pp. 98–103. SciTePress, Noordwijkerhout (2011)
 12. Hermans, F., Pinzger, M., van Deursen, A.: Domain-Specific Languages in Practice: A User Study on the Success Factors. In: *Model Driven Engineering Languages and Systems*, pp. 423–437. Springer Berlin Heidelberg, Berlin (2009)
 13. Kolovos, D., Rose, L., Abid, S., Paige, R., Polack, F., Botterweck, G.: Taming emf and gmf using model transformation. In: *Model Driven Engineering Languages and Systems, LNCS*, vol. 6394, pp. 211–225. Springer Berlin Heidelberg (2010)
 14. Liu, D., Zic, J.: Cloud#: A Specification Language for Modeling Cloud. In: *IEEE 4th CLOUD*. pp. 533–540. IEEE, Washington, DC (Jul 2011)
 15. Loutas, N., Peristeras, V., Bouras, T., Kamateri, E., Zeginis, D., Tarabanis, K.: Towards a Reference Architecture for Semantically Interoperable Clouds. In: *IEEE 2nd CloudCom*. pp. 143–150. IEEE, Indianapolis (Nov 2010)
 16. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. *ACM Computing Surveys* 37(4), 316–344 (Dec 2005)
 17. Petcu, D.: Multi-Cloud: expectations and current approaches. In: *MultiCloud '13*. pp. 1–6. ACM Press, Prague (2013)
 18. Petcu, D., Macariu, G., Panica, S., Crăciun, C.: Portable Cloud applications—From theory to practice. *Future Generation Computer Systems* (Jan 2012)
 19. Ranabahu, A., Maximilien, E.M., Sheth, A.P., Thirunarayan, K.: A domain specific language for enterprise grade cloud-mobile hybrid applications. In: *SPLASH '11 Workshops*. pp. 77–84. ACM Press (Oct 2011)
 20. Satzger, B., Hummer, W., Inzinger, C., Leitner, P., Dustdar, S.: Winds of Change: From Vendor Lock-In to the Meta Cloud. *IEEE Internet Computing* 17(1), 69–73 (Jan 2013)
 21. Sheth, A., Ranabahu, A.: Semantic Modeling for Cloud Computing, Part 1. *IEEE Internet Computing* 14(3), 81–83 (May 2010)
 22. Silva, G.C., Rose, L.M., Calinescu, R.: A Systematic Review of Cloud Lock-In Solutions. In: *IEEE 5th CloudCom*. pp. 363–368. IEEE, Bristol, UK (Dec 2013)

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

Ta'iid 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

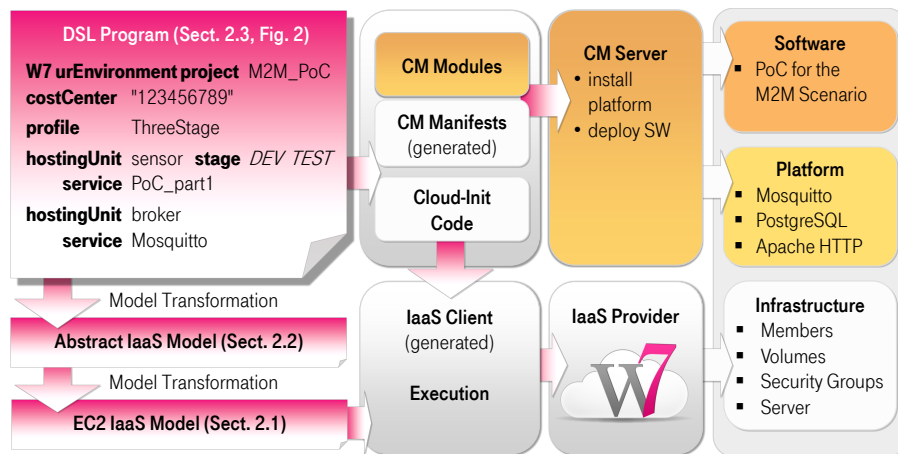


Figure 1: Overview of the Model-based Approach for the Automated Provisioning

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 Puppet⁹ 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-level 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 Mosquitto¹⁰ 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 Security¹². Other `hostingUnits` similarly host other services such as PostgreSQL¹³ 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

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

```

(a) A Customized Cloud Stack

(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: Winery¹⁵ [5] enables users to graphically model service topologies.

AWS CloudFormation¹⁶ 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 Chef¹⁷ (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 JuJu¹⁸: 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 RabbitMQ¹⁹. 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) DiffModel²⁰ 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.: Open-TOSCA - 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., Engelmänn, 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

UML-based Cloud Application Modeling with Libraries, Profiles, and Templates*

Alexander Bergmayr, Javier Troya, Patrick Neubauer,
Manuel Wimmer, and Gerti Kappel

Business Informatics Group, Vienna University of Technology, Austria
{bergmayr,troya,neubauer,wimmer,kappel}@big.tuwien.ac.at

Abstract. Recently, several cloud modeling approaches have emerged. They address the diversity of cloud environments by introducing a considerable set of modeling concepts in terms of novel domain-specific languages. At the same time, general-purpose languages, such as UML, provide modeling concepts to represent software, platform and infrastructure artifacts from different viewpoints where the deployment view is of particular relevance for specifying the distribution of application components on the targeted cloud environments. However, the generic nature of UML's deployment language calls for a cloud-specific extension to capture the plethora of cloud provider offerings at the modeling level. In this paper, we propose the *Cloud Application Modeling Language (CAML)* to facilitate expressing cloud-based deployments directly in UML, which is especially beneficial for migration scenarios where reverse-engineered UML models are tailored towards a selected cloud environment. We discuss *CAML's* realization as a UML internal language that is based on a model library for expressing deployment topologies and a set of profiles for wiring them with cloud provider offerings. Finally, we report on the use of UML templates to contribute application deployments as reusable blueprints and identify conceptual mappings between *CAML* and the recently standardized TOSCA.

Keywords: Cloud Computing, Model-Driven Engineering (MDE), Cloud Modeling, UML, Language Engineering

1 Introduction

Cloud computing has recently emerged as a new possibility how software can be made available to clients as a service. For software vendors, this is appealing as cloud environments [3] have the benefit of low upfront costs compared to a traditional on-premise solution and operational costs that scale with the provisioning and releasing of cloud offerings. They may range from low-level infrastructure elements, such as raw computing nodes, over higher level platforms, such as a Java execution environment on top of a cloud infrastructure, to ready-to-use software deployed on a platform. As a result, current cloud environments are diverse in nature and show various levels of virtualization they operate on. Recent cloud modeling approaches already capture a considerable set of domain-specific concepts to support different scenarios: description of

* This work is co-funded by the European Commission under the ICT Policy Support Programme, grant no. 317859.

cloud-based applications [17] and their deployments [5, 13, 21], optimization of such deployments [14, 18], provisioning of cloud resources [10], or automating the scalability of cloud environments [9, 11]. At the same time, general-purpose languages, such as UML, provide modeling concepts to represent software, platform and infrastructure artifacts from different viewpoints. Hence, providing extensions to UML that satisfy current cloud modeling requirements appears beneficial, especially when cloud-oriented migration scenarios [4] need to be supported where reverse-engineered UML models are tailored towards a selected cloud environment.

