FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

An Architecture Style for Cloud Application Modeling

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Alexander Bergmayr
Registration Number 0255171

to the Faculty of Informatics
at the TU Wien

Advisor: Priv. Doz. Dr. Manuel Wimmer

The dissertation has been reviewed by:

Priv. Doz. Dr. Manuel Wimmer	Prof. Dr. Dr. h. c. Frank Leymann

Vienna, 4th April, 2016

Alexander Bergmayr

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

An Architecture Style for Cloud Application Modeling

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

Alexander Bergmayr
Matrikelnummer 0255171

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Priv. Doz. Dr. Manuel Wimmer

Diese Dissertation haben begutachtet:

Priv. Doz. Dr. Manuel Wimmer	Prof. Dr. Dr. h. c. Frank Leymann

Wien, 4. April 2016

Alexander Bergmayr

Erklärung zur Verfassung der Arbeit

Alexander Bergmayr
Maria-Tusch-Straße, A-1220 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 4. April 2016

Alexander Bergmayr

Acknowledgements

I had the good fortune of getting to know Manuel Wimmer back in 2007. Since then we stayed in contact. When I joined the research group of Gerti Kappel in 2012, he guided me in my research and advised me on writing my dissertation. I am grateful to Manuel Wimmer for his extraordinary support in the last four years and to Gerti Kappel, who provided to me an inspiring environment for doing research. Thank you, Manuel and Gerti! I am indebted to Frank Leymann, who reviewed my dissertation. Thank you, Frank! Angelika Kebhart gave generously of her time to correct my dissertation. Thank you, Angi! I thank Martin Fleck for his final proofreading. Thank you, Martin! I thank the many people who contributed directly or indirectly to this dissertation: Michael Grossniklaus, Philip Langer, Tanja Mayerhofer, David Madner, Patrick Neubauer, and Javier Troja. I am deeply grateful to my family for all the patience, support, and love. Thank you, Nadine, Gustav, and Frida! No, no, I do not forget you Mum. Thank you!

Abstract

UML is a widely adopted open standard to create architectural models from multiple viewpoints for various domains. Its language-inherent extension mechanism is being applied to systematically integrate domain-specific concepts via libraries and profiles because they are indispensable for model-based engineering (MBE). Cloud computing is an appealing target domain for MBE. Modern cloud environments support a relatively high degree of automation in service provisioning, which allows cloud users to dynamically acquire services required for deploying cloud applications. On the other hand, MBE aims at increasing the automation of application development by a systematic tool-supported refinement of high-level models towards a target platform and the environment underneath. The selected platform and environment designate the technical target domain whose concepts have to be captured on the model level in order to enable the refinement of architectural models. Modeling concepts and tools along with a set of constraints on how they can be used denote an *architecture style*. Providing an architecture style for cloud application modeling based on UML including tools that exploit automated processes of both cloud computing and MBE is thus highly desirable. Due to the generic nature of UML, it does however not provide cloud modeling concepts by default and existing UML tools do not yet adequately support cloud-specific model refinement.

To address these deficiencies, the goal of this thesis is to realize cloud-specific extensions to UML and a toolset that together form an architecture style for developing cloud applications. In particular, we place emphasis on the automation of development processes and their effectiveness in producing truly useful models. Four main contributions are presented to achieve this goal. First we systematically review current cloud modeling languages (CMLs) and investigate major cloud environments to derive a core set of features inherent to the cloud computing domain. They serve as the basis for developing the UML-based *cloud application modeling language (CAML)*, which is the second contribution. CAML supports semi-automatic model refinement towards the Java platform and three major cloud environments via dedicated libraries and profiles. The third contribution addresses the automatic translation of UML architecture models refined by CAML into TOSCA, a recently adopted standard that aims at automating application provisioning and management. Combining UML and TOSCA closes the gap between architecture modeling and application provisioning. As model transformations are key to automate the refinement and translation of model-based artifacts, maintaining those transformations and their produced artifacts is addressed by the fourth contribution. We exploit incremental transformation to co-evolve existing models with changes to transformations. In addition to the conceptual contributions, we provide proof-of-concept implementations as open-source projects and present case studies for evaluating not only their practical relevance but also aspects such as quality and performance.

Kurzfassung

UML ist ein weitverbreiteter offener Modellierungsstandard für unterschiedliche Domänen, der es durch seinen sprachinhärenten Erweiterungsmechanismus ermöglicht, domänen-spezifische Konzepte systematisch zu integrieren. Die dadurch entwickelten Bibliotheken und Profile sind unabdingbar für den Einsatz von Model-based Engineering (MBE). Cloud Computing ist eine vielversprechende Zieldomäne für MBE. Heutige Cloud-Umgebungen bieten einen relativ hohen Automatisierungsgrad für die Serviceprovisionierung, der es Benutzern erlaubt Services für das Deployment einer Cloud-Anwendung dynamisch zu akquirieren. Ein Kernziel von MBE ist die Automatisierung der Anwendungsentwicklung durch systematische werkzeuggestützte Verfeinerung von Architekturmodellen für eine Zielplattform und die darunterliegende Umgebung. Die gewählte Plattform und Umgebung bestimmen die technische Zieldomäne deren Konzepte auf der Modellierungsebene erfasst werden müssen, um eine Modellverfeinerung zu ermöglichen. Die Bereitstellung geeigneter Modellierungskonzepte und Werkzeuge in Form eines *Architecture Styles* basierend auf UML ist deshalb wünschenswert. Standardmäßig stellt UML jedoch keine Modellierungskonzepte für Cloud Computing zur Verfügung und existierende UML Werkzeuge bieten keine ausreichende Unterstützung für eine cloud-spezifische Modellverfeinerung.

Um diese Limitierungen zu adressieren, werden in dieser Arbeit cloud-spezifische Erweiterungen für UML und ein Werkzeugsatz vorgestellt, die zusammen einen Architecture Style für die Entwicklung von Cloud-Anwendungen bilden. In einem ersten Schritt werden existierende Modellierungssprachen und Umgebungen im Cloud Computing Bereich untersucht, um Kernaspekte der adressierten Domäne abzuleiten. Auf dieser Grundlage wird die UML-basierte *Cloud Application Modeling Language (CAML)* entwickelt. CAML unterstützt mit Hilfe geeigneter Bibliotheken und Profile die semi-automatische Modellverfeinerung für die Java Plattform und drei führende Cloud-Umgebungen. Zudem befasst sich diese Arbeit mit der automatischen Übersetzung von Architekturmodellen repräsentiert in UML und verfeinert durch CAML nach TOSCA, ein kürzlich verabschiedeter Standard mit dem Ziel die Anwendungssprovisionierung und das Anwendungsmanagement zu automatisieren. Durch das Zusammenwirken von UML und TOSCA kann die Lücke zwischen Architekturmodellierung und Anwendungssprovisionierung geschlossen werden. Da in der automatisierten Verfeinerung und Übersetzung von modellbasierten Artefakten die Modelltransformation eine Schlüsselrolle einnimmt, wird in dieser Arbeit auch die Modellevolution auf Grund von Transformationsanpassungen betrachtet. Die in dieser Arbeit vorgestellten Konzepte werden als proof-of-concept Implementierungen in Form von open-source Projekten zur Verfügung gestellt und anhand von Fallstudien evaluiert. Dabei wird nicht nur die praktische Relevanz der vorgestellten Arbeit demonstriert sondern auch Aspekte wie Qualität und Performanz untersucht.

Contents

CHAPTER 1

Introduction

With the emergence of cloud computing, the effort required to get access to environments with the scale of large distributed data centers has tremendously been reduced. Provisioning resources of a cloud environment [BGPCV12] can be carried out on-demand [Ley11] via the Web without the need to negotiate with the cloud provider because their offerings are considered as commodities that are readily available as a service and consumable over the network even on per hour basis. For instance, Amazon Web Services (AWS)[1], which appeared in 2006, offers compute services for less than 2 cents per hour without any long-term commitments, where the provisioning of a running virtual machine on a raw computing node takes only a few minutes. Therefore, cloud environments are appealing for software companies and engineers because of the low upfront costs compared to a traditional on-premise solution and operational costs that scale with the consumed services. They range from low-level infrastructure services, *e. g.,* raw compute capacity, over higher level platform services, *e. g.,* a fully managed Java runtime as provided by the Google App Engine (GAE)[2], on top of a cloud infrastructure, to ready-to-use software, *e. g.,* Sales Cloud offered by Salesforce[3], hosted on a cloud platform. Companies are no longer forced to plan far ahead for the provisioning of their applications [AFG+10] because the large-scale data centers of today's cloud providers ensure that the required cloud services to operate those applications are available. Cloud services are not only provisioned as their demand increases, *e. g.,* the number of user requests exceeds a defined threshold, but also released once their demand decreases, *e. g.,* the number of user requests falls below a defined threshold. This elastic capability of cloud environments [Owe10] reduces the risk of under and over-provisioning of services and enables applications deployed in the cloud to scale in and out dynamically [VRB11]. As a result, modern cloud environments support a relatively high degree of automation in service provisioning not only to achieve the dynamic scalability of cloud applications but also to allow clients to acquire services on demand required for the deployment of those applications. The deployment of an application determines the connection between the application components and the target environment on

[1] Amazon Web Services (AWS): `http://aws.amazon.com`
[2] Google App Engine (GAE): `https://cloud.google.com/appengine`
[3] Salesforce: `http://www.salesforce.com`

top of which the implementations of those applications are eventually executed. Deploying and configuring applications is usually a complex multi-faceted task as it cuts across the boundaries of different layers, *e. g.,* infrastructure and platform, and requires expertise from various areas, *e. g.,* development and operations [EKK+06, WBL14]. Like in any engineering discipline, modeling is a powerful means of software engineering to master complexity and communicate effectively. Model-based engineering (MBE) [Ken02, AK03, BCW12, MLM+13, LMM+15] emphasize the importance of models throughout the entire application life-cycle.

From a development perspective, models are considered as the primary artifacts [Béz05b, BCW12] to capture the essence of an application by means of abstraction [Kra07], thereby hiding lower level implementation details. Ideally, implementations can be generated for possibly multiple environments from a single set of architecture models. Typically, this set of models is progressively transformed into models that capture environment-specific features. In the context of cloud computing, those features must convey the peculiarities of the targeted cloud environment to enable model transformers to generate appropriate implementations [Sel12]. A promise of model-based approaches is that model transformers can produce the implementations for a variety of environments from high-level models, thereby investments in creating them are retained [GS03] even though the target environment beneath is changed. For instance, a typical application scenario is the generation of environment-specific implementations from a high-level data model, where the environment may refer to a relational database and a particular data access framework such as the Java Persistence API (JPA)[4]. Changing the environment for which the Java-based implementation has been generated, *e. g.,* by the highly scalable datastore of the GAE, would require re-generating the implementation for the new environment. As a result, models become valuable assets in the development of cloud applications for providing abstractions over those applications and the diverse target environments. Moreover, they support the transition of applications from on-premise environments to cloud environments and between cloud environments [FR10, SR10] through rigorous model transformation techniques [SK03, Sch06]. This implies that the applications must be (re-)distributed across a single or even multiple cloud environments, which in turn necessitates the provisioning of the required cloud services. The desired state of the application and service provisioning is usually captured in terms of a deployment model [TMW+05] which is considered as part of the overall application architecture [CGL+03]. From an operations perspective, it triggers the actions carried out in a provisioning process. This explains also the need for aligning the two perspectives [WBL14].

UML is a widely adopted open standard to create architectural models [MLM+13, LMM+15] from multiple viewpoints for various domains. Its lightweight extension mechanism is being applied to systematically integrate domain-specific concepts into UML via profiles and libraries [Par10, Sel07, Sel12]. Now, with the emergence of cloud computing, a new target domain appealing for architecture modeling has appeared. This poses the general question whether UML is suitable to model cloud application architectures. In this thesis, we hypothesize that appropriately applying UML can render cloud application architectures more valuable for architects, developers, and operators to design and implement those applications and execute their provisioning.

[4] JPA: http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html

1.1 Problem statement

Domain-specific extensions to architectural languages are just as indispensable as are libraries for programming languages. Developing language concepts and tools specific to a domain is hard in particular if the addressed domain is highly diverse and constantly progressing as cloud computing is. However, an elegant architecture of a cloud application is of limited value unless it exploits concepts of the target cloud computing domain. Due to the generic nature of UML, it does neither provide cloud-specific modeling concepts by default nor guidance how the existing modeling concepts can be used to represent cloud application architectures. Therefore, this thesis is concerned with providing a cloud-specific architecture style for UML. An architecture style comprises a set of element and relationship types tailored to a particular domain together with with a set of constraints on how they can be used to design a class of systems[5]. It also supports that class of system design with style-specific tools, analysis, and implementations [GBI+10]. In the context of this thesis, the domain is cloud computing and the class of systems refers to cloud applications. UML is used as the base language whose element and relationship types are tailored to the cloud computing domain. The following four research challenges (*RC1–RC4*) are addressed by this thesis.

1.1.1 Elicitation of cloud modeling concepts (*RC1*)

The first challenge that is addressed by this thesis concerns the classification and comparison of existing languages for representing cloud applications on the model level. They provide a complementary set of modeling concepts to deal with the diversity of current cloud environments. Moreover, a standard called TOSCA [OAS13a, BBKL14a] for representing portable cloud applications [BBLS12] and supporting their life cycle management were adopted by the OASIS standardization body[6] in late 2013. All those languages can be considered as a domain-specific language (DSL) [MHS05] where the domain refers to cloud computing. We collectively refer to them as cloud modeling language (CML). There is, however, still little consensus in the research community on what a CML is, what aspects of a cloud application and the target cloud environment including the runtime platform for executing the application components should be modeled by a CML, which of the currently existing CMLs is best suited for a particular problem, and how they relate to existing languages for architecture modeling such as UML. Having classified and compared existing CMLs would provide a useful source of inspiration for collecting a common set of cloud modeling concepts based on which extensions for UML can be developed [Sel07]. Moreover, lessons learned from this investigation would support the identification of the potential extension points of UML for integrating cloud-specific features.

1.1.2 Refinement of cloud architecture models (*RC2*)

The second challenge that is addressed by this thesis concerns the refinement of high-level models towards a selected platform and a target cloud environment, thereby allowing engineers to capture platform and environment-related decisions, *e. g.,* the selection of a cloud storage

[5] Based on the definition of the Software Engineering Institute, http://www.sei.cmu.edu

[6] OASIS: https://www.oasis-open.org

solution, already on the model level. Even though UML provides modeling concepts to represent software, platform and infrastructure artifacts from different viewpoints, they neither satisfy current cloud modeling requirements nor provide features of platforms currently available for cloud environments. This is mainly because UML is a general-purpose language and hence its modeling concepts must be generic rather than specific to a certain domain such as cloud computing. Therefore, providing cloud-specific extensions to UML appears beneficial. They would not only allow exploiting the full expressive power of UML to model cloud applications but also a seamless integration of cloud-specific features into existing or possibly reverse-engineered UML models. The latter would be especially useful for "modernization to the cloud" scenarios, where reverse-engineered UML models [TP04] are refined towards a certain platform and cloud environment in a forward-engineering process [HRW11, HWRK11]. Properly refined architecture models would pave the way for generating implementations from different views, *e. g.,* class, component, and deployment, provided that model transformers capable to deal with environment-specific features are available. Moreover, deployment models that capture environment-specific features would explicitly represent the interconnection between the cloud application and its environment and thus specify the desired state of the application provisioning.

1.1.3 Model-based cloud application provisioning (*RC3*)

The third challenge that is addressed by this thesis concerns the automation of cloud application provisioning where UML deployment models refined towards the target cloud environment are considered as input to the provisioning process. Since the adoption of TOSCA by OASIS, a standard is available for modeling portable cloud applications with the aim to automate their provisioning. However, until now, an effective conceptual mapping between UML and TOSCA as a basis for an automated transformation between the two languages is still missing. As a result, the translation is currently carried out in a tedious manual step, which is only achievable if engineers are familiar with the peculiarities of both languages and capable to identify the correspondences between them. Bridging UML and TOSCA would allow engineers to translate UML architectural models into corresponding TOSCA deployment models based on which the provisioning process can be enacted by a TOSCA-compliant runtime container [BBKL14a]. From a UML perspective this is beneficial as it would allow the application provisioning for UML deployment models. On the other hand, TOSCA deployment models can be considered in the light of UML, thereby gaining insights into the components manifested by deployed artifacts and how they are realized from a structural viewpoint. Obviously, model transformations play an important role for not only bridging UML and TOSCA but also refining cloud architecture models towards the target cloud environment.

1.1.4 Maintaining cloud artifact repositories (*RC4*)

The fourth challenge that is addressed by this thesis concerns the efficient re-execution of a changed transformation chain because changes can invalidate previously produced models usually maintained in a central repository. As a result, changes to a transformation chain need to be propagated to existing models. Chaining transformations generally fosters reuse in the sense that existing transformations are loosely coupled by a more coarse-grained transformation [KSW+15].

Such a loose coupling ensures that the engineers still have the control over the execution of the single transformations. The latter is important not only to pause and resume a transformation chain in case of engineers need to intervene, *e. g.,* manual changes need to be carried out on produced models, but also to partially re-execute the transformation chain as a result of changes to one or more transformations. An incremental transformation execution can significantly reduce the runtime overhead particularly when computation-intensive transformations are marginally revised. Moreover, it can ensure that manual changes to existing models prior the transformation re-execution and external references to them via identifiers are retained because the models are only updated instead of entirely re-created.

1.2 Aim of the thesis

This thesis aims at realizing extensions to UML and a corresponding toolset that together form an architecture style for developing cloud applications with a particular emphasis on the automation of development processes and their effectiveness in producing truly useful and usable models. The architectural style proposed in this thesis is grounded in a model-based framework that defines its core building blocks.

Considering Figure 1.1, the framework provides a *cloud library* for representing the *architecture model* of a cloud application from an environment-independent perspective. Modeling concepts of the cloud library are imported by the environment-independent architecture model. Several views on the architectural model are conceivable. For instance, a component view reveals the high-level structure of a cloud application, whereas a class view shows how components are in fact realized. The interconnection of components with the target cloud environment is usually represented by a deployment view. The latter implies already a refinement step [Wir71, MG06] provided that the cloud-specific features injected to the architecture model are grounded in the target environment. Those features are captured by the *cloud profile* of the framework. It collects the cloud services offered by cloud environments, thereby allowing engineers to explicitly represent the allocation of components to them. A model-based refinement of components and their realizing classes towards a platform hosted on top of the target cloud environment is also foreseen by the framework as it enables model transformers to produce platform-specific application code including method bodies even from a structural viewpoint. To accomplish the refinement towards a platform, dedicated *target platform libraries* and *profiles* are employed for capturing platform-specific features on the model level. The *refined architecture model* embodies the result of the transition from the platform-independent level to the platform-specific level. The differentiation between a platform-independent model (PIM) and a platform-specific model (PSM) has in fact been introduced by MDA[7] initiative of OMG[8]. It essentially proposes a staged-transformation process from a PIM to a PSM. In addition to the PIM and PSM, MDA defines also a computation-independent model (CIM) which is typically used to provide the context in which the system is embedded and to analyze high-level system requirements. It provides thus a business-centric view on the system, which is however out of the scope of this thesis.

[7] Model-Driven Architecture (MDA): www.omg.org/mda
[8] OMG: http://www.omg.org

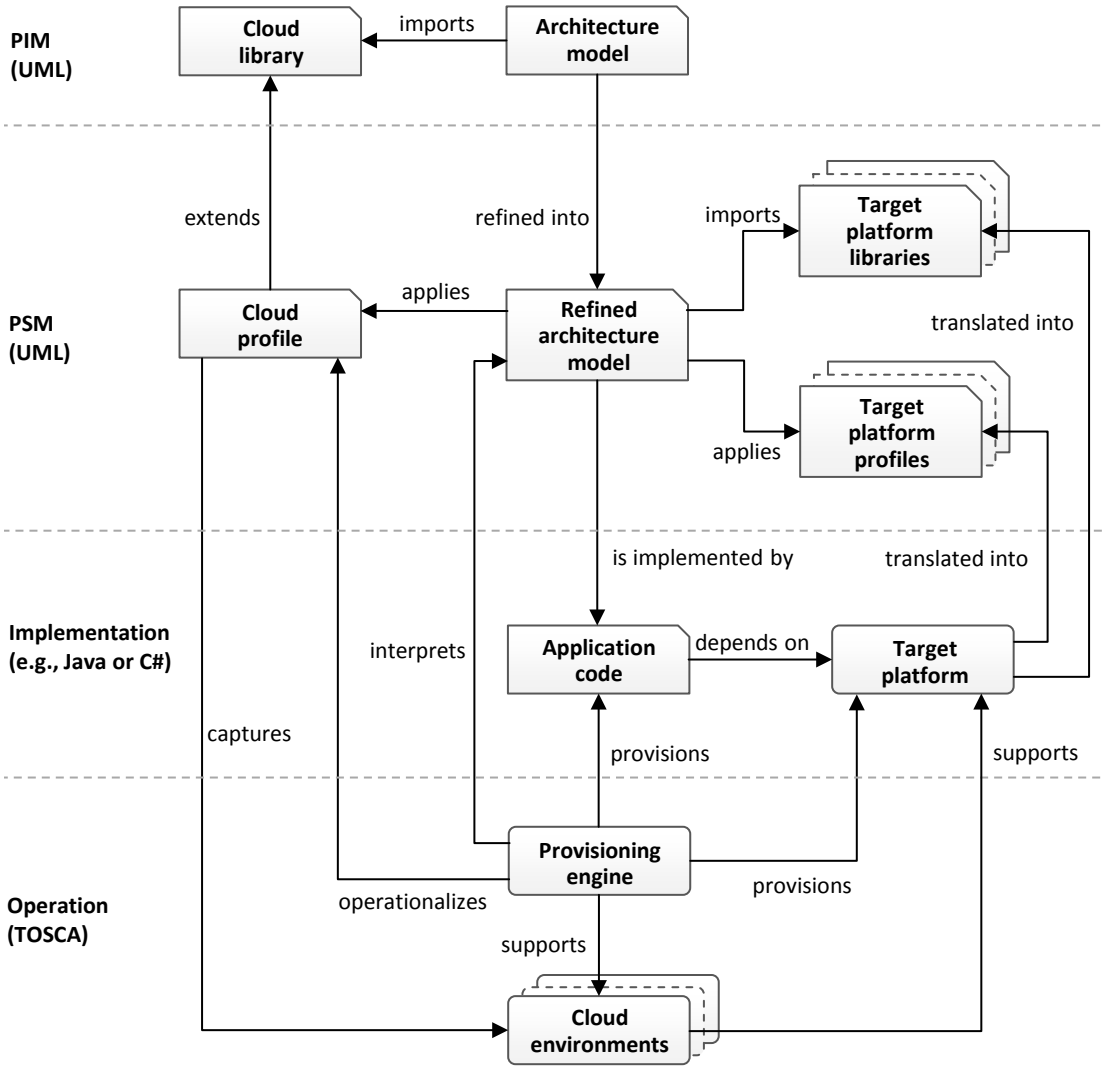

Figure 1.1: Conceptual framework

The PSM is the basis for developing the *application code* which obviously depends on the *target platform* already specified on the model level. To what extent the application code can be automatically generated from the refined architecture model heavily depends on the capabilities of the employed model transformers, the platform-specific features captured on the model level, and which aspects of an application are in fact modeled. As the refined architecture model represents the desired state of the application provisioning in terms of the deployment view, it is considered as input to the *provisioning engine* for enacting the respective provisioning process. The provisioning engine requires access to all implementation-related artifacts that must be transferred to the target cloud environment. It executes the provisioning of the deployable artifacts from the implementation layer and the services of the target cloud environment as specified by

the deployment model. In this way, the provisioning engine operationalizes the cloud services captured by the cloud profile on the model level.

It is important to note that the proposed framework does not impose any process how and in which order the particular artifacts from the various layers must be produced. Rather it shows how its constituting building blocks relate to each other which also reveals that all the introduced layers exploit model-based artifacts expressed either in UML or TOSCA. This emphasizes the importance of models and model-based techniques, especially model transformation, for the architectural style proposed by this thesis. Model transformation is considered as a key enabler to automate the generation of artifacts from different layers. For instance, target platform libraries and profiles on the model level can automatically be generated provided that an appropriate model transformer is available. Generating libraries and profiles for a certain platform may even be carried out independent of a concrete application scenario but solely for collecting them in a central repository to make them available.

The following four main research questions (*RQ1–RQ4*) are going to be answered by this thesis.

RQ1: *What are the main modeling concepts for the cloud computing domain and how do current cloud modeling languages (CML) differ from UML?*

RQ2: *What are the extension points of UML for integrating cloud-specific domain modeling concepts and how can they be exploited to represent architectural models of cloud applications?*

RQ3: *What are the language correspondences between UML and TOSCA and how can the provisioning of UML deployment models be automated based on TOSCA?*

RQ4: *What are adequate techniques to maintain cloud modeling artifacts produced by transformations and how can those artifacts be automatically co-evolved with changes to transformations?*

In total, four scientific contributions are going to be presented by this thesis to answer the posed research questions.

1.2.1 Scientific contribution 1: Systematic review of cloud modeling languages

We systematically review the diverse features currently provided by existing CMLs with the goal to support engineers in selecting the CML which fits the needs of their application scenario and setting and distill a core set of modeling concepts inherent to all of them, see *RC1*. Furthermore, we discuss how CMLs differ from architecture description languages (ADLs) because the influence of ADLs on current CMLs is obvious. In order to classify and compare existing CMLs we present a relatively concise framework with a main emphasis on their modeling capabilities and the toolset which comes with them. As the systematic review of CMLs presented in this

thesis follows the guidelines of Kitchenham and Charters [KC07], we present the detailed process according to which the review was conducted. The gained results of this systematic review and the experiences and needs of several research projects [BRF+15], ARTIST [BBC+13], MODA-Clouds [ANM+12] and PaaSage [JHS13] form the basis of developing cloud-specific extensions to UML.

1.2.2 Scientific contribution 2: Cloud application modeling

In order to support a flexible cloud-specific refinement process from high-level architecture model down to a concrete implementation of it, we propose the *Cloud Application Modeling Language (CAML)* accompanied with a toolset to render it useful, see *RC2*. It consists of a set of libraries and profiles dedicated to target platforms supported by cloud environments (see Figure 1.1), where the focus is on the Java platform.

CAML's toolset is capable to automatically generate UML libraries and profiles from existing Java libraries. Whereas the structural aspects of a Java library are directly captured by a corresponding UML library on the model level, for Java libraries which embrace annotations a UML profile is generated in addition. As this necessitates overcoming existing heterogeneities that, *e. g.,* refer to the target specification of Java annotations and other peculiarities of how Java annotation types are declared, CAML provides a conceptual mapping between UML's profile language and Java's annotation language. As a result, it enables the generation of specific stereotypes for corresponding annotations and a dedicated stereotype for the library itself, which in turn leverages library-specific profiles that are intended to be applied in the context of class and component modeling. The tool set of CAML provides extensions to the Eclipse UML generator for Java, such that stereotypes of library-specific UML profiles are translated into corresponding annotations. Furthermore, it provides transformation chains that exploit library-specific profiles to produce application behavior, *e. g.,* method bodies of CRUD operations for entities that are persisted.

When turning the focus from target platforms to the cloud environments underneath, CAML provides a set of profiles that capture cloud services offered by well-known cloud environments such as Amazon AWS, Google Cloud Platform, and Microsoft Azure. They are collected in terms of a common cloud profile which is intended to be applied for refining an environment-independent deployment model towards a selected target cloud environment. The deployment model is considered as part of the overall architecture of a cloud application. In addition to the environment-specific profiles, CAML also provides a cloud library which provides abstractions over services offered by cloud environments. Combining the library approach with the notion of environment-specific UML profiles results in a powerful cloud modeling solution. The cloud library is used to model environment-independent deployment models, whereas the environment-specific profiles are applied to those models for accomplishing the refinement. The applied profiles allow hiding details of the target cloud environments from the deployment models and dynamically switching between them by (un-/re-)applying the respective profiles. The cloud library along with the profiles specific to cloud environments is exploited by CAML's model transformers to produce environment-specific deployment descriptors usually required for configuration purposes of cloud services.

1.2.3 Scientific contribution 3: Cloud application provisioning

We propose a conceptual mapping between UML and TOSCA as a basis for a fully automatic transformation between the two languages, see *RC3*. In this endeavor, we address both the intensional and extensional level [Küh06] of deployment models as the deployment viewpoint is provided by both languages. Obviously, this favors its use in the mapping process. Moreover, the cloud-specific extensions to UML provided by CAML are addressed in the mapping process. They enable the generation of meaningful TOSCA models from models represented in UML as the cloud-specific features are injected via CAML's cloud library and profile. In this sense, CAML can also be considered as a bridge between UML and TOSCA, which explains why we called our approach CAML2TOSCA. Considering the conceptual framework in Figure 1.1, the refined architecture model expressed in UML is considered as input to the CAML2TOSCA model transformer. It produces a TOSCA standard compliant model which can be interpreted by TOSCA-based provisioning engines, such as OpenTOSCA [BBH+13]. We show how the CAML2TOSCA model transformer can be integrated into a tool chain leveraging architecture modeling for and application provisioning to the cloud.

1.2.4 Scientific contribution 4: Cloud model patching

To enable output models to be co-evolved with changes in transformations, we propose to infer in-place patch transformations from revised out-place transformations for existing output models [BTW14], see *RC4*. A patch transformation propagates changes of a revised out-place transformation to the respective output models without re-creating them from scratch. They are only updated [MG06] according to the changes in the revised transformation by following an in-place execution strategy that enables *model patching*. A patch transformation is automatically produced based on the notion of a higher order transformation [VP04, ALS08, TJF+09], a diff model [KDRPP09] that captures the differences between the two version of a transformation, and a classification of transformation change types with their respective co-changes. Our approach fills the gap between current research on incremental transformations [JE04, HLR06, JT10, RBÖV08, RK12, EKK+13], where changes in input models are propagated to existing output models, and co-evolution of transformations with evolving metamodels, where changes in the metamodels are propagated to transformations [LBNK09, IPM12, GDA12, RIP13].

1.3 Scientific approach

This thesis is carried out according to commonly accepted research methods in the area of software engineering [Sha02, Sha03]. For that reason, we formulate four main research questions that are motivated by practical challenges of realizing innovative techniques and tools in the light of MBE and providing a common ground for bridging software engineering and cloud computing [JAP13]. Table 1.1 categorizes the formulated research questions according the conducted *type of research*, the produced *type of results*, and the *type of validation* carried to provide evidence that research results are sound. As a result, knowledge is yielded by a constructive methodological approach that also adheres to the main guidelines of Hevner *et al.* [HMPR04].

Research question	Type of research	Type of result	Type of validation
$RQ1$	Generalization or characterization	Descriptive model	By instantiation
$RQ2$	Method or means of development	Technique, tool, and language	Methodological evaluation Quality evaluation Performance evaluation Practical relevance by example
$RQ3$	Method or means of development	Procedure and tool	
$RQ4$	Method or means of development	Technique and tool	Quality evaluation Performance Evaluation

Table 1.1: Scientific approach classified according to dimensions of Shaw [Sha02, Sha03]

1.4 Application scenario

The potential benefits of cloud environments are certainly not only appealing in the development and provisioning of new applications, but also in the modernization of existing applications [FH11, LFM+11, ANM+12, BBC+13] hosted on an on-premise environment. In this context, modernization to the cloud refers to the transition of an application from a traditional non-cloud environment to a modern cloud environment, thereby taking advantage of the novel opportunities of cloud environments, *e. g.,* advanced scalable data persistence solutions, to modernize software. A transition typically implies adaptations to application components [ABLS13] because the optimization opportunities of cloud environments can often only be exploited properly if the software complies with the peculiarities imposed by such environments, *e. g.,* the unique datastore solution of the GAE. The need for adapting software induces also the need to understand it [MJS+00, CDPC11], which, in turn, requires representations of the application or at least parts of it in another form or at a higher level of abstraction [CI90, BCDM14]. How such a scenario can be carried out in general is described by Kazman's "horseshoe" [KWC98], which is a theoretical model that captures the main processes of evolving applications. We interpret this theoretical model in the light of an advanced MBE approach [FR13] to introduce a moderately complex application scenario consisting of a reverse engineering and forward engineering phase. UML profiles and libraries play an important role in the application scenario as they enable models annotated with platform-specific and environment-specific information [PBMH12].

