Refactoring Architecture Models for Compliance with Custom Requirements

Ta'id Holmes
Technology & Innovation, Deutsche Telekom AG
Darmstadt, Germany
t.holmes@telekom.de

Uwe Zdun
Faculty of Computer Science, University of Vienna
Vienna, Austria
uwe.zdun@univie.ac.at

ABSTRACT

In the process of software-intensive systems engineering, architectures need to be designed that are compliant to the requirements. For this, architects need to examine those requirements with regard to their architectural impact. Accessing and interpreting the requirements is however not always possible, for instance if custom requirements are yet unknown at the time when the architecture is modeled. Ideally, architectural knowledge as derived from custom requirements could be imposed upon architecture models. This paper proposes a novel concept for automated refactoring of architecture models in order to meet such requirements by formalizing architectural knowledge using model verification and model transformations. Industrial application within a telecommunications service provider is demonstrated in the domain of cloud application orchestration: service providers are enabled to autonomously customize solutions predefined by vendors according to their own internal requirements.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Computing methodologies → Modeling methodologies**; *Model verification and validation*; • **Software and its engineering → Requirements analysis**; **Model-driven software engineering**; *Domain specific languages*; *Orchestration languages*; *Constraint and logic languages*; • **Security and privacy** → *Firewalls*;

ACM Reference Format:
Ta'id Holmes and Uwe Zdun. 2018. Refactoring Architecture Models for Compliance with Custom Requirements. In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18), October 14–19, 2018, Copenhagen, Denmark.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3239372.3239379

1 INTRODUCTION

In the process of engineering of software-intensive systems various design decisions have to be taken. Generally, these are influenced by functional or non-functional requirements of customers. Sometimes design decisions are also based on considerations of engineers regarding qualitative, non-functional properties of the aspired solution such as maintainability or flexibility. Often, the design of a solution is manifested in an architecture model (e.g., software architecture in software engineering, business process in business process management, or cloud architecture in cloud computing). The architecture model, thus, has to comply with the requirements. Architects, e.g., cloud, enterprise, software, or systems architects, need to examine requirements towards their architectural impact. In this process they need to access and interpret requirements, acquire architectural knowledge, and use such for deriving compliant architectures.

In case new requirements arise and architectural changes are required, architectures have to be reevaluated and revised if necessary. As architectural changes are costly and encompass many risks, they should be avoided. For a vendor of a solution (like a service vendor) that is used by many different customers (like service providers) this means that it is crucial to know about the requirements of a customer beforehand. Incorporating custom requirements drives costs and prolongates delivery times. Also it implies the need for a customer to disclose (internal) requirements. Finally, custom requirements may emerge at a later point in time and are subject to change. For mastering complexity, many engineering domains and methodologies utilize models as central artifacts. Here, we focus on models that reflect architectural aspects, especially software architecture and cloud architecture models.

Figure 1 depicts the key concepts with regard to the context of this work. From a customer (e.g., a service provider) point of view, the architecture of a solution (e.g., a cloud application) has to comply to certain requirements (e.g., originating from security policies or architectural guidelines). That is, the requirements have implications towards the architecture of the solutions. Therefore, architectural knowledge can be derived from such requirements. When such knowledge is incorporated into the architecture, compliance with the requirements can be established.

While this can be performed on a case-by-case basis manually, an automated approach can be achieved by formalizing appropriate model transformations. In general, model transformations automate the enrichment, refactoring, or translation of models. Here, the model transformations capture the architectural knowledge as derived from requirements for refactoring models. Refactoring in this context means that the model transformations only introduce changes to the internal structure of software to improve certain software quality characteristics without changing its observable behavior (cf., [9, 22]). For automating such refactoring, first, relevant sources for this architectural knowledge need to be identified in the requirements; that is, those requirements that have an impact

MODELS '18, October 14–19, 2018, Copenhagen, Denmark

ACM ISBN 978-1-4503-4949-9/18/10…$15.00
https://doi.org/10.1145/3239372.3239379

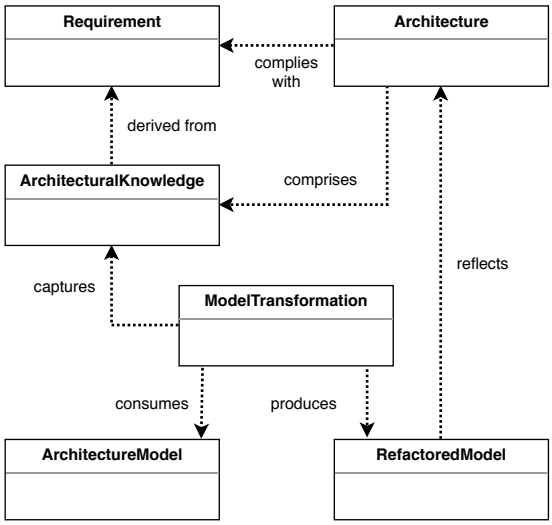

Figure 1: Architectural Compliance through Refactorings

on the architecture of a solution need to be identified. Next, architectural knowledge has to be derived from requirements. Finally, this architectural knowledge needs to be expressed and formalized in terms of model verification and model transformations.

The idea of this paper, thus, is to exploit model transformations to impose architectural knowledge as derived from custom requirements upon models. This is demonstrated in the context of cloud computing using cases from an industrial service provider. As a result, service providers are enabled to customize solutions independently according to internal requirements. Other benefits are that custom requirements do not have to be disclosed and the change impact is minimized. The remainder is structured as follows: The next section gives an introduction into the application domain of cloud application orchestration and motivates the work by highlighting and illustrating the problems. Next, Sect. 3 presents a number of industrial use cases. Section 4 discusses on the benefits, risks, and limitations of our refactoring approach. Related work is compared in Sect. 5 and Sect. 6 concludes.

2 AUTOMATING CLOUD APPLICATION ORCHESTRATION MODEL REFACTORING

In this paper, we study the general problem explained in Sect. 1 in the context of the specific domain of provisioning in the cloud, derived from industrial use cases such as the one discussed in Sect. 3, using models that broadly can be categorized as cloud and software architecture models[1]. In particular, orchestration models[2] of cloud applications are used for automating the provisioning. As such, they need to reflect the capabilities and requirements of the internal services and their relationships in form of service topologies. Architectural knowledge as derived from security policies and architectural guidelines needs to be incorporated into respective models as well. Such requirements are often specific to service providers. Therefore, there is a need for customizing applications. This hinders development and deployment of off-the-shelf cloud applications (and, in the context of telecommunication, virtual network functions (VNFs) operated in the cloud). Worse, in case of changing requirements, e.g., a new security policy, the impact goes beyond one stakeholder involving the necessity for time-consuming and costly alignment between a service provider and multiple vendors of cloud applications.