However, to date, effective UML-based support for modeling cloud application deployments that are wired with cloud provider offerings is still missing. As a result, on-premise deployments expressed in UML can hardly be turned into cloud-based deployments without neglecting the intended usage of UML. In the ARTIST project [4], we are particularly confronted with this problem as we work towards a model-driven engineering approach for modernizing applications by novel cloud offerings, which involves deploying them or at least some of their components on a cloud environment. Ideally, the design choices of a cloud-based deployment are expressed at the modeling level, which calls for an appropriate language support in the light of UML. While in this way, not only the full expressive power of UML can be exploited, also a seamless integration of cloud-specific models into existing UML models is ensured.

In this paper, we propose the *Cloud Application Modeling Language (CAML)* [7] to enable representing cloud-based deployment topologies directly in UML and refining them with cloud offerings captured by dedicated UML profiles. Thereby, a clear separation is achieved between cloud-provider independent and cloud-provider specific deployment models [1], which is in accordance with the PIM/PSM concept. In our case, the “platform” refers to the cloud provider. We developed profiles for two major cloud providers¹ and integrated them into a common cloud profile. Inspired from common cloud computing literature [2, 3, 12], recent cloud modeling approaches [5, 9, 15, 17, 18, 21] and cloud programming approaches², we developed *CAML*'s model library that facilitates developing base deployment topologies to which cloud offering profiles are applied. The benefits of realizing *CAML* as an internal language of UML are threefold: (i) UML provides a rich base language for the deployment viewpoint, (ii) “cloudifying” UML models is facilitated without the need to re-model existing applications, and (iii) profiles in UML allow hiding details of cloud provider offerings from models and dynamically switching between them by (un-/re-)applying respective cloud provider profiles.

We motivate the practical value of *CAML* by means of a deployment scenario in Section 2. In Section 3, we give the design rationale of *CAML* and provide insights into its model library and the covered UML profiles whereas in Section 4, we discuss the employment of UML templates as reusable deployment blueprints. The operationalization of *CAML* by means of a mapping to the recently accepted TOSCA standard is dedicated to Section 5. Finally, in Section 6 we discuss work related to *CAML* before we conclude in Section 7.

¹ Amazon AWS: <http://aws.amazon.com> and Google Cloud Platform: <http://cloud.google.com>

² Deltacloud: <https://deltacloud.apache.org> and jclouds: <http://jclouds.apache.org>

2 Motivating Deployment Scenario

To motivate the benefits of employing UML as the host language for realizing *CAML*, we give an overview of UML's structural viewpoints that support representing application deployments by means of a reference application³ of the ARTIST project. We take the viewpoint of the application components and their deployment. Figure 1a depicts some components of our application, an excerpt of their realizing classes and the manifestation of these components by deployable artifacts. A possible on-premise deployment for them is presented in Figure 1b. It covers instances of the two deployable artifacts and connects them to a Java-based middleware and a relational DBMS, which are in turn deployed onto a node with specified (virtual) machine characteristics. The model elements of the deployment are instances of the custom types defined in the component viewpoint (see Figure 1a) and the deployment viewpoint (see Figure 1c), respectively. With the emergence of cloud offerings and the demand to exploit them, deployment models need to be expressive enough to capture such offerings. This is exactly the idea of *CAML*. Because it is realized in terms of lightweight extensions to UML, *CAML* models are applicable to UML models and so to our modeled reference application as depicted in Figure 1. In Sections 3 and 4, we present cloud-based deployments for our reference application.

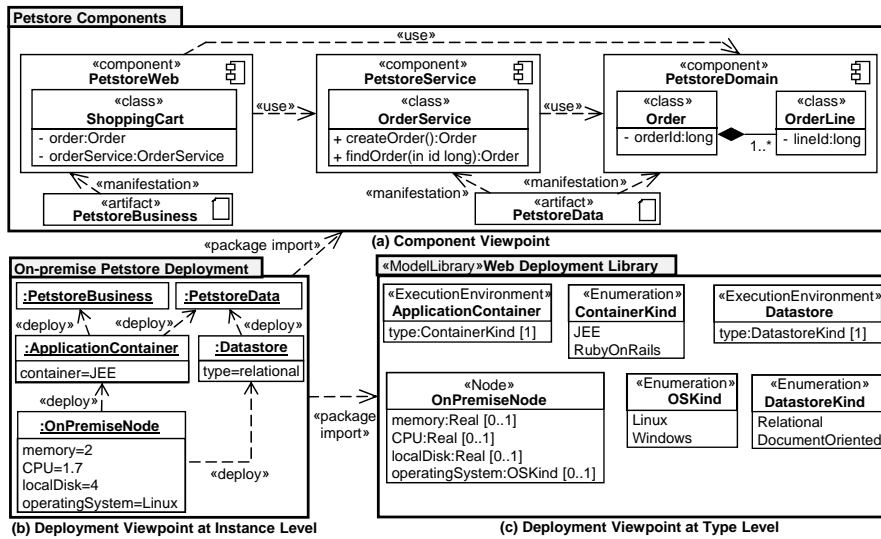


Fig. 1: CAML Use-Case

3 Cloud Application Modeling

With *CAML*, we propose lightweight extensions to UML for modeling cloud application deployments that are seamlessly applicable to UML models, such as component models, typically created throughout software modeling activities. The intended purpose of

³ It is based on the Java Petstore: <http://www.oracle.com/technetwork/java/index-136650.html>

CAML is to express deployment topologies by common cloud modeling concepts and to enable the wiring of such models with concrete cloud provider offerings. This wiring is achieved by applying a dedicated *CAML Profile* to a deployment model expressed in terms of the *CAML Library*. As a result, a clear separation between cloud-provider independent and cloud-provider specific models is achieved. Selecting cloud provider offerings at the modeling level for a concrete deployment becomes a matter of applying the respective stereotypes. The overall set of stereotypes encompass the possible design choices provided by *CAML* regarding cloud provider offerings.

3.1 Model Library for Cloud Deployment Topologies

As presented in Figure 2, the *CAML Library* is built around the concept of *cloud offering*. It is considered as a virtual resource that is expected to be supported by a cloud environment once the wiring with a concrete cloud offering has been performed. More specifically, three offering types capture common cloud environment capabilities. A *cloud node* provides compute capacity and operates at a certain level of virtualization [3]. From an infrastructure-level perspective, cloud nodes come with an operating system, while when turning this perspective to the platform level they also provide middleware, such as a web server and an application container. In case of the latter, the platform is fully managed by a cloud environment. With dedicated scalability strategies, the elastic nature of a cloud environment is managed. For instance, cloud nodes can automatically be acquired depending on the number of incoming requests. Clearly, acquiring and releasing cloud nodes can also be manually controlled. The second offering refers to the *cloud storage* capabilities of cloud environments which provide diverse solutions for structuring application data [12] and increasing their availability by relaxing consistency [22]. Finally, a *cloud service* is considered as a ready-to-use *cloud offering* that is provisioned and managed by a cloud provider. For instance, a load balancer that distributes requests to cloud nodes is an infrastructure-related *cloud service*, while a task queue for long running processes is a platform-related *cloud service*. To represent offering-to-offering connections, *communication channels* are employed while *cloud configurations* enable modifying the assumed conventions of a cloud environment. For instance, an automatic scaling strategy can be configured with boundaries of minimum and maximum running cloud nodes. Generally, instantiated elements of the *CAML Library* are refined to concrete cloud provider offerings via dedicated stereotypes.