By employing UML profiles and libraries, models independent of a platform and environment are refined towards the required target platform and environment. Turning this forward engineering perspective into a reverse engineering perspective, existing applications can be represented as UML models that capture platform and environment-specific information. In a reverse engineering process, this additional information can be exploited to facilitate comprehension [CDPC11], whereas models refined to a target platform and environment pave the way for generating richer application code in a forward engineering process [Sel12].

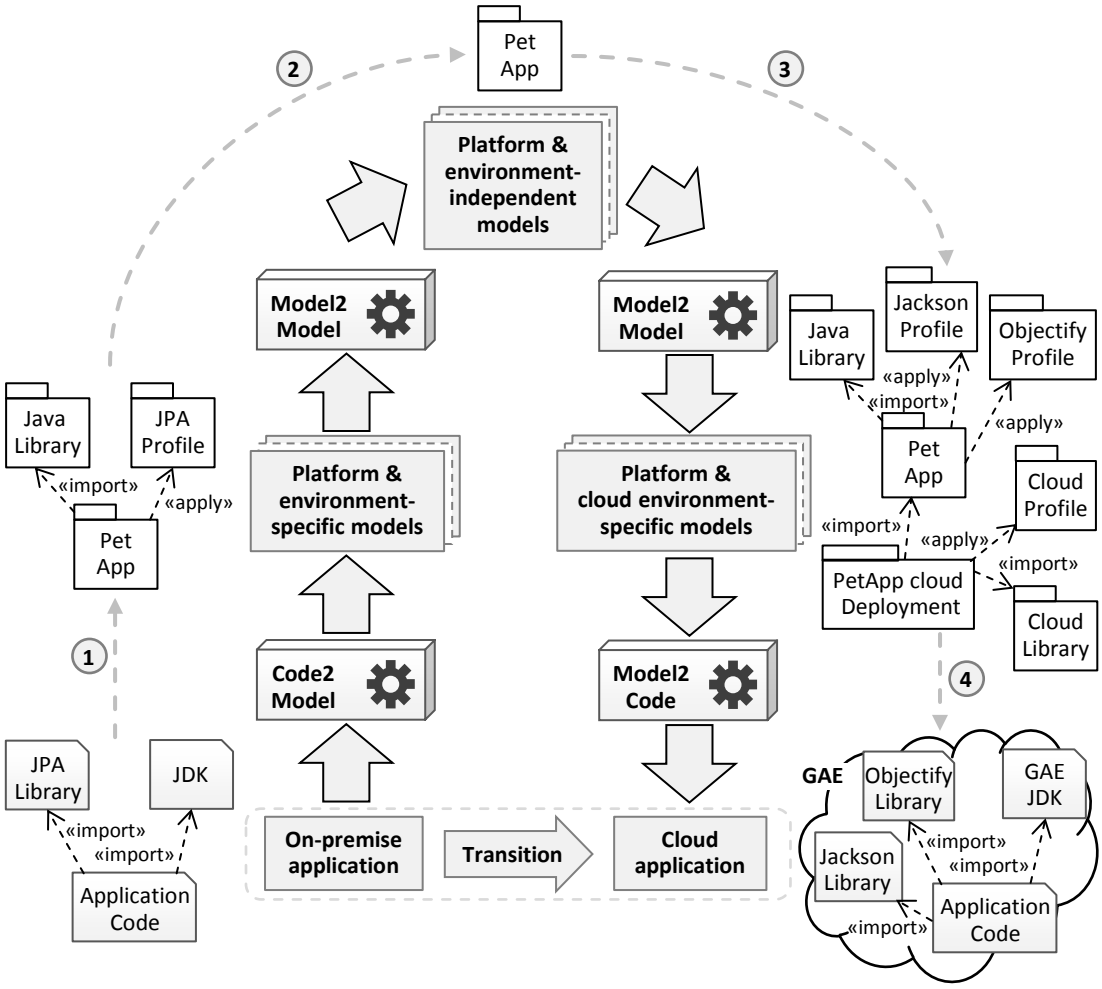

Figure 1.2: Application scenario used throughout the thesis

The inner part of Figure 1.2 gives an overview of combining both processes for realizing the transition of an existing application towards a cloud environment. In particular, we consider a slightly extended version of the well known Java Petstore[9] application (PetApp) which is expected to be hosted on Google's App Engine. The PetApp consists of three deployable components [Szy03]. They refer to the Petstore web application (PetWeb), an administration dashboard (PetAdmin), and support for calculating and visualizing sales statistics (PetStats). Both the PetAdmin and PetStats component depend on the PetWeb component because they require access to the domain classes such as `Order` and `OrderLine`. To demonstrate a concrete adaptation step required to host the PetApp on the App Engine, we focus on the data persistence solution of the original PetApp and how it needs to be changed to meet the peculiarities imposed by of the App Engine's datastore solution. The idea is to replace the JPA-based solution of the

[9]Java Petstore: http://oracle.com/technetwork/java/index-136650.html

original PetApp by a solution based on Objectify[10], thereby realizing a change of the data access platform as typically required for "moving-to-the-cloud" scenarios [ABLS13]. In addition, we demonstrate how a REST-based client can be generated for domain classes managed by Objectify. The presented solution is based on Jackson[11], which provides data processing tools for a variety of formats including the JavaScript Object Notation (JSON)[12].

Now that we have introduced the application scenario, in the following a brief summary is given how it is used throughout the remaining chapters. In Chapter 3, we demonstrate the representational capabilities of existing CMLs according to the PetApp. As some CMLs support reverse engineering processes in addition to forward engineering ones in a similar way compared to CAML, the choice of the "PetApp-to-the-cloud" scenario fits well for this purpose. Then, in Chapter 4 we provide insights into how the transition of the PetApp to the Google App Engine can be accomplished by applying CAML, whereas its provisioning is presented in Chapter 5. In Chapter 7 we again refer to the PetApp for demonstrating the practical relevance of the architecture style for cloud application modeling presented in this thesis.

1.5 Structure of the thesis

This thesis is structured according to its main contributions. In the following, an overview of this thesis is given by briefly describing the elaborated contents of each chapter. As some of the contributions of this thesis are already published in peer reviewed workshops, conferences, and journals, the contents presented in this thesis build on the contents of published research work. The following chapter overview includes information about which contents are already published.

Chapter 2: Preliminaries. In this chapter, we briefly introduce the main characteristics of cloud computing and give a short overview of three major cloud environments: Amazon AWS, Google Cloud Platform, and Microsoft Azure. Thereafter, we discuss the role of UML in the context of architecture modeling by putting emphasis on three modeling viewpoints most relevant for this thesis: (*i*) class viewpoint, (*ii*) component viewpoint, and (*iii*) deployment viewpoint. Finally, we consider how UML can be extended with domain concepts from an language engineering perspective. As UML profiles play an important role in this thesis, we go into detail of how they are defined and applied. In this respect, we also discuss Java's annotation concept in the light of UML profiles.

Chapter 3: Review of cloud modeling languages. This chapter presents the first contribution of this thesis. We systematically review existing CMLs according to four core dimensions of a classification and comparison framework. To satisfy the common guidelines to conduct a systematic literature review, we explain in detail the applied review process. As a result of the CML comparison, not only features of existing CMLs are pointed out for which extensive support is already provided but also in which they are deficient, suggesting a research agenda for the future. Some of these suggestions can obviously be considered as motivation for the scientific

[10] Objectify: https://code.google.com/p/objectify-appengine
[11] Jackson: http://jackson.codehaus.org
[12] JSON: www.json.org/

contributions 2 to 4 of this thesis. This chapter builds on research work in which we investigate some CMLs with an emphasis on their extensional capabilities [BWKG14].

Alexander Bergmayr, Manuel Wimmer, Gerti Kappel, and Michael Grossniklaus. Cloud Modeling Languages by Example. In *Proc. of Intl. Conf. on Service-Oriented Computing and Applications (SOCA)*, pages 137–146, 2014

Chapter 4: Cloud application modeling. This chapter introduces CAML, which is the second main contribution of this thesis. In the first part of this chapter, we give an overview of CAML's core building blocks and provide insights into how custom UML types can be extended via stereotypes. The generation of UML profiles from Java-based target platforms is presented in the second part of this chapter, while the third part deals with the development of cloud-specific extensions to UML. This chapter builds on research works in which we present a conceptual mapping between Java and UML [BGWK14b, BGWK15, BGWK16] and demonstrate a fully automatic transformation chain on top of it [BGWK14a]. Moreover, this chapter relies on research work in which we introduce some of CAML's modeling capabilities [BTN+14].

Alexander Bergmayr, Michael Grossniklaus, Manuel Wimmer, and Gerti Kappel. JUMP – From Java Annotations to UML Profiles. In *Proc. of Intl. Conf. on Model Driven Engineering Languages and Systems (MoDELS)*, pages 552–568, 2014

Alexander Bergmayr, Michael Grossniklaus, Manuel Wimmer, and Gerti Kappel. UML Profile Generation for Annotation-based Modeling. In *Proc. of Software Engineering & Management: Multikonferenz der GI-Fachbereiche Softwaretechnik (SWT) und Wirtschaftsinformatik (WI)*, pages 101–102, 2015

Alexander Bergmayr, Michael Grossniklaus, Manuel Wimmer, and Gerti Kappel. Leveraging Annotation-based Modeling with JUMP. *Software and Systems Modeling*, 2016. to appear

Alexander Bergmayr, Michael Grossniklaus, Manuel Wimmer, and Gerti Kappel. Bridging Java Annotations and UML Profiles with JUMP. In *Proc. of Demo Track of Intl. Conf. on Model Driven Engineering Languages and Systems (MoDELS)*, pages 1–5, 2014

Alexander Bergmayr, Javier Troya, Patrick Neubauer, Manuel Wimmer, and Gerti Kappel. UML-based Cloud Application Modeling with Libraries, Profiles, and Templates. In *Proc. of Intl. Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE)*, pages 56–65, 2014

Chapter 5: Cloud application provisioning. In this chapter, the third contribution of this thesis is presented. It is concerned with the conceptual mapping between UML and TOSCA, where CAML is exploited as the bridge between the two languages. We give a brief introduction of TOSCA's core language concepts and clarify how the intensional and extensional modeling levels introduced by UML and TOSCA relate to each other. Thereafter, we present the language correspondences based on which we implemented a model transformation that automates the

translation of UML models into TOSCA models. We discuss this model transformation in the light of an application provisioning process that starts with a high-level architecture model represented in UML. This chapter builds on research work in which we propose an approach for combining UML and TOSCA [BBK+16]. Moreover, it builds on a transformation chain capable to produce MOF[13]-based metamodels from XML Schemas [NBM+15].

Alexander Bergmayr, Uwe Breitenbücher, Oliver Kopp, Manuel Wimmer, Gerti Kappel, and Frank Leymann. From Architecture Modeling to Application Provisioning for the Cloud by Combining UML and TOSCA. In *Proc. of Intl. Conf. on Cloud Computing and Services Science (CLOSER)*, 2016. to appear

Patrick Neubauer, Alexander Bergmayr, Tanja Mayerhofer, Javier Troya, and Manuel Wimmer. XMLText: From XML Schema to Xtext. In *Proc. of Intl. Conf. on Software Language Engineering (SLE)*, pages 71–76, 2015

Chapter 6: Cloud model patching. This chapter is concerned with the fourth contribution of this thesis. It presents patch transformations capable of co-evolving existing models with changes in the model transformation that produced those models. We give an overview of different co-evolution scenarios in the context of model transformation evolution and show how incremental transformation execution can be exploited for efficiently maintaining models repositories in a non-invasive manner. This chapter builds on research work in which we present the approach underlying patch transformations [BTW14].

Alexander Bergmayr, Javier Troya, and Manuel Wimmer. From Out-Place Transformation Evolution to In-Place Model Patching. In *Proc. of Intl. Conf. on Automated Software Engineering (ASE)*, pages 647–652, 2014

Chapter 7: Evaluation. In this chapter, the evaluation of the artifacts developed in the course of this thesis is presented. We investigate the methods of current UML modeling tools to deal with platform and environment-specific extensions and compare them to CAML's methodological approach. As CAML is capable to automatically generate UML profiles from Java-based libraries, we report on their quality with respect to UML profiles used in practice. Moreover, we discuss its scalability for relatively large programming libraries. Furthermore, we evaluate the quality and performance of model patching by means of model transformations developed in the course of the ARTIST project and applied to its use cases. Finally, in order to demonstrate the practical relevance of the proposed architecture style for cloud application modeling, we report on its application in the context of of a modernization scenario to the cloud.

Chapter 8: Conclusion. In this chapter, the four main contributions of this thesis are briefly summarized and open challenges to be addressed in future work are outlined.

[13] Meta Object Facility [OMG11b]

CHAPTER 2

Preliminaries

To provide the necessary background for this thesis, we introduce *cloud computing* because it is the domain for which we present extensions to UML. In particular, we discuss the different levels of virtualization typically used to distinguish services offered by today's cloud environments. We support this discussion by looking at three major cloud environments whose features were analyzed in the course of this thesis to provide the foundation for the development of cloud-specific extensions to UML.

Some of its peculiarities with respect to deployment modeling are seized and discussed because this viewpoint of UML plays a particular role for the refinement of architecture models towards a target cloud environment. Even though UML's class and component viewpoint are just as important as the deployment viewpoint for this thesis, their use has already been discussed intensively in existing literature. For that reason, the main emphasis is placed on UML's deployment viewpoint when discussing its role in the context of *architecture modeling*.

Finally, we introduce two mechanisms of UML that are useful for extending its metamodel with supplementary types: profiles and libraries. Considering the former, we also discuss its role as general injection mechanism because one goal of this thesis is to bring annotations from programming to modeling. In this respect, we focus on Java's annotation mechanism and current methods of how it can be supported on the model level. Hence, the foundations for leveraging annotation-based modeling are mainly emphasized when we introduce *model-based engineering (MBE)*.

The remainder of this chapter is structured as follows. In Section 2.1, we introduce cloud computing and highlight some of its core characteristics most relevant for this thesis. Then, we discuss UML's capabilities for architecture modeling in Section 2.2. Finally, in Section 2.3, the typical artifacts that occur in MBE are introduced at first. Thereafter, we place emphasis on approaches for extending UML's metamodel.

2.1 Cloud services and environments

In cloud computing, infrastructure resources, such as processing power and storage, platforms, and software, are viewed as commodities that are readily available from large data centers operated by cloud providers. Cloud computing leverages service-oriented architectures to unify elements of distributed, grid, utility, and autonomous computing. One characteristic that sets cloud computing apart from these existing approaches is the *dynamic provisioning* of resources offered by cloud providers as services. Those services are exposed by the providers' cloud environments. Usually they are offered according to a certain quality of service. The quality of a technical service can be expressed in various ways. Often cloud providers define only the availability of their services. From a user perspective, the quality of a cloud service is ideally at least equivalent [VW12] to a non-cloud solution.

Cloud users can provision and release cloud services on demand and pay only for what they have actually consumed. This "pay-as-you-go" principle benefits both the cloud user and the cloud provider. From the user perspective, the risk of under or over-provisioning is avoided as the provisioned cloud services can elastically scale [VRB11] with the user's demand. In contrast, the cloud provider profits from an economy of scale and can offer cloud services at a price that is usually lower than the one of a non-cloud solution [Wal09]. Cloud providers can utilize their resources to capacity by optimizing the work load scheduling of the different co-located cloud users. To achieve this elasticity, *virtualized* resources are virtualized [AFG+09]. Hence, a key concept that enables cloud computing is virtualization. The possible spectrum of virtualization that can be offered by a cloud environment is wide. On one end of the spectrum, compute capacity is virtualized by providing a raw virtual machine. Since compute services constitutes virtualization at a relatively low level in the computing stack, they provide little management and maintenance. Therefore, cloud users of such services are, for example, responsible to provide the complete software stack from the operating system upwards, configure it, and keep all the components of the stack up to date. On the other end of the spectrum, the virtualization comprises in addition to low level compute capacity a fully-managed software stack that is inherently tailored to a certain type of applications. Such a software stack may even provide complete and ready-to-use applications or a collection of general components that can be assembled into an end-user application without programming by branding, customization and configuration, similar to mash-ups in the context of web services.

In theory, the spectrum of virtualization is continuous and all possible trade-offs are imaginable. In practice, however, cloud services have converged to three rather discreet points on this spectrum. They are commonly referred to as *Infrastructure-as-a-Service (IaaS)*, *Platform-as-a-Service (PaaS)*, and *Software-as-a-Service (SaaS)*, where IaaS provides low-level virtualization and less management and SaaS provides high-level virtualization and more management, while PaaS resides in between the two. Figure 2.1 provides an overview of the three levels of virtualization including examples of cloud services for each level.

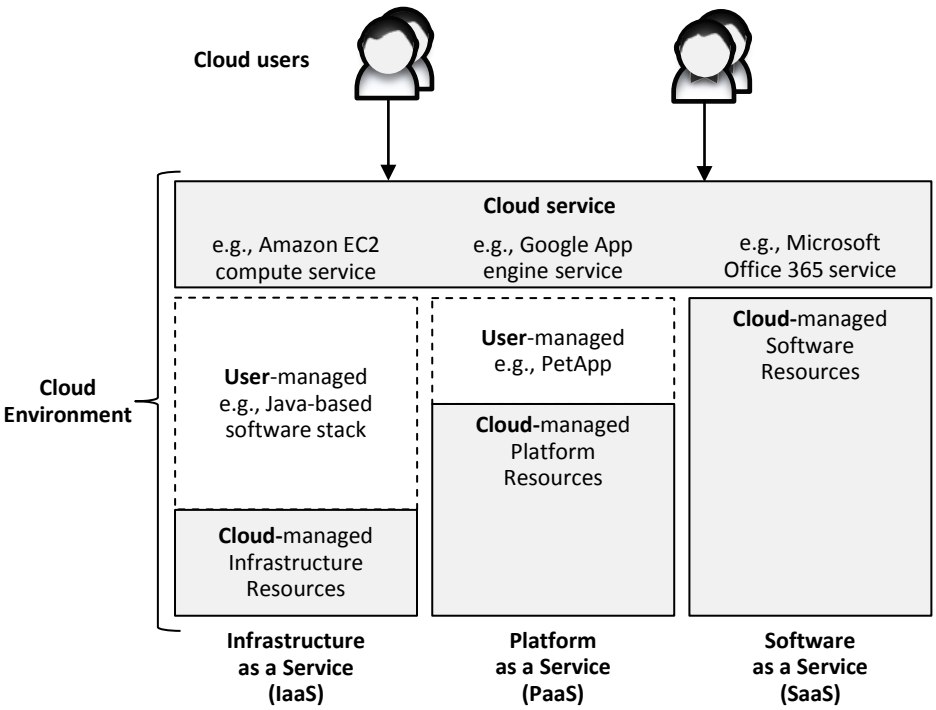

Figure 2.1: Levels of virtualization in cloud computing

Infrastructure-related cloud services are typically concerned with the provisioning of compute units consisting of a processor, memory and local storage. Those compute units can be networked by defining their addresses and the connections between them. For instance, Amazon's EC2 service offers virtualized compute units that look in fact much like physical hardware. Another option that is commonly available on the infrastructure level is shared storage in the form of a basic block storage, a database, or a scalable data store. Such elements can be found in the services of most major cloud environments. In the case of Amazon, for example, block storage is offered as part of EC2, while their Simple Storage Service (S3), SimpleDB, and DynamoDB provide scalable data management. The latter are different cloud storage services that can be accessed via dedicated APIs.

Cloud services on the platform level provide a pre-configured and managed execution environment in terms of tools, supported programming languages, and available APIs. Often those services replicate execution environments in the cloud that are already commonly used in practice, such as web servers, web application servers, or database servers. For instance, Microsoft Azure is a mixture between an infrastructure and a platform cloud environment. On the one hand, it is possible to work with bare Windows or Linux virtual machines, similar to Amazon EC2. On the other hand, Microsoft Azure also provides cloud services for media delivery, web sites and web applications, which are realized using byte-code virtualization based on the .NET framework. Additionally, two types of data management services are available. The SQL Database service is based on Microsoft SQL Server and supports tables and BLOBs, whereas the Big Data service is based on Apache Hadoop. In contrast to Microsoft Azure, Google's App Engine virtualizes

a complete software stack that targets automatic scalable web applications that require high availability. It is proprietary to Google, but provides application programming interfaces that are similar to well-known frameworks. For example, their Datastore API is similar to JDBC and supports database access through an SQL-like language called GQL. The Google cloud ecosystem further offers a series of services for data management and data analysis. Google Cloud Storage is similar to Amazon S3, whereas the key-value datastore of Google's App Engine is similar to DynamoDB. For the on-line transaction processing (OLTP) use case, Google offers Cloud SQL, which is based on MySQL with synchronous replication and support for joins. For the off-line analytical processing (OLAP) use case, clients can use Google Big Query which is based on Cloud Storage and can evaluate MapReduce as well as SQL tasks without joins.

Finally, software-related cloud services operate on the highest level of the three types of cloud computing platforms. Examples of ready-to-use cloud software include Microsoft Office 365 and Google Drive. Platforms such as Force.com are examples of platforms that can be adapted at a very high-level to build end-user applications.

Orthogonal to the classification of cloud environments into infrastructure, platform, and software, they can be distinguished according to their availability to the public [MG11]. In the case of a public cloud environment there is a clear distinction between an organization or company that acts as the cloud provider and other organizations, companies, or individuals that use the offered services. We refer to the latter as cloud users. In contrast, a private cloud is used and operated in-house by the same company or organization and hence not available to the public. Hybrid clouds are composed from several different cloud services, regardless of whether they are private or public. There are many use cases in which companies or organizations opt for using a hybrid cloud. Security considerations, for example, might require that sensitive data are managed and processed in a private cloud, while data that are not restricted can be handled by a public cloud. Another reason to use multiple cloud environments might be the different capabilities of cloud services [Pet14]. For example, companies may try to split their data management between several cloud environments for the purpose of combining the best available transactional and analytical processing. Finally, if several organizations have common concerns or goals, they can share an environment as a community cloud. As a consequence, a community cloud is a mix of a private and a public cloud environment.

2.2 Architecture viewpoints

Already since the late 1980s architecture models are used to capture and communicate the high-level structure of an application along with the behavior of structural elements as specified in the interactions among them [BRJ05]. For that reason, a plethora of languages starting with box-and-line notations to describe applications in terms of components and connectors have been proposed since then. Today, UML is a well-adopted language for modeling the architecture of applications from various viewpoints [LMM+15]. Each model view delivers a certain perspective on the application architecture, thereby addressing the different concerns of involved stakeholders. This thesis is mainly concerned with three viewpoints for modeling the architecture of a cloud application: (*i*) class viewpoint, (*ii*) component viewpoint, and (*iii*) deployment viewpoint. How these viewpoints can be supported by UML models is demonstrated in Figure 2.2.

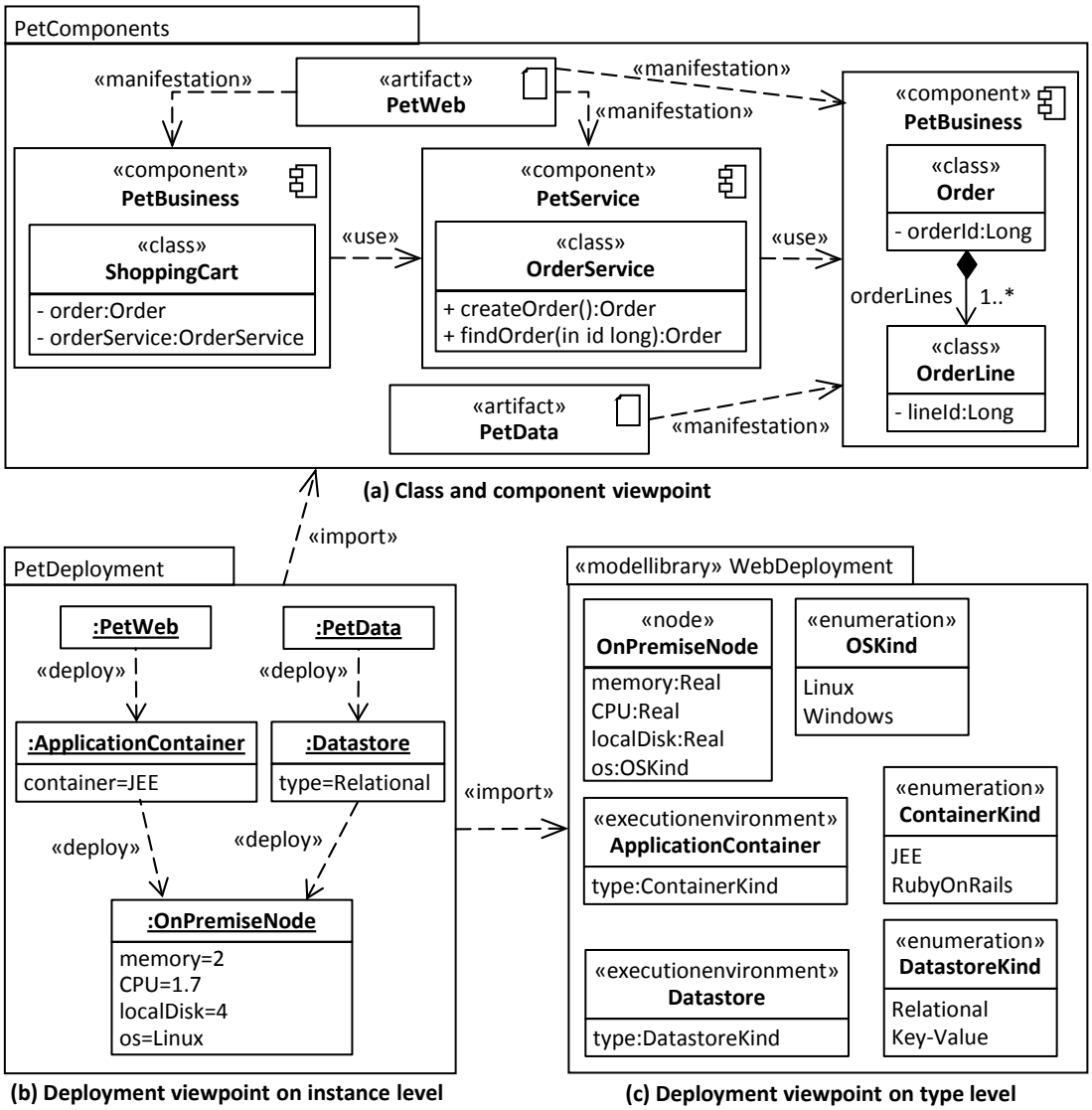

Figure 2.2: Demonstration of class, component and deployment viewpoint

It shows three sub-components (PetBusiness, PetService and PetData) of the PetWeb component and an excerpt of their realizing classes (see Figure 2.2(a)). The components are manifested by deployable artifacts instead of directly allocated to a deployment target. This additional level of indirection allows the manifestation of one or more components into an artifact that refers to the actual implementation of the manifested components. For instance, a "JAR file" is a deployable artifact in the context of Java-based web development. Artifacts are allocated to respective deployment targets that are capable to interpret or execute them (see Figure 2.2(b)). In fact, the PetWeb artifact is allocated to a Java-based application container which in turn is deployed on a node with certain (virtual) machine characteristics. Similarly, the PetData artifact

is allocated to relational DBMS that runs on a separate node. Both nodes are connected to each other to allow communication between the various artifacts and targets comprised by the deployment model. All these artifacts and targets are instances of the custom types created by the component model (see Figure 2.2a) and the common web deployment model (see Figure 2.2c), respectively. The latter provides the types of the deployment targets instantiated by the concrete deployment model of the PetApp. A deployment model on the type level describes all possible deployment configurations. In contrast, a deployment model on the instance level captures a concrete deployment configuration for a certain application. Engineers that model concrete deployment configurations instantiate the types of the common web deployment model and assign values to the features provided by those types. For them, the differentiation between *ontological* and *linguistic typing* [AK07] is irrelevant. They can directly benefit from the fact that custom types or even hierarchies of them can be established without modifying the UML metamodel. However, from a language engineering perspective, it is important to understand which meta-classes are in fact instantiated, how are custom types assigned to created instances, and what are the implications of the two typing dimensions. As this thesis deals with developing cloud-specific language extensions to UML, the differentiation between ontological and linguistic typing is briefly discussed by means of the deployment viewpoint.

Figure 2.3 captures UML's meta-classes relevant for deployment modeling. The meta-class `InstanceSpecification` is used to model artifacts and targets of a concrete deployment configuration. It inherits from the meta-classes `DeploymentTarget` and `DeployedArtifact`, which are the member ends of the `Deployment` meta-class. As a result, instances of the meta-class `InstanceSpecification` can be related via a deployment relationship. Considering the deployment configuration in Figure 2.2(b), all the artifacts and targets are instances of the meta-class `InstanceSpecification`. The relationships between them (annotated by «deploy») are instances of the `Deployment` meta-class. Meta-classes of the UML metamodel enable linguistic typing. In order to assign a custom type to an instance of `InstanceSpecification`, ontological typing is required. In UML, the `classifier` property of `InstanceSpecification` allows custom types to be linked to instances. For instance, the PetWeb artifact defined in the component model of Figure 2.2(a) is the ontological type of the respective artifact instance in the deployment configuration of Figure 2.2(b). Even though an instance may have both a linguistic type as well as an ontological type, in UML they are treated differently. Linguistic typing is realized in the sense of a true "class/object" instantiation known from object-oriented programming. In contrast, ontological typing is solely based on a plain reference between the two members. Thus, the actual semantics underlying this reference, which is in fact also a "class/object" instantiation, is only weakly defined. The different realization of linguistic and ontological typing, the first one may be considered as first-class type, while the latter as a second-class type, has a strong effect on the functionality of UML's extension mechanism as it solely allows linguistic types to be extended by default.

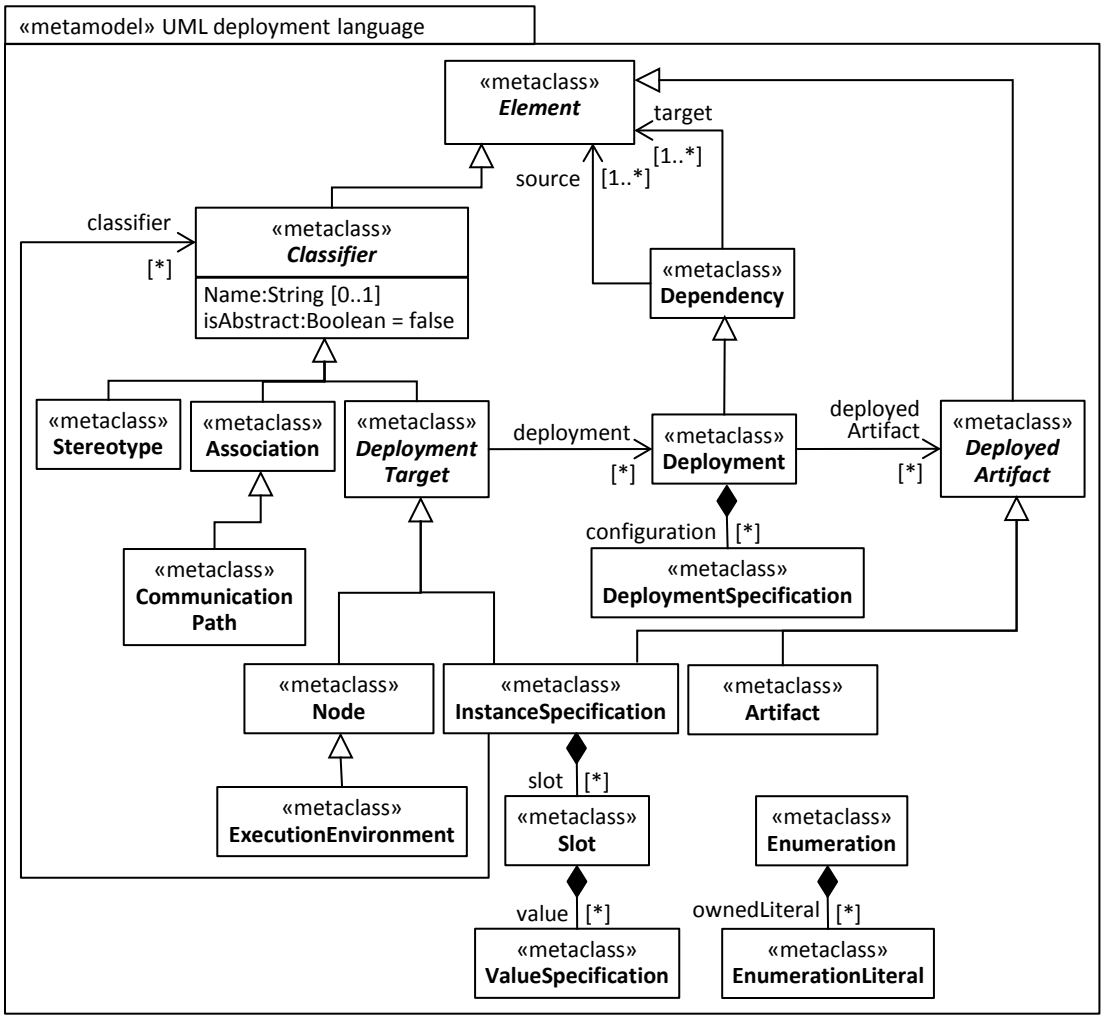

Figure 2.3: Meta-classes of UML deployment viewpoint

2.3 Model-based engineering

Model-based Engineering (MBE) advocates the use of models to raise the level of abstraction and model transformations to increase the degree of automation in software engineering [Sel08, BCW12]. Models are used to represent a certain kind of information, *e. g.,* a model of an application that exists (descriptive) or that should be realized (prescriptive). Transformations enable the manipulation of models in a systematic manner for a given purpose. In this sense, they are the active part in MBE. Following the "everything is a model" doctrine of MBE [Béz05b], transformations are in fact also models. This general notion of a model brings several benefits to MBE. For instance, in case transformations are subject to manipulation, considering them in terms of a model is extremely useful. As a result, transformations can be passed as input to a transformation and produced from it as output. A *higher order transformation (HOT)* is a model

transformation whose input and/or output are themselves transformations [VP04, TJF+09, ALS08].