2.1 Background and Motivation

In order to profit from the promises of the cloud computing paradigm and for exploiting capabilities offered by a cloud platform, various properties are expected from a modern, cloud-native (cf. [7]) application. Among others, it should make use of available building blocks, expose highly-available services, and scale horizontally. For achieving these objectives it is necessary to decompose and distribute the application (cf. Move-to-Cloud problem [20]). In case of a Web application, e.g., the various tiers and services such as database, application, and web servers are decoupled and placed in separate (virtual) server instances. For the provisioning of resources as required by the application (e.g., compute, storage, and network), respective requirements need to be formulated and communicated to the platform. For the former a cloud modeling or orchestration language can be used (cf. [3]). It provides a way to express an orchestration model. This model, comprising the service topology of the application, can be consumed by an orchestration engine and is transformed into a deployment plan. Cloud application orchestration (AO), thus, automates the provisioning process of applications based on an orchestration model[3].

In the past, much emphasis, from both research and industry, has been placed on making cloud applications portable (cf. [6]), e.g., for avoiding vendor lock-in of a cloud platform (provider). Being agnostic to the later may realize technical portability; from a business point of view, however, this is not sufficient for permitting applicability of the very same application across a market. The reason for this is that – in addition to varying target cloud platforms that often still require additional portability effort – service providers have individual established policies, rules, and guidelines. These result in requirements towards applications and in particular their orchestration models (the latter is in focus of this work). A security guideline, e.g., may demand that firewalls must protect services which are not critical with regard to latency. For this packet filters may be mandated but in some cases application firewalls with deep packet inspection such as Web application firewalls must be deployed. The objective may be to protect services exposed to the public Internet; in other cases, also internal networks and hosts shall be protected from each others. Regardless of the various rules, the orchestration model has to reflect a compliant architecture in form of a service topology.

This example already indicates the complexity problem that a vendor of a potential off-the-shelf cloud application has to face as the application has to be customized or tailored for service providers.

[1]Please see Sect. 4 for a discussion about the generalizability of the results to other kinds of models and domains.

[2]Cloud application orchestration models are referred to as *orchestration models* in this paper.

[3]For simplicity this paper focuses on the conceptual level, i.e., orchestration models (conforming to an abstract domain-specific language (DSL)) and does not examine artifacts such as a template or a VNF descriptor (written in a concrete DSL).

Fulfilling the various mandatory requirements of a service provider individually introduces a dependency within the engineering process, negatively impacts the time to market, and last but not least is expensive from a financial point of view. In case of changing service provider requirements, multiple applications originating from various vendors are generally impacted. This calls for alignments and maintenance – and as a result it generates work and costs.

The orchestration model, as a matter of fact, is tightly coupled with different requirements through architectural knowledge. Partly, such knowledge is the result of the decomposition of the cloud application; to another part, as described, it derives from individual policies, rules, and guidelines of a service provider. An orchestration model – as effectively used for the deployment – needs to comprise all of the architectural knowledge for being compliant with all of the requirements irrespectively of their backgrounds.

Instead of developing the (effective) orchestration model upfront, this paper rather proposes automated model refactoring: An (initial) orchestration model, for instance as the result of a decomposition of the cloud application, is refactored according to additional requirements at a later stage. For realizing this, architectural knowledge is derived from custom requirements and expressed in terms of model verification and model transformations. Architectural knowledge is then injected into a model during the delivery process of a cloud application. As a result, a cloud application can be described in terms of an orchestration model by a vendor independently from a service provider. Also, a service provider can perform refactoring autonomously without the need to disclose internal requirements or changes of such.

2.2 Delivery Process of Cloud Applications

The delivery process of a cloud application generally comprises the following stakeholders: 1) an independent software vendor (ISV) that develops, tests, and maintains the application; 2) a cloud platform provider that provides an infrastructure as a service (IaaS) or platform as a service (PaaS) operating datacenters for hosting (software as a service (SaaS)) applications; 3) a service provider that purchases the application from the vendor, onboards it onto and operates it in the cloud; and finally, 4) end-customers that make use of the application.

In a telecommunication context this delivery process also applies to virtual network functions (VNFs) operated in the cloud and subscribers. Examples of VNFs are various router services such as Dynamic Host Configuration Protocol (DHCP)[4], Domain Name System (DNS)[5], and Address Family Transition Router (AFTR)[6] servers as well as more complex applications such as Evolved Packet Core (EPC)[7] and IP Multimedia Subsystem (IMS)[8].

Telecommunications service providers (TSPs) operate VNFs as part of their (core) network. For this a cloud platform in terms of an IaaS solution can be deployed. In this case, a TSP performs both the roles of the platform as well as of the service provider.

Replacing physical appliances, VNFs are developed by different ISVs for the TSP market. From this point of view common, cloud-based VNFs can be regarded as specialized off-the-shelf cloud applications. Although the functionality of a VNF as deployed at different TSPs may be identical, practically, it requires customization and tailoring towards the TSPs, however. This is because of individual requirements of respective TSPs that need to be met and that impact the architecture of applications. As a result, development of a VNF for the TSP market becomes difficult as not all different variability can be foreseen by vendors. Last but not least, the need to customize and tailor applications drives costs. Thus, this work is motivated by the necessity to customize cloud applications in general and VNFs in particular according to custom requirements. Central in this respect is the orchestration model that describes an application's required resources, services, and their relationships. Thus, AO and AO languages are described next.

2.3 Cloud Application Orchestration and Configuration Management

A cloud application orchestrator realizes the provisioning of an application's required infrastructure resources as well as deployment of the application itself. For this, an orchestration model is consumed and processed by an AO engine. Such a model can be described with cloud modeling languages (cf. [3]). Some examples are Cloud Application Modeling Language [4] (CAML), CloudML [8, 13], Heat Orchestration Template (HOT)[9] DSL, and the OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) [6]. TOSCA provides a metamodel for describing the service topology and orchestration of cloud applications and also defines Simple Profiles in YAML [25] and for Network Functions Virtualization (NFV) [24]. The latter, particularly relevant for TSPs, uses concepts from the European Telecommunications Standards Institute (ETSI)[10] while mapping them to cloud applications.

In addition to orchestration languages that are strong in automating the provisioning of infrastructure resources, configuration management (CM) solutions can be used for the deployment of software services as well as the automation of lifecycle operations (cf. [32] for a study of the interplay of TOSCA with various CM solutions).

A best practice for supporting portability that both AO languages and CM solutions have in common is to detach environment specific values from central artifacts. This way, orchestration templates and CM files can be reused across deployments in conjunction with respective environment files. This permits deployment of the same application in different stages of a continuous delivery pipeline such as for testing and production. Also it permits ISVs to deliver applications without the need to know the details of a target cloud platform. Yet, as will be shown next, this practice addresses only a part of the considered problem.

2.4 Architecture-Impacting Requirements

The model, described by means of an AO language, is tightly coupled to the architecture of the cloud application. In this regard it is important to note that an orchestration model is described by the ISV (already). This is because it is a crucial part of the delivery of the cloud application, describing its service topology, i.e., the various components, their requirements, capabilities, and relationships.