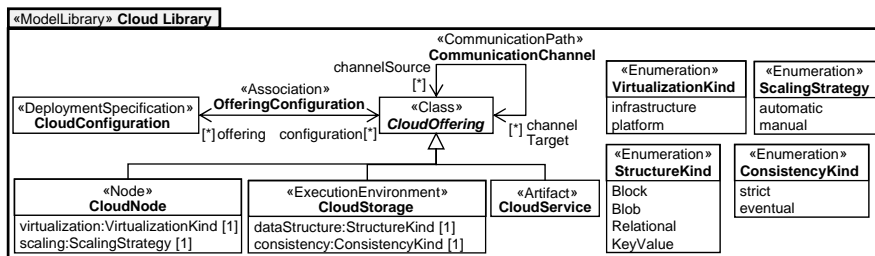


Fig. 2: Cloud Library of CAML

3.2 Profiles for Cloud-Provider Specific Deployments

With *CAML Profiles*, we provide a set of UML stereotypes that enable wiring cloud deployment topologies with concrete offerings of cloud providers. Basically, a stereotype embodies a concrete offering at the modeling level and captures its features in terms of properties. Figure 3 presents some stereotypes specific to the cloud offerings of the Google App Engine (GAE) and Amazon AWS. Common cloud offerings that are shared by both providers are lifted to the *common cloud profile*. Considering *instance types*, they are supposed to be applied to cloud nodes to wire them to a concrete cloud offering, such as a “Frontend Instance” (e.g., *GAEF1*) that hosts a Java-based middleware managed by Google’s App Engine. In turn, cloud offerings are refined by what we call meta-profiles. With the notion of meta-profiles, we facilitate refining them with technical-related details, such as the performance of instance types, and business-related information [8], like the costs of cloud offerings.

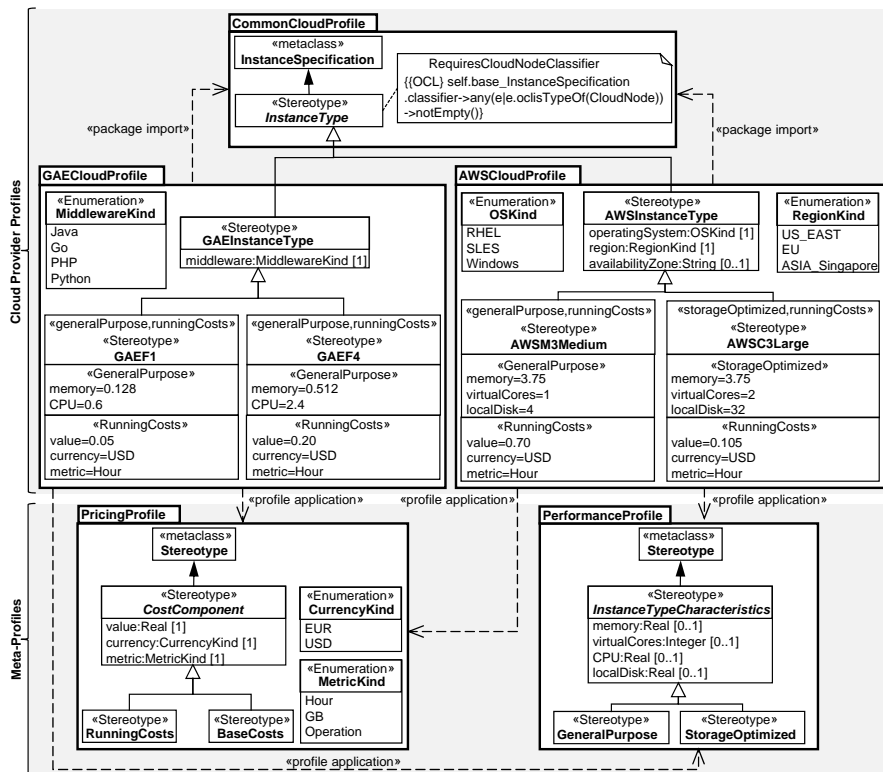


Fig. 3: CAML Profiles and Meta-Profiles

3.3 CAML By-Example

To demonstrate how *CAML* is applied, Figure 4 presents a possible deployment topology and refinement towards a GAE-based cloud deployment of our introduced use case (cf. Figure 1). In a first step, we modeled the deployment topology. It consists of two automatically scaled cloud nodes and a key-value cloud storage for managing the application data in an eventually consistent way. As the cloud nodes are specified as platform-level offering, we directly deployed the application components onto them. Then, in a second step, we applied the GAE profile and the respective stereotypes to refine the deployment model towards concrete cloud offerings provided by the GAE. As a result, the modeled cloud nodes refer to the *F1* and *F4* instance types that host a Java-based middleware. The configuration attached to these cloud nodes constrains the maximum number of idle cloud nodes. Finally, GAE's key-value datastore is employed for the required cloud storage capabilities.

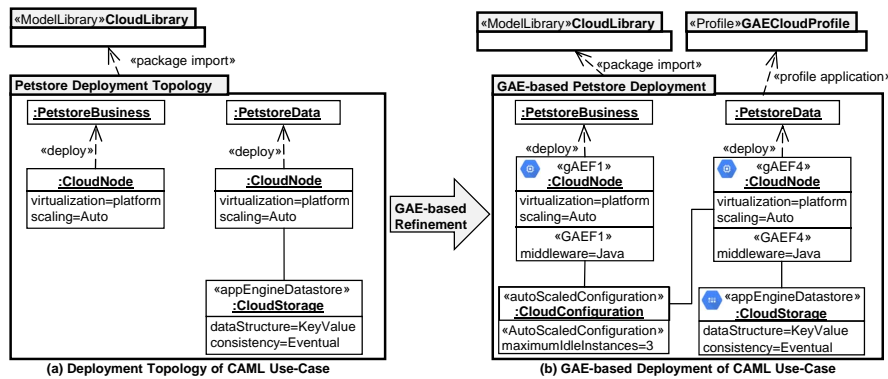


Fig. 4: Reference Application deployed onto Google App Engine

3.4 Prototypical Implementation

To show the feasibility of *CAML*, we have implemented an Eclipse-based prototype, which exploits extension points. In this way, developers can directly use *CAML* in Eclipse tools, such as Papyrus⁴, or access its library and profiles in terms of a resource, which is helpful for the development of transformations. *CAML* together with all artifacts used in this paper are publicly available at our project web site [7]. In addition, together with our industrial partner SparxSystems, we have also implemented a first version of *CAML* for Enterprise Architect⁵. This provides first evidence that our proposed approach for developing a UML internal cloud modeling language based on a library and profiles is feasible and current modeling tools with UML support provide the necessary features to support *CAML* models.

⁴ Papyrus: <http://www.eclipse.org/papyrus>

⁵ Enterprise Architect: <http://www.sparxsystems.at>

4 Reusable Deployment Blueprints as UML Templates

As *CAML* is based on UML, its reuse mechanisms can be applied for cloud application deployments. This is particularly useful for providing frequently occurring deployment patterns as predefined UML templates. To show their usefulness and give first evidence of *CAML's* expressivity, we developed 10 templates as reusable deployment blueprints, most of them are based on Amazon's best practices⁶. We modeled their inherent topology with *CAML's* cloud library and refined them with stereotypes from the cloud profile dedicated to Amazon. The developed blueprints are available at our project website [7]. To demonstrate the use of a blueprint, we show how our reference application is bound to a template, which refers in our case to a 2-tier web architecture [12]. To reuse the predefined template, the deployable artifacts need to be bound to the template parameters. Figure 5 depicts the component viewpoint of our reference application and the respective *CAML* template. It consists of two cloud nodes that refer to the "M3Medium" offering of Amazon. Their location is required to be in Europe while the operation system needs to be Linux. For reliability reasons, they are placed in different availability zones. Requests that arrive at the cloud nodes are first handled by a load balancing service, which enables a higher fault tolerance of the application. The number of running cloud nodes is automatically managed by Amazon as expressed by the scalability strategy. Only the minimum number of running cloud nodes and their adjustment is configured. Both cloud nodes are connected to a cloud storage that in turn is replicated to improve data availability. Finally, as Amazon cloud nodes operate at the infrastructure level, the required middleware for our reference application is defined. In fact, we directly reused it from the on-premise deployment given in Figure 1.