Modeling languages that are used to create models and upon which model transformations are defined play an important role in MBE. They are often specified by means of a metamodel. Roughly speaking, a metamodel is a model that captures the abstract syntax of a language. Additional constraints on the syntactical elements provide contextual validation rules for the language. For instance, the UML metamodel along with OCL[1] constraints is one obvious example in this respect. Another example is the TOSCA metamodel which plays also an important role in the context of this thesis.

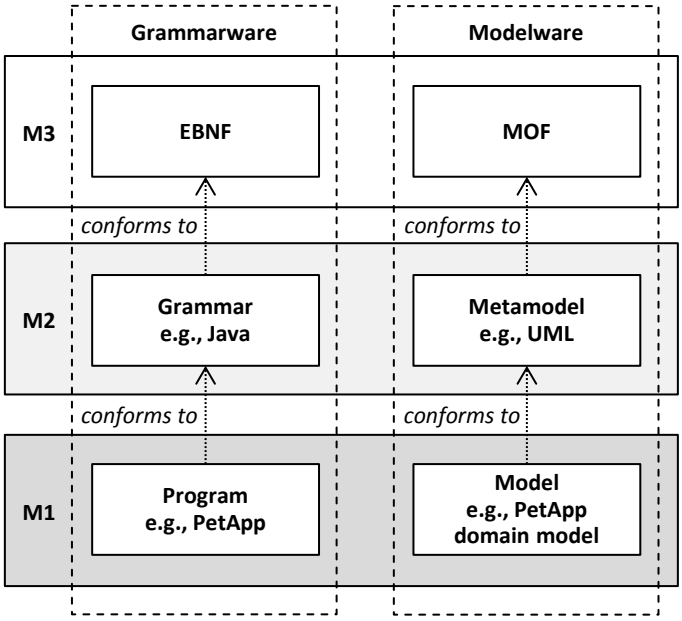

Figure 2.4: Meta-layers of grammarware and modelware

2.3.1 Models, metamodels, and transformations

A model is built up from elements that comprise values stored in attribute slots and references to other elements. The underlying structure of a model is a graph. In the context of MBE, models usually conform to a metamodel. Validating the conformance of a model to a metamodel is of particular interest in scenarios where models are processed. A metamodel is itself a model that defines a potentially infinite set of valid models. Metamodels are often represented by means of UML class-diagram-like notations in combination with OCL to describe the abstract syntax of a modeling language. The same pattern can be applied for validating the conformance of metamodels against the model used to create them. Such a model is standardized by OMG's MOF. For instance, UML is a MOF-based metamodel. It conforms thus to the metamodel specified by MOF. A popular implementation of the essential part of MOF (EMOF) is Ecore, the top-level

[1] Object Constraint Language: http://www.omg.org/spec/OCL

model of EMF[2]. For instance, the Eclipse-based implementation of UML is built on top of EMF, where its metamodel is expressed in Ecore. When considering the different layers of the general meta-architecture introduced by MOF, they show similarities to the meta-layers used in the field of programming language development. The terms "modelware" and "grammarware" are often used to distinguish between the two technical spaces [KLV05, Béz05a]. A valid program conforms to a grammar which is usually expressed in terms of a meta-language. For instance, EBNF is a prominent example of a meta-language. MOF takes a similar role in MBE. Figure 2.4 illustrates the meta-layers of both technical spaces along with concrete examples.

Turning the focus now on transformations, they are indispensable for manipulating models in MBE. Originally, the notion of model transformation has been proposed to automate the transition from models to code. Today, this core technique of MBE is used to automate a multitude of different MDE tasks. Some tasks that are nowadays often solved by (semi-)automated model transformations are summarized in the following.

- **Model refinement**: An abstract model is transformed into a more concrete model
- **Model modernization**: A model is transformed in order to operate in a new target environment
- **Model refactoring**: The internal structure of a model is transformed for the purpose of improvement while the external observable behavior is preserved

The core ingredients of a model transformation are illustrated in Figure 2.5. This pattern [CH06a] describes the systematic transformation of an *input model* conforming to an *input metamodel* into an *output model* conforming to an *output metamodel*.

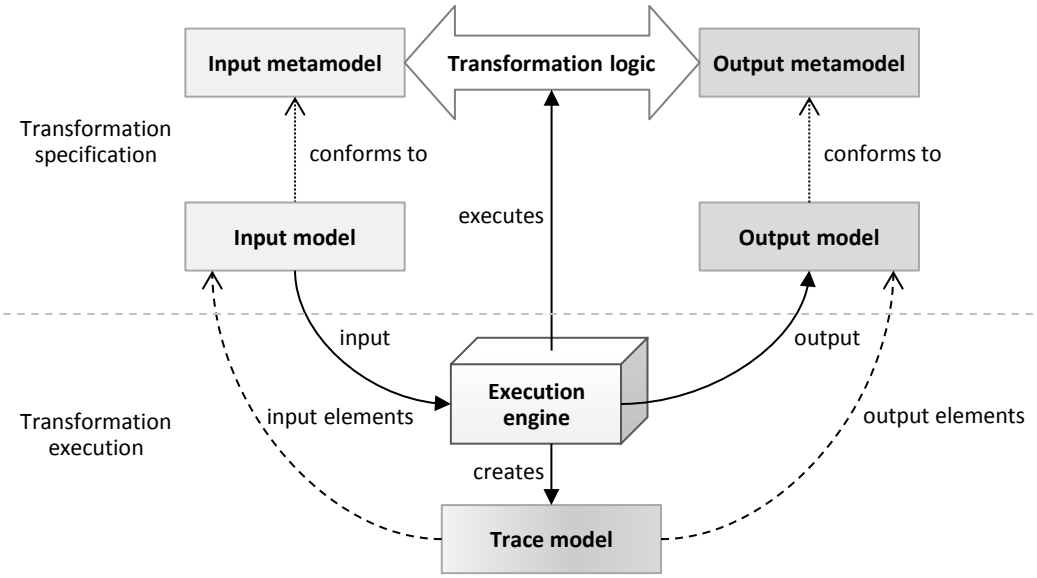

Figure 2.5: Model transformation pattern

[2]Eclipse Modeling Framework: http://www.eclipse.org/modeling/emf/?project=emf

A model transformation is developed based on the metamodels of the models that are intended to be transformed. The source metamodel and the target metamodel need not necessarily be different. For instance, a UML model that is refined into a more concrete UML model is an example of a transformation where both the source and target metamodel refers to UML. The *transformation logic* implemented in some kind of model transformation language is executed by a dedicated *execution engine*. It reads the input models and generates the output models by applying the transformation. In addition, most transformation execution engines are also capable of producing a *trace model*. It connects the input elements transformed into corresponding output elements via trace links. The trace information is especially useful for transformation development, *e. g.,* it can be exploited for debugging and testing purposes. Furthermore, it can be exploited to trace model elements through different abstraction levels and to analyze change impact. In general, model transformations can deal with several input and output models, thus, although the one-to-one transformation case is the most used one, it is only one out of several cases.

As quite diverse MDE tasks are supported by model transformations, several languages different characteristics emerged in the last decade. Most importantly, their underlying paradigm can be classified in declarative, imperative, and hybrid. Furthermore, the execution possibilities of transformations is of major interest. While some languages enable *uni-directional* execution only, others are capable to transform in both directions and to match and synchronize existing input and output models. A prominent hybrid model transformation language used in academia as well as industry is ATL[3] [JABK08].

2.3.2 Profiles and libraries for extending UML

Extensions to UML can be realized by profiles and libraries [Sel07]. A library in UML comprises custom model elements intended to be reused by other models including profiles. For instance, the common web deployment model in Figure 2.2 is defined as a library (annotated by «modellibrary»). It provides a variety of custom types to be reused by a concrete deployment configuration. On the other hand, UML profiles can be considered as general injection mechanisms for varying purposes. For instance, a UML profile is often used to specify variation points of general UML semantics, explicitly document environment-specific design decisions, introduce classifiers in addition to the standard UML classifiers (*i. e.,* it is used as a classification mechanism), and capture platform-specific terminology (*i. e.,* it is used as an annotation mechanism). In this thesis, UML's profile mechanism is primarily used to capture features of selected cloud environments and platform-specific terminology of Java-based libraries. For instance, the UML specification contains a simplified Enterprise Java Beans (EJB)[4] profile to discuss the benefits of the profile mechanism and demonstrate its application.

We use a similar approach to briefly introduce UML's profile mechanism, but turn the focus on the JPA, which is employed by our PetApp example. In fact, we place emphasis on a single JPA concept that is used to annotate persistable domain classes: `Entity`. Its declaration in Java is shown in Listing 2.1. The `Entity` of the JPA is realized in terms of an annotation that can be applied to declared domain classes (see `@Target` annotation which refers to Java types).

[3] ATL: https://eclipse.org/atl

[4] EJB: http://www.oracle.com/technetwork/java/javaee/ejb/index.html

Listing 2.1: Declaration of `Entity` annotation

```
1 package javax.persistence;
2 import java.lang.annotation.*;
3
4 @Target(ElementType.TYPE)
5 public @interface Entity {
6   String name() default "";
7 }
```

Listing 2.2: Application of `Entity` annotation

```
1 package ...;
2 import javax.persistence.Entity;
3
4 @Entity(name = "Order")
5 public class Order {
6   ...
7 }
```

To demonstrate how the declared annotation (type) can be applied, we consider the `Order` class of the PetApp. Placed orders are required to be persisted. Thus, the `Entity` annotation is applied to the `Order` class. Listing 2.2 illustrates the declaration of `Order` class along with the applied `Entity` annotation.

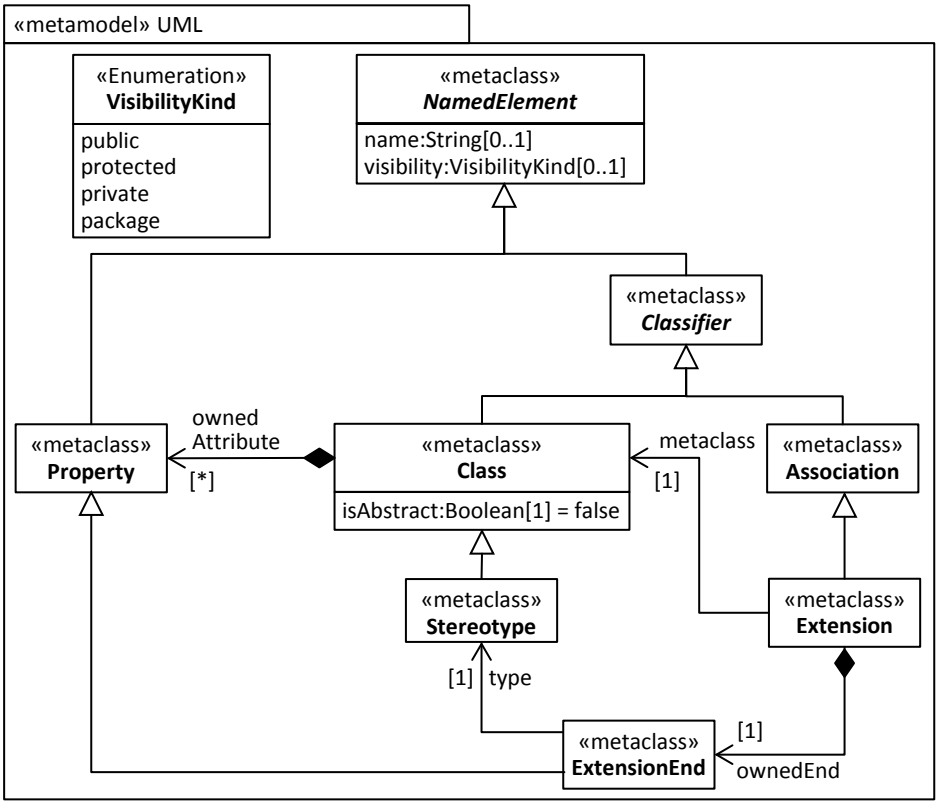

Figure 2.6: Metamodel of UML profiles

Turning now the perspective from Java to UML, we show how the `Entity` concept can be realized by means of a UML profile. With the introduction of UML 2, the profile mechanism has been significantly improved compared to the beginnings of UML [FFV04, Sel07]. In particular, a profile modeling language has been incorporated in the UML language family to precisely define how profiles are applied on UML models and how stereotypes are applied to elements of those models. Figure 2.6 depicts the core elements of UML's `Profiles` package and relates them to the `Classes` package of UML. Several classifiers, relationships, and features are omitted. We refactored some relationships for reasons of comprehensibility [BWRZ13]. For instance, in the standard UML metamodel `Class` inherits indirectly from `NamedElement`, hence we reduced the intermediate meta-classes forming a deep inheritance hierarchy.

A `Stereotype` is a specific meta-class used to extend meta-classes of the UML metamodel. This enables platform-specific concepts to be injected into instances of meta-classes that are extended by a defined stereotype. The `Stereotype` meta-class specializes the meta-class `Class`. Hence, it inherits modeling capabilities such as properties. Similar to `AnnotationTypes`, an instance of a `Stereotype` is identified by a `name` and modified by an optional `visibility` and the mandatory `isAbstract` property. A defined stereotype references the extended meta-classes via instances of the `Extension` relationship. The `Extension` relationship inherits from the `Association` meta-class. As a result, it is a binary relationship with two association ends where both are realized by a `Property`. The property that points to the extended meta-class is contained by the defined stereotype, whereas the extension contains the other association end. It realizes the reference from the extended meta-class back to the defined stereotype. This back reference is represented by the `ExtensionEnd` meta-class, which inherits from `Property`.

Having introduced the core meta-classes of UML's profile langugage, the corresponding UML representations of the declared and applied `Entity` concept are shown in Figures 2.7 and 2.8.

They demonstrate the stereotype application to the `Order` class and the `Entity` definition by a stereotype. Considering the latter, it comprises as expected the string-typed `name` property

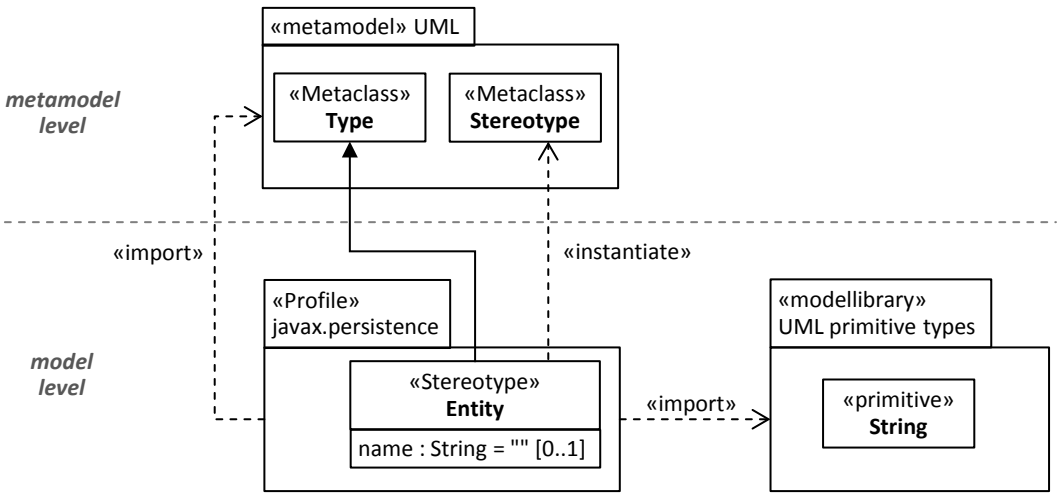

Figure 2.7: Declaration of `Entity` stereotype

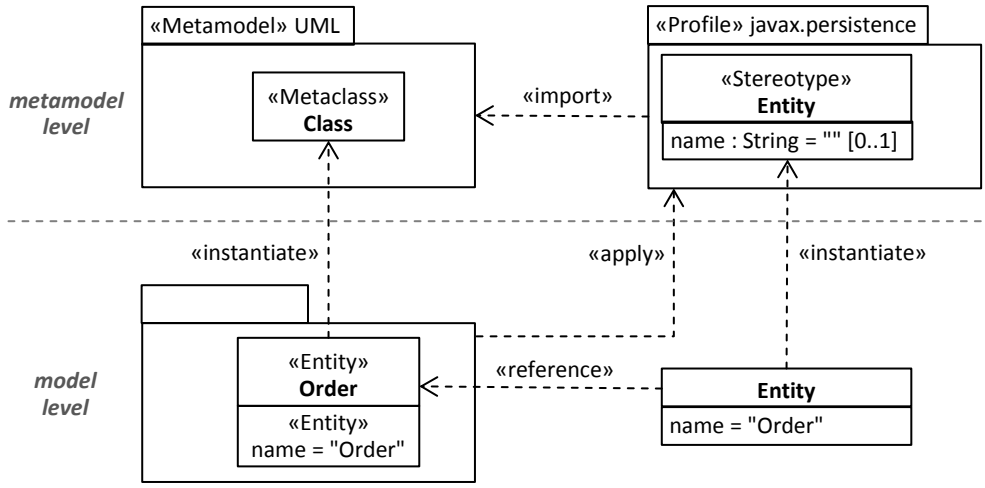

Figure 2.8: Application of `Entity` stereotype

corresponding to the annotation type element of the `Entity` annotation type. UML provides a model library of primitive types, which comprise a `String` type among others. Hence, the primitive types library of UML is imported by the defined UML profile. To ensure that the `Entity` stereotype provides at least similar capabilities as the *Entity* annotation type, the `Extension` relationship references the UML meta-class `Type`. Once the required stereotypes have been defined, they can be applied to instances of the extended meta-class and its sub-meta-classes.

In order to apply the defined `Entity` stereotype to the `Order` class, the profile comprising it must be applied to the `Order`'s package at first. Applying a stereotype means that it is instantiated similar to any other meta-class that is used to create elements on the model level, *e. g.,* the `Order` class which is an instance of the meta-class `Class`. Hence, a declared stereotype can be considered as part of the metamodel level if the focus is on the stereotype application [AKHS03]. A stereotype instance references the element on the model level to which the respective stereotype has been applied. In our example, the `Order` class is thus referenced by an instance of the declared `Entity` stereotype. An excerpt of the serialized PetApp model in Listing 2.3 shows the result of applying the `Entity` stereotype to the `Order` class in serialized form (see line 24 for the stereotype instance).

Listing 2.3: Serialized PetApp model

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <xmi:XMI ...>
3    <!--
4         Excerpt of PetApp model covering the Order entity and the application of
5         the JPA profile.
6    -->
7    <uml:Model xmi:id="_Irs5cOQcEeStU_XOIGwR-w" name="pet-app">
8      <packagedElement xmi:type="uml:Class" xmi:id="_Irs5p-QcEeStU_XOIGwR-w" name="
           ↪Order">
9        <ownedAttribute xmi:id="_Irs5qOQcEeStU_XOIGwR-w" name="id" visibility="
             ↪private">
10         <type xmi:type="uml:PrimitiveType" href="pathmap://UML_LIBRARIES/
               ↪JavaPrimitiveTypes.library.uml#long"/>
```