[4] https://ietf.org/rfc/rfc2131.txt
[5] https://ietf.org/rfc/rfc1034.txt
[6] https://ietf.org/rfc/rfc6333.txt
[7] https://www.3gpp.org/the-evolved-packet-core
[8] https://www.3gpp.org/IMS
[9] https://wiki.openstack.org/heat
[10] https://www.etsi.org

Table 1: Sources and Owners of Requirements Impacting the Architecture of Applications

Requirement Sources	Vendor	Platform Provider	Legislator	Service Provider	End-Customer
cloud application	X				
target technologies		X			
datacenter design		X			
datacenter details		X			
legal interception			X		
security policies				X	
architectural guidelines				X	
operational requirements				X	
service level agreements				X	X

Basic architectural knowledge, thus, results from the decomposition of the application. Besides this source, further architectural knowledge must be derived from custom requirements and needs to be reflected in the service topology of an orchestration model. Such requirements can originate from a service provider's internal security policies and architectural guidelines. Please note that for TSPs, compliance with such (self-imposed) requirements is essential as it generally presents an imperative prerequisite for the approval of the production use of VNFs. Finally, target technologies and systems may impose further requirements that need to be respected as well.

Often, the authoritative source of custom requirements are (internal) papers (cf. [31]). Besides informal descriptions of the requirements, derived architectural knowledge is usually delineated as well. Yet, details on how to apply this knowledge, e.g., to an AO language, cannot be expected to be included. This is because such documents really focus on conveying principles. Thus, the respective transformation needs to be rendered by experts such as cloud architects.

Table 1 depicts different sources of requirements and indicates respective owners. The ISV conducts the decomposition as part of the engineering of the cloud application (first row). As the table shows, however, there are various other sources that have different owners, yet impacting the architecture and as a result the orchestration model.

One problem thus is that while there is but one orchestration model there exist multiple requirement sources having different owners. In particular, an ISV does not own all of the requirements that impact the architecture of the application. Still, the ISV is expected to deliver an orchestration model that reflects a compliant architecture.

The second row in the table relates to target technologies such as IaaS and software-defined networking (SDN) solutions. Details of these technologies may not be transparent to the orchestrator and, thus, may need to be reflected in the orchestration model. Moreover, the use of certain AO languages or extensions thereof is required and knowledge of available and enabled modules as well as deployed software versions of the solutions may be crucial as well. For example, for the provisioning of resources different model elements or a different syntax may need to be used between IaaS releases. Availability of certain modules such as for monitoring may provide additional options how to design the application. Related design decisions, in turn, may have taken place and the use of an available option may be mandated by an architectural guideline. Besides deployed cloud solutions, an orchestration model usually needs to consider the design of a datacenter (third row) such as the presence of certain availability zones, host aggregates, and virtual networks. Details of a datacenter such as IP addresses may not be required initially, but may be provided at deployment using environment files.

Legislation usually demands TSPs to provide an application programming interface (API) for legal interception (LI) of a VNF. Of course, this additional service not only affects the architecture of a VNF itself but also needs to be protected appropriately. Depending on respective requirements this further impacts the architecture.

Security policies in general may mandate the insertion of packet filter and/or application firewalls into the architecture in order to protect services and server instances. In addition or alternatively basic packet filtering can be configured at an IaaS level. Other requirements can relate to the placement of server instances and the use of storage services.

Next, architectural guidelines may stipulate the use of particular service solutions such as for high availability (HA) or DNS. Also, they can comprise various conventions for a tenant such as virtual networks and the use of IPv6 and/or an IP dual stack as well as harmonized (internal) IPv4 addresses.

Operational requirements imply some kind of operations, administration, and maintenance (OAM) access as well as a unified logging, monitoring, and reporting approach for cloud applications. A continuous delivery (CD) pipeline may stipulate further requirements.

Finally, service level agreements (SLAs) between a service provider and end-customers are an integral part of a cloud(-native) application. For realizing respective service availability architectural patterns such as HA load balancers can be applied. While every cloud application may bring its own solution, again, a service provider may want to see a unified adoption of best-practice solutions. For meeting respective SLAs, policies can be defined in an orchestration model besides following architectural principles in a service topology. For this key performance indicators (KPIs) are referred and actions such as for scaling out can be triggered automatically by an orchestrator. A presumption for this is the monitoring and reporting of KPIs. Requirements for such monitoring, in turn, may be part of architectural guidelines.

Table 2: Comparison of Delivery Process Scenarios

Efforts	Traditional Collaboration Model	Provider-Side Derivation	Model Refactoring
$E1$: disclosure of requirements	necessary	NOT necessary	NOT necessary
$E2$: derivation of architectural knowledge	INDIVIDUALLY per ISV	ONCE	ONCE
$E3$: implementation of architectural knowledge	MANUAL per application	MANUAL per application	AUTOMATED
$E4$: compliance checks	MANUAL per application	MANUAL per application	AUTOMATED
Total Effort	$E1 + E2 * n_{ISV} + (E3 + E4) * n_{Apps}$	$E2 + (E3 + E4) * n_{Apps}$	$E2_{Refactoring}$

2.5 Delivery Process Scenarios

Let us suppose ISVs are expected to deliver compliant applications including respective orchestration models using the documents that comprise the requirements: First, such documents need to be disclosed to the ISVs. Next, the ISVs need to read and understand the requirements, clarify open questions with the service provider, derive architectural knowledge, and incorporate such into respective orchestration models. Having received the cloud application from the vendor the service provider may want to validate the deliverable for conformance. In case of changing requirements such as derived from a new datacenter design, the service provider needs to communicate this to the ISVs he purchased applications from. The change impact needs to be analyzed and adaptations need to be conducted by each ISV and – delivered to the service provider – validated once again.

As an optimization to the scenario above, a service provider may – instead of communicating the requirements – only share the (tangible) implications towards the architecture. That is, architectural knowledge is derived by the service provider already out of all the custom requirements. The result needs to be shared in an appropriate form with the ISVs. This eliminates the need for the ISVs to interpret the requirements individually on their own avoiding misunderstandings as well. Instead, architectural knowledge is derived once from the requirements by the service provider. Respective documents comprising those requirements do not have to be disclosed to ISVs in this case. When there is a change of requirements, the service provider needs to conduct the analysis of the change impact, derive new architectural knowledge, and communicate this to the ISVs.

As an alternative to both scenarios that demand manual work while involving multiple stakeholders a more radical approach is chosen as presented next. Ideally, ISVs could describe an application such as the service topology as a result of the decomposition in terms of an orchestration model independently. In particular they should not need to incorporate architectural knowledge as derived from custom requirements, e.g., of a service provider. Similarly, a service provider would be able to impose custom requirements on a given orchestration model for achieving respective compliance.

Table 2 summarizes the key efforts of the three different scenarios such as the disclosure of requirements ($E1$) or the derivation of architectural knowledge ($E2$) that has to be performed individually per ISV in case of the traditional scenario. Please note, that the refactoring approach permits automation in regard to both the implementation of the architectural model ($E3$) and respective compliance checks ($E4$), while the other scenarios involve manual work per application. While the effort for formalizing the architectural knowledge in terms of model transformations is higher than in the other scenarios (i.e., $E2_{Refactoring} > E2$); the approach scales not only over the number of ISVs (n_{ISV}) but also the number of applications (n_{Apps}). That is, in contrast to the other scenarios, the total effort is not a function of these but a constant.