⁶ Amazon Architecture Center: <https://aws.amazon.com/architecture>

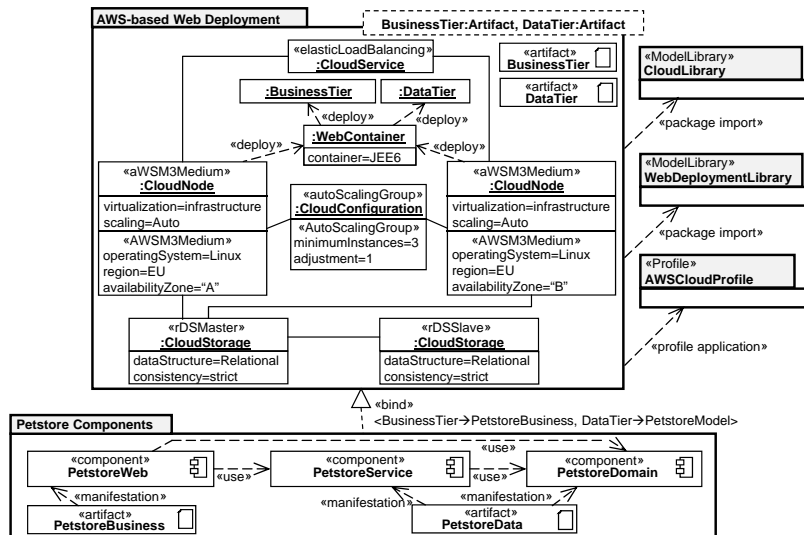


Fig. 5: Reusable Deployment Template for AWS

5 Interoperability between CAML and TOSCA

One major aspect in model-based engineering is to place models as first-class entities in the engineering process. Ideally, they should be turned into executable or interpretable artifacts. Regarding the deployment viewpoint, it appears desirable to translate the respective models into descriptors and scripts that are passed to provisioning engines for cloud environments. For instance, a GAE-based deployment requires specific descriptors for defining the assignment of application modules to a concrete instance type. This assignment can certainly be derived from a *CAML* model. At the same time, there are ongoing efforts in standardizing the representation of cloud-based application deployments. The recently accepted TOSCA standard aims at supporting portable cloud applications. With the notion of management plans, emerging TOSCA-compliant engines are capable to interpret such deployment topologies and initiate the provisioning of defined service templates [5]. Clearly, this is also of practical value for *CAML* models. For that reason, we present an initial mapping between *CAML* and a subset of *TOSCA*. Generally, in *TOSCA*, two modeling concepts are prevalent: *template* and *type*. Templates embody the elements of a deployment topology while types expose the properties and relationships for which concrete values are provided by templates. In this sense, types are considered as reusable entities that can inherit from each other. Figure 6 depicts a concrete *TOSCA* model expressed in *Vino4TOSCA* [6] for an excerpt of our GAE-based application deployment (cf. Figure 4). To represent the *TOSCA* template for the stereotyped *CAML* cloud node, the pertinent *TOSCA* types need to be created: “*CloudNode*” and “*GAEF1*”. The latter is derived from the former as in *TOSCA* a template can only have a single type. Similarly, the deployed application component is represented by a *TOSCA* template. Finally, the deployment relationship type is required for connecting the deployed application component to the cloud node at the template level.

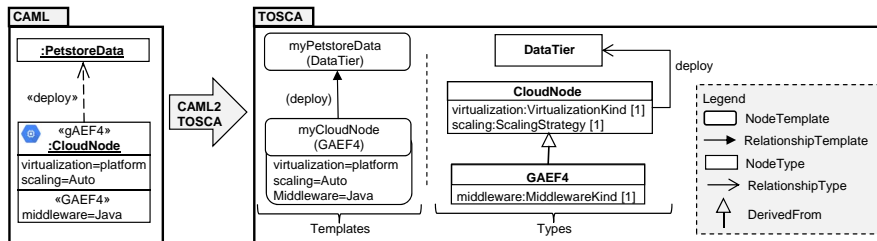


Fig. 6: Mapping between *CAML* and *TOSCA*

6 Related Work

Cloud modeling approaches with the purpose of achieving the wiring of applications with concrete cloud offerings are most closely related to *CAML*. Modeling concepts of these approaches [5, 9, 15, 18, 21] are reflected by *CAML* on a level of abstraction that facilitates to represent design decisions for cloud-based application deployments. As a

result, modeling concepts of these approaches, e.g., required to achieve the optimization of an application deployment (cf., [15]) or to express elasticity rules (cf., [9]), are not completely captured by *CAML*. However, *CAML* enables expressing cloud application deployments that are seamlessly applicable on UML models usually created throughout software modeling activities as it is realized as a UML internal language. As a result, well-connected modeling views on cloud applications from a cloud-provider independent perspective as well as a cloud-provider specific perspective are supported. The refinement of modeling views is enabled by profiles for cloud providers. This additional typing dimension provided by such profiles and the exploitation of a multi-viewpoint language to realize *CAML* differentiates it from existing cloud modeling approaches and the recently standardized TOSCA.

To the best of our knowledge, the only approach providing cloud modeling support within UML is MULTICLAPP [17]. It proposes a UML profile for the purpose of representing components that are expected to be deployed onto a cloud environment by applying cloud-provider independent stereotypes to them. Hence, these stereotypes do not support wiring components with cloud provider offerings, which is different to *CAML* as stereotypes are applied to achieve exactly that wiring.

CloudML-UFPE [16] provides modeling concepts to represent cloud offerings connected with the internal resources of a cloud environment. Similarly to approaches [10, 11, 19], which propose modeling concepts to represent resources internally managed by a cloud environment, the focus is set on the cloud provider perspective. As a result, such modeling approaches support cloud providers to model their environments, which is out of the scope of *CAML*.

Finally, it is worth mentioning that approaches, such as Deltacloud⁷ and jclouds⁸, provide an abstraction layer on top of cloud-provider specific programming libraries. They can be considered as transformation targets for cloud modeling approaches to automate the provisioning of modeled application deployments.

7 Conclusion and Future Work

We have presented *CAML* as a UML internal language based on a library, profiles, and templates. Currently, it is employed by the ARTIST project to model deployments of large applications used in practice. In this respect, cloud providers that operate at both infrastructure level and platform level are targeted. Although the realization and initial evaluation of *CAML* seems promising, several lines of future work need to be investigated. First, we aim for an automated maintenance of provider-specific profiles with, for instance, performance or pricing information based on web information extraction techniques. Second, we intend to provide a simulator for *CAML* to provide prediction about non-functional properties such as costs and performance. In this respect, we plan to explore how FUMML can be employed to provide behavioral semantics for *CAML* in a similar way as we use it to define behavioral semantics for metamodels [20]. Finally, we aim for interoperability with current cloud modeling approaches by providing dedicated transformations or a UML profile.