```
      </ownedAttribute>
    </packagedElement>
    <profileApplication xmi:id="_Irs6L-QcEeStU_XOIGwR-w">
      <eAnnotations xmi:id="_Irs6MOQcEeStU_XOIGwR-w" source="http://www.eclipse.
          ↪org/uml2/2.0.0/UML">
        <references xmi:type="ecore:EPackage" href="platform:/.../jpa2_profile.
            ↪profile.uml#_TEHZ8iWxEeOCj6LiK5WXEg"/>
      </eAnnotations>
      <appliedProfile href="platform:/.../jpa2_profile.profile.uml#
          ↪_TEHZ8CWxEeOCj6LiK5WXEg"/>
    </profileApplication>
  </uml:Model>
  <!--
       Instance of the Entity stereotype. The base_Type captures the id of the
       Order entity to the stereotype is applied.
   -->
  <jpa2.javax.persistence:Entity xmi:id="_Irs6i-QcEeStU_XOIGwR-w" base_Type="
      ↪_Irs5p-QcEeStU_XOIGwR-w"/>
</xmi:XMI>
```

CHAPTER 3

Review of cloud modeling languages

Recently, several CMLs emerged with a complementary set of concepts to represent cloud applications on the model level and to address the diversity of today's cloud environments and the services they offer. Moreover, the TOSCA standard for representing portable cloud applications and supporting their life cycle management were adopted by OASIS in late 2013. There is however still little consensus in the research community on what a CML is, what aspects of a cloud application and the target cloud environment should be modeled by a CML, and which of the currently existing CMLs is appropriate for a particular problem. For example, there are CMLs that place emphasis on virtual machine (VM) configuration required for the provisioning of compute services with custom software stacks (*e. g.,* the approach by Nhan *et al.* [NSJ12]). Others address in addition networking aspects such as custom addressing and segmentation of launched VMs (*e. g.,* CloudNaas [BASS11]). While those languages solely target the infrastructure layer of a cloud environment, there are also CMLs that turn the focus more on the platform layer (*e. g.,* StratusML [HLT11]). Independent of the addressed service layer [BGPCV12], there are CMLs which support the representation of elasticity rules in order to trigger the provisioning of a compute service if a certain threshold is exceeded (*e. g.,* RESERVOIR-ML [CEM+12]) or provide dedicated tools to allow cloud users seeking for compute services that satisfy their requirements in terms of performance and costs (*e. g.,* CloudMIG [FFH13]).

Consequently, there is an urgent need to investigate the diverse features currently provided by CMLs and classify and compare them according to a common framework with the goal (*i*) to support cloud users in selecting the CML which fits the needs of their application scenario, *e. g.,* migration or optimization, and setting, and (*ii*) to investigate language characteristics and concepts as they are of particular relevance for this thesis. As a result of the comparison, not only features of existing CMLs are pointed out for which extensive support is already provided but also deficiencies of current CMLs are identified.

Existing surveys by Papazoglou and Vaquero [PV12] and Sun *et al.* [SDA12] mostly analyze general description languages for service-oriented architectures and low-level formats for resource virtualization with respect to their applicability to cloud computing. Generic service description language can definitely be considered as a source of inspiration of current CMLs as they are

often capable to capture services offered by cloud environments. Existing formats for resource virtualization are exploited by some CMLs for model serialization. As a result, models created by a CML can directly be interpreted by a cloud environment that support the selected format. One of the obvious reasons why most of the current existing CMLs were not considered by the surveys of Papazoglou and Vaquero and Sun *et al.* is that the majority of CMLs emerged around or shortly after they carried out their surveys. In the survey of Silva *et al.* [SRC13] a systematic literature review regarding existing solutions that address the "vendor lock-in" problem in the context of cloud computing is presented, whereas Jamshidi *et al.* [JAP13] conducted an SLR of cloud migration research. However, their surveys do not focus on CMLs.

The systematic literature review on CMLs presented in this chapter builds upon the results of existing efforts. It is further influenced by insights gained from investigating individual CMLs, language concepts relevant in the context of architecture modeling and software modeling, features of current cloud environments, and experiences and needs of several research projects [BRF+15], ARTIST [BBC+13], MODAClouds [ANM+12] and PaaSage [JHS13]. The process according to which the review was carried is grounded in the guidelines suggested by Kitchenham and Charters [KC07]. We discuss how CMLs differ from architecture description languages (ADLs) because the influence of ADLs on current CMLs is obvious. Furthermore, we present a relatively concise classification framework for CMLs with a main emphasis on their characteristics from a language engineering perspective, capabilities to model cloud application from different viewpoints. Moreover, we also investigate the tools which come with them as they render a CML truly usable and useful. Finally, we discuss to what extent existing CMLs are capable to deal with the modernization scenario introduced in Figure 1.1. As we propose in this thesis cloud-specific extensions to UML, we include also UML in this discussion and draw up the phases of the application modernization in which cloud-specific extensions to UML are most relevant and beneficial. Some of the main findings of reviewing current CMLs are summarized in the following.

High diversity in current cloud modeling languages. Current CMLs pursue different sometimes even specific goals, propose hence diverse modeling concepts and show various levels of maturity. At the same time, there is a common theme among them as the majority of CMLs are capable of representing the structure of cloud applications in terms of components and their deployment on cloud services. Still, a well-connected mix of CMLs is currently not available.

Little attention paid to standard modeling languages. Even though general-purpose languages such as UML provide modeling concepts to represent application from a variety of viewpoints, only one CML currently provides extensions to UML. With the relatively recent adoption of the TOSCA standard, it appears even more desirable to align modeling approaches that emerged in the area of cloud computing and software engineering for providing continuous modeling support.

Primary focus of CMLs on design-time aspects. The majority of currently existing CMLs is primarily used for representing design-time artifacts such as application components manifested by deployable artifacts or deployment targets. However, considering run-time aspects on the model level appears also promising for capturing the current status and workload of a certain

provisioned compute service. This run-time information can be exploited for various tasks such as adaptation or optimization.

Considerable set of tools for current CMLs. Current CMLs span a broad spectrum of tools supporting engineers in the design, development, and provisioning of cloud applications. Overall, existing CMLs have placed the greatest emphasis on modeling, generation, and provisioning of cloud applications and the least on analysis and refinement.

Lack of interoperability between CMLs. The exchange of models between CMLs requires currently manual effort for replicating those models in the different languages. Model transformations for an automated model exchange are not available even though scenarios such as application modernization would benefit from a combined set of tools provided by different CMLs.

Heterogeneous language engineering background. From a language engineering perspective, two dominant meta-languages are used to realize CMLs: MOF and XML Schema. Some CMLs are realized by means of a grammar-based approach while others employ meta-language of proprietary language workbenches such as Micrsoft's DSL tools. The heterogeneities imposed by the different meta-languages used to implement them certainly impedes to achieve interoperability between CMLs.

The remainder of this chapter is structured as follows. The framework according to which we classify and compare currently existing CMLs is defined in Section 3.1. In Section 3.2 we present the process that was applied to conduct this systematic literature review, whereas in Section 3.3 we give insights into the results we obtained from classifying and comparing the selected CMLs. In this respect, we also present the obtained results from considering CMLs in the light of application modernization to the cloud with the goal to draw up the phases of the modernization process in which cloud-specific extensions to UML are most relevant and beneficial. A summary of this chapter is given in Section 3.4 before related surveys are discussed in Section 3.5.

3.1 Review framework

The diversity of features provided by todays cloud environments and existing challenges cloud adopters are faced with [BPBB14] has led to several approaches that propose a CML. They have different origins, pursue different goals, and hence provide a complementary and diverse set of language features. Still, a closer study of the set of features they propose and their main purpose shows that there is a common theme among them, which is used as a guide in elaborating our framework to classify and compare CMLs. To establish a thorough framework, the features of individual CMLs and work in the field of cloud computing that discuss core domain concepts were studied. Furthermore, common characteristics of modeling languages were extracted from work in the area of language engineering. Whereas to a great extent, our classification and comparison framework captures categories supported by all or most existing CMLs, we also argue for features that are only supported by a few of them. They have either been identified in the literature as

important to develop cloud applications or have resulted from our own experience gained from participating in the ARTIST project and collaborating with the MODAClouds [ANM+12] and PaaSage [JHS13] projects. Finally, to validate that our framework is of practical relevance, we analyzed features of current cloud environments (*e. g.,* Amazon AWS, Google Cloud Platform, and Microsoft Azure) and concepts of programming libraries (*e. g.,* jclouds[1] and Deltacloud[2]) that provide an abstraction layer on top of cloud environments. In fact, such libraries enable cloud users to connect to cloud environments for carrying out the software provisioning of cloud-based applications. Now that we have discussed how our framework has been developed, Figure 3.1 depicts its main categories and, where appropriate, provides possible manifestations for them. We developed a metamodel for our framework, which enables us to provide a model conforming to this metamodel for the results of each reviewed CML. Providing the framework in terms of a metamodel allows extensions and modifications, which is crucial in a field that is still largely in its infancy.

Considering the *language scope*, we give a concise summary of the pragmatics for each reviewed CML and classify them according to widely accepted categories of cloud environments, *i. e.,* IaaS, PaaS, SaaS, considered as a target. Common *language characteristics* refer to the syntax and semantics of a CML, how it has been realized, and the different kinds of typing mechanisms (*i. e.,* linguistic and ontological) that are supported. The different typing mechanisms seem to be not only relevant from a language engineering perspective but also from an application one as ontological typing allows extensions to a language without manipulating its definition. *Modeling capabilities* of a CML turn the focus on the core domain concepts to model the structure of a cloud application, the services of a cloud environment required to operate a cloud application, and the interconnection between the application and its environment. Cloud applications need to be decomposed into components because the deployment of a cloud application usually enforces to distribute them across a single or even multiple cloud environments [ASLW14, Pet14]. As a result, a CML must support cloud users to model two essential concerns: (*i*) cloud environments in terms of their offered services and (*ii*) cloud application structure in terms of components and their deployment on cloud services. The deployment of application components on cloud services defines their interconnection. Clearly, several other technical concerns, *e. g.,* elasticity, and non-technical concerns, *e. g.,* pricing, are desirable, but not sufficient to argue that a given language is not a CML. At the same, representing an application's structure is not uncommon in the context of architecture modeling [CGL+03]. In fact, architecture description languages (ADLs) provide concepts to model the high-level structure of an application [MT00, MRRR02]. What differentiates now a CML from an ADL? A CML can be considered as an ADL for a particular domain. However, syntactic elements of a CML capture cloud computing vocabulary, which is usually not the case for a general purpose ADL. As a result, the semantics given to a CML is more specific compared to the semantics of general purpose ADLs intended to be applied to arbitrary domains. For instance, the semantics of a CML can be grounded in translators to executable languages or frameworks in the cloud computing context (*e. g.,* Google's App Engine or Orleans of Microsoft[3]) or engines that initiate the provisioning of modeled cloud services

[1] jclouds: https://jclouds.apache.org
[2] Deltacloud: https://deltacloud.apache.org
[3] Orleans: http://research.microsoft.com/en-us/projects/orleans

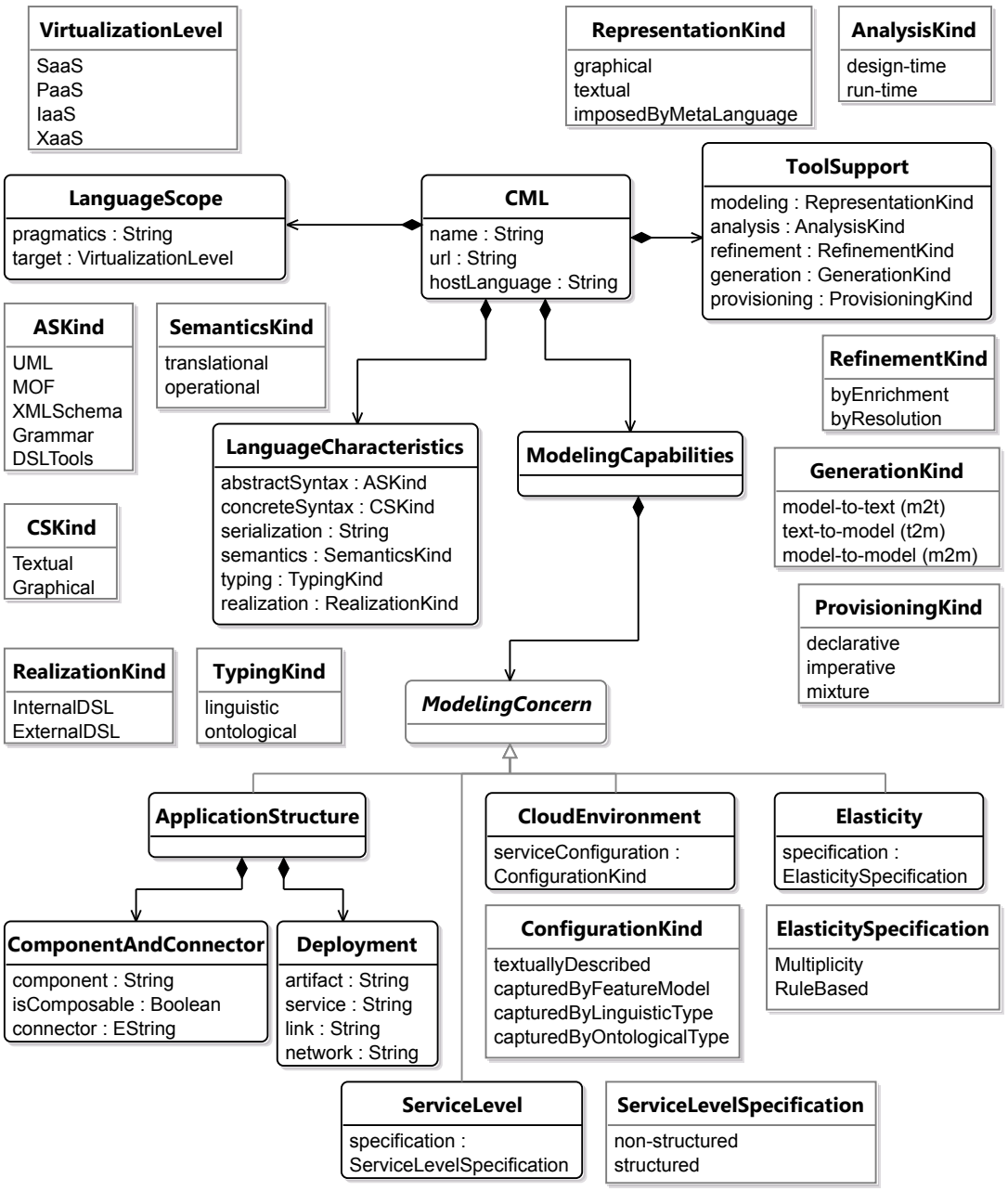

Figure 3.1: Classification and comparison framework for CMLs

including the application components on top of them. The latter motivates the importance of explicitly representing the deployment of a cloud application as it specifies the desired state which triggers the provisioning process. Hence, the semantics given to a language and the two essential capabilities of representing the structure of a cloud application and their deployment on cloud services enable us to determine whether or not a certain language is a CML. Finally, even though the suitability of a CML is independent of whether and what kinds of *tool support* it provides, an accompanying toolset will render a CML both more usable and useful.

3.1.1 Language scope

Pragmatics

The *pragmatics* of a CML refers to its intended purpose including the overall goal that is pursued. The intended purpose of a modeling language can range from sketching software architectures that need to be developed over specifying blueprints for manually realizing application components to creating models for the purpose of generating implementations or directly interpreting or even executing them. Models are not only applied in a generative manner, but more and more they are used analytically in software engineering, *e. g.,* for design-space exploration, optimization, validation, or even for verification. It is worth noting that there is a strong influence of the pragmatics on language characteristics [KKP+09], such as syntax and semantics, and how the language is realized.

Target

Cloud environments considered as *target* of a CML can be differentiated according to the commonly accepted layers of virtualization [AFG+10]: *infrastructure*, *platform*, and *software*. The higher the degree of virtualization is, the more is usually managed by a cloud environment and the less is controlled by a cloud user. For instance, Google's App Engine is a fully-managed platform service in the sense that the application container and the programming language runtime is pre-configured and cannot be manipulated by the cloud user. This means that aside from the application-related artifacts deployed on the App Engine service, the other artifacts related to the platform down to the infrastructure are immutable and controlled by Google. This is certainly of interest to a cloud user in order to select services that operate at the expected virtualization layer. For instance, in the context of software modernization to the cloud, an on-premise environment is partly or even completely replaced by a cloud environment where in practice the typical scenario requires "wiring" both environments [ABLS13]. A concrete scenario may refer to a cloud application whose frontend is hosted on Amazon's platform service Beanstalk, utilizes the software service Google Maps, and connects to a user-controlled MySQL backend system that runs in a virtual machine hosted on an Amazon EC2 infrastructure service. In this work, we investigate the capabilities of CMLs to represent artifacts related to the different service layers: infrastructure-as-a-service (IaaS), platform-as-a-service (PaaS), and software-as-a-service (SaaS). We use the abbreviation "XaaS" to refer to all layers.

3.1.2 Language characteristics

Syntax

The *abstract syntax* of a modeling language defines its concepts and how they relate to each other. It is the common basis of a modeling language since the elements of the abstract syntax are mapped to their *concrete syntax*, *serialization syntax*, and a proper semantic domain. Considering the concrete syntax, it is concerned with the form [Moo09] of a modeling language and defines how abstract elements are realized in a concrete representation. Decorating abstract syntax elements with concrete ones usually increases the readability and intuitive handling of a modeling language. A modeling language may have one or more *textual* or *graphical* syntaxes to represent models. In this work, we investigate solely which kind of notation is provided by a CML. To persist or interchange models, they are encoded according to the *serialization syntax* of the modeling language.

Semantics

The *semantics* gives meaning to the syntactic elements of a modeling language. Most definitions of semantics are functions that map the abstract syntax elements of one language onto elements of a well-understood formal semantic domain, where the degree of formality may range from plain English to rigorous mathematics [HR04]. Defining the semantics of a modeling language is far from trivial as it involves a decision about a proper semantic domain, a mapping from valid syntactic elements to a selected semantic domain [HR04], and the finding of an agreement between stakeholders thereon. Therefore, most modeling languages do unfortunately not have a rigorously defined semantics that goes beyond natural language specifications, even though it is in general a undisputed requirement for the definition of a modeling language. In particular in the light of the growing number of domain-specific languages, this requirement becomes even more important. In practice, however, a useful approach is to implement transformers that *translate* models of a given language into models of a commonly understood or executable language such as Java or C#. Another approach is to implement an interpreter that directly *operates* on the models. In the context of CMLs, a model-based provisioning engine is a concrete example of an interpreter. It is important to note that one could implement for a single CML more than one provisioning engine that may behave differently. This raises the question whether or not a provisioning engine should be considered as part of a CMLs semantics definition. However, a provisioning engine gives meaning to modeled cloud services in the sense that it relates them to concrete services of a cloud environment once the provisioning process has been enacted.

Typing

As pointed out by [AK07], two different kinds of classification mechanisms need to be considered in developing modeling languages. *Linguistic* classification refers to the commonly accepted approach that a user-defined model (token model [Küh06]) is directly expressed by instantiating types which define a modeling language, *e. g.,* in terms of a metamodel. Linguistic types determine which models are valid instances of a modeling language definition. However, to support engineers to create custom types or even hierarchies of them without modifying directly

the modeling language definition, the notion of *ontological* classification has been introduced. Ontological types can be considered as extensions to a modeling language even though in contrast to linguistic types they are not grounded in the language definition. Instead they are defined by means of (linguistic) types of the modeling language and often provided in terms of a custom type library to foster their reuse across different application scenarios. As a result, ontological types may capture vital features that are however relevant in a specific context only and thus lifting them to a linguistic type appears unfavorable[4]. This is certainly of relevance for CMLs. Current cloud environments offer a considerable set of diverse services including processing power usually provided by virtual machines that can be provisioned on-demand. Different virtual machine types may be defined in terms of ontological types while keeping the language definition unchanged. Furthermore, they may capture not only common features but also vital peculiarities imposed by a cloud environment such as the availability zone in which a particular virtual machine instance must be provisioned and the operating system hosted by it. For instance, these features are required to provision a virtual machine in Amazon's cloud environment, whereas in the context of the Google App Engine the operating system of a virtual machine is pre-defined and the distribution of virtual machine replicas is automatically accomplished without granting custom configurations.

Realization

Two different approaches are common for *realizing* a modeling language in general [Fow10] and a CML in particular. Either the language is developed on top of an existing usually general-purpose language or it is developed from scratch [MHS05]. Considering the former, they are *internal* in the sense that the selected host language provides the base elements for which extensions and constraints are developed. In contrast, *external* modeling languages have their own custom concepts without explicit relationships to any existing language. Generally, there is no simple answer when to create an internal or external modeling language. However, design guidelines [KKP+09] and patterns [MHS05] have been proposed to aid engineers in developing DSLs.

3.1.3 Modeling capabilities

As different stakeholders are usually involved in the development of a cloud application, their concerns need to be covered by the models created with a CML. Essentially, a concern is a stakeholder's interest that pertains to the development of an application, its operation or any other matters that are critical or otherwise important [vdBCC05][5]. A critical task in the development of a cloud application is its decomposition into deployable components because they must eventually be distributed across a cloud environment or even multiple ones. Hence, the capability of a CML to represent the *structure* of a cloud application certainly matters.

[4]The linguistic type is often considered as first-class, whereas the ontological as second-class.

[5]A concern is usually supported by a modeling viewpoint, see Section 2.2.

Application structure

A component is basically a unit of computation in an application, whereas interactions among components are represented by connectors [MT00]. Components and connectors are used to describe the high-level structure of an application in terms of a component configuration. *Composing* components and connectors into another more abstract component is beneficial in particular to hide complex structures of a cloud application.

To deploy a cloud application on the selected target environment, its application components need to be allocated to cloud *services*. More precisely, what needs to be allocated to the cloud services are the implementations of those components. The notion of a deployable *artifact* supports exactly the reference between logical components and connectors to their implementations. Artifacts are supposed to manifest any number of components and connectors. For instance, a cloud application implemented in Java is possibly packaged into several archives, *i. e.,* "JAR files". Those archives can be represented on the model level via artifacts. Allocating them to a cloud service, *e. g.,* a compute service including a Java platform, should have the effect that the "JAR files" are physically allocated to the provisioned service. Furthermore, as cloud services interact with each other, modeling capabilities are required to explicitly connect them. We use the term *link* to refer to this capability. Artifacts deployed on possibly connected cloud services constitute what is usually called a deployment configuration[6]. To ensure that connected cloud services can in fact interact with each other, properties related to *networking* concerns need often to be explicitly specified. For instance, a cloud service at the infrastructure-level needs to be assigned to a virtual network and possibly connected to a middlebox to ensure that it can be accessed by other cloud services.

It is important to note that we aim to investigate the modeling capabilities of CMLs to represent component, connector and deployment concerns on a per concept basis. As a result of this investigation, the different vocabulary introduced by current CMLs is classified according to the common terms of our framework, where the string-valued properties allow us to collect the concrete terms used by them. This effort is a first step towards achieving interoperability between existing CMLs and a core set of common cloud modeling concepts upon which semantic relations among different CMLs can be defined [MMPT10].

Cloud environment

From a cloud application deployment perspective, a modeled cloud service embodies in fact a concrete service offered by the target cloud environment. For instance, several compute services located in different availability zones and a storage service may be required to provide a reliable and scalable cloud applications. The compute services may refer to Amazon's EC2 offering and the storage service to its DynamoDB datastore solution. As a result, services offered by a *cloud environment* need to be available on the model level. Several possibilities are conceivable to represent a configuration of cloud services required for the deployment of a cloud application. Clearly, they can be described in *textual* form. Providing them in a structured form would

[6]The term "topology" is often used in this context as well. A deployment configuration or topology is a connected graph that describes deployment artifacts along with targets and relationships between them from a structural perspective.

certainly ease their interpretation by tools, *e. g.,* engines that initiate the provisioning of compute and storage services based on a deployment topology. In our framework, we distinguish between three structured-based approaches for capturing cloud services: *feature model*, *linguistic types*, and *ontological types*. Considering the former approach, existing compute and storage services may be captured as features of a certain cloud environment denoting the root concept of the model. In case of the latter two approaches, a cloud service is captured in terms of a type as part of a CML. Depending on the typing mechanism a CML supports, a cloud service type is either directly built-into the language definition (*i. e.,* linguistic type) or realized as a custom type supplementing the definition of a language without modifying it (*i. e.,* ontological type).

Elasticity

As a main incentive of using cloud services is the capability of cloud environments to scale them with a user's demand [VRB11], a concern that matters is *elasticity*. Lower and upper-bounds of cloud service instances can be specified by a *multiplicity* associated to the modeled service. To specify more sophisticated strategies when a cloud service must be provisioned or released, a rule-based approach [KDR14] tend to be more powerful compared to specify service multiplicity. The elastic nature of cloud environments is also exploited to utilize them to capacity by optimizing the work load scheduling of the different co-located cloud users with consideration to their required quality of service.

Service level

A concern that matters is the specification of service levels, *e. g.,* referring to latency, availability and security of a cloud service. Ideally, the quality of a cloud service is at least equivalent [VW12] to what can be expected if an on-premise environment is employed to host applications instead of a cloud environment. Currently, only a few of the reviewed CMLs support modeling concepts for capturing service levels at a rather high level. As a result, we distinguish in our framework whether a service level is captured by means of a *structured* approach or it is described in natural-language, *i. e.,* a *non-structured* approach is employed.

3.1.4 Tool support

Modeling support

The means provided for the use of a CMLs notation and the validation of created models according to its syntax and semantics is subsumed under modeling support. Depending on how the notation is defined for a CML either *graphical* or *textual* representations of models are provided [Moo09]. In rare cases both kinds of representations can be used. Obviously, this requires that a textual as well as a graphical notation is available for a CML. The notation of a CML may also be *imposed* by the meta-language used to realize it [NBM+15]. For instance, XML-based CMLs for which no further modeling support is available force engineers to express their models directly in XML. If a CML is realized as internal DSL, it should ensure the portability between the modeling tools that support the selected host language of the CML. For instance, UML-based CMLs should be applicable for any standard conformance UML modeling tool. This can be achieved if UML's

extension mechanisms are appropriately employed for developing a CML [Sel07]. Furthermore, the multiple concerns supported by a CML should ideally be manageable by multiple views (*e. g.,* dedicated views for the component and deployment concern) while ensuring consistency across the various created views for the same cloud application [MT00].

Analysis support

To evaluate or predict certain (non-functional) properties of an application, *e. g.,* operational costs or performance, before it is hosted on a cloud environment is certainly a major incentive to use a CML. Moreover, selecting an adequate set of services from possibly multiple cloud environments is labor-intensive not only because of inevitable trade-offs between, *e. g.,* operational costs and performance, but also the enormous design space that needs to be explored for an optimal deployment of a cloud application [HLS+13]. Analysis support for cloud applications and their underlying environments has thus been addressed by CML toolset developers. In addition to *design-time* analysis, support for analyzing cloud applications at *run-time* is of particular interest because they may be migrated among cloud environments if a certain quality of service can no longer be guaranteed.

Refinement support

Explicit refinement support can ensure that modifications to models expressed by a CML are carried out in a stepwise systematic manner. Model refinement can be considered as a process of transforming a given high-level model into a more concrete model. For instance, deployment topologies of cloud applications are often modeled independently of the target cloud environment in a first step. The refinement of the deployment topology towards the target cloud environment is conducted in a second step [ANM+12]. This approach is particularly beneficial if a cloud application needs to be migrated between environments. The high-level models representing the cloud application are retained and *enriched* by environment-specific information in order to accomplish the refinement. For instance, environment-specific information can be captured in terms of custom (ontological) types [AGK09] or profiles [LWWC12]. Refinement may also include the process of discovering appropriate concrete solutions that are already available, *e. g.,* an application service that is hosted on a cloud environment. To enable this kind of refinement, both the requirements of high-level models and the capabilities of existing more concrete models must be appropriately described, such that the former can be *resolved* according to the latter.

Generation support

Applying generative techniques is promising because executable artifacts can be produced for possible multiple target cloud environments from a single set of (architectural) models. Even though the environment for which artifacts were generated may change over time the investment in creating models is retained [GS03] provided that generators are capable to produce those artifacts for the new environment. This includes not only the generation of implementations for the application itself but also deployment scripts and vice versa, *i. e.,* the generation of models from lower level code artifacts. Furthermore, a deployment plan expressed in terms of a workflow

model may be generated from a deployment configuration in order to enact the application provisioning. In this work, we distinguish between three kinds of generative techniques [CH06b] possibly supported by a CML: *model-to-code*, *code-to-model*, and *model-to-model*.

Provisioning support

One key characteristic of a modern cloud environment is the capability of dynamic service provisioning. Cloud users can provision and release cloud services on demand and pay only for what they have actually consumed. Automating the processes of service provisioning and releasing them and possibly (re-)deploying application-related artifacts including the required middleware on top of those services is aimed at a provisioning engine. Considering the support for application and service provisioning in the light of Talwar's classification, current CML's inherently apply a model-based approach as they represent a deployment configuration in terms of a model. In case it is directly interpreted by a provisioning engine, the approach can be characterized as *declarative* because the created model describes only what has to be provisioned, but without providing any details about how the provisioning shall be executed. In contrast to a declarative approach, an *imperative* approach explicitly prescribes how the provisioning must be executed. For instance, a deployment script[7] or a workflow model is often used to capture the respective provisioning actions[8].

Considering the two approaches from the perspective of a CML user, the declarative approach is less invasive compared to the imperative approach because it requires describing solely the desired state of the provisioning in terms of a deployment configuration. How the desired state is reached is completely hidden from the users. Since this loss of control is not always desirable, a *mix* of the two approaches is supported by some CMLs. For instance, if a CML supports in addition to a deployment configuration the specification of deployment scripts that shall be executed at a certain point during the provisioning of an application component, then both approaches can be combined to a certain degree. Such approaches typically employ life-cycle definitions that subdivide the provisioning of an application component into multiple phases. Those definitions provide a hook for custom scripts or other implementations that must be executed in a certain phase.

3.2 Review process

To conduct the systematic review of CMLs, we followed the guidelines recommended by Kitchenham and Charters [KC07]. The review commenced in mid-2014 in the context of the ARTIST project, where some CMLs with no claim for completeness where demonstrated in the setting of a cloud migration scenario [BWKG14]. However, this demonstration revealed already an initial set of CMLs which was useful in several phases of the review process. For instance, we used this initial set of CMLs to assess the quality of the search queries we formulated in an early phase of the review process as those CMLs had to be covered by the obtained records. Furthermore, we

[7] A deployment script is sometimes executed at the remote environment that is considered as the target of the application provisioning

[8] A workflow model usually captures the data flow and the control flow among actions in an explicit way

exploited them to develop a list of keywords required to conduct a keyword-based search as part of the study selection phase. The main phases we carried out in the course of the review process are described in the following Sections 3.2.1 to 3.2.4.

3.2.1 Research questions

The aim of this systematic literature review (SLR) is to provide an overview of current CMLs, classify their main characteristics and core capabilities including the tool set they support, and identify the gaps and future research directions for CML development. The overall objective is defined by five research questions as follows.

SLR – Research question 1: *What are the main purposes of current CMLs?*
SLR – Research question 2: *What are their characteristics from a language engineering perspective?*
SLR – Research question 3: *What core cloud modeling capabilities do they provide?*
SLR – Research question 4: *What toolset is accompanied with existing CMLs?*
SLR – Research question 5: *What are the capabilities of current CMLs for application modernization to the cloud?*

3.2.2 Data sources and search strategy

In this systematic review, the electronic databases recommended by [PFMM08,KBB+09,KPB+10] have been used to search for primary studies[9]. We decided to search for publications in the period from 2006 to 2015 as Amazon Web Services (AWS) was officially launched in 2006 and the term "cloud computing" appeared around 2007 [VW12]. Since cloud computing is a highly diverse research topic, determining on the publication sources to search for relevant research works is a difficult task.

Hence, instead of taking this decision solely based on our expertise and knowledge in the area of cloud computing, model-driven engineering, and related research areas, we additionally formulated a search query with the main aim to figure out the publication sources where a CML may have been published. Based on the topic of this systematic review and the research questions proposed in Section 3.2.1, we defined the terms of the search queries according to the recommendations of [KC07]. We considered the terms "model", "modeling", "modelling", "language", "ontology", "profile", and "domain" as the main constituents of the search query. In addition, we limited the search to studies that are indexed by the keyword "cloud computing". After several tests, we selected the search query that returned the largest result set. Depending on the electronic database, the syntax of the search query obviously differs.

Table 3.1 summarizes the exact search queries we executed against the selected electronic databases including the number of records we received as a result. Based on the obtained initial set of records, we determined on the set of publication sources by a manual selection process. We selected those publication sources that seemed to be relevant for the purpose of this review. For

[9]We only discarded Google Scholar (http://scholar.google.com) from the list of recommended electronic databases as it hardly allows publications to be downloaded in a batch process and a suitable format.

Electronic database	Search query	Records
ACM Digital Library http://portal.acm.org	"query": {(model, modeling, modelling, language, ontology, profile, domain) AND keywords.author.keyword:(+"cloud computing")} "filter": {"publicationYear":{"gte":2006 }},{owners.owner=GUIDE}	3,208
IEEE Xplore http://ieeexplore.ieee.org	((model OR modeling OR modelling OR language OR ontology OR profile) AND "Author Keywords":"cloud computing") ? and refined by Year: 2005–2016	4,020
ScienceDirect http://www.sciencedirect.com	(model OR modeling OR modelling OR language OR ontology OR profile) and KEYWORDS("cloud computing")[All Sources(Computer Science)]	899
Scopus http://www.scopus.com	ALL (model OR modeling OR modelling OR "language" OR ontology OR profile OR domain) AND KEY(cloud computing) AND PUBYEAR > 2005 AND SUBJAREA (comp)	13,284
SpringerLink http://www.springerlink.com	"cloud computing" AND (model OR modeling OR modelling OR language OR ontology OR profile) within "Computer Science" AND 2005–2016	12,341

Table 3.1: Search queries executed against electronic databases

instance, we discarded sources dedicated to topics such as “e-health applications and services” or “green computing”. The selected publication sources are summarized in Tables 3.2 to 3.4. They were used to limit the records of publications considered in study selection process.

Workshop	Acronym
Intl. Workshop on Model-Driven Engineering for High Performance and CLoud Computing	MDHPCL
Intl. Conference on Cluster Computing Workshops and Posters	Cluster Workshops
Intl. Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems	MESOCA
Intl. Workshop on European Software Services and Systems Research - Results and Challenges	S-Cube
Intl. Workshop on Modeling in Software Engineering	MiSE
Intl. Workshop on Analysis and Programming Languages for Web Applications and Cloud Applications	APLWACA
Intl. Workshop on Security in Cloud Computing	Cloud Computing
Intl. Workshop on Distributed Cloud Computing	DCC
Intl. Workshop on Cloud Services, Federation, and Open Cirrus Summit	FederatedClouds
Intl. Workshop on Hot Topics in Cloud Services	HotTopiCS
Intl. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems	MASCOTS
Intl. Workshop on Multi-Cloud Applications and Federated Clouds	MultiCloud
Intl. Enterprise Distributed Object Computing Workshop	EDOCW
Intl. Workshop on Grid Computing	-
Intl. Workshop on Cloud Computing Platforms	CloudCP
Intl. Workshop on Cloud Data and Platforms	CloudDP
Intl. Workshop on Software Engineering Challenges of Cloud Computing	CLOUD@ICSE
Intl. Workshop on Model-Driven Engineering on and for the Cloud	CloudMDE
Intl. Workshop on on Model-Driven Engineering for High Performance and CLoud computing	MDHPCL

Table 3.2: International workshops considered for the review

Conference	Acronym
Symp. on Cluster Computing and the Grid	CCGRID
Asia-Pacific Services Computing Conf.	APSCC
Conf. on Computer Science and Service System	CSSS
Conf. on Cloud Computing in Emerging Markets	CCEM
Conf. on Cluster Computing	CLUSTER
Latin American Conf. on Cloud Computing and Communications	LatinCloud
Conf. on Cloud Computing and Intelligence Systems	CCIS
Conf. on Cloud Computing and Services Science	CLOSER
Conf. on Cloud Computing	CLOUD
Conf. on Cloud Computing Technology and Science	CloudCom
Conf. on Web Services	ICWS
Conf. on Model-Driven Engineering and Software Development	MODELSWARD
Symp. on Modeling, Analysis, Simulation of Computer and Telecommunication Systems	MASCOTS
Conf. on Software Engineering	ICSE
Europ. Software Engineering Conf./Symp. on the Foundations of Software Engineering	ESEC/FSE
Conf. on Software Engineering and Service Science	ICSESS
Conf. on Software and Data Technologies	ICSOFT
Conf. on Software Engineering and Applications	ICSOFT-EA
Computer Science and Engineering Conf.	ICSEC
Conf. on Communications and Information Technology	ICCIT
Conf. on Services Computing	SCC
Asia-Pacific Services Computing Conf.	APSCC
Conf. on Service-Oriented Computing and Applications	SOCA
Conf. on Utility and Cloud Computing	UCC
Conf. on Web Services	ICWS
Conf. on Cloud and Service Computing	CSC
Conf. on P2P, Parallel, Grid, Cloud and Internet Computing	3PGCIC
Asia Pacific Cloud Computing Congress	APCloudCC
Symp. on Cloud and Services Computing	ISCOS
Conf. on Internet Computing for Engineering and Science	ICICSE
Conf. on Cloud and Green Computing	CGC
Conf. on Cloud and Ubiquitous Computing and Emerging Technologies	CUBE
Conf. on Cloud Computing and Big Data	CLOUDCOM
Conf. on Information Science and Cloud Computing Companion	ISCC-C
Conf. on Information Science and Cloud Computing	ISCC
Conf. on Cloud Engineering	IC2E
Conf. on Future Internet of Things and Cloud	FiCloud
Symp. on Service-Oriented System Engineering	SOSE
Europ. Conf. on Web Services	ECOWS
Conf. on Grid and Cloud Computing	GCC
Conf. on Cluster Computing	ICCC
Conf. on System of Systems Engineering	SoSE
Conf. on Cloud Computing Technologies, Applications and Management	ICCCTAM
Conf. on Cloud Computing and Internet of Things	CCIOT
Conf. on Service Science	ICSS
Symp. on Cloud Computing	SoCC
Conf. on Internet and Web Applications and Services	ICIW
Enterprise Distributed Object Computing Conf.	EDOC
Conf. on Systems, Programming, Languages and Applications: Software for Humanity	SPLASH
World Congress on Services	SERVICES
Conf. on World Wide Web	WWW
Conf. on Distributed Applications and Interoperable Systems	DAIS
Europ. Conf. on Service-Oriented and Cloud Computing	ESOCC
Conf. on Model-Driven Engineering Languages and Systems	MoDELS

Table 3.3: International conferences considered for the review

Journal	Acronym	Publisher
Communications of the ACM	CACM	ACM
Computing Surveys	CSUR	ACM
Transactions on Internet Technology	TOIT	ACM
Transactions on Software Engineering and Methodology	TOSEM	ACM
Communications of the Association for Information Systems	CAIS	AIS
Advances in Information Sciences and Service Sciences	AISS	CIS
Computer Systems Science and Engineering	-	CRL
Computer Communications	-	Elsevier
Computer Standards and Interfaces	-	Elsevier
Future Generation Computer Systems	FGCS	Elsevier
Information and Software Technology	IST	Elsevier
Journal of Systems and Software	JSS	Elsevier
Journal of Visual Languages and Computing	JVLC	Elsevier
Intl. Journal of Web Information Systems	IJSWIS	Emerald
IBM Journal of Research and Development	IBM RD	IBM
IBM Systems Journal	Systems	IBM
Computer		IEEE
Cloud Computing	-	IEEE
Communications Letters	-	IEEE
Internet Computing	-	IEEE
Software	-	IEEE
Transactions on Computers	TC	IEEE
Transactions on Cloud Computing	TCC	IEEE
Transactions on Services Computing	TSC	IEEE
Transactions on Software Engineering	TSE	IEEE
Intl. Journal of Grid and Utility Computing	IJGUC	Inderscience
Intl. Journal of Web and Grid Services	IJWGS	Inderscience
Intl. Journal of Network Management	-	John Wiley & Sons
Computer Journal	-	Oxford Journals
Communications in Computer and Information Science	CCIS	Springer
Computing	-	Springer
Cluster Computing	-	Springer
Journal of Cloud Computing	JoCCASA	Springer
Journal of Internet Services and Applications	-	Springer
Service Oriented Computing and Applications	-	Springer
Software and Systems Modeling	SoSym	Springer
Intl. Journal of Cooperative Information Systems	IJCIS	World Scientific
Intl. Journal of Software Engineering and Knowledge Engineering	IJSEKE	World Scientific
Computer Science and Information Systems	ComSIS	-
Computing and Informatics	-	-
Journal of Internet Technology	JIT	-
Journal of Object Technology	JOT	-

Table 3.4: Journals considered for the review

3.2.3 Study selection

In order to select the most relevant and important studies, inclusion and exclusion criteria were developed in a first step, see Section 3.2.3. They were applied in several stages of the study selection process as described in Sections 3.2.3 to 3.2.3.

Inclusion and exclusion criteria

Studies relevant for this review must propose language concepts for the purpose of modeling cloud applications. Those concepts must be defined in terms of a grammar or a metamodel. For

instance, even though in the work of Sun *et al.* [SHSW12] a toolkit for managing cloud services is proposed, the language concepts used to represent them have not actually been defined but rather sketched by means of a single example only. Research works that introduce modeling methodologies independent of a CML (*e. g.,* MADCAT [INS+14]) are also excluded by this criterion.

Furthermore, proposed language concepts must enable engineers to model a cloud application independent of the concrete target cloud environment. Shielding models from possible changes of target cloud environments is one main requirement of a modeling language in general [AK03] and so also desirable for a CML. However, this does not mean that a CML should not provide capabilities for creating environment-specific models at all. Ideally, it allows engineers to refine environment-independent models into models specific to the target cloud environment [ANM+12], which is commonly known as the transition from a platform-independent model (PIM) to a platform-specific model (PSM) in model-driven engineering (MDE). Languages that solely support PSMs, *i. e.,* they are directly bound to a cloud environment such as Amazon's CloudFormation[10] and OpenStack's HOT[11], are thus excluded by this review. However, such languages are potential transformation targets for CMLs to automate the provisioning of modeled application deployments. In this respect, approaches such as Deltacloud and jclouds may also be considered as they provide an abstraction layer on top of the programming libraries provided for cloud environments. With dedicated connectors a variety of cloud environments are supported by these approaches.

To limit the scope of this review, we consider studies that propose language concepts mainly applied by the user rather than by the provider of a cloud environment. One of the obvious reasons for this decision is that approaches addressing the cloud provider perspective tend to connect proposed language concepts with internal resources of a cloud environment. This is hardly possible for a CML targeting cloud users as providers of a cloud environment usually offer cloud services to their users without giving much details of the internal resources underlying those services if at all. For instance, SCORCH [DWS11] assumes a scenario in which auto-scaling is realized by provisioning and releasing pre-instantiated virtual machines from a queue, where its optimizer aims at determining the length of this queue and the configuration of the pre-instantiated virtual machines in the queue. Moreover, SCORCH is based on several computational models that specify, for example, the energy consumption of resources and the costs for consuming such resources, which is certainly of interest for providers of a cloud environment as they mainly benefit from utilizing their resources to capacity by co-locating different cloud users on the same infrastructure resources. Providing explicit specifications of such infrastructure resources is supported by Cloud# [LZ11] for the purpose of improving the understanding of how resources in a cloud environment are virtualized, scheduled and isolated from each other.

The inclusion and exclusion criteria that were applied in this systematic review are described as follows.

Inclusion criteria

1. Studies that report on language concepts for the purpose of cloud application modeling

[10] CloudFormation: https://aws.amazon.com/de/cloudformation

[11] Heat Orchestration Template (HOT): http://docs.openstack.org/developer/heat

2. Proposed language concepts that are suitable to create models independent of target cloud environments
3. Studies that address the cloud user perspective

Exclusion criteria

1. Studies or approaches that are bound to a cloud environment
2. Studies that address the cloud provider perspective
3. Studies that are not written in English

The selection of the primary studies was carried based on the initial set of records we obtained from executing the search queries against the data sources given in Table 3.1. To extract the studies relevant for this review from the total number of studies, we passed through several pruning stages, where in each stage the number of studies was significantly reduced compared to the result of the previous stage. Overall, four researchers conducted the four pruning stages as illustrated in Figure 3.2.

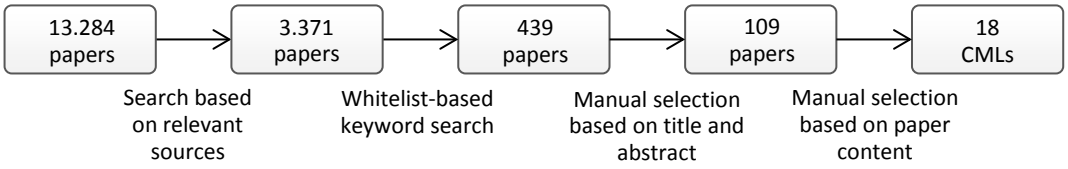

Figure 3.2: Study selection process

Pruning stage 1: Search limited to relevant publication sources

Based on the initial set of records obtained from the electronic databases, we extracted those studies that were published in the selected publication sources, see Tables 3.2 to 3.4. Additionally, the removal of duplicates as a result of using different electronic databases was carried out in this stage. Duplicates were basically identified by considering the title, authors, and publication year of a study. Overall, in this stage, we selected around 25 percent of the total number of records for the next pruning stage.

Pruning stage 2: Whitelist-based keyword search

In order to accomplish this stage, we defined in a first step a list of keywords from which at least one must be present in a certain study to consider it for the next pruning stage. We elaborated the whitelist on the basis of CMLs we were already aware of [BWKG14] and our own experience in the area of model-driven engineering and cloud computing. Table 3.5 summarizes the keywords we defined for the whitelist. After conducting the second pruning stage, we ruled out around 85 percent of studies, thus the number of studies were considerably pruned. The reduction to a manageable number of studies at this stage was important as the two remaining stages are hardly achievable automatically and so were conducted manually.

Keyword		
[application|service] developer	[application|service] definition	deployment topology
[application|service] mode(l)ler	[application|service] architecture	infrastructure(-)as(-)a(-)service
[application|service] engineer	[application|service] topology	platform(-)as(-)a(-)service
[application|service] requirement	resource provisioning	software(-)as(-)a(-)service
[application|service] component	resource description	language engineering
[application|service] deployment	cloud service [engineering]	problem(-)oriented(-)language
[application|service] mode(l)l(ing)	cloud service	domain(-)specfic(-)language
[application|service] life(-)cycle	cloud application	domain modelling
[application|service] provisioning	cloud mode(l)l(ing) [language]	meta(-)mode(l)ling
[application|service] distribution	deployment mode(l)l(ing) [language]	meta(-)model

Table 3.5: Keywords of the elaborated whitelist.

Pruning stage 3: Manual selection based on title and abstract

As it is too often rather difficult to determine the relevance of a study for a systematic review from solely considering its title, we decided to evaluate in this stage both the title and abstract of each study against the inclusion and exclusion criteria. Generally, we acted in a conservative manner in this stage as in some cases even more information in addition to the title and abstract of a study is required to determine whether a study is relevant for this review. Finally, in this pruning stage we ruled out around three quarter of the studies, thus leaving 108 studies for the fourth pruning stage.

Pruning stage 4: Manual selection based on study content

In the final pruning stage, we carefully read the remaining studies under the consideration of the goal of this systematic review and the defined inclusion and exclusion criteria. After all we selected 18 relevant studies [12] that are presented in Table 3.6. The table provides in addition to a representative name of each selected approach the main publications from which the data has been extracted relevant for this review.

3.2.4 Data extraction

In a first step, we converted our proposed classification and comparison framework for CMLs into several spreadsheets. They were used for the purpose of collecting the relevant data. Basically, the single columns of the created spreadsheets were derived from the properties of the classes constituting the framework. Following this approach, the properties determine the concrete data items that we had to extract from the selected studies. For instance, the scope of a certain CML is described by a single row which captures its pragmatics and the kind of target cloud environments that are supported. The scope of a CML has been derived from the natural language descriptions of the respective studies. Similar for the toolset provided by a CML, we extracted the relevant data from the available studies. Concerning the characteristics of a CML and its modeling capabilities we extracted them from the language definition. Considering the process of collecting and interpreting the data itself, a series of consensus meetings were held with the goal to carefully analyze each reviewed CML according to the properties of the prepared tables. As

[12] CAML is not included in this selection.

CML	References
Blueprint	[NLT+11, NLPvdH12]
Caglar *et al.*	[CASG13]
clADL	[PR13, HPR14]
CloudDSL	[SRC14]
CloudMIG	[FH10, FH11, FHS13, FFH13]
CloudML-Sintef	[FRC+13, FSR+14]
CloudML-UFPE	[GES+11]
CloudNaas	[BASS11]
GENTL	[ARXL14, ARSL14]
Holmes	[Hol14, Hol15]
MOCCA	[LFM+11]
MULTICLAPP	[GMMC13b, GMMC13a]
Nhan *et al.*	[NSJ12]
PDS	[LSS+13]
RESERVOIR-ML	[CEM+12]
StratusML	[HLT11, HT14, HT15]
TOSCA	[OAS13a, OAS13b, BBKL14a]
VAMP	[ECBP11, ECB+11]

Table 3.6: Selected approaches

a result of this process, we established the basis necessary to answer the research questions as defined in Section 3.2.1.

3.3 Results

Until now several CMLs have been proposed. In the following, we classify and compare them along the dimensions of the presented review framework (see Section 3.1) with the main aim to answer the defined research questions (see Section 3.2).

3.3.1 What are the main purposes of current CMLs?

Considering the pragmatics of current CMLs several interesting aspects can be observed as described in the following.

Pragmatics

The majority of CMLs deal with the description of cloud deployment configurations. CloudMIG aims at migrating on-premise deployment configurations into optimal cloud deployment configurations and assuring that those configurations conform to the target cloud environment. Optimization of cloud deployment configurations is also addressed by GENTL and MOCCA. While GENTL places emphasis on cost efficient application provisioning, MOCCA primarily deals with the distribution of cloud application components to multiple target cloud environments. CloudML-Sintef exploits cloud deployment configurations not only at design-time but also at run-time for the purpose of model-based reconfigurations of provisioned cloud services. CloudNaas places emphasis on capturing networking aspects (*e. g.,* addressing and segmentation of compute services at the infrastructure layer) by cloud deployment configurations, whereas PDS

CML	Pragmatics	Target
Blueprint	Cloud service composition and description of deployment configurations	XaaS
Caglar *et al.*	Cloud service simulation and description of deployment plan configurations	IaaS
clADL	Architecture description of interactive cloud services and generation of implementations for the cyber-physical systems domain	XaaS
CloudDSL	Description of deployment configurations	XaaS
CloudMIG	Application migration to the cloud with emphasis on optimal deployment configurations and their conformance with target cloud environments	PaaS IaaS
CloudML-Sintef	Automated provisioning of multi-cloud applications and re-configuration of provisioned cloud services at run-time	XaaS
CloudML-UFPE	Description of cloud services	IaaS
CloudNaas	Description of deployment configurations with emphasis on network aspects	IaaS
GENTL	Description of deployment configurations with emphasis on cost-efficient application provisioning	XaaS
Holmes	Description of deployment configurations and their automated provisioning	IaaS
MOCCA	Optimal (re)arrangement of (existing) deployment configurations for application provisioning to multiple target cloud environments	XaaS
MULTI-CLAPP	Application code generation for target cloud environments from component configurations	XaaS
Nhan *et al.*	Feature model based software stack (re-)configuration and their automated provisioning	IaaS
PDS	Deployment plan generation from described deployment configurations	IaaS
RESERVOIR-ML	Description of deployment configurations with emphasis on application-triggered elasticity rules for infrastructure-related cloud services	IaaS
StratusML	Generation of executable deployment descriptor and run-time adaptation rule from described deployment configurations	SaaS PaaS
TOSCA	Description of portable composite cloud applications for their automated provisioning and life-cycle management	XaaS
VAMP	Automated provisioning of distributed cloud applications with emphasis on support for establishing communication between components	IaaS

Deployment configuration: connected graph of deployment artifacts, targets, and their relationships (see Section 3.1.3)
Deployment plan: imperative description of the provisioning process
Component configuration: connected graph of components and connectors (see Section 3.1.3)

Table 3.7: Language scope

exploits them for generating deployment plans. Deployment plans[13] are suggested by TOSCA to describe the process used to create and terminate cloud services and to manage them throughout

[13]In the TOSCA specification the term “management plan” is used.

their whole lifetime. In the approach of Caglar *et al.* deployment plan configurations are created based on previous simulation results obtained from CloudSim [CRB+11].

CMLs that are capable of describing cloud deployment configurations support also the representation of cloud services. In contrast to these CMLs, CloudML-UFPE places emphasis solely on describing cloud services without providing dedicated concepts to model a cloud deployment configuration. Still, cloud services described by CloudML-UFPE can be considered as a potential source for describing cloud deployment configurations. The composition of cloud services is addressed by Blueprint.

Several CMLs aim at automating the provisioning of cloud services and possible application components deployed on top of them. CloudML-Sintef comes with a dedicated provisioning engine. Such an engine is also available for TOSCA. Other approaches (Holmes, Nhan *et al.* and PDS) rely on configuration management systems, such as Cloud-Init or Chef, whereas StratusML generates deployment descriptors for platform-related cloud services (*e. g.,* Azure App Service). In contrast to those approaches, VAMP exploits OVF to describe VM configurations including application components. It provides a dedicated protocol for exchanging configuration parameters (*e. g.,* remote addresses and ports) between remote VMs in order to establish the communication among application components. Generally, generative techniques play an important role in automated provisioning to the cloud because a variety of artifacts (*e. g.,* deployment plans or scripts, runtime models, and VM images) are automatically produced. Aside from generating deployment or provisioning-related artifacts, the goal of MULTICLAPP is to generate application code for the Java environment. Generation of cloud application code is also support by clADL. It proposes an architecture style for modeling interactive cloud services in the context of cyber-physical systems (*e. g.,* services that process sensor data from industrial production machines). Cloud environments are considered as the deployment target for those services.

A few CMLs place emphasis on the representation of elasticity rules (RESERVOIR-ML, StratusML) capable to trigger the provisioning of cloud service at application run-time.

Target

Almost half of the CMLs are capable to represent cloud services at any of the three service layers. CMLs that target the IaaS layer mainly deal with the description of compute services or the configuration of VMs. CloudMIG provides cloud service descriptions for both layers IaaS and PaaS, whereas StratusML considers the PaaS layer up to the SaaS layer.

Summary of CML scope

Current CMLs pursue different goals and show various levels of maturity. Still, they also show similarities with respect to their pragmatics. For instance, the majority of CMLs deal with the description of deployment configuration and some of them even support automated application provisioning. On the other hand, the observed diversity of the current CMLs is beneficial in the sense that a broad spectrum of application scenarios is supported. At the same time, the exchange of models between approaches and provided tools, respectively, is hardly supported. As a result, a well-connected mix of existing CMLs is currently not available. The finding of a common ground between the current approaches is thus highly desirable. GENTL did already

a first step in this direction. Mappings from Blueprint and TOSCA to GENTL are presented in the work of Andrikopoulos *et al.* [ARSL14]. In this respect, the semantics of the CMLs play a major role [KKK+06] since useful mappings, which are the basis for language interoperability (*cf. e. g.,* [MMP08]), can otherwise hardly be identified. A common metamodel [ACB05] may serve as a useful means in such an endeavor.

Another interesting aspect is that most CMLs are used solely at design-time, whereby the representation of the cloud application at run-time is outside the scope of most CMLs. For instance, run-time information may provide the current status and workload of a certain provisioned compute service. CloudML-Sintef is capable of annotating the design-time model with run-time information, which allows run-time adaptations to be performed not only by human operators but also reasoning engines, which manipulate models at run-time automatically. This capability is facilitating "models@run-time" [BBF09], which is an architectural pattern for dynamically adaptive systems that leverages upon models at both design-time and run-time.

3.3.2 What are the characteristics of CMLs from a language engineering perspective?

Main language characteristics of current CMLs are summarized in Table 3.8. While they appear to be primarily relevant for language engineers, some of them may also be of interest for the users of a CML. For instance, the notation of a CML clearly affects its users.

Syntax