3 USE CASES

Let us now consider a cloud application – based on a real world example of a service provider with custom requirements – consisting of an HA load balancer, web servers, and a cluster of database servers. Figure 2 gives a graphical overview of the application indicating various connections between components. After a client initiates a service request to the HA endpoint, a load balancer forwards the request to a web server. In the following and for processing the request, a database connection is made use of. PostgreSQL[11] is deployed as a relational database management system and high availability of the database cluster as well as of the database endpoint is established via pgpool-II[12]. For realizing the HA web endpoint, keepalived[13] is used in conjunction with HAProxy[14]. That is, HAProxy realizes the load balancing by forwarding requests to (available) nginx[15] web servers and keepalived makes sure that the HA endpoint with a predefined (virtual) IP address remains reachable. In case the server instance fails that used to receive requests, another instance takes over the IP address.

For describing the cloud application and its service topology a cloud AO language such as TOSCA can be utilized (cf. Sect. 2.3). For brevity and demonstration purposes Frag [33], a dynamic language, is used in this paper for defining both a rudimentary textual DSL and for expressing a conforming model. Listing 1 defines main concepts that are used in the following for expressing a model reflecting the cloud application. The first statement imports the Frag package FMF, a modeling framework to build language models for DSLs. Similarly to the Unified Modeling Language [23] (UML) and the Eclipse Modeling Framework [29] (EMF), FMF provides concepts such as classes (with attributes and optional superclasses) and associations (having roles and multiplicities).

After the provisioning of infrastructure resources CM deploys services onto associated server instances. For the parameterization of these services variables can be used. The ports that are associated

[11] https://postgresql.org
[12] https://pgpool.net
[13] http://keepalived.org
[14] https://haproxy.org
[15] https://nginx.org

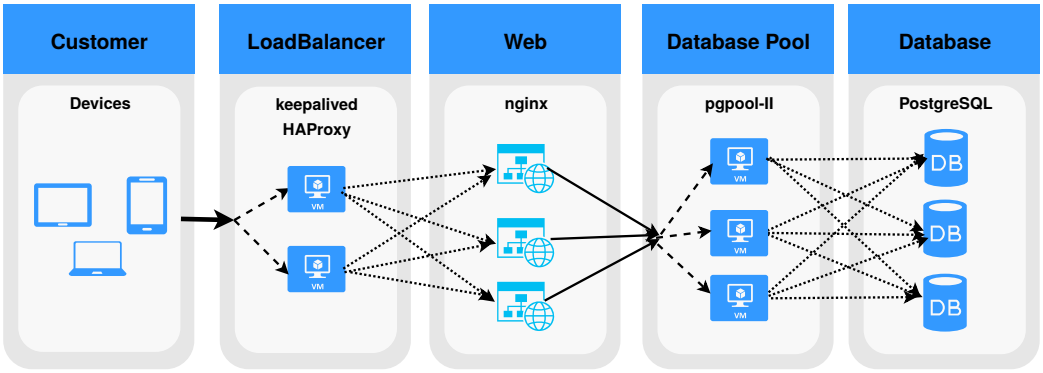

Figure 2: A Multi-Tier Cloud Application (Example)

with services can also be used for the CM of the server instances or firewalls. Computational and memory demands are stored in server instances as well as the name of a boot image and optionally the availability zone where it is to be located. Dependencies within a service topology can be specified using requirements. In addition to the generic service instance type, some more specific types such as load balancer, web, and database servers are defined. The concept of a connection point is used for IP addresses. The `ha` property indicates if the connection point realizes high availability.

Listing 1: Application Orchestration Metamodel (Example)

```
import mdd.FMF

FMF::Class create ServiceVar -attributes { value String }
FMF::Class create ServicePort -attributes {
    protocol int
    port int
}
FMF::Class create Service
FMF::Composition create ServicePorts -ends {
    {Service} {ServicePort -roleName ports -multiplicity *}
}
FMF::Composition create ServiceVars -ends {
    {Service} {ServiceVar -roleName vars -multiplicity *}
}
FMF::Class create Instance -attributes {
    cpu int
    ram int
    image String
    az String
}
FMF::Class create Network
FMF::Composition create Networks -ends {
    {Service} {Network -roleName nets -multiplicity *}
}
FMF::Class create Requirement
FMF::Association create Requirements -ends {
    {Instance} {Requirement -roleName requires -multiplicity *}
}
FMF::Association create Services -ends {
    {Instance} {Service -roleName services -multiplicity *}
}
FMF::Class create ConnectionPoint -attributes { ha boolean }
FMF::Class create IP -superclasses ConnectionPoint
    -attributes { addr String }
FMF::Class create Firewall -superclasses Instance
FMF::Class create LoadBalancer -superclasses Instance
FMF::Class create WebServer -superclasses Instance
FMF::Class create DBPool -superclasses Instance
FMF::Class create DBServer -superclasses Instance
```

Using the example DSL, the web application is described in Listing 2. First a service port and service for PostgreSQL are specified and used in a database server instance. The object is copied for a total of three instances. Next, an endpoint for the database service is defined using an internal IP address. It is used for the parameterization of pgpool-II. For high availability and a quorum three database pools are created specifying the database servers as requirements. Similarly, nginx web servers are defined that list the database pools as dependencies. Finally, the (web) service endpoint with a public IP address is specified and used as a virtual IP for configuring keepalived. HAProxy that is hosted on the same instances lists the web servers for forwarding requests. Note that this description – apart from the IP addresses of the endpoints that can be outsourced into an environment file – could be the result of the decomposition of the application as conducted by an ISV. It formally describes elements from the service topology. Thus, a cloud application orchestrator can use information from such a model for provisioning the infrastructure resources and for deploying the services using a CM system (cf. Sect. 2.3).

Listing 2: Cloud Application Orchestration Model (Example)

```
ServicePort create PostgreSQL_Port -protocol 6 -port 5432
Service create PostgreSQL -ports { PostgreSQL_Port }
DBServer create PostgreSQL_0 -services { PostgreSQL }
PostgreSQL_0 create PostgreSQL_1
PostgreSQL_0 create PostgreSQL_2

IP create DBEndpoint -addr "10.20.30.45" -ha
ServiceVar create DB_vIP -value [DBEndpoint get addr]
Service create PgpoolII
    -ports { PostgreSQL_Port } -vars { DB_vIP }
DBPool create PgpoolII_0 -services { PgpoolII }
    -requires { PostgreSQL_0 PostgreSQL_1 PostgreSQL_2 }
PgpoolII_0 create PgpoolII_1
PgpoolII_0 create PgpoolII_2

ServicePort create HTTP_Port -protocol 6 -port 80
ServicePort create HTTPS_Port -protocol 6 -port 443
Service create Nginx -ports { HTTP_Port HTTPS_Port }
WebServer create Nginx_0 -services { Nginx }
    -requires { PgpoolII_0 PgpoolII_1 PgpoolII_2 }
Nginx_0 create Nginx_1
Nginx_0 create Nginx_2

IP create WebEndpoint -addr "93.184.216.34" -ha
ServiceVar create Web_vIP -value [WebEndpoint get addr]
Service create Keepalived -vars { Web_vIP }
Service create HAProxy -ports { HTTP_Port HTTPS_Port }
LoadBalancer create LoadBalancer_0
    -services { Keepalived HAProxy }
    -requires { Nginx_0 Nginx_1 Nginx_2 }
LoadBalancer_0 create LoadBalancer_1
```

Table 3 lists some custom requirements. Based on the introduced application, the following use cases demonstrate the described methodology and illustrate how the orchestration model

Table 3: Some Custom Requirements of the Service Provider Towards Applications

Requirement	Description
R1	An edge server instance SHALL be protected by a packet firewall.
R2	All connection points SHALL realize high availability.
R3	All server instances SHALL be reachable via Secure Shell (SSH)[a] through a jump host and an OAM[b] network.
R4	Check_MK[c] SHOULD be deployed as a monitoring solution.
R5	Check_MK agents SHALL replace Nagios[d] agents.