⁷ <https://deltacloud.apache.org>

⁸ <https://jclouds.apache.org>

References

1. Ardagna, D., Nitto, E.D., Mohagheghi, P., Mosser, S., Ballagny, C., D'Andria, F., Casale, G., Matthews, P., Nechifor, C.S., Petcu, D., Gericke, A., Sheridan, C.: MODAClouds: A Model-Driven Approach for the Design and Execution of Applications on Multiple Clouds. In: MISE Workshop (2012)
2. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I., Zaharia, M.: A View of Cloud Computing. *CACM* 53(4) (2010)
3. Badger, M.L., Grance, T., Patt-Corner, R., Voas, J.M.: Cloud Computing Synopsis and Recommendations. Tech. rep., NIST Computer Security Division (2012)
4. Bergmayr, A., Bruneliere, H., Cánovas Izquierdo, J.L., Gorroñogoitia, J., Kousiouris, G., Kyriazis, D., Langer, P., Menychtas, A., Orue-Echevarria Arrieta, L., Pezuela, C., Wimmer, M.: Migrating Legacy Software to the Cloud with ARTIST. In: CSMR (2013)
5. Binz, T., Breitenbücher, U., Kopp, O., Leymann, F.: TOSCA: Portable Automated Deployment and Management of Cloud Applications. In: Advanced Web Services (2014)
6. Breitenbücher, U., Binz, T., Kopp, O., Leymann, F., Schumm, D.: VINO4TOSCA: A Visual Notation for Application Topologies based on TOSCA. In: OTM (2012)
7. CAML: Project Web Site (2014), <http://code.google.com/a/eclipselabs.org/p/caml>
8. Cardoso, J., Barros, A., May, N., Kyla, U.: Towards a Unified Service Description Language for the Internet of Services: Requirements and First Developments. In: SCC (2010)
9. Chapman, C., Emmerich, W., Márquez, F.G., Clayman, S., Gallis, A.: Software Architecture Definition for On-Demand Cloud Provisioning. *Cluster Comput.* 15 (2012)
10. Chatziprimou, K., Lano, K., Zschaler, S.: Towards a Meta-model of the Cloud Computing Resource Landscape. In: MODELSWARD (2013)
11. Dougherty, B., White, J., Schmidt, D.C.: Model-Driven Auto-Scaling of Green Cloud Computing Infrastructure. *FGCS* 28 (2011)
12. Fehling, C., Leymann, F., Retter, R., Schupeck, W., Arbitter, P.: Cloud Computing Patterns - Fundamentals to Design, Build, and Manage Cloud Applications. Springer (2014)
13. Ferry, N., Rossini, A., Chauvel, F., Morin, B., Solberg, A.: Towards Model-Driven Provisioning, Deployment, Monitoring, and Adaptation of Multi-cloud Systems. In: CLOUD (2013)
14. Frey, S., Fittkau, F., Hasselbring, W.: Search-based Genetic Optimization for Deployment and Reconfiguration of Software in the Cloud. In: ICSE (2013)
15. Frey, S., Hasselbring, W.: The CloudMIG Approach: Model-Based Migration of Software Systems to Cloud-Optimized Applications. *Advances in Software* 4 (2011)
16. Gonçalves, G., Endo, P., Santos, M., Sadok, D., Kelner, J., Merlander, B., Mångs, J.E.: CloudML: An Integrated Language for Resource, Service and Request Description for D-Clouds. In: CloudCom (2011)
17. Guillén, J., Miranda, J., Murillo, J.M., Canal, C.: A UML Profile for Modeling Multicloud Applications. In: ESOC (2013)
18. Leymann, F., Fehling, C., Mietzner, R., Nowak, A., Dustdar, S.: Moving Applications to the Cloud: An Approach Based on Application Model Enrichment. *IJCIS* 20(3) (2011)
19. Liu, D., Zic, J.: Cloud#: A Specification Language for Modeling Cloud. In: CLOUD (2011)
20. Mayerhofer, T., Langer, P., Wimmer, M., Kappel, G.: xMOF: Executable DSMLs based on fUML. In: SLE (2013)
21. Nguyen, D.K., Lelli, F., Taher, Y., Parkin, M., Papazoglou, M.P., van den Heuvel, W.J.: Blueprint Template Support for Engineering Cloud-Based Services. In: ServiceWave (2011)
22. Vogels, W.: Eventually consistent. *CACM* 52(1) (2009)

MDEForge: an extensible Web-based modeling platform*

Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Amleto Di Salle, Ludovico Iovino, and Alfonso Pierantonio

DISIM - University of L'Aquila, Italy
{name.lastname}@univaq.it

Abstract. Model-Driven Engineering (MDE) refers to the systematic use of models as first class entities throughout the software development life cycle. Over the last few years, many MDE technologies have been conceived for developing domain specific modeling languages, and for supporting a wide range of model management activities. However, existing modeling platforms neglect a number of important features that if missed reduce the acceptance and the relevance of MDE in industrial contexts, e.g., the possibility to search and reuse already developed modeling artifacts, and to adopt model management tools as a service. In this paper we propose MDEForge a novel extensible Web-based modeling platform specifically conceived to foster a community-based modeling repository, which underpins the development, analysis and reuse of modeling artifacts. Moreover, it enables the adoption of model management tools as software-as-a-service that can be remotely used without overwhelming the users with intricate and error-prone installation and configuration procedures.

1 Introduction

Model-Driven Engineering (MDE) refers to the systematic use of models as first class entities throughout the software development life cycle. Model-driven approaches shift development focus from code expressed in third generation programming languages to models expressed in domain-specific modeling languages [1]. MDE increases productivity and reduces time to market by enabling the development of complex systems using models defined with concepts that are much less bound to the underlying implementation technology and much closer to the problem domain.

Over the last few years, many MDE technologies have been conceived for developing domain specific modeling languages, and for supporting a wide range of model management activities. The relevance of MDE is evidenced also by the increasing interest in many scientific endeavours, active technology projects, and numerous industrial projects ranging from direct applications of MDE concepts and tools, to those developing its foundations [2]. Even though existing MDE technologies provide practitioners with facilities that can simplify and automate many steps of model-based development processes, empirical studies show that some barriers still exist for the wider adoption of MDE technologies [3]. Among the main issues that currently hamper a wide adoption of MDE there are at least the following:

* This research was supported by the EU through the Model-Based Social Learning for Public Administrations (Learn Pad) FP7 project (619583).

- the support for discovery and reuse of existing modeling artefacts is very limited. As a result, similar transformations and other model management tools often need to be developed from scratch, thus raising the upfront investment and compromising the productivity benefits of model-based processes. For instance, when modelers identify a need for a domain-specific modeling language, it is quite common to implement it from scratch instead of reusing already developed languages that might satisfy their requirements;
- modelling and model management tools are commonly distributed as software packages that need to be downloaded and installed on client machines, and often on top of complex software development IDEs (e.g. Eclipse).

In this paper we propose MDEForge, an extensible modeling framework specifically conceived to address the issues previously mentioned. In particular, MDEForge consists of a set of core services that permit to store and manage typical modeling artefacts and tools. Atop of such services it is possible to develop extensions adding new functionalities to the platform. All the services can be used by means of a Web access and by a REST API that permits to adopt the available model management tools as software-as-a-service.

The paper is structured as follows: Section 2 discusses the motivations of the paper by considering already existing works. Section 3 presents the architecture of MDEForge and makes an overview of its main components. Two extensions of the platform are presented in Section 4. Section 5 concludes the paper and discusses some research perspectives.