Considering how the abstract syntax of CMLs are represented, two meta-languages seem to be dominant in the field of cloud computing: MOF and XML Schema. The majority of CMLs provide a MOF-based metamodel [14], whereas one of them uses UML as a host language (MULTICLAPP). Around a quarter define their modeling elements in terms of XML Schemas (BLUEPRINT, CloudML-UFPE, PDS, TOSCA, VAMP). The remaining languages follow either a grammar-based approach (clADL, CloudNaas, Holmes) or rely on Microsoft's DSL tools (StratusML).

Regarding the concrete syntax of CMLs, the majority provides either a graphical notation or a textual one. A few of them provide both. In case of CloudML-UFPE and VAMP models need to be expressed directly in XML. Even though the TOSCA standard does not define a graphical notation, with Vino4TOSCA [BBK+12] service templates can be visually represented. The representational capabilities of some of the reviewed CMLs are demonstrated in [BWKG14].

While the serialization format of a language is usually imposed by the meta-language used to define it (*e. g.,* XMI is the standard interchange format for MOF-based metamodels), some CMLs support alternative formats in addition mainly for the purpose of compatibility between tools. For instance, the provisioning engine of CloudML-Sintef requires models serialized in the JSON format.

[14] Since Ecore is a reference implementation of the essential part of MOF, CMLs which provide an Ecore-based metamodel follow a MOF-based approach.

CML	Syntax			Semantics	Realization	Typing
	Abstract	Concrete	Serialization			
Blueprint	XML schema	graphical	XML	operational	external	linguistic
Caglar *et al.*	MOF	graphical	custom XML	operational	external	linguistic
clADL	Grammar	textual	custom	translational	internal MontiArc	linguistic
CloudDSL	MOF	graphical	XMI	English prose only	external	linguistic
CloudMIG	MOF	graphical	XMI	operational	external	linguistic ontological
CloudML-Sintef	MOF	textual graphical	XMI JSON	operational	external	linguistic ontological
CloudML-UFPE	XML schema	textual in XML	XML	operational	external	linguistic
CloudNaas	Grammar	textual	custom	operational	external	linguistic
GENTL	MOF	graphical	XML	operational	external	linguistic ontological
Holmes	Xtext[1] grammar	textual	custom XMI	operational	external	linguistic ontological
MOCCA	MOF	graphical	XMI	operational	external	linguistic
MULTI-CLAPP	UML	graphical	XMI	translational	internal UML	linguistic ontological
Nhan *et al.*	MOF Feature model	textual graphical	XMI JSON SXFM	operational	external	linguistic
PDS	XML schema	textual	XML Ruby	operational	external	lingusitic
RESERVOIR-ML	MOF	textual graphical	XMI XML	operational	external	linguistic
StratusML	Microsoft DSL toolkit	graphical	XML	operational	external	linguistic
TOSCA	XML schema	textual graphical	XML YAML	operational	external	linguistic ontological
VAMP	XML schema	textual in XML	XML	operational	internal OVF	linguistic

[1] Xtext is a language engineering framework: https://eclipse.org/Xtext

Table 3.8: Language characteristics

Semantics

Turning the focus from the syntactical aspects of the CMLs to their semantics, the majority of approach come with a tool set (see Table 3.11) that directly interprets or executes the models of a

CML, *e. g.,* a provisioning engine. In this case, the semantics of the CMLs is defined based on an operational approach. A translational approach is applied for defining the semantics of clADL and MULTICLAPP. The former is grounded in the FOCUS [BS01] which enables the formal specification of distributed systems in terms of components communicating via channels. The communication between the components is formally represented by the concept of streams [RR11]. MULTICLAPP provides a mapping to Java for the purpose of generating application code from models. Finally, in case of CloudDSL neither an operational nor a translational semantics is defined. However, it seems that mapping from CloudDSL to TOSCA is currently developed.

Realization

Four out of the 18 reviewed CMLs are realized as internal domain-specific languages. MULTICLAPP is embedded in UML by providing generic cloud modeling concepts in terms of a profile. Additionally, MULTICLAPP comes with a feature model for capturing concrete cloud services. clADL is realized on top of MontiArc [HRR12], which is a textual domain-specific language for modeling distributed interactive systems, whereas VAMP is integrated into OVF.

Typing

More than a third of the reviewed CMLs support in addition to linguistic typing also ontological typing. From a language engineering perspective, all CMLs basically rely on a spanning hierarchy [AK05] where ontological types are considered orthogonal to linguistic types. Thus, ontological types are referenced by instances of linguistic types instead of directly instantiated. The latter would require a stacking hierarchy [AK05].

Summary of CML characteristics

One aspect that requires consideration refers to the interoperability between CMLs. The heterogeneities imposed by the different meta-languages used to implement them certainly impedes such an endeavor. Realizing a mapping between two CMLs defined with different meta-languages would require the implementation of a technical bridge in addition to the definition of language correspondences. Hence, the use of different meta-languages for realizing CMLs poses a challenge for exchanging models between them. CMLs for which solely an XML Schema is provided without a human-usable notation, the users are bound to the verbose angle-bracket syntax which is complex in terms of human-comprehension and therefore impedes maintainability [Bad00]. For instance, XMLText [NBM+15] provides a semi-automatic approach for generating Xtext-based grammars with a human-readable textual concrete syntax while ensuring backwards compatibility to the original XML-based representation.

Proposals for new CMLs or extension to them come ideally together with a machine-interpretable and human-usable language definition. The latter is preferably available in a commonly accepted format. Once they are defined, sharing them via an open repository, *e. g.,* AtlanMod's Metamodel Zoo[15] or ReMoDD[16], allows them to be easily accessed. It may also

[15] Metamodel Zoo: http://www.emn.fr/z-info/atlanmod/index.php/Zoos
[16] ReMoDD: http://www.cs.colostate.edu/remodd/v1

further stimulate their reuse in the development of new languages or additional features for them.

Another interesting aspect is that currently little attention is paid to general-purpose software modeling languages, such as UML, even though they provide modeling concepts to represent software, platform, and infrastructure artifacts from different viewpoints. Only MULTICLAPP is realized on top of UML. With the relatively recent emergence of TOSCA and its standardization by OASIS, it appears obvious that aligning cloud modeling approaches with existing software modeling approaches in order to provide continuous modeling support is highly required [JAP13].

3.3.3 What core cloud modeling capabilities do CMLs provide?

Probably most important to the users of a CML are their modeling capabilities. A black-box view on the modeling concerns addressed by current CMLs is summarized in Table 3.10. The results of a closer investigation of the component and deployment viewpoint are given in Table 3.9.

Application structure

As expected, all the reviewed CMLs provide capabilities to model the structure of a cloud application from a deployment viewpoint. Around half of the CMLs support in addition to the deployment viewpoint also the component viewpoint. Reasons for including the latter viewpoint are manifold. For instance, representing the interaction between a cloud application and cloud services at the SaaS layer requires the consideration of application components. Connectors are used to represent the interconnection among them and with cloud services. Having components explicitly represented is also of particular relevance for application modernization to the cloud if components are replaced by cloud services already offered by the selected target environment. Furthermore, components may need to be re-allocated to deployable artifacts if the cloud application is migrated only partially.

In addition to the component and deployment viewpoints of a cloud application, CloudMIG also addresses the class viewpoint. As CloudMIG aims at supporting application migration to the cloud, it captures technical constraints of cloud environments (*e. g.,* directly instantiating Java's `Thread` class is not permitted on the Google App Engine) against which conformance checks are carried out automatically. Dedicated validators operate at the class level of KDM models in order to investigate whether the application satisfies all the captured environment constraints. KMD models are directly associated to cloud nodes which explains the absence of components and connectors in CloudMIG.

Considering the component and deployment viewpoints in more detail (see Table 3.9), the high diversity of the vocabulary used by the reviewed CMLs is revealed. At the same time, it shows that even though various different syntactical terms have been proposed, they can be classified at least according to a high-level categorization. Almost all CMLs that deal with both viewpoints component and deployment support artifacts as a means of allocating concrete implementations to cloud services. In case of PDS, the artifact is specific to Java-based applications. RESERVOIR-ML and StratusML do not distinguish between logical components and their implementations in terms of deployable artifacts. Components are directly allocated to cloud services where the interconnection between them is solely modeled from the deployment viewpoint by connecting cloud services to each other. As a result, connectors between components can hardly be modeled

CML	Application structure	Cloud environment services	Elasticity	Service level
Blueprint	Component Deployment	textually described	Multiplicity	structured by policies
Caglar *et al.*	Deployment	linguistic types	✗	✗
clADL	Component Deployment	ontological types	✗	✗
CloudDSL	Deployment	textually described	✗	✗
CloudMIG	Class Deployment	ontological types	Rule-based	✗
CloudML-Sintef	Component Deployment	ontological types	Multiplicity	✗
CloudML-UFPE	Deployment	linguistic types	✗	✗
CloudNaas	Deployment	linguistic types	✗	✗
GENTL	Component Deployment	ontological types	✗	structured by annotations
Holmes	Deployment	ontological types	✗	✗
MOCCA	Component Deployment	ontological types	✗	✗
MULTI-CLAPP	Component Deployment	feature model	✗	structured by properties
Nhan *et al.*	Deployment	linguistic types	✗	✗
PDS	Deployment	linguistic types	✗	✗
RESERVOIR-ML	Component Deployment	linguistic types	Rule-based	✗
StratusML	Deployment	linguistic types	Rule-based	✗
TOSCA	Component Deployment	ontological types	Multiplicity	structured by policies
VAMP	Component Deployment	linguistic types	✗	✗

* Blueprint does not directly support to represent service levels but instead exploits existing languages such as WS-Policy or SLAng.

Table 3.9: Cloud modeling concerns

in particular if a variety of components are allocated to cloud services.

Regarding cloud service, it is important to note that as a result of the varying pragmatics of the CMLs, they refer to different virtualization layers (see the superscript annotation determining the service layer). For instance, CMLs targeting the IaaS layer mainly focus on the representation

of compute services and their configuration in terms of virtual machine characteristics such as CPU, memory and local disc space. As a result, they can hardly deal with cloud services at the PaaS or SaaS layer. Moreover, deployment models represented by CMLs that also support PaaS or even SaaS related cloud services may only be partially translated to CMLs with a focus on cloud services at the IaaS layer. Clearly, the border between cloud services at the IaaS and PaaS layer is becoming blurred with the emergence of new cross-layered services (*e. g.,* Azure App Service on different virtual machines). Still, the service layers appear to be relevant to consider. For instance, defining useful correspondences between the language concepts with the goal to achieve better interoperability between CMLs requires considering the supported target cloud environment.

Finally, considering the artifact concepts of TOSCA, it distinguishes between deployment artifact and implementation artifact. The former refers to the implementations of application components, whereas the latter is used to capture the implementation of a management operation. A management operation is attached to a deployment model and executed in the context of the application provisioning, *e. g.,* "start" of a provisioned cloud service or "install" of an application-related artifact.

Cloud environment

Considering how cloud environment services are represented by CMLs, the majority captures them in terms of linguistic or ontological types that are instantiated to model the cloud application structure from a deployment or component viewpoint or even both. Generally, CMLs that advocate ontological types over linguistic types for capturing cloud services provide a richer collection of pre-defined custom service types mainly because their definition is less intrusive compared to defining linguistic types. Capturing cloud services in terms of a feature model can be considered as an orthogonal approach compared to providing service types. From a language engineering perspective, a feature model is not expressed by means of a CML but rather in one of the existing notations (*cf. e. g.,* [CE00]). Both MULTICLAPP and the approach of Nhan *et al.* exploit feature models to let engineers configure the target cloud environment by selecting the required cloud services. They are considered as a feature of a cloud environment denoting the root concept of the feature model. As feature models are often used to explicitly represent the commonalities and variabilities of a certain domain, they appear to be a useful source to define cloud service types assuming that the analyzed domain refers to cloud computing. The variabilities indicate the information required to instantiate a cloud service type in a CML [MHS05].

Elasticity

Turning the focus on how the elastic nature of cloud environments can be treated at model-level, less than half of the CMLs allow engineers to define upper and lower-bounds for service instances or elasticity rules based on which new service instances are provisioned or released. Considering the CMLs supporting a rule-based approach, they all rely basically on the notion of ECA rules that are triggered by monitors observing the cloud environment. Events may refer to increasing virtual machine workload when considering the infrastructure level or growing response time of application servers operating at the platform level. CloudMIG seems to focus solely on

CML	Component & connector			Deployment		
	Component	Connector	Service	Link	Networking	Artifact
Blueprint	Blueprint	Resource-Requirement	Blueprint^{X}	Require	✗	Implementation-Artifact
Caglar *et al.*	✗	✗	VM^{I}	✗	✗	Cloudlet
clADL	Component^{C}	Prov./Req. Interfaces	Runtime^{X}	Endpoint	✗	Artifact
CloudDSL	✗	✗	Service^{X}	Endpoint	✗	Resource
CloudMIG	✗	✗	CloudNode^{I}	Cloud Link	Addressing	KDM model
CloudML-Sintef	Component	Relationship	External-Component^{X}	Hosting	Addressing	Resource
CloudML-UFPE	✗	✗	Node^{I}	Link	Addressing	✗
CloudNaas	✗	✗	Group^{I}	Virtual Net	Addressing Segmentation NetworkService	✗
GENTL	Component	Connector	Component^{X}	Connector	✗	✗
Holmes	✗	✗	Hosting Unit^{P,I}	Ports	Security-Groups	Service
MOCCA	Component	Component-Relation	Virtual-System	Network Port Profiles	Addressing	Artifact
MULTI-CLAPP	Cloud Artefact Element^{C}	Prov./Req. Interfaces	Cloud Artefact Interface^{X}	✗	✗	Cloud Artefact
Nhan *et al.*	✗	✗	VMNode^{I}	Connection	Addressing	Software-Component
PDS	Service	✗	Node^{I}	Database connection	Segmentation	War file
RESERVOIR-ML	Component	✗	Virtual-Machine-Descriptor^{I}	Network Port Profiles	Addressing Segmentation	✗
StratusML	Service	✗	Task^{S,P}	Connection	✗	✗
TOSCA	Node-Template	Relationship-Template	Node-Template^{X}	Relationship-Template	Addressing	Deployment-Artifact Implementation-Artifact
VAMP	Component	Binding	Dynamic-Virtual-System^{I}	Network Port Profiles	Addressing	✗

^{C} Component is composable
^{I} Service at infrastructure layer
^{P} Service at platform layer
^{S} Service at software layer
^{X} Services of all three layers can be modeled

Table 3.10: Component and deployment viewpoint

infrastructure-level events whereas both RESERVOIR-ML and StratusML do also consider events created at the platform-level.

Service level

Regarding the representation of service levels, only few CMLs have reported on how they can be captured. Blueprint exploits existing languages such as WS-Policy[17] or SLAng[18] to express QoS constraints, *e. g.,* "response time < 3sec" or "Data storage is only within the Netherlands". GENTL is capable of representing those QoS constraints in terms of typed annotations. MULTICLAPP provides some stereotypes capable to capture QoS constraints by means of key-value pairs along with an operator that can be selected from a predefined enumeration. It covers standard relational operators such as "equal to", "greater than", or "less than". Finally, TOSCA supports the definition of policies for expressing non-functional behavior or a kind of QoS that a node type can declare to expose. A declared policy type can be instantiated via a policy template used within a node template and processed by a TOSCA container. For instance, policies are processed during service provisioning to guarantee that the provisioned services of a cloud environment satisfy the requirements captured by policies [WWB+13].

Summary of CML modeling capabilities

The reviewed CMLs provide a considerable set of modeling concepts to support a variety of viewpoints relevant in the context of cloud application and service modeling. Still, modeling concepts for capturing non-functional aspects are under-represented. The spectrum of non-functional requirements is certainly broad, however directly attaching information, such as service levels and pricing [CBMK10], to the deployment artifacts and targets may be a further improvement [Gli07]. As a result, technical and non-technical aspects are brought together in a possibly single view, which can support the selection of an appropriate cloud provider and the optimization of the application provisioning. Regarding the latter aspect, CloudMIG and GENTL cover pricing information of cloud environments for the optimization of deployment configuration.

Considering the modeling concepts of Table 3.10 in the light of interoperability, it gives a first impression of how they relate to each other. Still, it is a first step in this direction as achieving interoperability between the CMLs requires also the consideration of concept specializations and possible available custom types. For instance, the TOSCA primer provides a set of pre-defined custom types that may not only be of value for TOSCA users, but cloud application engineers in general. As a result, conceptual mappings between the CMLs as a basis for accomplishing interoperability among the CMLs need to consider both levels intensional and extensional [Küh06][19]. This requires a good understanding of the modeling levels addressed by a CML, which may include not only the design-time models but also run-time models [RdLGN15].

[17] http://www.w3.org/TR/ws-policy

[18] http://uclslang.sourceforge.net

[19] Custom types are at the intensional level. They are instantiated at the extensional level by assigning concrete values to their features.

CML	Modeling	Analysis	Refinement	Generation	Provisioning
Blueprint	Graphical model editor	✗	resolution	✗	✗
Caglar *et al.*	Grapical model editor	✗	✗	m2t: Deployment script Simulation code	imperative
clADL	Textual model editor	✗	✗	m2t: Monticore facility	✗
CloudDSL	Graphical model editor	✗	✗	✗	✗
CloudMIG	Graphical model editor	design-time: Conformance checking and Deployment optimization	✗	t2m: Java-KDM	✗
CloudML-Sintef	Textual & graphical model editor	✗	enrichment by ontological types	m2m: Runtime model m2t: Adaption script	declarative
CloudML-UFPE	Directly represented in XML	✗	✗	✗	✗
CloudNaas	✗	✗	✗	m2t: network rules	declarative
GENTL	Graphical model editor	design-time: Deployment optimization	enrichment by ontological types	✗	✗
Holmes	Textual model editor	✗	enrichment by ontological types	m2t: Deployment script	declarative
MOCCA	Graphical model editor	design-time: Deployment optimization	✗	m2m: Deployment plan	declarative
MULTI-CLAPP	Arbitrary UML model editor	✗	refinement by UML profile	m2t: Service adapter m2m: Deployment plan	✗
Nhan *et al.*	Graphical feature configuration editor	✗	resolution	m2t: Deployment script	declarative
PDS	Directly represented in XML	✗	✗	✗	declarative
RESERVOIR-ML	Textual & graphical model editor	✗	✗	m2t: Deployment script	declarative
StratusML	Graphical multi-view model editor	✗	✗	m2t: Deployment descriptor[1] Adaptation rules	✗
TOSCA	Graphical model editor	✗	enrichment by ontological types and Implementation artifacts	m2m: Deployment plan	mixture
VAMP	Directly represented in XML	✗	✗	m2t: VM image	declarative

[1] A deployment descriptor is directly interpreted by the cloud environment.

Table 3.11: Tool support

3.3.4 What toolset is accompanied with existing CMLs?

Given the fact that the first CML was published in 2010, a considerable set of diverse tools beyond model editors for CMLs is already available as summarized in Table 3.11.

Modeling support

The majority of CMLs provide a custom textual or graphical model editor. Both CloudML-Sintef and RESERVOIR-ML provide a textual as well as a graphical notation to represent models. The latter uses a textual approach for representing elasticity rules. UML-based CMLs (MULTICLAPP) can be used by arbitrary modeling tools supporting UML and its profile mechanism. Most current UML modeling tools provide multi-view model editors. Also, StratusML comes with a multi-view model editor implemented on top of Microsoft's DSL tools. Even though the TOSCA specification does not define a standard graphical notation, Winery [KBBL13] is a web-based model editor capable to visually represent TOSCA models. For some of the reviewed CMLs, a dedicated model editor is missing. CloudML-UFPE, PDS, and VAMP are XML-based languages where models are intended to be represented directly in XML. In case of CloudNaas, only a parser is available for its language, hence engineers are forced to use a plain text editor to represent network policies.

Analysis support

Only two of the reviewed CMLs bring analysis support to deployment topologies. CloudMIG's toolset is capable to validate a cloud application against constraints defined over cloud environments (see Section 3.3.3). Furthermore, it supports engineers seeking for the pareto optimal set of deployment configurations with the objective to minimize response time, the number of violations of an upper-bound response time, and costs. In fact, the solutions in the pareto optimum are a trade-off between performance and costs. MOCCA addresses the problem of distributing cloud application components over a single or multiple cloud environments. For that reason, a deployment topology is translated into a labeled connected graph where the labels capture the information (*e. g.,* data throughput per time unit or the workload of an application component) according to which the set of optimal graph partitions are calculated by means of a cohesion metric.

Refinement support

Considering the toolset of CMLs for refinement tasks, the Blueprint approach enables the resolution of defined service requirements by performing a string-based matching against service offerings. As a result of this resolution process, explicit interconnections between concrete service blueprints are derived. A resolution process is also carried out by the approach of Nhan *et al.* in order to derive a valid configuration of a feature model from an initial one. In fact, the resolution is achieved by additionally selecting required features that are not covered by the initial configuration. As some CMLs propose to model a cloud application independent of a cloud environment in a first step (CloudML-Sintef, MULTICLAPP, the approach of Holmes, StratusML, and TOSCA), they employ dedicated approaches to achieve a refinement towards the target cloud

environment. In essence, all of them exploit predefined environment-specific information that is associated to an environment-independent model. How the environment-specific information is in fact captured and associated to a model from a technical perspective varies between the CMLs mainly because they are realized on top of different platforms. Furthermore, three of the CMLs (CloudML-Sintef, the approach of Holmes and StratusML) employ matching techniques to semi-automatically achieve the refinement step.

Generation support

The majority of CMLs exploit generative techniques. Caglar *et al.* provide a code generation facility capable of producing simulation code for CloudSim. Similarly, clADL[20] and MULTICLAPP[21] support automated forward engineering capabilities. Model-based reverse engineering is supported by CloudMIG, where the generated models are represented by KDM. The generation of deployment scripts, descriptors and plans from deployment configuration is supported by several CMLs (see the approaches of Caglar *et al.*, Holmes, and Nhan *et al.*, MOCCA, MULTICLAPP, RESERVOIR-ML, StratusML, TOSCA). Virtual machine images can be generated from deployment configuration by the VAMP. They are represented in OVF. A few approaches deal with environment-related artifacts such as a runtime model (see CloudML-Sintef), adaption scripts or rules that allow the manipulation of provisioned cloud services at run-time (see CloudML-Sintef, StratusML), and network rules capable of re-provision virtual networks to support different fail-over scenarios (see CloudNaas).

Provisioning support

Finally, more than half of the reviewed CMLs come with tool support capable of automatic application and service provisioning. Most of them suggest a declarative approach, thereby reducing the effort for engineers to explicitly describe the provisioning actions. Interestingly, the CML of Caglar *et al.* supports the opposite approach. Their CML provides modeling concepts for expressing the provisioning actions in a strict imperative style. The TOSCA-compliant run-time container OpenTOSCA [BBH+13] combines the declarative and imperative approach [BBK+14]. It enables automated plan-based provisioning and management of cloud applications, which allows imperative deployment scripts to be integrated into a deployment plan. The latter is generated from a deployment configuration represented in terms of TOSCA.

Summary of CML tool support

Current CMLs span a broad spectrum of tools supporting engineers in the design, development, and provisioning of cloud applications. Moreover, the existing set of tools enable them to deal with different application scenarios, *e. g.,* migration of a non-cloud application to a cloud environment, distribution of application components in a single or multi-cloud environment, optimization of cloud applications. Having categorized the core tools provided by existing CMLs can be considered as a first step towards a canonical CML toolkit [MT00]. Overall, existing

[20] clADL exploits Monticore's code generation facility [KRV10]

[21] MULTICLAPP focuses on code generation for the Java platform

CMLs have placed the greatest emphasis on modeling, generation, and provisioning of cloud applications and the least on analysis and refinement.

3.3.5 What are the capabilities of current CMLs for application modernization to the cloud?

Taking existing literature [CDPC11, FH11, LFM+11] into consideration, we derived five activities as summarized by the modernization process illustrated in Figure 3.3. First, a deep understanding of the legacy application is required [CDPC11] before any other step in the process can be conducted. Once a target cloud environment is selected, adaptations on the legacy application are performed, if required. Adaptations may be required to meet the goals of the migration and to exploit the novel cloud-based technologies offered by today's cloud environments. The latter may also involve overcoming constraints imposed by different service layers that hinder the appropriate functioning of the migrated application in a cloud environment. In particular, applications to be deployed at the platform and software layer typically have to interface with the proprietary framework of the cloud environment, whereas arbitrary dependencies can be deployed and used at the infrastructure layer. Nevertheless, PaaS is appealing as it often provides better support to automate the deployment and the management of cloud applications. For example, the Google App Engine currently does not support EJB, but offers a degree of automatic scalability [AFG+09] through a strict cloud application structure, which is inherently difficult to achieve at the infrastructure layer, where, for instance, Amazon operates. Depending on the service layer of the selected cloud service(s) and the pursued migration type [ABLS13], the deployment covers different aspects of an application, such as user interface, business logic, or data management, when considering a classical three-layer architecture. After the deployment is prepared, the required cloud services can be provisioned to finally run the migrated application in the cloud. The support offered by the selected approaches for each activity is indicated in Figure 3.3 (see gray boxes). Most approaches support more than one activity.

Get understanding

As conceptual models provide excellent means for getting an understanding of applications, CloudMIG supports their discovery from legacy applications by building on MoDisco [BCJM10]. It enables the discovery of KDM models from Java applications. CloudMIG provides a tree-based structure to represent KDM models, which can be linked to a deployment model of the legacy application. Representing legacy applications at model-level is also supported by MOCCA. It provides modeling concepts for representing the architecture and the deployment of legacy applications. Clearly, UML appears to be useful for increasing the engineer's current understanding of legacy applications as it is capable to represent from a variety of viewpoints. Generally, the selected modeling language plays an important role because the (reverse-engineered) models must support engineers in analyzing the application from which the models originate. For instance, if engineers are already familiar with UML because they use it for application development, creating UML models of legacy applications appears favorable provided that they allow engineers to carry out the required analysis for the purpose of comprehension.

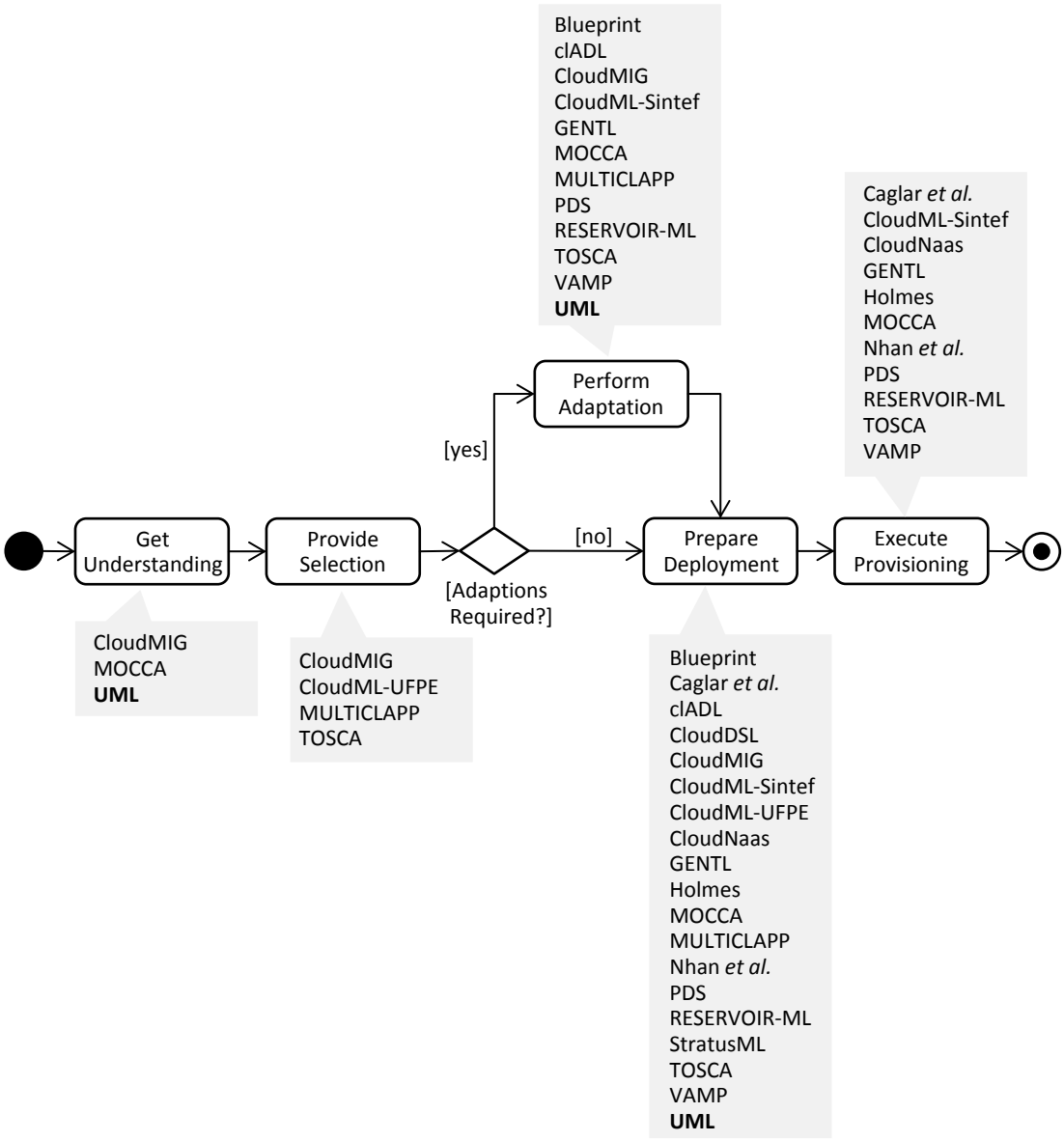

Figure 3.3: Modernization process and support for activities of CMLs and UML

Provide selection

CloudMIG, CloudML-UFPE and MULTICLAPP aim at providing descriptions of cloud services for the purpose of cloud environment selection. CloudML-UFPE places the emphasis on describing infrastructure-related cloud services, whereas ClouMIG provides environment profiles for cloud services at the infrastructure as well as the platform layer. In addition to selecting cloud services, CloudMIG features conformance checking of legacy applications with respect to potential cloud environments. Even though MULTICLAPP is realized on top of UML, it captures cloud service descriptions in terms of a feature model rather than directly in UML via, *e. g.,* a UML profile. UML itself does not provide environment-specific descriptions by default mainly because it is a general purpose modeling language. Finally, it is important to note that even though all reviewed CMLs provide capabilities for capturing cloud services they are predominantly used to represent deployment models without the goal to collect and integrate those descriptions into environment-specific profiles or libraries. Most of the CMLs lack such a mechanism. TOSCA comes with a set of predefined environment-independent node types (*e. g.,* "WebServer" and "DBMS") and relationship types (*e. g.,* "DependsOn" and "HostedOn"). Similar Nhan *et al.* provide environment-independent software stack configurations in terms of a feature model.

Perform adaption

The required adaptations for a successful modernization to the cloud can be diverse and related to different service layers of cloud environments. This activity of the modernization process refers to possible modifications of the application. As a result, CMLs supporting it must be expressive enough to represent at least application components. Often their internal structure or behavior is even required for carrying out the necessary modifications. CMLs capable to capture application components seem to be potential candidates for this activity and so UML. As it is a multi-viewpoint modeling language, a variety of model views on applications are supported. At the same time, mainly due to its generic nature, UML does not come with built-in support for modeling cloud-specific aspects of applications required to run in the cloud. MULTICLAPP's UML profile is a first step in this direction, as it is capable to annotate components with stereotypes that are expected to be deployed onto one or multiple environments.

Prepare deployment

All of the reviewed CMLs provide modeling concepts to prepare the deployment of an application where the target is a cloud environment. Deployment models created by current CMLs describe the desired state of the application provisioning [TMW+05]. They describe the result of the application provisioning in a declarative way, *i. e.,* how the provisioning is in fact carried out is not explicitly prescribed by them. As UML provides also a generic deployment language, it can be employed to describe a cloud-based application deployment. Clearly, cloud-specific features are not supported by UML's standard deployment modeling concepts. As a result, on-premise application deployments expressed in UML can only be turned into cloud-based application deployments if cloud-specific extensions are available. From this point of view,

UML's deployment language is similar to CMLs which support ontological typing, as they also require declaring custom cloud-specific types prior to creating concrete deployment models.

Execute provisioning

As todays cloud environments allow cloud service to be provisioned dynamically via dedicated service interfaces, it appears obvious that those interfaces can be exploited by CMLs for automating the application provisioning. Around two thirds of the CMLs support application provisioning to the cloud. They come with dedicated tools that are capable to accomplish the application provisioning based on a deployment model. Some of them build on existing configuration management tools (*e. g.,* Chef, Puppet, or Cloud-Init), whereas for others full-fledged provisioning engines have been developed. A cloud-based provisioning engine that directly supports UML deployment models is yet not available.

3.3.6 Threats to validity

There are two main threats that may jeopardize the internal validity of this systematic literature review. First, the search of potential relevant research works was carried out on a set of publication sources that we determined. Since cloud computing is a highly diverse research topic, we may have excluded publication sources in which research works relevant to this review are published. In order to reduce the possibility of missing publication sources, we formulated in a first step a relatively generic search query for the purpose of obtaining a set of potential relevant publication sources. From this obtained initial set of publication sources, we excluded those sources that seemed to be out of the scope of this review in a second manual step. Second, the definition of keywords used to prune the set of studies is a critical task in the sense that to many studies are already excluded before the manual search process is carried out. In order to reduce the risk of defining a keyword list that is too restrictive, we carefully tested it with a set of studies we were aware of from previous work. Concerning external threats to validity, we cannot claim any results regarding language features outside of our classification and comparison framework even though some CMLs are realized as extensions to existing general-purpose model languages or architecture description languages.

3.4 Summary

Several CMLs accompanied by a considerable set of tools have been proposed so far. As they address the diversity of modern cloud environments and their services, existing CMLs pursue different goals, differ in scope, and provide thus complementary modeling concepts. Consequently, the investigation of the diverse features currently provided by CMLs is of high interest in general and for this thesis in particular.

In this chapter, we presented a common classification and comparison framework with the goal (*i*) to support cloud users in selecting the CML which fits the needs of their application scenario and setting, and (*ii*) to investigate language characteristics and concepts as they are of particular relevance for this thesis. In this respect, we are not only interested in features of existing CMLs for which extensive support is already provided but also in which they are deficient.

Review framework and process. We introduced a relatively concise framework for classifying and comparing current CMLs. The framework is realized in the form of a metamodel, which allows extensions and modifications. This appears to be crucial in a field that is still largely in its infancy. In total, we have reviewed 18 CMLs according to the presented classification and comparison framework. For conducting this systematic review, we followed the guidelines recommended by Kitchenham and Charters [KC07].

Study results and limitations. The results of classifying and comparing existing CMLs are summarized in Tables 3.7 to 3.11 along with detailed descriptions of our findings and respective summaries thereon. Thereafter, we discussed to what extent existing CMLs are capable to deal with the application modernization scenario introduced in Figure 1.1, which includes processes of both reverse engineering and forward engineering. Since in this thesis cloud-specific extensions to UML are proposed, we included UML in this discussion and drew up those phases of the modernization process in which standard UML lacks support for cloud application modeling.

3.5 Related surveys

In the work of Papazoglou and Vaquero [PV12], the need for knowledge-intensive cloud services that comprise metadata (*e. g.,* offered services, quality of a service, available service level agreements, technical service specification) of cloud environments is discussed. Currently, the metadata are spread over and confined to the different virtualization levels (*i. e.,* IaaS, PaaS, SaaS) of such environments. As a consequence, Papazoglou and Vaquero argue for a language that supports the description, the definition of constraints over such descriptions, and the manipulation of cloud services and their metadata. They identify and analyze (modeling) languages that fall into these three categories. The set of selected languages spans a broad spectrum, ranging from general languages used in the context of service-oriented architecture (*cf. e. g.,* [MLP08]) to low-level formats describing web resources (*e. g.,* RDF) or virtual resources (*e. g.,* OVF). We share the approach of Papazoglou and Vaquero to use the virtualization layers as introduced by the NIST [BGPCV12] to categorize existing CMLs regarding the target cloud environment they support. However, we focus exclusively on modeling languages tailored to the cloud computing domain, hence claim to be what we call a CML. As a result, we use more fine-grained criteria to analyze existing CMLs compared to the work of Papazoglou and Vaquero.

Sun *et al.* [SDA12] present a survey of service description languages that examines seven different aspects: domain, coverage, purpose, representation, semantics, intended user, and feature[22]. By analyzing common modeling language characteristics (*i. e.,* coverage, purpose, semantics), their capabilities (*i. e.,* representation) and intended users with respect to the cloud computing domain, we cover all the aspects of this article. In contrast to our work, Sun *et al.* do not further refine the domain aspect, which is due to the fact that their scope goes beyond cloud computing, which includes languages used in the context of service-oriented architecture (*e. g.,* SoaML [OMG12]) or semantic web (*e. g.,* OWL-S [W3C04]).

[22]It is used to capture additional informal comments over a language rather than to provide a feature-based analysis [KCH+90]

Jamshidi *et al.* [JAP13] conducted a systematic literature review of cloud migration research in which they classified 23 publications from 2010 to 2013 according to 12 analysis dimensions. They conclude that cloud migration research is still in its early stages, but their study also provides evidence that the maturity of the field is increasing. Jamshidi *et al.* do not focus on modeling techniques and languages in the cloud computing context, which distinguishes their work from ours. Nevertheless, they cite the need for a common research agenda between cloud computing and software engineering researchers, which further motivates our work.

Silva *et al.* [SRC13] conducted also a systematic literature review regarding existing solutions that address the "vendor lock-in" problem in the context of cloud computing. They point out that the dependency on a certain cloud environment is a major obstacle to cloud adoption [DWC10]. From an initial set of 721 primary studies 78 were selected and categorized according to 25 solution types dealing with the portability of cloud applications and how the interoperability between offered cloud services can be improved. Even though some of the introduced solutions types indicate that modeling techniques and languages can counteract portability and interoperability challenges, Silva *et al.* do not further categorize or compare them in terms of more fine-grained criteria. As some of the reviewed CMLs particularly aim for portable cloud applications, the work of Silva *et al.* also further motivates our work.

CHAPTER 4

Cloud application modeling

The general-purpose language UML provides modeling concepts to represent software, platform and infrastructure artifacts from different viewpoints. It can thus be applied to a variety of domains including cloud computing. Its language-inherent extension mechanism enables domain knowledge to be captured by profiles. Numerous profiles are available today [Par10], some of which were even standardized by OMG [OMG14a]. Their practical value has been recognized in industry, because modern modeling tools offer already predefined stereotypes covered by profiles. Stereotypes are considered as a major ingredient for current model-based software engineering approaches [BCW12] by providing modeling concepts supplementary to the UML standard metamodel without directly manipulating it. Hence, providing extensions to UML that satisfy current cloud modeling requirements appears beneficial, especially when cloud-oriented modernization scenarios need to be supported where reverse-engineered UML models are refined towards a selected cloud environment. A UML profile seems to be well suited to support an environment-specific refinement provided that it captures at least the required cloud services. Thereby, a clear separation can be achieved between environment-independent and environment-specific models [ANM+12], which is in accordance with the PIM/PSM concept [MFBC12] where the "platform" refers to the selected cloud environment in general and a certain cloud service in particular.

For instance, Google's environment offers a cloud storage service called App Engine Datastore for managing application data. Objectify is the recommended programming library to access the storage service. It is implemented in Java and must be deployed along with the cloud application's component(s) realizing the data management tier, which typically includes entities that are persisted and service classes capable to manipulate them. As a result, several modeling viewpoints are involved when the refinement towards the App Engine Datastore service is carried out. Most notably from an application structure perspective, three key modeling viewpoints can be identified that require careful consideration: (*i*) *class viewpoint*, (*ii*) *component viewpoint*, and (*iii*) *deployment viewpoint*. Regarding the former viewpoint, persistable entities represented by means of UML classes must be properly annotated, such that Objectify recognizes them accordingly. In Java, this task is accomplished by applying the respective Objectify annotations to the entities.

Turning the focus from the class viewpoint to the component viewpoint, all the libraries which are directly and indirectly used by the classes must be associated with the components realized by those classes to ensure that they function appropriately once deployed on a cloud environment. To accomplish the latter, the deployable artifacts which manifest the components must be linked to the required cloud services. Considering Amazon's DynamoDB cloud storage service, a similar refinement process is applicable if the Objectify library is replaced by the DynamoDB Java library, which also exploits annotations, and the Amazon cloud environment is used instead of Google's environment. Even Microsoft's cloud environment can be targeted by applying this refinement process provided that the Entity Framework is used as a data management library and Azure SQL cloud service is selected to actually manage the data. In case of the Azure environment, C# must be employed instead of Java for implementation purposes. Attributes in C# resemble the notion of annotations in Java. They are exploited by the Entity framework in a similar manner as the Objectify and the DynamoDB library embrace annotations.

In order to support a flexible refinement process from high-level PIMs over possibly several PSMs down to a concrete implementation of them, we have developed the *Cloud Application Modeling Language (CAML)* accompanied with a tool set to render it useful in practice. CAML consists of a collection of UML profiles dedicated to libraries of a target platform such as Java or C#. In fact, those libraries are exploited in terms of an annotation mechanism [Sel12], which leverages *annotation-based modeling*, where defined stereotypes show similar capabilities as annotations over program elements [ESM05, NP07]. Considering the large number of possible annotations at the programming level, manually developing the corresponding UML profiles would only be achievable by a huge development and maintenance effort. For that reason, CAML provides a fully automatic transformation chain to generate UML profiles from programming libraries that embrace annotations, where the focus is on the Java platform. This necessitates overcoming existing heterogeneities that, *e. g.,* refer to the target specification of Java annotations and other peculiarities of how Java annotation types are declared. While current modeling tools are capable to deal with Java annotations in UML, the mapping realized by CAML overcomes some existing limitations of their capabilities. It allows annotations to be applied in a controlled UML standard-compliant way as the generated stereotypes extend exactly the required UML meta-classes. CAML realizes a mapping between Java's annotation language and UML's profile language. As a result, it enables the generation of specific stereotypes for corresponding annotations and a dedicated stereotype for the library itself, which in turn leverages library-specific profiles that are intended to be applied in the context of class and component modeling. From a tooling perspective, profiled UML models, *i. e.,* models to which profiles are applied, pave the way for model transformers to generate richer application code from PSMs [Sel12]. CAML's tool set provides extensions to the Eclipse UML generator for Java, such that stereotypes of library-specific profiles are translated into corresponding annotations. Furthermore, it provides transformation chains that exploit library-specific profiles to produce application behavior, *e. g.,* method bodies of CRUD operations for entities that are persisted.

In addition to library-specific profiles, CAML provides a set of profiles that capture cloud services offered by well-known cloud environments such as Amazon AWS, Google Cloud Platform, and Microsoft Azure. As they are intended to be applied for refining environment-independent deployment topologies towards a selected cloud environment, the peculiarities of how those

topologies are modeled in UML need to be carefully investigated for properly developing them. In this respect it is important to understand that concrete deployment topologies are typically modeled in UML on the *instance level*. The main benefit of this approach from a user perspective is that custom types or even hierarchies of them can be established without modifying the UML metamodel. Defined custom types can then be assigned to elements at the object level (see Section 2.2). This approach is exploited by CAML in the sense that it provides a library of common cloud modeling types which are abstractions over services of cloud environments, *e. g.,* cloud storage service. Combining the library approach with the notion of environment-specific UML profiles results in a powerful cloud modeling solution. The cloud library is used to model environment-independent deployment topologies at the object level, whereas the environment-specific profiles are applied to those topologies for accomplishing the refinement towards the target cloud environment without direct modifications to the base elements and their features as stereotypes are only associated to them. This additional typing mechanism shows its benefit when a refinement needs to be changed, *e. g.,* Amazon's DynamoDB cloud storage service is used instead of the cloud storage service offered by Google, because it requires only un-applying and re-applying the respective stereotypes. The dynamic ability of a UML profile even allows the same model to be refined towards several platforms and environments without affecting the underlying model [Sel07]. In case of the modeled cloud storage service, it is decorated by the applied stereotype. The stereotype encapsulates the environment-specific features, whereas the common features of the cloud storage service are provided by the type of the cloud library assigned to it. The cloud library along with the profiles specific to cloud environments is exploited by CAML's model transformers to produce environment-specific deployment descriptors usually required for configuration purposes of cloud services. The main characteristics of CAML are summarized in the following.

Embedded in the sense that it is a UML internal language. By realizing CAML as lightweight extensions to UML it (*i*) directly exploits UML's rich language support, (*ii*) facilitates a cloud-oriented refinement process for UML models, and (*iii*) enables dynamically switching between target platforms and cloud environments by (un-/re-)applying the respective profiles.

Generic with respect to generating library-specific profiles. The mapping between Java and UML realized by CAML is generic because any declared annotation type by a programming library can be represented in terms of a corresponding stereotype.

Generic with respect to representing deployment topologies. CAML's cloud library enables deployment topologies to be represented independent of a target cloud environment. The cloud library is generic in the sense that it provides abstractions over cloud environments. Their concrete services are captured by profiles specific to cloud environments.

Extensible with respect to cloud environment profiles. The cloud environment profiles provided by CAML are integrated via a common cloud profile that hides the peculiarities of how custom types in UML can be extended. Abstract stereotypes covered by the common cloud profile define possible extension points of CAML.

The remainder of this chapter is structured as follows. In Section 4.1, we motivate the practical value of CAML by means of the application scenario introduced in Section 1.4. A high-level architectural overview of CAML including a discussion of how custom library types can be extended by stereotypes of cloud environment profiles is given in Section 4.2. Sections 4.3 and 4.4 are devoted to the profiles and libraries constituting CAML. A summary of this chapter is given in Section 4.5 before work related to CAML is discussed in Section 4.6.

4.1 Motivation

To motivate the benefits of CAML, we use the application scenario as introduced in Section 1.4 and demonstrate the practical value of platform and environment-specific profiles for engineers faced with an application modernization towards the Google App Engine. Figure 4.1 depicts an excerpt of the reverse-engineered PSM of the PetApp, whereas the platform refers to Java in general and the applied platform-specific profiles in particular. These profiles can be exploited to accomplish the generation of a sliced PIM that sets the focus solely on the domain classes because they are annotated with JPA stereotypes in the PSM. Even better, some JPA stereotypes can be interpreted in terms of native UML concepts which not only strengthens the domain classes' independence of a platform but also increases their accuracy because *identifiers*, *compositions*, and more precise *multiplicities* can be explicitly captured. These improvements of the PIM demonstrate the practical value of considering platform-specific profiles in the context of a model-based reverse engineering process.

Profiles specific to a platform can also leverage the refinement of the PIM towards a platform without the need to identify mappings between the respective platforms. A profile dedicated to Objectify can support refining the domain classes of the reverse-engineered PIM towards the Google App Engine because applied stereotypes are capable of denoting entities to be persisted or embedded by another entity and properties which are uniquely identified. From the produced Objectify-based PSM, annotated application code can be generated by also interpreting applied stereotypes in the context of a forward engineering process. For instance, method bodies for CRUD operations can be generated for domain classes as they are indicated by the respective stereotypes and generated code elements can be automatically annotated (see Section 7.4 for excerpts of generated application code). Hence, platform-specific profiles act as an enabler for model-based reverse and forward engineering processes.

Having adapted the domain classes of the PetApp, it needs to be deployed on the cloud environment of Google as assumed by our scenario. Google's App Engine service offers several different virtual machine types that are priced based on an hourly rate. Furthermore, it provides several data storage solutions including a key-value datastore. Ideally, all the environment-related deployment decisions can be captured on the model level. Therefore, profiles specific to cloud environments appear to be highly desirable. Considering the deployment topology in Figure 4.2, a profile dedicated to Google's cloud platform can be exploited to express that for both modeled compute service instances, an *App Engine* service that hosts a Java-based platform on top of

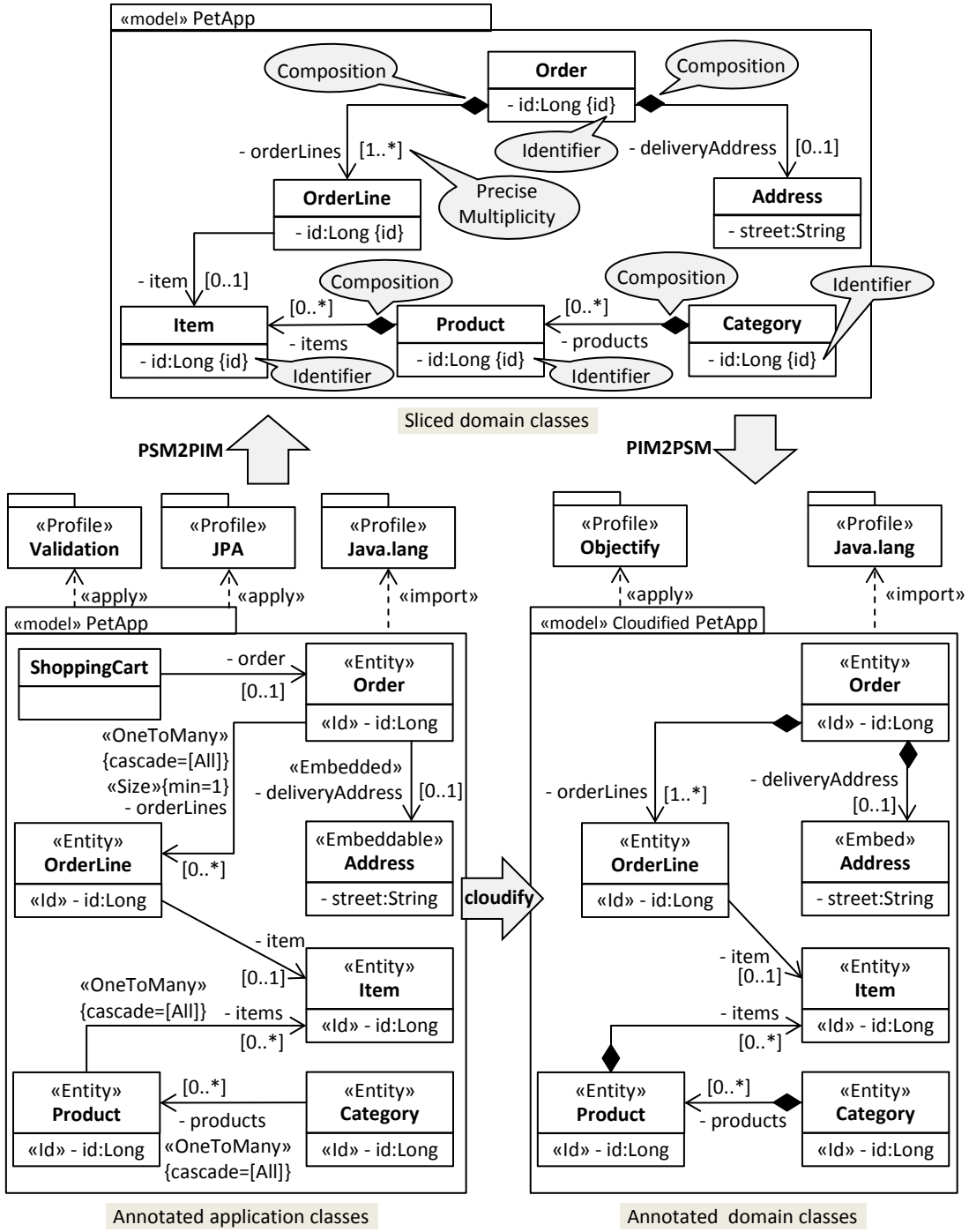

Figure 4.1: Benefits of annotation-based modeling in forward and reverse engineering processes

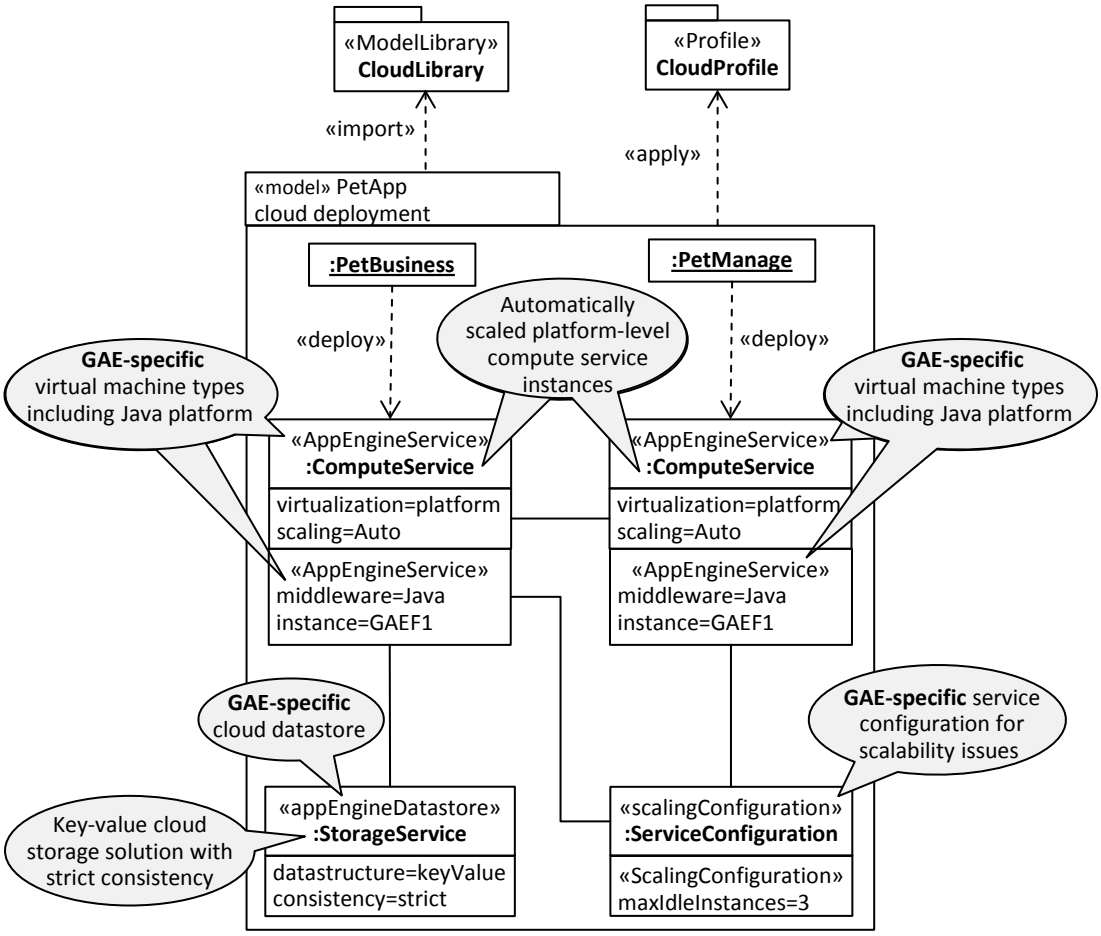

Figure 4.2: Benefits of cloud-specific extensions to UML

F1 virtual machine instances must be provisioned. Furthermore, one of the compute service instances must be capable to access Google's cloud datastore. To ensure that provisioned compute service instances are also released once they become idle, the modeled service configuration defines the upper limit of idle instances. Capturing the environment-related information in terms of profiles would allow engineers to model deployment topologies that are independent of a target environment because the respective stereotypes can be used in a refinement step in a similar way to platform-specific stereotypes. In a first step, an environment-independent deployment topology can be modeled by instantiating common cloud modeling types, such as `ComputeService` and `StorageService`, provided in terms of a dedicated library. Then, in a second step, the modeled deployment topology can be bound to the target cloud environment by applying the respective profile. As a result, environment-specific profiles along with a library of common cloud modeling types enables engineers to exploit UML's deployment viewpoint for modeling cloud applications as part of model-based forward engineering processes.

4.2 UML-based language for cloud application modeling

CAML aims at supporting engineers to model architectural decisions of cloud applications and the target cloud environment on top of which those applications are required to be executed. UML was selected as the host language to realize CAML. The main reasons for this design decision are manifold.

1. UML is a widely adopted standardized language that provides multiple viewpoints to represent software, platform, and infrastructure artifacts by means of common modeling concepts.
2. Its lightweight extension mechanism enables domain, platform, and environment-specific concepts to be integrated into UML via profiles and libraries in a systematic way.
3. A considerable set of industrial strength tools is available today for UML which together form an ecosystem that can be applied to UML-based approaches without any extra effort.

Recent work of Lago *et al.* [LMM+15] provides evidence that UML and UML profiles are used by many organizations and engineers to model the architecture of their applications. Their survey revealed that 86 percent of the respondents' organizations use UML or UML Profile for architecture modeling. The modeling language of choice plays an important role because created models must support the understanding of design decisions and their communication to the stakeholders involved in application design and should also promote automatic tasks [MLM+13] including the generation of implementations from (UML) models and vice versa as part of MDE processes [WHR14]. Clearly, the latter calls for mature tool support [WHR+13]. Today, several open-source and commercial modeling tools with a diverse tool set exist. CAML extends UML in a standard compliant way, such that current modeling tools that support UML are capable to adopt our developed extensions.

The core library and profiles constituting CAML are presented in Figure 4.3. As expected, all the building blocks of CAML are instances of the UML metamodel. Its profiles are defined by means of UML's profile language, some of which with the intention either to capture platform and environment-specific concepts for refinement purposes or to annotate other profiles. In particular, the pricing and performance profile are meant to be applied solely to profiles dedicated to capture services of cloud environments. Whereas the former profile enables descriptions of cloud services annotated with costs, high-level performance characteristics for such services can be represented by means of the latter profile. To offer an interface for contributing other cloud environment profiles to CAML, the common cloud profile provides abstractions over these specific profiles. Abstract stereotypes of the common cloud profile extend the custom types defined by the cloud library. They in turn instantiate meta-classes belonging to the deployment viewpoint of UML. Custom types of the cloud library can be used via the ontological instantiation mechanism supported by UML. Considering the cloud deployment topology of our application scenario in Figure 4.2, the modeled platform-level nodes are instances of the type `CloudNode`, which is provided by the cloud library. At the same time, they are instances of the UML meta-class `InstanceSpecification` for the purpose of realizing their embodiment. These two forms of typing are introduced in Section 2.2. From a language engineering point of view, this

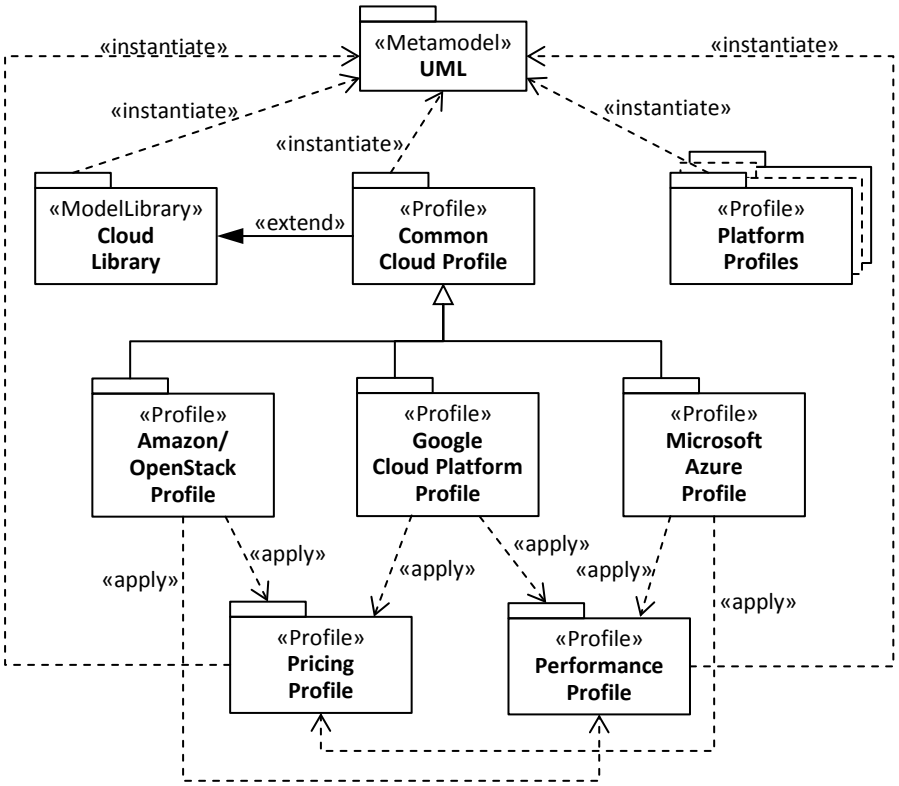

Figure 4.3: CAML architecture

differentiation is important to understand because UML's profile mechanism enables the extension solely of meta-classes.

As a result, custom types as defined via libraries in UML cannot directly be extended by stereotypes. Hence, the extension relationship between the cloud library and the common cloud profile requires a thorough consideration. For that reason, Figure 4.4 gives insights into how CAML deals with the challenge of extending custom UML types in addition to meta-classes as supported by default.

We take again one of cloud node instances of the PetApp's cloud deployment topology as an example. Its linguistic type refers to the meta-class `InstanceSpecification` whereas from an ontological perspective it is of type `CloudNode`. Obviously, the applied stereotype `GAEF1` should be applicable to instances of `CloudNode`. Hence, the stereotype of the cloud profile in the upper part of Figure 4.4 directly extends the custom `CloudNode` type. However, if we define the extension relationship without specific treatment the stereotype would not be applicable to cloud node instances because UML's stereotype application mechanism does not consider custom types as valid targets. Instead, only meta-classes are considered as valid targets. Hence, the extension relationship pointing to the custom `CloudNode` type needs to be redefined, such that the `GAEF1` stereotype extends the UML meta-class which is instantiated by cloud node instances. In other words, the extension relationship needs to follow the path of the linguistic instantiation

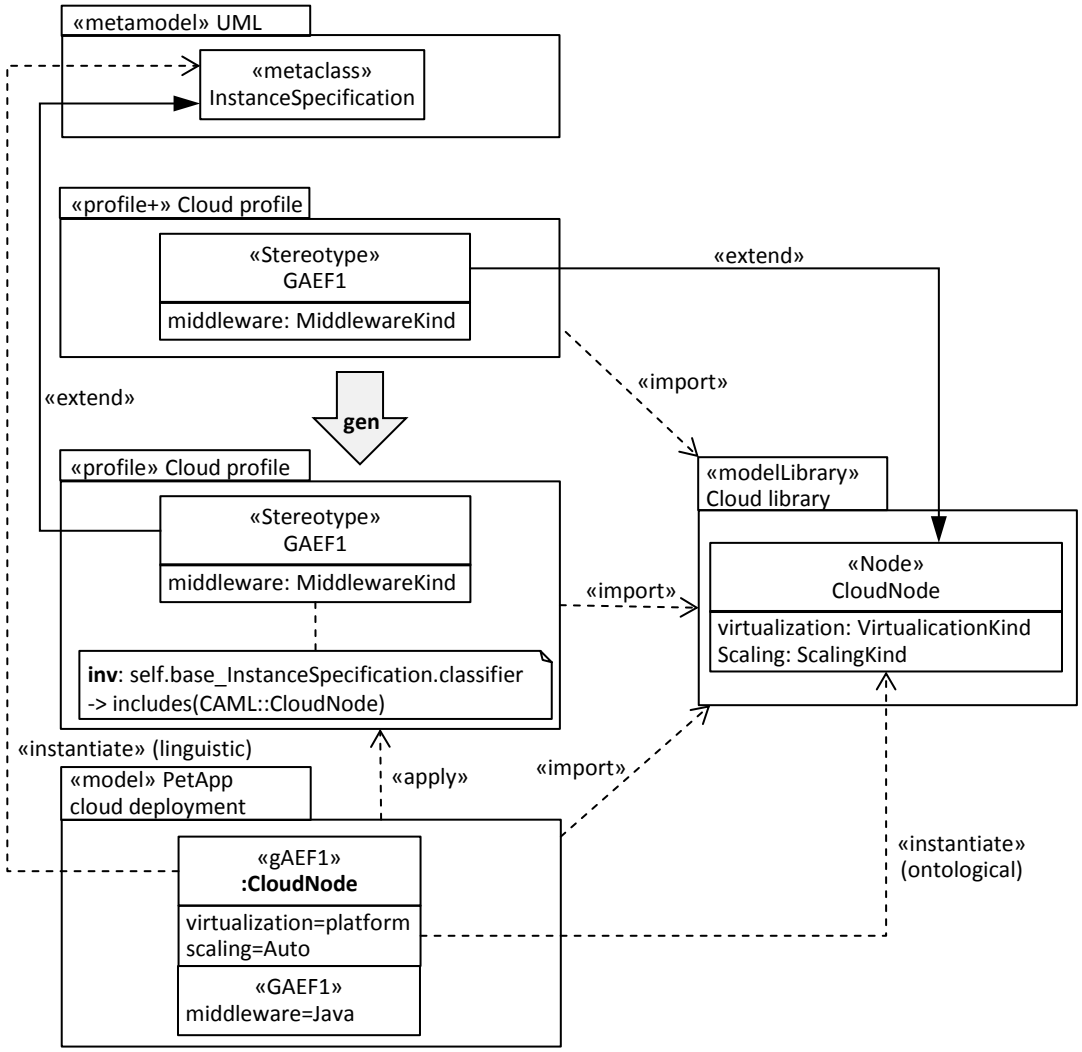

Figure 4.4: Pattern for extending custom types in UML via stereotypes

instead of the ontological one. To effectively support ontological extensions, a pre-processing step is carried out by CAML. If an extension relationship is defined along the path of ontological instantiation the extended custom type is replaced by the meta-class `InstanceSpecification` to accomplish an applicable stereotype definition. `InstanceSpecification` is the common reference meta-class that has been introduced by UML to embody custom types and let them point to an ontological type. To guarantee that the defined stereotypes are only applicable to the extended custom types, a specific OCL constraint is introduced. It restricts the application of the defined stereotype to instances of the custom type, *i. e.,* concrete instance specifications that point to the extended custom type. The `GAEF1` stereotype covered by the profile in the middle of Figure 4.4 presents the solution generated by CAML for the profile in the upper part. As a result, CAML supports engineers to develop concise UML profile definitions even for ontological

extensions without requiring changes to existing UML tools because the provided generative approach ensures that extension relationships are appropriately redefined if necessary.

4.3 Extensions to UML for target platforms in the cloud

Target platforms offered by a cloud environment often replicate existing platforms including supported programming languages, available programming libraries, and standardized applications containers that are already commonly used in practice. One prominent example is the Java platform. From a platform point of view, CAML targets engineers who employ UML in order to realize reverse engineering and forward engineering processes where application artifacts are implemented in Java. The investigation of realizing mappings between these two technical spaces [KBA02, JCD+12] has already a long tradition [EHSW99, HBR00, NNZ00, KSS+02]. A major contribution of CAML is the consideration of Java annotations and UML profiles in the mapping process. Recent novelties of Java 8 regarding *repeating annotations* are included in the mapping because it leads us to revisit how stereotypes are defined and applied in profile applications [LWWC12]. In this respect, we discuss pros and cons of three significantly different solutions to support *repeating stereotypes* in analogy to *repeating annotations* and modifications required to the current UML 2.4 formal specification and its Eclipse-based reference implementation that are implied by two of them. CAML supports the generation of UML profiles with repeating stereotypes. To give insights into our generative approach, we discuss the conceptual mapping underlying it and elaborate effective solutions to overcome existing heterogeneities between Java and UML. As a result, it allows engineers to "jump" from Java libraries to UML profiles. All the platform-specific UML profiles that we have generated throughout the evaluation of CAML were contributed to the Eclipse UML Profiles Repository (UPR) [UPR15]. Our aim is to share them with existing community portals in general and the Eclipse modeling community in particular.

4.3.1 Repeating stereotypes

Since Java 8, *repeating annotations* enable the same annotation to be repeated multiple times in the place where it is declared. Obviously, this repeatable application of annotations has an effect on determining the multiplicity of the `ExtensionEnd` contained by the `Extension` relationship. In case of repeating annotations, the multiplicity should be 0..*, which expresses that the corresponding stereotype can also be applied to base elements[1] multiple times. However, the UML standard introduces an OCL constraint that explicitly hinders the application of the same stereotype to the same base element more than once (see page 683 of the UML standard [OMG15]). As shown in Listing 4.1, the OCL constraint restricts the upper bound of the extension end to 1.

Listing 4.1: Multiplicity constraint on `ExtensionEnd`