[a] https://ietf.org/rfc/rfc4254.txt
[b] operations, administration, and maintenance
[c] https://mathias-kettner.de
[d] https://nagios.com

is refactored for covering the requirements. The first step consists of analyzing the architectural impact. That is, the requirement is interpreted by a cloud architect, architectural knowledge is derived, and a model transformation is implemented for covering one or more of such requirements.

3.1 Enriching a Service Topology with Firewalls

This use case provides a first insight into how automation can be achieved technically by presenting a simplified example. The first requirement (R1) demands that edge server instances, i.e., publicly reachable server instances, need to be protected by a packet firewall. Reflecting on and interpreting this requirement, a cloud architect derives the following architectural knowledge: All public service endpoints need to be protected by packet firewalls that only permit communication as related to respective services. For transforming a model accordingly, service endpoints that are reachable from the Internet need to be identified first. For each of these, a new server instance for hosting a firewall can be inserted between the Internet and the service endpoint (in case it is not already existing).

Transformation 1: Firewall Protection of Edge Instances

```
wizard Firewall_Protection {
    guard : self.isKindOf(Model!ao::ConnectionPoint) and
            self.isPublic() and not self.isProtected()
    title : "inserting a firewall for protecting " + self.name
    do { self.owningModel.insertFirewall(self) }
}
```

Transformation 1 highlights a respective implementation using the Epsilon Wizard Language [18] (EWL). This model-to-model transformation language is well suited for refactorings and permits a compact expression of rules. A matching of relevant model elements is done using the guard statement. Here all kinds of connection points (including IPs) are looked for that are public and that are not (already) protected. Helper operations in the Epsilon Object Language [17] (EOL) are used both in the guard as well as in the do part permitting such a compact and comprehensible form. In addition to model transformation, a verification can be implemented similarly reusing the guard expression, e.g., with the Epsilon Validation Language (EVL). EVL actually also permits model modification for fixing models and thus can be adopted as a better alternative to EWL for realizing model refactorings (used in the following)[16].

[16] It is possible to convert such transformations from EWL to EVL (cf. end of Sect. 3.2).

Besides, verification can also help to point out missing compliance in cases where automated transformation is not (yet) available. In addition to hard constraints, an EVL critique can be used for reporting details of an architecture that does not meet a non-critical requirement.

3.2 Establishing High Service Availability

The second requirement (R2) demands high availability to be established (also) for the (newly introduced) firewall functionality. Thus, the public connection point is used for a keepalived service that is installed on the firewall server instance. Finally, the instance is duplicated for establishing the redundancy.

Transformation 2: High Availability of Connection Points

```
context High_Availability {
    guard : self.isKindOf(Model!ao::ConnectionPoint)
    constraint C2_derived_from_R2 {
        check : self.ha
        message : "R2 is not met by " + self.name
        fix {
            title : "new server instance with keepalived"
            do {
                var service : new Service("keepalived");
                service.vars.add(self.addr);    // vIP
                var instance : new Instance;
                instance.services.add(service);
                self.owningModel().cloneInstance(instance);
                self.owningModel().refactorCP(self);
}   }   }   }
```

The described steps are realized in Transformation 2 using EVL. The guard statement identifies all connection points and the specified constraint checks if the ha property is set. Similarly to EWL, the do part of the constraint's fix comprises the model modifications for realizing the transformation. The actual model verification thus, occurs in the check statement while the model transformation is defined in the fix.

An excerpt of the result after both transformations have been applied on the original model is shown in Fig. 3. First, a firewall functionality was introduced that was then duplicated for establishing high availability together with an appropriate service (keepalived).

Ideally, implementation of the modifications realize idempotency. This way, transformations can be reapplied on a model without changing the results. That is, parts of the architecture that already comply with the requirements as covered by a transformation are not modified. Please note that in the example using EWL this is ensured by the last (negated) operand of the guard statement. The

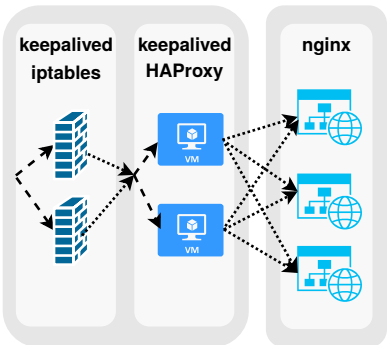

Figure 3: Refactored Model Addressing R1 and R2

same operand can be used in an analogous manner in an EVL `check`. This permits conversion of the transformation from EWL to EVL.

3.3 Operations, Administration, and Maintenance Access

An operations team needs to be able to access, inspect, and administrate service instances. For this an OAM access is required. Note that the original model does not cover OAM at all. Instead it is limited to a basic service topology of the application. It does not (need to) incorporate different concerns than the decomposition of the application. This use case, thus, demonstrates that is is possible to separate different concerns during the engineering. In particular, ISVs can focus on describing the basic service topology of the application and service providers that own requirements related to OAM can formalize the derived architectural knowledge for realizing a respective model transformation.

The third requirement (R3) mandates that access to service instances must be possible via SSH through a OAM network and a jump host. Thus, all server instances and a jump host have to provide an SSH service and must be connected to an OAM network. In case a server instance is not yet connected to the OAM network, the network and a jump host is created if not existent and the instance is attached to it .

Figure 4 displays the result after applying the described transformation. In addition to OAM other operational requirements such as those for logging and monitoring can be addressed similarly. For example a monitoring agent can be associated as a `Service` with instances. For meeting R2 (i.e., high availability of the SSH server), Transformation 2 can be (re)applied. The before mentioned idempotency property can help transformations to be used in a transitive manner, i.e., the same model can be obtained by applying transformations in different order.

3.4 Unified Adoption of Best Practice Solutions

For realizing a certain architectural pattern – such as those for establishing high availability as we have seen in the first example[17] – there are generally various solutions. A service provider may want to adopt a particular solution for a specific pattern (see R4). In some

[17] In contrast to the HTTP load balancer with keepalived, pgpool-II internally makes use of watchdog, another HA solution while managing a virtual IP.