2 Background and Motivation

The artefacts and tools that are typically involved when applying MDE approaches are those shown in Fig. 1: *editors* are used to create *models* that in turn are manipulated to generate other models or even *code*. *Model repositories* are employed to enable the re-use of already specified models. The whole ecosystem is developed according to different kinds of relations (e.g., conformance) with corresponding *metamodels*.

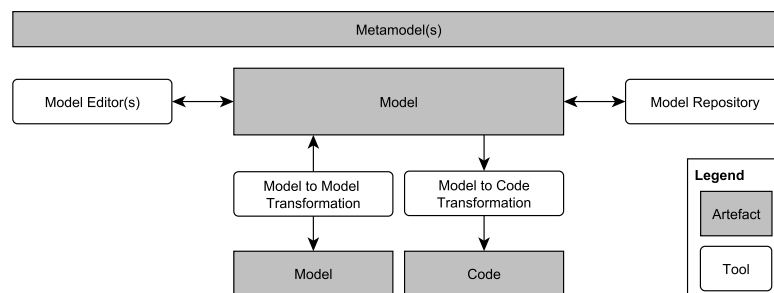


Fig. 1. Main MDE artefacts and tools

	Managed Artefact	Main purpose	Typical deployment scenario
AMOR [5]	Model	Model versioning	Desktop application
Bizycle [6]	Model	Integration of software components	Desktop application
CDO	Model	Storage	Client-Server application
EMFStore [7]	Model	Model versioning	Client-Server application
GME [8]	Model	Storage	Client-Server application
ModelBus [9]	Model	Model versioning	Client-Server application
Morse [10]	Model	Model versioning	Software-as-a-service
ReMoDD [11]	Any	Documentation	Web-based interaction

Table 1. Overview of existing MDE tools providing storage features

Even though existing modeling platforms (e.g., EMF [4]) provide developers and users with the tools shown in Fig. 1 they neglect a number of important features that if missed the acceptance and the relevance of MDE in industrial contexts might be reduced. In particular, in this paper we focus on the limited support for the re-use of already developed modeling artefacts¹ and for enabling the adoption of modeling tools as-a-service. To this end in this section we overview existing works that are related to MDEForge concerning the re-use of modeling artefacts (Sect. 2.1), and the possibility to use model management tools as a service (Sect. 2.2).

2.1 Reuse of modeling artefacts

In this section we discuss state-of-the-art approaches (see Tab. 1) for providing repositories of modeling artefacts, and outline outstanding research challenges for achieving a comprehensive solution to the problem of properly managing the persistence of models and the deployment and discovery of any kind of model management tools to enable their reuse and refinement.

AMOR - Adaptable Model Versioning [5]: it is an attempt to leverage version control systems in the area of MDE. AMOR supports model conflict detection, and focuses on intelligent conflict resolution by providing techniques for the representation of conflicting modifications as well as suggesting appropriate resolution strategies.

Bizycle [6]: it is a project aiming at supporting the automated integration of software components by means of model-driven techniques and tools. Among the different components of the project, a metadata repository is also provided in order to manage and store all the artefacts required for and generated during integration processes, i.e, external and internal documentation, models and metamodels, transformation rules, generated code, users and roles.

*CDO*²: it is a pure Java model repository for EMF models and meta models. CDO can also serve as a persistence and distribution framework for EMF-based application systems. CDO supports different kinds of deployments such as embedded repositories, offline clones and replicated clusters. However, the typical deployment scenario consists

¹ Hereafter with the terms *modeling artefacts* we include also *modeling tools*.

² <http://www.eclipse.org/cdo/>

of a server managing the persistence of the models by exploiting all kinds of database backends (like major relational databases or NoSQL databases), and an EMF client application.

EMFStore [7]: it is a software configuration management system tailored to the specific requirements of versioning models. It is based on the Eclipse Modeling Framework and it is an implementation of a generic operation-based version control system. EMFStore implements, in the modeling domain, the typical operations implemented by SVN, CVS, Git for text-based artefacts, i.e., change tracking, conflict detection, merging and versioning. It consists of a server and a client component. The server runs standalone and provides a repository for models including versioning, persistence and access control. The client component is usually integrated into an application and is responsible for tracking changes on the model, and for committing, updating and merging.

GME - Generic Modeling Environment [8]: it is a set of tools supporting the creation of domain specific modeling languages and code generation environments. A repository layer is also provided to store the developed models. Currently, MS Repository (an object oriented layer on top of MS SQL Server or MS Access) and a proprietary binary file format are supported.

ModelBus [9]: it consists of a central bus-like communication infrastructure, a number of core services and a set of additional management tools. Depending on the usage scenario at hand, different development tools can be connected to the bus via tool adapters. Once a tool has been successfully plugged in, its functionality immediately becomes available to others as a service. Alternatively, it can make use of services already present on the ModelBus. Among the available services, ModelBus also includes a built-in model repository, which is able to version models, supports the partial check-out of models and coordinates the merging of model versions and model fragments;

Morse - Model-Aware Repository and Service Environment [10]: it is a service-based environment for the storage and retrieval of models and model-instances at both design- and run-time. Models, and model elements are identified by Universally Unique Identifiers (UUID) and stored and managed in the Morse repository. The Morse repository provides versioning capabilities so that models can be manipulated at runtime and new and old versions of the models can be maintained in parallel;

ReMoDD - Repository for Model-Driven Development [11]: it is a repository of artefacts aiming at improving MDE research and industrial productivity, and learning experience of MDE students. By means of a Drupal Web application, users can contribute MDE case studies, examples of models, metamodels, model transformations, descriptions of modeling practices and experience, and modeling exercises and problems that can be used to develop classroom assignments and projects. Searching and browsing facilities are enabled by a Web-based user interface that also provides community-oriented features such as discussion groups and a forum.

According to Tab. 1 the majority of existing approaches focus on providing support for the persistence of models. Only ReMoDD supports other kinds of modeling artefacts, like transformations, and metamodels. However, the main goal of ReMoDD is to support learning activities by providing documentation for each stored artefact. Consequently, ReMoDD cannot be used to programmatically retrieve artefacts from the

repository or more generally cannot be adopted as software-as-a-service to search and reuse already existing modeling artefacts. Most of the discussed approaches require local installation and configuration. Only ReMoDD and Morse do not require to be installed locally. In particular, the modeling artefacts stored in ReMoDD can be searched and browsed through a Web-based application. Morse provides developers with the possibility to use it as a service.

2.2 Model management tools as service

The motivation of our work is shared also in [12] where authors propose the Modeling as a Service (MaaS) initiative as an approach to deploy and execute model-driven services over the Internet. This initiative is aligned with SaaS principles, since consumers do not manage the underlying cloud infrastructure and deal mostly with end-user systems. Interestingly, in [13] authors investigate the problem of transforming very large models in the Cloud by addressing two phases: i) model storage, and ii) model transformations execution in the Cloud. For both aspect authors identify a set of research questions, and possible solutions.

Even though there are different attempts [14] and projects (e.g., the EU MONDO project³) related to the adoption of Cloud infrastructures to apply MDE, the area is still mostly unexplored. In line with the MaaS initiative we want to contribute to this research area with an extensible modeling framework that enables the adoption of model management and analysis tools as service. As discussed later in the paper the framework is at its early stages and we have not addressed yet aspects like workload and capacity management that are typical in Cloud computing, however the results we have obtained so far are promising.