```
(self->lowerBound() = 0 or self->lowerBound() = 1) and self->upperBound() = 1
```

[1] A stereotype is applicable to a base element if it is an instance of a meta-class extended by the stereotype. Considering the `Entity` stereotype, it extends the meta-class `Type`. As a result, it can be applied to all instances of the meta-class `Type`.

When considering a stereotype as a means to *classify* base elements, the restriction on the upper bound of the extension end seems reasonable. Classifying the same base element twice by the same stereotype is obviously inappropriate. In contrast, when considering a stereotype as a means to *annotate* base elements, there are use cases for applying the same stereotype to a base element multiple times. For instance, in the context of model versioning dedicated stereotypes can be used to visualize changes to a model element, *e. g.,* "update class", and highlight potential conflicts, *e. g.,* contradicting updates to a class, as a result of concurrently edited model versions [BKL+10]. As updates to classes may be manifold, the respective stereotype is ideally applied to the changed class several times where each atomic change is captured by exactly one applied stereotype. To give another example, expressing several queries for an entity with the JPA profile requires applying the `NamedQuery` stereotype multiple times. As a result, even though *repeating stereotypes* in analogy to repeating annotations are currently not supported by standard UML, they are still desirable.

To realize repeating stereotypes, several solutions are conceivable. Table 4.1 summarizes three such possible solutions and shows pros and cons for all of them. Concerning the UML metamodel and tools that depend on it, we refer to the Eclipse-based reference implementation.

Solution	Stereotype		Changes in UML metamodel and tools	Backward compatibility	
	Repeatable application	Container		UML metamodel	Tools
Composition of multiple stereotypes	not supported only contained by a dedicated stereotype	explicitly modeled	not required	yes	yes
Emulation of repeating stereotypes	supported but contained by a dedicated stereotype	automatically generated	yes, moderate effort	no	yes
Native support for repeating stereotypes	supported	not required	yes, relatively high effort	no	no

Table 4.1: Possible solutions for repeating stereotypes

The first solution is fully compliant to the current UML standard. In fact, it does not actually apply several stereotypes to a base element. Instead a dedicated stereotype acts as a container for the repeating stereotypes. This solution foresees that the container stereotype is explicitly created by the modeler. As a result, changes to the UML metamodel and tools built on top of its API are not required because the repeating stereotypes are only referenced by their container stereotypes rather than applied to base elements. On the contrary, however, standard operations, for instance, to apply stereotypes and retrieve them, are not applicable by this solution for repeating stereotypes as they provide the expected result only for stereotypes that are applied following the standard procedures.

This drawback is compensated by the second solution. It emulates repeating stereotypes as a result of slight modifications to the operations provided for stereotypes. Even though, similarly to the first solution, a container stereotype is exploited also by the second solution, this container

is automatically generated on demand. Moreover, as a result of the modifications required by this solution, all standard operations for stereotypes are applicable also to repeating stereotypes. However, the extension ends pointing to them need to be multivalued to ensure that they can be applied multiple times. Consequently, this solution neglects the multiplicity constraint of the `ExtensionEnd` meta-class, which in turn leads to profiles that do not fully conform to the current UML metamodel. Still, the compatibility with existing tools can be ensured because the required changes can completely be hidden by the UML metamodel API. This backward compatibility cannot be maintained by the third solution. To natively support repeating stereotypes without providing a dedicated container stereotype requires not only changes in the UML metamodel API but also how they are represented and edited by the tools. For instance, applied stereotypes are represented according to unique categories to which also their features are assigned, where the category is derived from the name of a stereotype. Applying the same stereotype multiple times to the same base element would result in a single category to which all the features of the applied stereotypes are assigned.

To demonstrate how the profiles with repeating stereotypes of the three discussed solutions differ from each other, we refer again to the `NamedQuery` annotation of the JPA. Listing 4.2 shows its declaration as a repeatable annotation, whereas Listing 4.3 declares the required container annotation. For compatibility reasons, in Java 8, repeating annotations are stored in a container annotation that is automatically generated by the Java compiler once the annotation is applied.

Listing 4.2: Declaration of `NamedQuery` repeating annotation