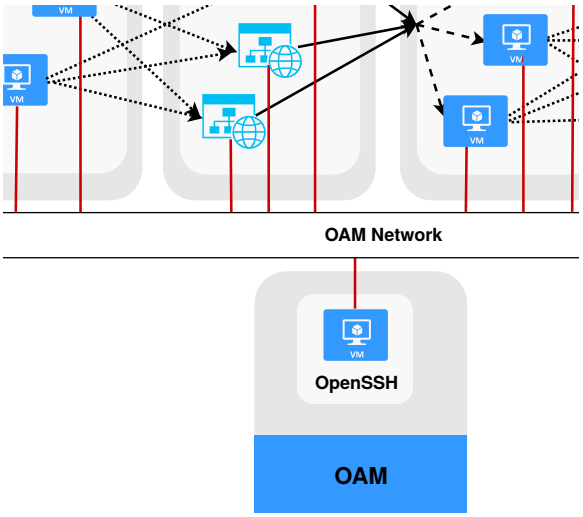

Figure 4: Excerpt of a Model Addressing R3

cases a certain solution shall be avoided and needs to be replaced by an alternative (see R5). A motivation for adopting common solutions can be the desire for a unification of services. This in turn can be demanded by an operations team that does not want to operate various systems that are similar in their function, yet different to operate.

Transformation 3: Unification of Monitoring

```
context Monitoring_Solution {
   guard : self.isKindOf(Model!ao::Instance)
   critique C4_derived_from_R4 {
      check : not self.services.exists(e|e.name = "Check_MK_agent")
      message : "R4 is not met by " + self.name
   }
   constraint C5_derived_from_R5 {
      check : self.services.exists(e|e.name = "NRPE")
      message : "R5 is not met by " + self.name
      fix {
         title : "replacing Nagios agent with Check_MK"
         do {
            var nrpe = self.services.selectOne(e|e.name = "NRPE");
            var service : new Service("Check_MK_agent");
            service.vars.add(nrpe.vars);
            self.services.add(service);
            self.services.remove(nrpe);
} } } }
```

In all these cases it is necessary to identify existing architectural patterns and apply refactoring for realizing the conformance. Translating this into EVL for meeting R5, Transformation 3 first matches server instances and tests if a Nagios agent is associated with a server instance. If this is the case then this agent is to be replaced according to R5 with a Check_MK agent as realized in the `do` part of the `fix`.

Various other scenarios for adopting common solutions are possible: In case OpenStack is to be used as IaaS solution, the load balancing could be realized using the load balancing as a service (LBaaS) from OpenStack Networking (Neutron). For adopting LBaaS as a common solution for cloud applications, the respective parts of orchestration models (keepalived and HAProxy server instances in our example) need to be replaced with the determined service. Similarly, firewall functionality can be realized in a different way

using OpenStack's SecurityGroups and SecurityRules. Replacing a database service such as PostgreSQL or adapting a different web server, however, may not easily be possible without testing the cloud application and adapting configurations at least. More experience from practice needs to be gained in this regard to better report on the chances and limitations of such refactoring.

3.5 Reflections and Remarks

This section showcased the applicability of model refactoring to cloud application orchestration models. For this, different use cases covering some non-functional requirements related to security and service availability as well as requirements introducing new service functionality (for OAM, logging, and monitoring) were demonstrated. All of these use cases commonly need to be supported for a production use. To realize them in an automated way and impose respective architectural knowledge on orchestration models promises substantial cost and effort savings (i.e., economic benefits).

Originated within and out of an industrial setting, the context, problem, and innovative approach, addresses a real world, highly relevant problem. In this course, two different cloud applications (one of these facing thousands of end customers in a pilot phase) have been successfully deployed into production.

The approach enables service providers to independently refactor orchestration models without having to disclose internal requirements to ISVs. While the model transformations for covering such requirements need to be implemented first, the examples showed that the derivation of the respective model verification and transformations can be easy to conduct. When foreseen in the delivery process of cloud applications and VNFs, ISVs can focus on delivering off-the-shelf solutions instead of tailoring solutions individually towards customers.

In case a model transformation has not been implemented yet, model verification can help to identify missing compliance in an automated manner. Note that in the examples given such checks comprise a necessary part of a transformation as realized with EVL. Thus, they can be formalized in a first step towards implementation. Yet, they deliver added value for an architecture impacting requirement already.

Please note that while an example DSL has been used in the presentation of the use cases, any AO language (see Sect. 2.3) together with its corresponding metamodel can be used as a basis. Similarly, the language for realizing model-to-model transformations – assuming it integrates with the DSL or its modeling technologies – can be chosen freely when adopting the approach.

4 DISCUSSION

Our approach introduces the first systematic approach for refactoring cloud application orchestration models. To the best of our knowledge, there are no other approaches that solve this problem using models and model transformations. While some other architecture-centric approaches exist in the field of cloud migration or general architectural refactoring (see e.g. [27, 28, 35]), none of them focuses on refactoring cloud application orchestration models or a closely related problem.

Our approach assumes that the architectural knowledge can be derived from the requirements in a detailed enough fashion. It might, however, be a substantial effort to transform the requirements into a tangible enough form to be usable in this way (depending on how the requirements are specified and on the domain experience of the architects). In a systematic engineering process, where architecturally significant requirements are routinely identified, as it is usual for major architectures of large enterprises, the requirements identification part of our approach should produce little to no extra effort.

Another limitation might be that not all requirements can easily be formalized and expressed as a model transformation. For some kinds of requirements such a formalization might be a substantial effort. Also dealing with multiple requirements that may be contradicting can be a challenge. While we expect conflicts to be already identified at an earlier stage, further research is needed to understand potential limitations in this regard better and, if needed, develop approaches to solve such issues.

In addition, in our approach we basically assume that the derivation of model transformations from requirements is straightforward. For some requirements this might not be the case. Still, model verification as the first step towards such implementation and as outlined in this paper and this approach can help to check and identify missing compliance. Another goal for further research is to derive systematic methods for deriving model transformations from requirements. We hope that approaches such as model transformation by example [30] could help to provide easy and save to apply solutions.

Please note that this paper is limited to conveying the idea of refactoring a model for establishing architectural compliance. Particularly – even if some first considerations regarding idempotency and transitivity have been raised in the last section – further questions in which order to apply model transformations and how to write non-destructive transformations and/or protect parts of the architecture are subject of future work. While these questions are interesting from a theoretical point of view and for a large scale application in practice, in our experiences it was easy to define an accurate sequence of transformations (e.g., apply `R2` for HA at last).

Our approach has been discussed in the context of cloud application orchestration models. However, we believe it to be generalizable to many other (software) engineering contexts, where compliance to requirements is needed in a similar fashion. That is, we see no reason why the approach should not be applicable for other kinds of architecture models in software-intensive systems and in domains other than cloud provisioning. We plan to perform further research in the direction of more generic architecture compliance based on models and model transformations in the future.

Overall, in our industrial use cases we have shown the applicability of our approach in practice, as well as reported in cost-effective experiences in applying the approach. While we have not studied cost-effectiveness in detail (e.g., using metrics or measurements), our use cases show clearly that little extra effort is needed in comparison to the total work required in these projects, so that our approach can be assumed to be applicable in such a project from a costs/benefits perspective, if it is repetitively needed. This observation might not be generalizable to all contexts; for instance, as discussed above, if architecturally significant requirements are not routinely identified in another company context, the needed

effort might be higher than the efforts we have observed in our experiences.