3 Overview of the MDEForge platform

In this section we present the MDEForge platform that has been conceived to overcome the issues discussed in the previous section. In particular MDEForge aims at:

- providing a community-based modeling repository, which underpins the development, analysis and reuse of any kinds of modeling artifacts not limited to only models;
- supporting advanced mechanisms to query the repository and find the required modeling artifacts;
- enabling the adoption of model management tools as software-as-a-service;
- being modular and extendible;

As shown in Fig. 2 the MDEForge platform consists of a number of services that can be used by means of both a Web access and programmatic interfaces (API) that enable their adoption as software as a service. In particular, *core* services are provided to enable the management of modeling artifacts, namely transformations, models, meta-models, and editors. Atop of such core services, extensions can be developed to add new functionalities. For instance, by exploiting the transformation and metamodel services

³ <http://www.mondo-project.org/>

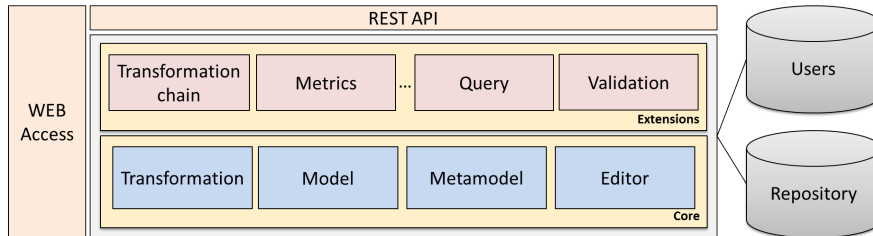


Fig. 2. Architecture of MDEForge

it is possible to extend the platform by adding a new service to enable the automated chaining of model transformations as discussed in Section 4.

We envision different kinds of users of the MDEForge platform and in our opinion they can be at least the following:

- *Developers of modeling artifacts:* as previously said we envision a community of users that might want to share their tools and enable their adoption and refinement by other users. To this end the platform provides the means to add new modeling artifacts to the MDEForge repository;
- *Developers of MDEForge extensions:* one of the requirements we identified when we started the development of MDEForge is about the modularity and extensibility of the platform. To this end we identified a set of core services that can be used to add new functionalities by means of platform extensions. In this respect, experienced users might contribute by proposing new extensions to be included in the platform;
- *End-users:* a Web application enables end-users to search and use (meta)models, transformations, and editors available in the MDEForge repository. Experienced users might use the REST API to exploit the functionalities provided by the platform in a programmatic way. For instance, tool vendors might exploit the functionalities provided by their tools by exploiting some of the transformations available in the MDEForge repository.

In the remainder of this section we give some details about the MDEForge repository (Section 3.1) and the available core services (Section 3.2).

3.1 The MDEForge Repository

The *Repository* component plays a key role in the MDEForge platform and it has been developed in order to store artifacts according to the metamodel shown in Fig. 3. In particular, the repository has been developed with the aim of managing any kinds of modeling artifacts (see the metaclass *Artifact* in Fig. 3). Each artifact refers to the corresponding type, e.g., model, transformation, metamodel, etc. The specification of the relation between a given artifact and the corresponding type is done by means of the *Relation* elements. In turn, each relation is typed by means of a corresponding *RelationType* element. By means of such modeling constructs it is possible e.g., to specify the *conformsTo* relation between a model *m1* and the corresponding metamodel *MM1* as shown in Fig. 4. Similarly, it is possible to specify any kinds of modeling

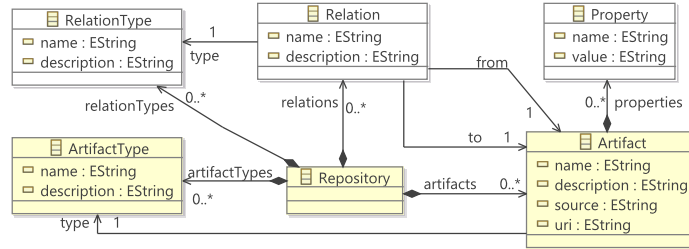


Fig. 3. Fragment of the MDEFForge Repository Metamodel

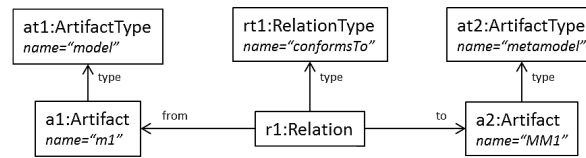


Fig. 4. Simple content of the Repository

elements together with their relations. For example, it is possible to represent also the execution engine of a given model transformation stored in the repository.

It is important to remark that all the artifacts stored and managed by the MDEFForge platform are related to the users that created them. The system permits also to make artifacts public, private, or limit their visibility to specific users.

3.2 The MDEFForge Core

In the following the core service previously mentioned and shown Fig. 5 are described. *Model Service*: it manages models in the repository, thus it permits users to upload, download, delete models, and even search models conforming to a specific metamodel; *Metamodel Service*: it provides the means to upload, download, delete metamodels, and find metamodels by a specific Universal Resource Identified (URI); *Transformation Service*: it manages the transformations in the repository, i.e., it permits to upload, download, delete transformations. Moreover, it permits to remotely execute

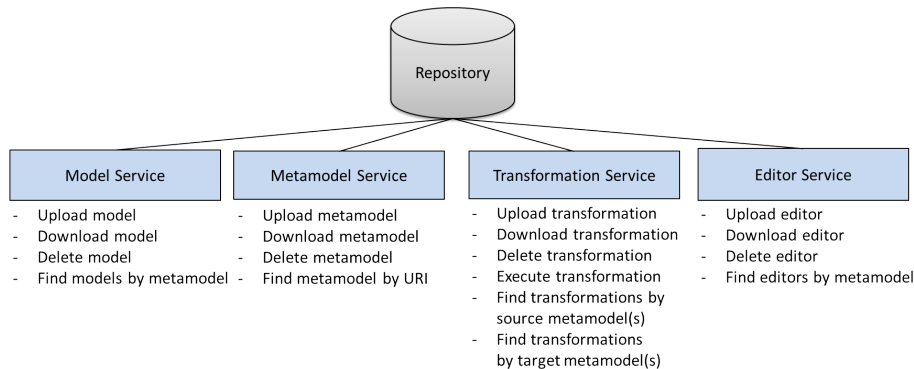


Fig. 5. Overview of the MDEFForge Core services

transformations, or even find transformations by specifying the source and/or target metamodel(s);

Editor Service: it permits to upload, download, or delete editors from the repository. It is important to remark that at this stage of the project we manage the JARs containing the implementation of a given editor (e.g., developed by means of GMF [15]). Consequently, the editor service permits users to upload the implementation code of a given repository to enable other users to find, download, and in case install them locally. In other words, at this stage of the project we do not refer to on-line and collaborative editors even though this represents an important extension of the platform that we intend to investigate in the near future.

A prototypical implementation of the MDEForge has been developed⁴ and even though it is still at an early stage, we managed to apply the platform to support two interesting problems: chaining model transformations, and measuring metamodels as discussed in the next section.

4 Examples of MDEForge extensions

In this section we present two examples of extensions we have developed atop of the core services of the platform and that have been successfully applied to deal with the problems of chaining of model transformations (Section 4.1) and to calculate metrics of metamodels to support the understanding of their characteristics (Section 4.2).

4.1 Automated chaining of model transformations

In [16] we have addressed the problem of automatically composing model transformations. In particular, we have proposed an approach to automatically discover and compose transformations: developers provide the system with the source models and specify the target metamodel. Then, by relying on the MDEForge repository, all the possible transformation chains are calculated. Importantly, in case of incompatible intermediate target and source metamodels, proper adapters are automatically generated in order to chain also transformations that otherwise would be discarded by limiting the reuse possibilities of available transformations.