```
package javax.persistence;
import java.lang.annotation.*;

@Target({ElementType.TYPE})
@Repeatable(NamedQueries.class)
public @interface NamedQuery {
  String name();
  String query();
}
```

Listing 4.3: Declaration of `NamedQueries` container annotation

```
package javax.persistence;
import java.lang.annotation.*;

@Target({ElementType.TYPE})
public @interface NamedQueries {
  NamedQuery [] value ();
}
```

Considering the possible profile solutions in Figures 4.5 to 4.7for the annotation declarations, we selected the notation used to represent associations and their member ends instead of extensions [LWWC12] to explicitly indicate the multiplicities of the extension relationships. The first profile depicted in Figure 4.5, allows multiple `NamedQuery` stereotypes to be composed by its container stereotype. As expected, the latter extends `Type`, where the multiplicity of the extension end pointing to the `NamedQueries` stereotype is 0..1. It indicates that the container stereotype can be applied once, which is sufficient because the composition relationship between the `NamedQueries` stereotype and the `NamedQuery` stereotype is multivalued.

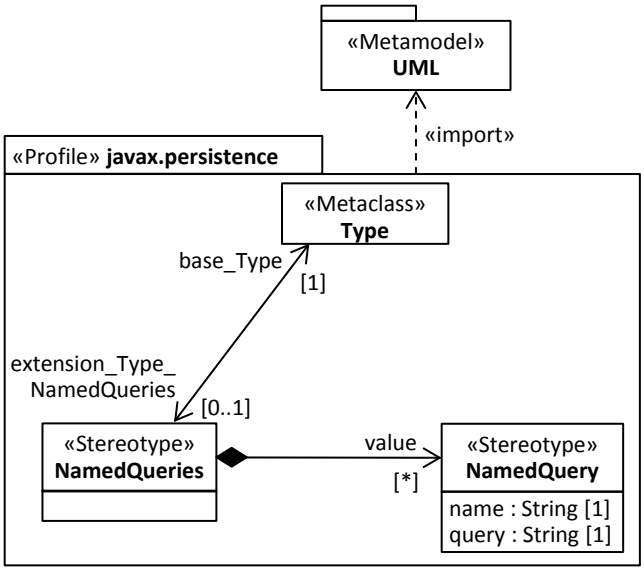

Figure 4.5: Profile for composing multiple stereotypes, see first solution in Table 4.1

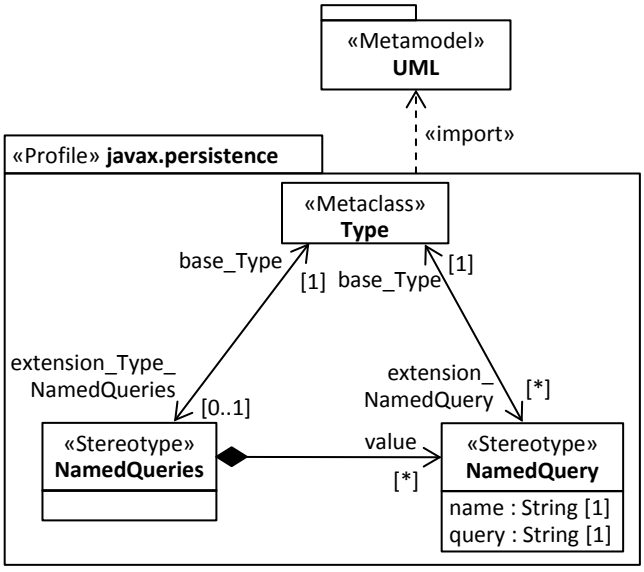

Figure 4.6: Profile for emulating repeating stereotypes, see second solution in Table 4.1

Similarly, the second profile shown in Figure 4.6 exploits a multivalued composition relationship to emulate repeating stereotypes. The main difference compared to the first profile is that the `NamedQuery` stereotype also extends `Type`, where the multiplicity of the extension end pointing to the stereotype is 0..*, which indicates that it is a repeating stereotype. As a result, the `NamedQuery` stereotype is applicable to base elements that are instances of the meta-class `Type`. It is important that the extension relationships of both defined stereotypes point to the same

meta-class because the container stereotype need to be applicable to exactly the same set of base elements as the repeating stereotype. In fact, this solution resembles the realization of repeating annotations in Java 8. From the perspective of a modeler, the second profile is more powerful compared to the first one, as the required container stereotype is managed in the background by the UML metamodel API and hence fully transparent to the modeler. The development effort is slightly higher and the profile more complex because an additional extension relationship is required for the repeating stereotype.

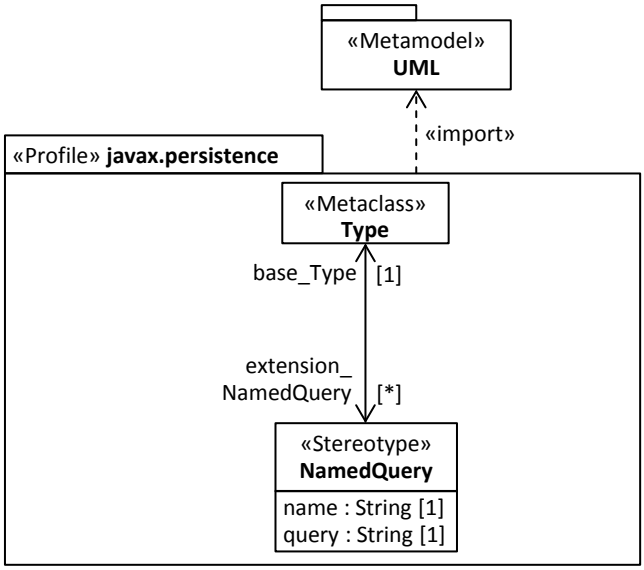

Figure 4.7: Profile for native support of repeating stereotypes, see third solution in Table 4.1

Finally, the profile envisaged for the native support of repeating stereotypes is given in Figure 4.7. It does not require a container stereotype to capture repeating stereotypes because they are assumed to be directly applied to the base elements. The main difference to the previous solution is that each applied repeating stereotype is captured by its own `StereotypeApplication` instead of composed by an artificially introduced container stereotype for reasons of backward compatibility. The latter can be considered as the trade-off between natively supporting repeating stereotypes and guaranteeing that the solution is compatible at least with tools that are built on top of the UML metamodel API.

Concerning support for the three discussed solutions of repeating stereotypes and the respective profiles, CAML allows the generation of these profiles by passing the respective configuration option. Hence, the modeler can decide which profile version should be generated from a Java library. Clearly, to emulate repeating stereotypes, a modified UML metamodel API is required whereas native support for them requires also modifications in the tools built on top of this API.

4.3.2 Generating UML profiles from Java libraries

We start our investigation for generating UML profiles from annotation-based Java libraries by presenting the process of CAML, as shown in Figure 4.8. The entry-point to the profile generation

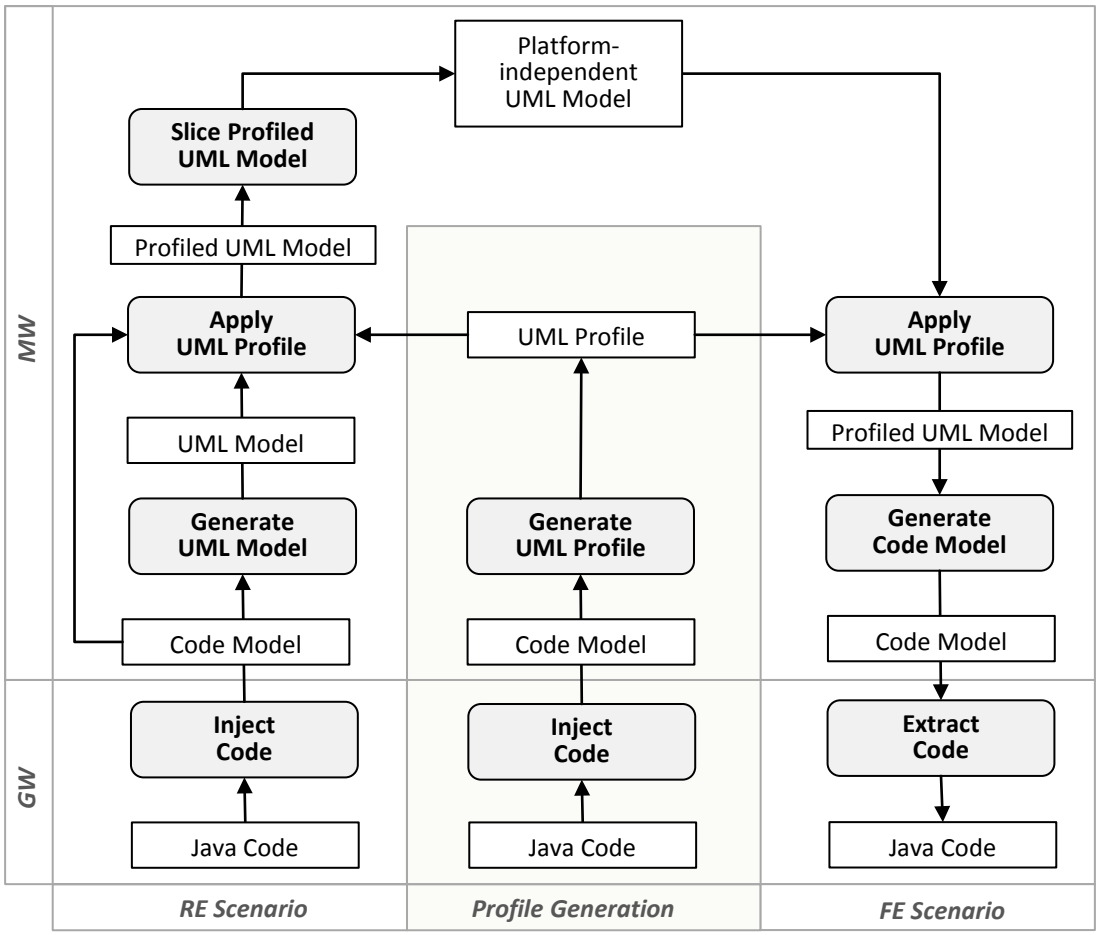

Figure 4.8: Processes for UML profile generation and application scenarios

is *Java Code* that is translated into a what we call *Code Model* conforming to MOF/EMF. As a result of this first step, the transition from a text-based representation into a model-based representation is accomplished. The generated *Code Model* is a one-to-one representation of the *Java Code* and the basis for generating a *UML Profile*, which captures Java annotation type declarations in terms of UML stereotypes (see middle of Figure 4.8). They serve as foundation to exploit profiles as an annotation mechanism [Sel12].

In case of the reverse engineering scenario, a *Code Model* is generated in a first step similar to the UML profile generation. The *Profiled UML Model* is generated from the *Code Model* by taking into account profiles that provide stereotypes corresponding to annotations in the *Code Model* (see left hand side of Figure 4.8). Annotated elements of the *Code Model* indicate the elements of the *Profiled UML Model* to which those stereotypes need to be applied. As demonstrated in Figure 4.1, stereotypes applied to elements of the *Profiled UML Model* may lead to a more accurate *Platform-independent UML Model* if they are appropriately interpreted by a model slicer [BCBB15]. This slicing step is specific to the interpreted stereotypes and

hence the profiles that cover them, whereas the steps of generating a *Profiled UML Model* and an intermediate *Code Model* is completely generic in the sense that any Java application and library can be translated into a *Profiled UML Model*.

A *Profiled UML Model* is accomplished in the case of the forward engineering scenario (see right hand side of Figure 4.8) by applying profiles to a *Platform-independent UML model*. While UML's profile mechanism is generic in the sense that arbitrary profiles can be applied to a UML model, automating the application of stereotypes to particular elements is certainly specific to the application scenario. In contrast, both the generation of the *Code Model* and the extraction of the *Java Code* are generic provided that the employed code generation facility supports stereotypes. Considering the generation of stereotypes compared to the reverse and forward engineering scenarios where those stereotypes are applied, the respective profile generation process operates at a different level as the processes supporting the two application scenario because declared stereotypes can be considered as part of the metamodel level instead of the model level [AKHS03] (see Figure 2.8). Following this classification into different levels, the profile generation is a meta-level process, which produces elements at the metamodel level.

Finally, bridging the two technical spaces [JCD+12] we are confronted with, *i. e.,* grammarware and modelware, is required for the two scenarios as well as CAML.

Bridging Technical Spaces

Transforming plain Java code into a UML-based representation requires overcoming the different encoding and resolving language heterogeneities. Concerning the first aspect, the Java code needs to be encoded according to the format imposed by the modeling environment [BW13]. Concerning the second aspect, a bridge between Java and UML based on translations requires a conceptual mapping between the two languages. Instead of directly translating plain Java code into a UML-based representation, the use of a two-step approach is preferable [HJSW10], which is also applied by CAML. In a first step, *Java code* is translated into a *Code model* that uses Java terminology and structures conforming to the Java metamodel provided by MoDisco [BCJM10]. An excerpt of the Java metamodel[2] is presented in Figure 4.9.

It shows the main meta-classes required to declare and apply annotations. Before annotations can be applied on code elements, they need to be declared in terms of annotation types. `AnnotationTypes` declare the possible annotations for code elements and may have, similar to Java interface declarations, optional `modifiers`. They are identified by a `name`. `AnnotationTypes` may themselves be subject for annotations. An `Annotation` references to its type and composes `ElementValuePairs`. They capture values passed to an annotation. Most importantly for the context of this work is the target annotation that is represented in the metamodel as an attribute for simplicity reasons. It is a meta-annotation because it can only be applied to declared annotation types to indicate the code elements that are valid bases for an application of an `AnnotationType`. The set of valid bases are captured by the literals of `ElementType` enumeration. Note that we omitted the newly introduced `TypeUse` and `TypeParameter` literals as they are considered as part of future work. Generally, UML does not support such annotations by default as it would require extending not only meta-classes but also meta-features, which is not

[2]The presented metamodel uses the terminology of the Java language specification (JLS8) [Ora15]

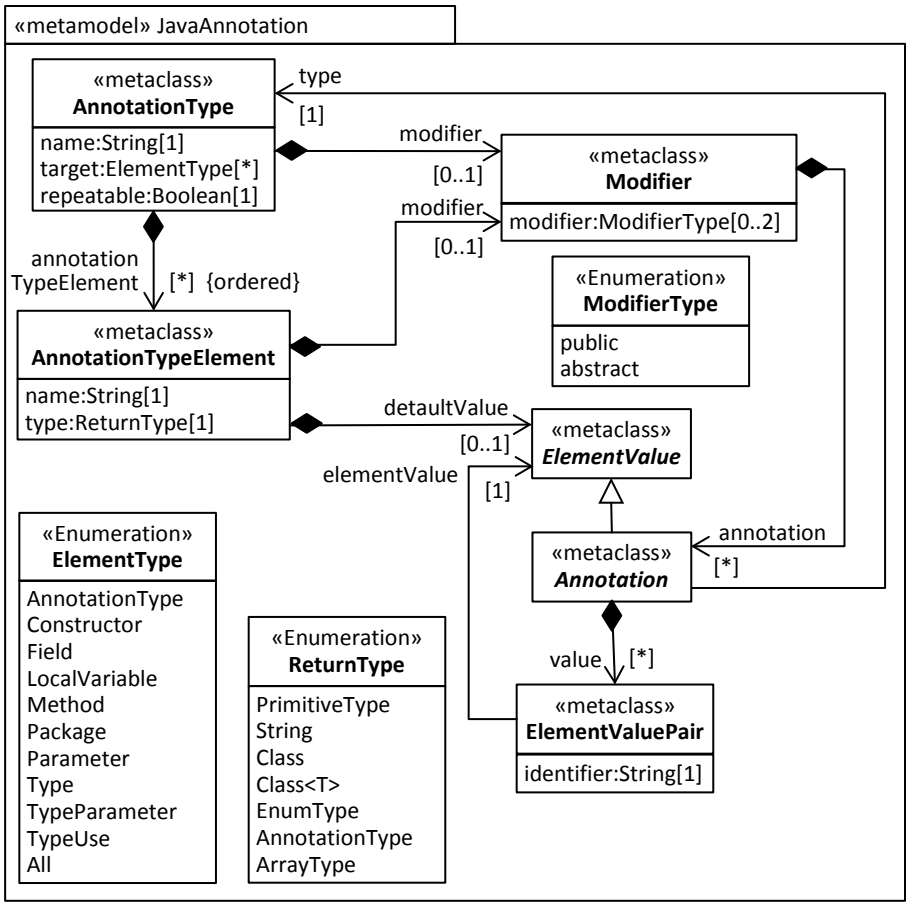

Figure 4.9: Metamodel of Java annotations

yet supported. The body of an annotation type declaration consists of zero or more `AnnotationTypeElements` for holding information of `AnnotationType` applications. They are declared in terms of method signatures with optional `modifiers`, a mandatory return `type` and `name`, and an optional `default value` that is returned if no custom value is set. The `default` value needs to conform to the defined return `type` of the `AnnotationTypeElement`. For instance, if the defined return `type` is `AnnotationType`, the `default value` needs to be an `Annotation`, which inherits from `ElementValue`. This abstract meta-class is specialized by other meta-classes, *e. g.,* `ConditionalExpression` to support the non-array `ReturnTypes` and `ElementValueArrayInitializer` to support one-dimensional arrays thereof. For the sake of brevity, these additional specializations of `ElementValue` are omitted.

The introduced meta-classes are instantiated for creating a *Code model* from Java code. A *Code model* is the basis for generating UML profiles and the input for the second step that is dedicated to resolving language heterogeneities by relying on the correspondences between the Java and UML metamodels.

Generating UML Profiles

To facilitate the generation of UML profiles, we present a conceptual mapping between Java's annotation language and the profile language of UML. Thereby, stereotypes play a vital role for representing annotation types on the model level as they enable their application in a controlled UML standard-compliant way. From a language engineering perspective, stereotypes only extend the required UML meta-classes and facilitate defining constraints and model operations, such as model analysis or transformations, because they can directly be used in terms of explicit types similar to a meta-class in UML. Our proposed mapping is generic in the sense that any declared annotation type can be represented by a stereotype.

Java	→	UML
AnnotationType a		**if**(**not** a.isContainerAnnotation() **or** a.isContainerAnnotation()[1] **and** requiresContainerStereotype()[2]) **add** Stereotype s s.name = a.name **add** Property p **for each** AnnotationTypeElement **in** a.annotationTypeElement
switch(a.modifier) **case** : public **case** : abstract **case** : annotation an **and not** an.type = Target **case** : annotation an **and** an.type = Target		 s.visibility = public s.abstract = false **apply** Stereotype **for** an.type **to** s **for each** ElementType et **in** a.target **add** Extension e **for each** Metaclass mc **in** et.getMetaclasses()[3] e.memberEnd = Set{p, f} **add** ExtensionEnd f f.name = "extension_".concat(mc.name).concat("_").concat(s) f.type = s f.aggregation = AggregationKind.composite f.lower = 0 f.upper = **if**(a.isRepeatable()) * **else** 1 **add** Property p **for each** Metaclass mc **in** et.getMetaclasses()[3] p.name = "base_".concat(mc.name) p.type = mc **if**(a.target.size() >= 1) 1 **else** 0[4] **if**(et = Constructor) **add** uml::Constraint *constructorConstraint*[4] **if**(et = Method) **add** uml::Constraint *methodConstraint*[5] **if**(et = Type) **add** uml::Constraint *typeConstraint*[6]

[1] See Listing 4.4 for further details.
[2] It provides a Boolean value of the decision taken by the modeler regarding repeating stereotypes.
[3] See Listing 4.5 for further details.
[4] `AnnotationTypes` that are intended to used only inside other annotations require a zero multiplicity (*e. g.,* `QueryHint` of the JPA).
[5] See Listing 4.6 for its specification.
[6] See Listing 4.7 for its specification.
[7] See Listing 4.8 for its specification.

Table 4.2: `AnnotationType` to `Stereotype` mapping

AnnotationType → Stereotype. The mapping presented in Table 4.2 provides the basis to generate an applicable `Stereotype` from an `AnnotationType`. First of all, it needs to be

decided if a stereotype should be generated at all, because container annotations required to declare repeating annotations may not in any case result in a corresponding container stereotype as discussed in the previous Section 4.3.1. For that reason, a corresponding container stereotype is generated from a container annotation only if it is required, which depends on the provided configuration parameter passed by the modeler. Whether an annotation type is exploited as a container annotation is indicated by the meta-annotation `Repeatable` applied to the annotation type for which the container annotation is declared. Listing 4.4 gives the respective function to check for container annotations.

Listing 4.4: Definition of *isContainerAnnotation*