A general alternative to our approach is to place the changes introduced by model refactoring into the orchestrator of a cloud application orchestration model, for instance using policies. While this might work well for smaller requirements, more substantial customizations might become hard to understand (and compose), as the original model and all (policy-introduced) changes need to be understood in order to comprehend the big picture. For this reason, we decided to follow the more systematic, model-based approach. This way, an effective model that is passed to an orchestrator already contains all of the details for the provisioning of infrastructure resources. Despite the benefits, all of the transformations can become part of an orchestrator, e.g., plugins being activated using AO language extensions.

5 RELATED WORK

This work leverages model refactoring as a kind of model transformation to automate meeting custom requirements in the context of cloud application orchestration models. To the best of our knowledge, there is no other approach yet that uses model transformations or model refactoring in this way.

A generic taxonomy of model transformations is provided in [22]. This taxonomy defines model refactoring as a special kind of transformation that performs "a change to the internal structure of software to improve certain software quality characteristics without changing its observable behavior" (definition adapted from [9]). In the context of this taxonomy our work can also be categorized as optimization or adaptation.

Much of the work on model refactoring focuses on implementing model-level refactoring. For example, Zhang et al. [34] introduce a model transformation engine for realizing generic and domain-specific model refactoring. Mens [21] and Biermann et al. [5] discuss different approaches using graph transformations for model refactoring. In principle, any such model transformation approach for model refactoring can be used in the context of our approach. A generic evaluation of these approaches can be found in [16].

Closer to our work are approaches that utilize model refactoring for generic changes to design models. For example, Rossi et al. [26] use model refactoring to transform old fashioned Web applications to support more modern rich Internet application interfaces. Another Web application model refactoring approach is presented in [11]. Arendt et al. [2] utilize model refactoring in a simple case study to compare different technologies. While all of these approaches use model refactoring in a similar way as our approach, none of them considers the specific customization problem, the application domain of cloud application orchestration, or the generic compliance to requirements aspect of our work.

The closest to our notion of compliance to requirements are approaches that use external design constraints in their model refactoring approaches. For instance, France et al. [10] introduce a pattern-based model refactoring approach using design knowledge embodied in design patterns as a basis for refactoring design models. Arcelli et al. [1] introduce an approach for anti-pattern-based model refactoring for software performance improvement. Glitia et al. [12] use repetitive model refactoring for design space exploration of signal processing applications. That is, trade-offs in the usage of storage and computation resources and in parallelism guide the repetitive model refactoring. The use of the patterns, anti-patterns or trade-offs in those approaches is akin to the notion of compliance to requirements in our approach, but the authors do not use compliance to generic requirements, but just the structures of the (anti-)patterns or effects on certain quality trade-offs.

Non-model-based refactoring has been used in a number of approaches for migration to the cloud. For example, Kwon and Tilevich [19] introduce an approach for automated transitioning to cloud-based services. Hilton et al. [14] suggest an approach for refactoring local to cloud data types. Some of the cloud migration patterns in [15] are based on refactoring. All of these works lack the systematic model-based and architecture-centric approach that is central to our approach, however, as well as the concrete focus on cloud application orchestration.

A number of approaches introduce architecture-centric refactoring methods. For instance, Strauch et al. [28] discuss refactoring as one strategy for migrating enterprise applications to the cloud. In another work, Strauch et al. [27] introduce a pattern-based refactoring-based approach for migrating application data to the cloud. Zimmermann [35] discusses architectural refactoring as a strategy for software evolution and mentions applications in cloud computing in his work. These more architecture-centric approaches are closer in spirit to the systematic approach proposed in our work, but do not provide support for automatic or model-based refactoring. Also, these approaches lack all specifics introduced in our work with regard to cloud application orchestration.

6 CONCLUSION

In this paper, we studied the general problem of establishing architecture compliance to requirements in the context of the specific domain of cloud application orchestration with use cases from industrial practice. We suggested an approach for imposing architectural knowledge as derived from custom requirements onto architecture models. Our approach is based on automated refactoring of models for meeting custom requirements.

To the best of our knowledge, no other approach exists that uses model transformations in this way or provides similar capabilities in the context of cloud application orchestration.

In a number of industrial use cases we have shown the cost-effective applicability of our approach in practice. Limitations with regard to deriving architectural knowledge from the requirements, requirements formalization in model transformations, and transitivity and/or sequence of model transformations, as discussed in Sect. 4, can be mitigated to a certain extent and/or need to be addressed in future work. While we have focused our study on our own industrial background in cloud application orchestration, we believe our approach to be generalizable to other kinds of architecture models and domains.

Acknowledgments

The authors would like to thank reviewers for valuable feedback.

This work was partially supported by Austrian Science Fund (FWF) project ADDCompliance (no. I – 2885) and Austrian Research Promotion Agency (FFG) project DECO (no. 864707).

REFERENCES

[1] Davide Arcelli, Vittorio Cortellessa, and Catia Trubiani. 2012. Antipattern-based model refactoring for software performance improvement. In *Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures*. ACM, 33–42.

[2] Thorsten Arendt, Florian Mantz, Lars Schneider, and Gabriele Taentzer. 2009. Model refactoring in Eclipse by LTK, EWL, and EMF refactor: a case study. In *Model-Driven Software Evolution, Workshop Models and Evolution*.

[3] Alexander Bergmayr, Uwe Breitenbücher, Nicolas Ferry, Alessandro Rossini, Arnor Solberg, Manuel Wimmer, Gerti Kappel, and Frank Leymann. 2018. A Systematic Review of Cloud Modeling Languages. *ACM Comput. Surv.* 51, 1 (2018), 22:1–22:38. https://doi.org/10.1145/3150227 [accessed in Sept. 2018].

[4] Alexander Bergmayr, Javier Troya, Patrick Neubauer, Manuel Wimmer, and Gerti Kappel. 2014. UML-based Cloud Application Modeling with Libraries, Profiles, and Templates. In *2nd International Workshop on Model-Driven Engineering on and for the Cloud*. 56–65. http://ceur-ws.org/Vol-1242/paper7.pdf [accessed in Sept. 2018].

[5] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer, and Eduard Weiss. 2007. EMF model refactoring based on graph transformation concepts. *Electronic Communications of the EASST* 3 (2007).

[6] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. 2014. TOSCA: Portable Automated Deployment and Management of Cloud Applications. In *Advanced Web Services*, Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel (Eds.). Springer, 527–549. https://doi.org/10.1007/978-1-4614-7535-4_22 [accessed in Sept. 2018].

[7] Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. 2014. *Cloud Computing Patterns - Fundamentals to Design, Build, and Manage Cloud Applications*. Springer. https://doi.org/10.1007/978-3-7091-1568-8 [accessed in Sept. 2018].

[8] Nicolas Ferry, Alessandro Rossini, Franck Chauvel, Brice Morin, and Arnor Solberg. 2013. Towards Model-Driven Provisioning, Deployment, Monitoring, and Adaptation of Multi-cloud Systems. In *2013 IEEE Sixth International Conference on Cloud Computing, Santa Clara, CA, USA, June 28 - July 3, 2013*. IEEE Computer Society, 887–894. https://doi.org/10.1109/CLOUD.2013.133 [accessed in Sept. 2018].

[9] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA.