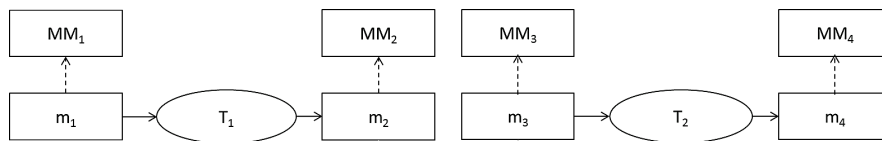


Fig. 6. Model transformation chain example

Figure 6 shows an explanatory model transformation chain. In particular, T_1 is a model transformation that generates models conforming to the target metamodel MM_2 from models conforming to MM_1 . Additionally, T_2 is a model transformation that generates models conforming to MM_4 from models conforming to the source metamodel MM_3 . In general, if the input metamodel of T_2 would be also the output metamodel of

⁴ www.mdeforge.org

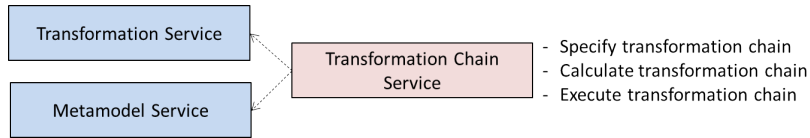


Fig. 7. Transformation Chain extension

T_1 , then these two transformations could be chained. However, under certain conditions, two transformations can be chained even though the output metamodel of the first transformation does not correspond to the input metamodel of the second transformation.

To support the techniques presented [16] we have developed the extension shown in Fig. 7 consisting of the *Transformation Chain* service that makes use of the *Transformation* and *Metamodel* core services discussed in the previous section. In particular, users specify the model to be transformed, the target metamodel by means of the *Specify transformation chain* operation. Such an input is used to calculate the transformation chains that subsequently can be executed.

4.2 Measuring metamodels

In [17] we have developed an approach to measure metamodels with the aim of understanding their typical characteristics by investigating the correlations of different metrics applied on a corpus of more than 450 metamodels. In particular, we proposed an approach for *a*) measuring certain metamodeling aspects (e.g., abstraction, inheritance, and composition) that modelers typically use; and *b*) for revealing what are the common characteristics in metamodeling that can increase the complexity of metamodels hampering their adoption and evolution in modeling ecosystems.

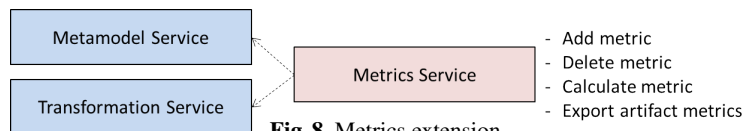


Fig. 8. Metrics extension

To perform such analysis we have developed an extension of the MDEForge platform to support the calculation of metrics as shown in Fig. 8. In [17] we discussed metrics calculated only on metamodels, however we are working also on the identification of possible correlations among transformation and metamodel characteristics. Finally, to enable the management of the calculated metrics, the added service permit to export CSV files encoding the values of all the calculated metrics. Generating CSV files enables the adoption of statistical tools like IBM SPSS, Microsoft Excel, and Libreoffice Calc for subsequent analysis of the generated data.

5 Conclusion and future works

In this paper we presented MDEForge, an extensible modeling framework supporting the creation of a community-based modeling repository, which underpins the development, analysis and reuse of modeling artifacts. The platform consists of core service that can be extended and all of them are remotely available as software as a service thus users are not overwhelmed with intricate and error-prone installation and configuration procedures. Two concrete extensions and applications of the platform have

been presented. As future work we intend to investigate issues that are typical in Cloud computing, e.g., scalability of the platform, and workload management. Moreover, we intend to implement further extensions for instance to support advanced queries on the repository. To this end we intend to investigate the integration of tools that have been recently proposed [18]. Moreover, we plan to extend the platform by adding services enabling collaborative modeling activities.

References

1. Schmidt, D.C.: Guest NOOPeditor's Introduction: Model-Driven Engineering. *Computer* **39** (2006) 25–31
2. Di Ruscio, D., Paige, R.F., Pierantonio, A.: Guest editorial to the special issue on Success Stories in Model Driven Engineering. *Science of Computer Progr.* (2014)
3. Whittle, J., Hutchinson, J., Rouncefield, M., Burden, H., Heldal, R.: Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem? In: *MODELS*. Volume 8107 of LNCS. Springer Berlin Heidelberg (2013) 1–17
4. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: *Eclipse Modeling Framework*. Addison Wesley (2003)
5. Brosch, P., Langer, P., Seidl, M., Wimmer, M.: Towards End-user Adaptable Model Versioning: The By-Example Operation Recorder. In: *Procs.of CVSM '09*, Washington, DC, USA, IEEE Computer Society (2009) 55–60
6. Kutsche, R., Milanovic, N., Bauhoff, G., Baum, T., Cartsburg, M., Kumpe, D., Widiker, J.: BIZYCLE: Model-based Interoperability Platform for Software and Data Integration. In: *Procs.of the MDTPi at ECMDA*. (2008)
7. Koegel, M., Helming, J.: EMFStore: a model repository for EMF models. In: *Software Engineering, 2010 ACM/IEEE 32nd Int. Conf. on*. Volume 2. (2010) 307–308
8. Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., Volgyesi, P.: The Generic Modeling Environment. In: *Workshop on Intelligent Signal Processing*. (2001)
9. Hein, C., Ritter, T., Wagner, M.: Model-driven tool integration with ModelBus. *Workshop Future Trends of Model-Driven* (2009)
10. Holmes, T., Zdun, U., Dustdar, S.: Automating the Management and Versioning of Service Models at Runtime to Support Service Monitoring. In: *EDOC*. (2012) 211–218
11. France, R., Bieman, J., Cheng, B.: Repository for Model Driven Development (ReMoDD). In: *Models in Software Engineering*. Volume 4364 of LNCS. Springer Berlin Heidelberg (2007) 311–317
12. Brunelière, H., Cabot, J., Jouault, F.: Combining Model-Driven Engineering and Cloud Computing. In: *MDA4ServiceCloud'10 Workshop co-located with ECMFA*. (2010)
13. Clasen, C., Didonet Del Fabro, M., Tisi, M.: Transforming Very Large Models in the Cloud: a Research Roadmap. In: *First International Workshop on Model-Driven Engineering on and for the Cloud*, Copenhagen, Denmark, Springer (2012)
14. Paige, R., Cabot, J., Brambilla, M., Chechik, M., Mohagheghi, P.: *Procs. of CloudMDE - First Workshop on MDE for and in the Cloud*. (2012)
15. Eclipse: Graphical Modeling Framework. <http://www.eclipse.org/gmf/> (2014)
16. Basciani, F., Di Ruscio, D., Iovino, L., Pierantonio, A.: Automated Chaining of Model Transformations with Incompatible Metamodels. In: *Procs. MODELS 2014 Accepted*. (2014)
17. Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: Mining metrics for understanding metamodel characteristics. In: *MiSE 2014 - ICSE Workshop*. (2014)
18. Szárnyas, G., Izsó, B., Ráth, I., Harmath, D., Bergmann, G., Varró, D.: IncQuery-D: A Distributed Incremental Model Query Framework in the Cloud. In: *Procs. MODELS 2014*, Valencia, Spain, Springer, Springer (2014) Accepted.