[10] Robert France, S Chosh, Eunjee Song, and Dae-Kyoo Kim. 2003. A metamodeling approach to pattern-based model refactoring. *IEEE software* 20, 5 (2003), 52–58.

[11] Alejandra Garrido, Gustavo Rossi, and Damiano Distante. 2007. Model refactoring in web applications. In *Web Site Evolution, 2007. WSE 2007. 9th IEEE International Workshop on*. IEEE, 89–96.

[12] Calin Glitia, Pierre Boulet, Eric Lenormand, and Michel Barreteau. 2011. Repetitive model refactoring strategy for the design space exploration of intensive signal processing applications. *Journal of Systems Architecture* 57, 9 (2011), 815–829.

[13] Glauco Estacio Gonçalves, Patricia Takako Endo, Marcelo Anderson Santos, Djamel Sadok, Judith Kelner, Bob Melander, and Jan-Erik Mångs. 2011. CloudML: An Integrated Language for Resource, Service and Request Description for D-Clouds. In *IEEE 3rd International Conference on Cloud Computing Technology and Science, CloudCom 2011, Athens, Greece, November 29 - December 1, 2011*, Costas Lambrinoudakis, Panagiotis Rizomiliotis, and Tomasz Wiktor Wlodarczyk (Eds.). IEEE Computer Society, 399–406. https://doi.org/10.1109/CloudCom.2011.60 [accessed in Sept. 2018].

[14] Michael Hilton, Arpit Christi, Danny Dig, Micha l Moskal, Sebastian Burckhardt, and Nikolai Tillmann. 2014. Refactoring local to cloud data types for mobile apps. In *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems*. ACM, 83–92.

[15] Pooyan Jamshidi, Claus Pahl, Samuel Chinenyeze, and Xiaodong Liu. 2015. Cloud migration patterns: a multi-cloud service architecture perspective. In *Service-Oriented Computing-ICSOC 2014 Workshops*. Springer, 6–19.

[16] Shekoufeh Kolahdouz-Rahimi, Kevin Lano, Suresh Pillay, Javier Troya, and Pieter Van Gorp. 2014. Evaluation of model transformation approaches for model refactoring. *Science of Computer Programming* 85 (2014), 5–40.

[17] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. 2006. The Epsilon Object Language (EOL). In *Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006, Proceedings (Lecture Notes in Computer Science)*, Arend Rensink and Jos Warmer (Eds.), Vol. 4066. Springer, 128–142. https://doi.org/10.1007/11787044_11 [accessed in Sept. 2018].

[18] Dimitrios S. Kolovos, Richard F. Paige, Fiona Polack, and Louis M. Rose. 2007. Update Transformations in the Small with the Epsilon Wizard Language. *Journal of Object Technology* 6, 9 (2007), 53–69. https://doi.org/10.5381/jot.2007.6.9.a3 [accessed in Sept. 2018].

[19] Young-Woo Kwon and Eli Tilevich. 2014. Cloud refactoring: automated transitioning to cloud-based services. *Automated Software Engineering* 21, 3 (2014), 345–372.

[20] Frank Leymann, Christoph Fehling, Ralph Mietzner, Alexander Nowak, and Schahram Dustdar. 2011. Moving Applications to the Cloud: an Approach Based on Application Model Enrichment. *Int. J. Cooperative Inf. Syst.* 20, 3 (2011), 307–356. https://doi.org/10.1142/S0218843011002250 [accessed in Sept. 2018].

[21] Tom Mens. 2006. On the use of graph transformations for model refactoring. In *Generative and transformational techniques in software engineering*. Springer, 219–257.

[22] Tom Mens and Pieter Van Gorp. 2006. A Taxonomy of Model Transformation. *Electr. Notes Theor. Comput. Sci.* 152 (2006), 125–142. https://doi.org/10.1016/j.entcs.2005.10.021

[23] Object Management Group, Inc. 2000. Unified Modeling Language (UML). https://omg.org/spec/UML [accessed in Sept. 2018].

[24] Organization for the Advancement of Structured Information Standards. 2016. *TOSCA Simple Profile for Network Functions Virtualization (NFV) Version 1.0*. Committee Specification Draft. OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC. https://docs.oasis-open.org/tosca/tosca-nfv/v1.0/tosca-nfv-v1.0.pdf [accessed in Sept. 2018].

[25] Organization for the Advancement of Structured Information Standards. 2017. *TOSCA Simple Profile in YAML Version 1.1*. Committee Specification Draft. OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC. https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.1/TOSCA-Simple-Profile-YAML-v1.1.pdf [accessed in Sept. 2018].

[26] Gustavo Rossi, Matias Urbieta, Jeronimo Ginzburg, Damiano Distante, and Alejandra Garrido. 2008. Refactoring to rich internet applications. a model-driven approach. In *Web Engineering, 2008. ICWE'08. Eighth International Conference on*. IEEE, 1–12.

[27] Steve Strauch, Vasilios Andrikopoulos, Thomas Bachmann, and Frank Leymann. 2013. Migrating application data to the cloud using cloud data. In *e 3rd International Conference on Cloud Computing and Service Science,(CLOSER)*. 36–46.

[28] Steve Strauch, Vasilios Andrikopoulos, Dimka Karastoyanova, Frank Leymann, Nikolay Nachev, and Albrecht Stäbler. 2014. Migrating enterprise applications to the cloud: methodology and evaluation. *International Journal of Big Data Intelligence 5* 1, 3 (2014), 127–140.

[29] The Eclipse Foundation. 2002. Eclipse Modeling Framework Project (EMF). https://eclipse.org/modeling/emf. https://eclipse.org/modeling/emf [accessed in Sept. 2018].

[30] Dániel Varró. 2006. Model transformation by example. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 410–424.

[31] Markus Völter, Daniel Ratiu, and Federico Tomassetti. 2013. Requirements as First-Class Citizens: Integrating Requirements closely with Implementation Artifacts. In *Proceedings of the 6th International Workshop on Model Based Architecting and Construction of Embedded Systems co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2013), Miami, Florida, USA, September 29th, 2013. (CEUR Workshop Proceedings)*, Iulian Ober, Florian Noyrit, Susanne Graf, and Gabor Karsai (Eds.), Vol. 1084. CEUR-WS.org. http://ceur-ws.org/Vol-1084/paper4.pdf [accessed in Sept. 2018].

[32] Johannes Wettinger, Uwe Breitenbücher, and Frank Leymann. 2014. Standards-Based DevOps Automation and Integration Using TOSCA. In *7th IEEE/ACM International Conference on Utility and Cloud Computing*. IEEE, 59–68. https://doi.org/10.1109/UCC.2014.14 [accessed in Sept. 2018].

[33] Uwe Zdun. 2010. A DSL toolkit for deferring architectural decisions in DSL-based software design. *Information & Software Technology* 52, 7 (2010), 733–748.

[34] Jing Zhang, Yuehua Lin, and Jeff Gray. 2005. Generic and domain-specific model refactoring using a model transformation engine. In *Model-driven Software Development*. Springer, 199–217.

[35] Olaf Zimmermann. 2015. Architectural refactoring: A task-centric view on software evolution. *IEEE Software* 32, 2 (2015), 26–29.