```
context AnntotationType def: isContainerAnnotation: Boolean =
Annotation.allInstances() -> select(a : Annotation | a.type.name = "Repeatable")
    ↪-> collect(value) -> exists(elementValue.type = self)
```

In cases where repeating stereotypes should be composed by a dedicated stereotype instead of directly applied to a base element, see first row in Table 4.1, or emulated in terms of repeating annotations, see second row in Table 4.1, a container stereotype is generated. Otherwise, its generation is neglected, see third row in Table 4.1. The function *requiresContainerStereotype* provides a Boolean value of the decision taken by the modeler.

Generally, the generation of a stereotype from an annotation type requires not only its signature to be considered, but also Java's `Target` meta-annotation. It determines the set of code elements an annotation type is applicable to. The `name` and, with two exceptions, the defined `modifiers` of an `AnnotationType` can straightforwardly be mapped to UML. First, the `abstract` modifier would lead to `Stereotypes` that cannot be instantiated if directly mapped. The problem is caused by Java's language definition. Although the `abstract` modifier is supported to facilitate one common type declaration production rule, it does not restrict the application of `AnnotationTypes`. To ensure the same behavior on the UML level, we never declare a `Stereotype` to be abstract. Second, because annotations are considered as modifiers, it needs to be ensured that the `Target` annotation is properly treated. In fact, the defined set of Java `ElementTypes` determines the required set of `Extensions` to UML meta-classes that specify the application context of the stereotypes. Listing 4.5 defines the correspondences between Java `ElementTypes` and UML meta-classes.

Listing 4.5: Definition of *getMetaclasses*

```
context ElementType def: getMetaclasses: Set(uml::Element) =
if(self = AnnotationType) then Set{uml::Stereotype}
else if(self = Constructor) then Set{uml::Operation}
else if(self = Field) then Set{uml::EnumerationLiteral, uml::Property}
else if(self = LocaleVariable) then Set{uml::Property}
else if(self = Method) then Set{uml::Operation, uml::Property}
else if(self = Package) then Set{uml::Package}
else if(self = Parameter) then Set{uml::Parameter}
else if(self = Type) then Set{uml::Type}
else if(self = All) then Set{uml::Class, uml::Enumeration, uml::Interface, uml::
    ↪Operation, uml::Package, uml::Parameter, uml::Property, uml::Stereotype}
```

Most of them correspond well to each other. Still, some constraints are required to precisely restrict the application scope of the generated `Stereotype` according to their intention. UML

does not explicitly support a constructor meta-class. The workaround is to map the `Constructor` to `Operation` and introduce a constraint that emulates the naming convention for constructors in Java, as depicted in Listing 4.6. Note that annotation types can have several target types. Thus, before validating the OCL constraint, we have to check which target is actually used in the application.

Listing 4.6: Constructor constraint

```
context generated Stereotype inv:
self.base_Operation.oclIsDefined() implies
self.base_Operation.name = self.base_Operation.oclContainer().oclAsType(uml::
    ↪Classifier).name
```

Similarly, the mapping of Java methods to UML requires a constraint as a declared method of an `AnnotationType`, *i. e.,* `AnnotationTypeElement`, is mapped to a `Property` rather than an `Operation` in UML. This is because such methods do not provide a custom realization but merely return their assigned value when they get called. `Properties` in UML provide exactly this behavior. Hence, the constraint in Listing 4.7 ensures that stereotypes generated from annotation types that target Java methods are applicable also to `Property` if they are contained by a Stereotype.

Listing 4.7: Method constraint

```
context generated Stereotype inv:
self.base_Property.oclIsDefined() implies
self.base_Property.oclContainer().oclIsTypeOf(uml::Stereotype)
```

Finally, we use a constraint to overcome the heterogeneity of Java's and UML's scope of `Type`. Consequently, stereotypes that extend `Type` are constrained to those elements that correspond to the set of elements generalized by Java's `Type`: `AnnotationType`, `Class`, `Enumeration` and `Interface`. The clear benefit of this approach is a smaller number of generated extension relationships between stereotypes and meta-classes in the profile. The constraint is depicted in Listing 4.8.

Listing 4.8: Type constraint

```
context ToBeGeneratedStereotype inv:
self.base_Type.oclIsDefined() implies
Set{uml::Stereotype,uml::Class,uml::Enumeration,uml::Interface} -> includes(self.
    ↪base_Type.oclType())
```

AnnotationTypeElement → Property. An `AnnotationTypeElement` is mapped to a `Property` as depicted in Table 4.3. Except for the fact that UML properties cannot be defined as abstract, `AnnotationTypeElements` straightforwardly correspond to `Properties`. In Java, `AnnotationTypes` cannot explicitly inherit from super annotations. Therefore, the `abstract` modifier is rarely used in practice. To fully support all return types of `AnnotationTypeElements`, we introduce a `Stereotype` to address the generic capabilities of `java.lang.Class`, which is not the case for UML's meta-class `Class`. Hence, we apply our custom `JGenericType` stereotype to properties with return type `Class<T>`.

Java	→	UML
AnnotationTypeElement a		**add** Property p p.name = a.name p.default = a.default p.lower = **if**(p.default.isEmpty()) 1 **else** 0
switch(a.modifier)		
case : public		p.visibility = public
case : abstract		– *no corresponding feature*
case : annotation an		**apply** Stereotype **for** an.type **to** p
switch(a.type)		
case : PrimitiveType		p.type = uml::PrimitiveType **for** a.type
case : Class		p.type = uml::Class
case : Class<T>		p.type = uml::Class **apply** javaProfile::JGenericType Stereotype **to** p
case : EnumType		p.type = uml::Enumeration
case : AnnotationType		p.type = uml::Stereotype
case : ArrayType		p.type = a.typeOfArray()[1] p.upper = 1

[1] Extracts the type of the array.

Table 4.3: `AnnotationTypeElement` to `Property` mapping

4.3.3 Collected UML profiles and their usage

To demonstrate concrete UML profiles that were generated from annotation-based Java libraries, Figures 4.10 and 4.11 show excerpts of the platform-specific profiles that are applied in the context of our application scenario. Whereas the JPA profile is applied in the reverse engineering process to generate a profiled UML model from the PetApp implementation, the Objectify profile captures the stereotypes applied to refine the platform-independent UML model towards Google's cloud environment as part of the forward engineering process. Considering the former profile, all the applied stereotypes to the PetApp domain model that refer to the JPA library are covered by the JPA profile. It was generated by means of the first solution regarding repeatable stereotypes (see Table 4.1) because a one-to-many composition relationship links the `NamedQueries` stereotype to the `NamedQuery` one, the former stereotype being the container for named queries applied to an entity. Also, the Objectify profile provides as expected all the stereotypes required for the "cloudified" version of the PetApp domain model.

Since early 2014, an Eclipse project is dedicated to develop a centralized repository that hosts standardized UML profiles, such as BPMN [OMG11a] and SoaML. The UML-Profile-Store complements the set of standardized profiles of the UML Profiles Repository (UPR) with profiles specific to the Java platform. Contributing these Java-specific profiles to the UPR appears obvious. Thereby, the Eclipse Modeling Community can access them via a common repository for standardized platform-independent profiles as well as profiles that are specific to a platform such as Java.

All the profiles specific to the Java platform are bundled in an Eclipse plug-in that currently provides 20 profiles, comprising in total over 700 stereotypes. To ensure that they can directly be used in the Papyrus UML modeling tool[3], the profiles are automatically registered via the

[3]Papyrus: https://eclipse.org/papyrus

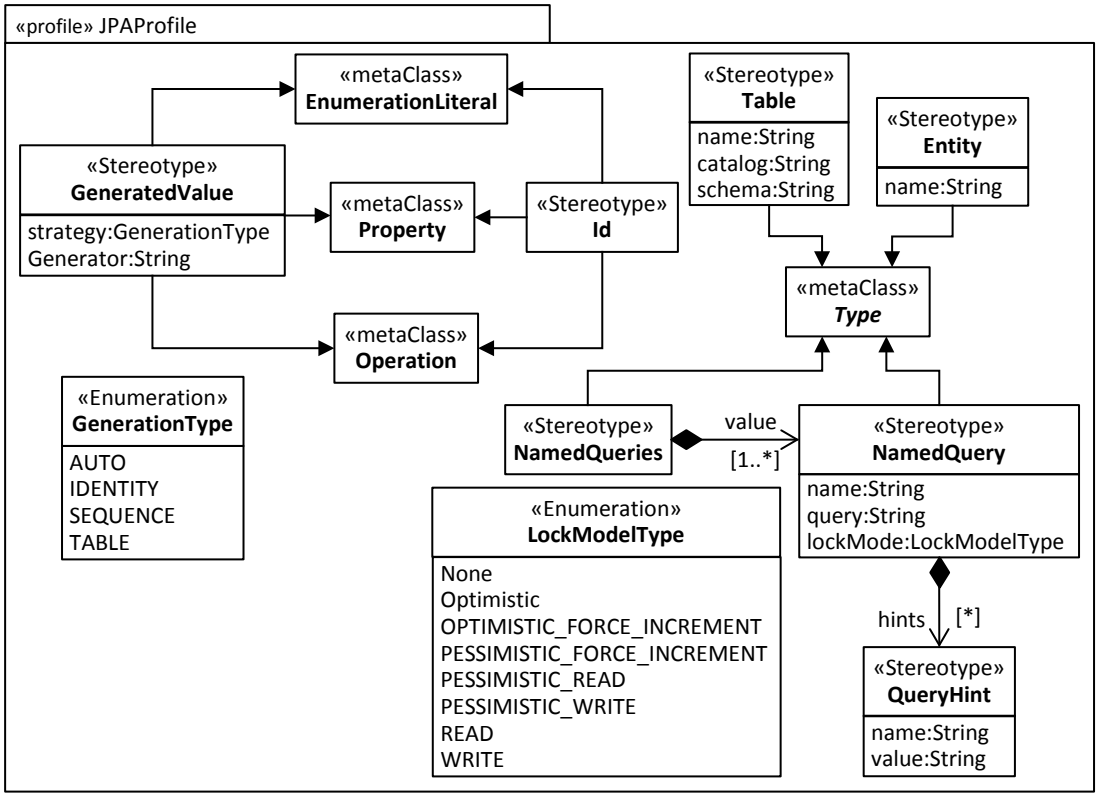

Figure 4.10: UML profile for JPA library

respective extension point offered by Papyrus. Also, all the profiles are defined in the sense that in addition to the profile definition itself Ecore-based metamodels are available for them. Those metamodels are automatically generated by the Eclipse UML reference implementation and mainly required for the stereotype application. If a stereotype is applied to a model element, an instance of the stereotype's meta-class is created along with a reference that associates the stereotype to the model element.

How the stereotype application is actually implemented by modeling tools which support UML may vary among them. This implies that the Ecore-based metamodels defined for the profiles may not necessarily be applicable by other tools which is not surprising because Ecore and the modeling framework built on top of it is grounded in Eclipse. Hence, the profiles are also offered in a tool neutral version, such that they are exchangeable among modeling tools. It just means that the Ecore-related meta-classes for the stereotypes are not included in the profiles but solely their UML native definitions.

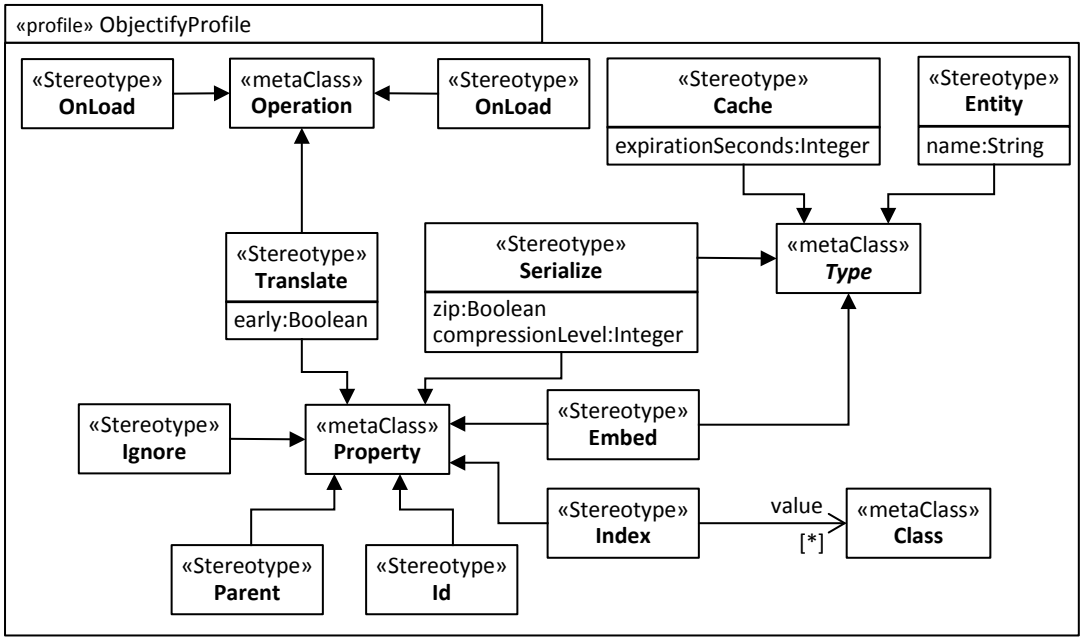

Figure 4.11: UML profile for Objectify library

4.4 Extensions to UML for target cloud environments

With the emergence of cloud services and the demand to exploit them, deployment topologies modeled in UML must be expressive enough to capture them. A major contribution of CAML are cloud-specific extensions to UML's generic deployment language. Developing extensions requires the investigation of the cloud computing domain with a particular emphasis on cloud services and their features, and the identification of useful abstractions over the gathered domain knowledge [MHS05]. The identified abstractions are the basis for designing and implementing CAML on top of UML. We discuss the steps carried out for extracting information related to offered cloud services and briefly summarize the phases performed for developing the extensions to UML based on our gained insight into the cloud service landscape. Then, we present the model library of CAML which allows engineers to model deployment topologies independent of a specific cloud environment. Because the ultimate objective is to support the provisioning of modeled cloud services, we show how profiles can close the gap between deployment topologies on the model level and target cloud environments offering the services to be provisioned. Finally, we present how templates in UML can be exploited for contributing deployment topologies as reusable blueprints.

4.4.1 Investigation of cloud environment services

Available information about cloud services is scattered across a variety of web pages hosted by the providers of cloud environments offering those services. A well-known approach to capture existing cloud services in a hierarchically structured form is feature-oriented domain

analysis (FODA) [KCH+90]. It promotes feature models to explicitly represent commonalities and variabilities of a problem domain. A commonly used feature diagram notation is introduced by Czarnecki & Eisenecker [CE00]. The root of a feature model denotes the concept that is modeled. It is decomposed into features, where child features depend on parent features. Dependencies between features impose constraints on how they can be combined: a feature is mandatory, alternative, or optional.

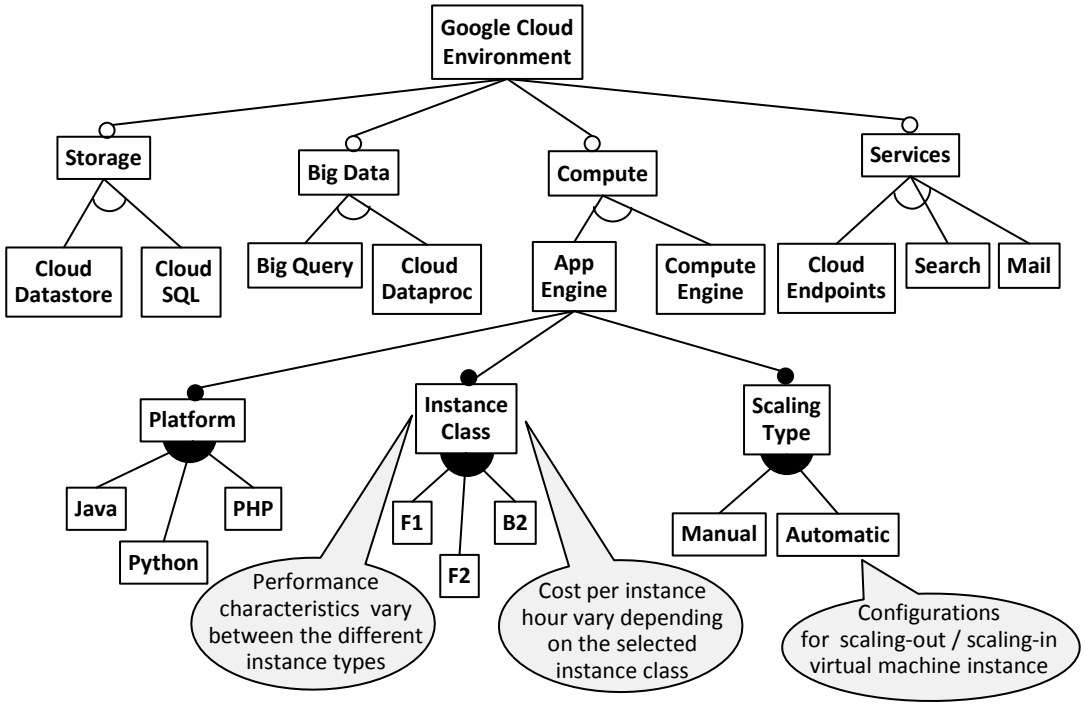

Figure 4.12: Feature model of the Google cloud environment

Considering an excerpt of the feature model (see Figure 4.12) we created based on the analysis of Google's cloud environment, the root of the model denotes the environment under investigation. It provides a variety of cloud services that belong to a certain category. For instance, the *App Engine* service is a *compute* service offered by Google's cloud environment. The App Engine service is a platform-level compute service because it provides in contrast to *Compute Engine* services also a platform that is immutable and fully managed by Google. Selecting the App Engine service requires selecting mandatory features that refer to the desired platform, the virtual machine type on top of which the platform is hosted, and the strategy for provisioning new virtual machine instances and releasing them. The latter can either be carried out manually by the cloud user or managed by the cloud environment in the sense that virtual machine instances are automatically provisioned or released based on certain thresholds. They can be manipulated depending on the needs of the cloud user. For instance, the App Engine enables cloud users to set a limit for the maximum number of idle virtual machine instances to ensure that instances are not only scaled-out but also scaled-in. Regarding the virtual machines offered by the App Engine

two further aspects are emphasized by the feature model: the performance characteristics of the various offered virtual machines differ and the costs of them for the cloud user varies. These cross-cutting aspects are not only relevant for Google's cloud environment but also for others as highlighted in the feature model (see Figures 4.13 and 4.14) of the cloud environments operated by Amazon and Microsoft.

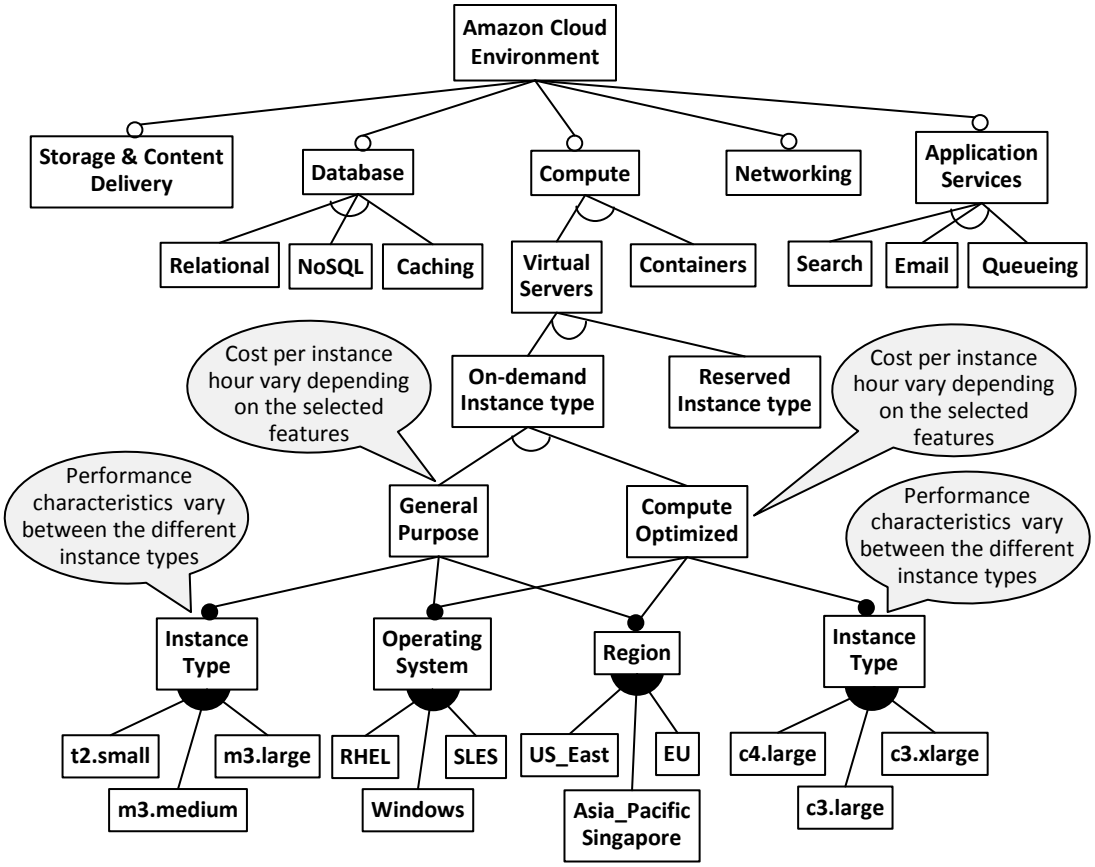

Figure 4.13: Feature model of the Amazon cloud environment

A variety of infrastructure-level compute services with different performance characteristics and costs are offered by Amazon's environment. Some features including the type of virtual machine, its operating system, and the region where it must reside are mandatory to select for the provisioning of a compute service. Even though the compute service of Microsoft is a mixture between infrastructure-level and platform-level offering, it requires similar to the Amazon's compute service the selection of the virtual machine type and its location. In summary, the gathered feature models of the three cloud environments point out the multitude of different cloud services which are currently offered by them. At the same time, several similar categories of cloud services are shared by all of them, *e. g.,* compute service, storage service, and application service. In case of the latter, Google refers to this category simply as "services" and Microsoft classifies their services according to more specific development areas, such as "web & mobile"

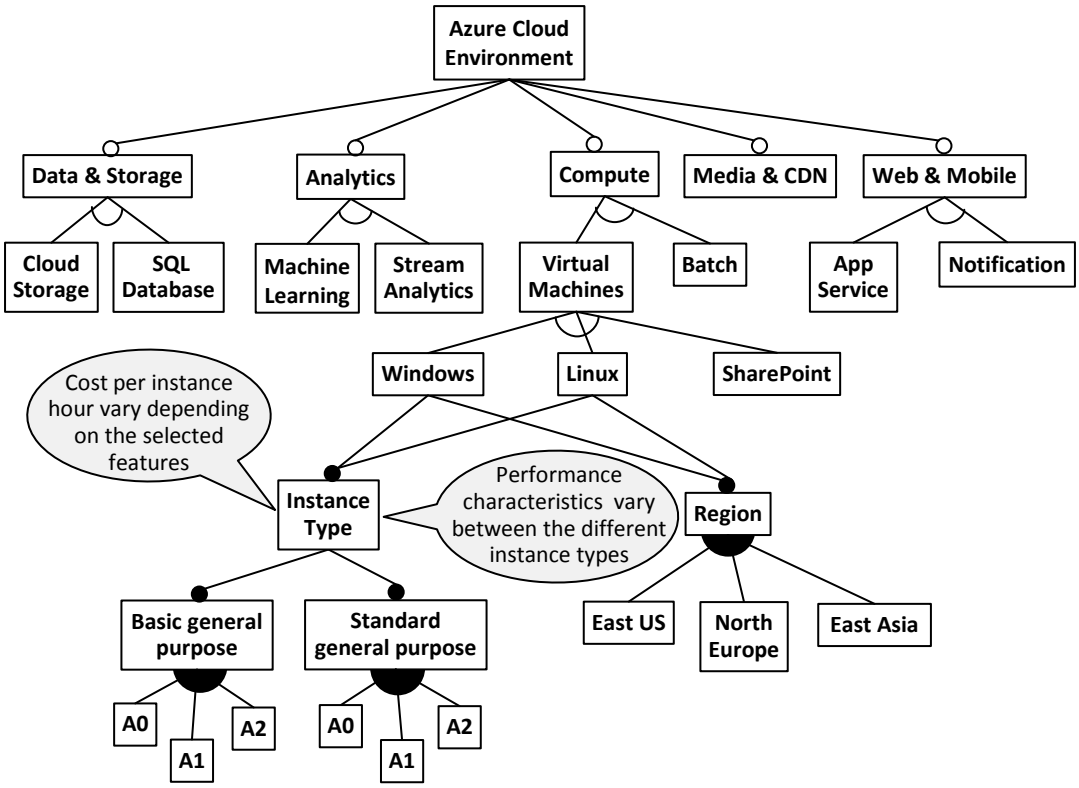

Figure 4.14: Feature model of the Microsoft cloud environment

and "media & CDN". However, given the fact that all these services are applications delivered as services [AFG+09], the term "application service" appears adequate to provide at least a common category for them.

4.4.2 Developing UML-based extensions: From problem space to solution space

The feature models gathered from the three selected cloud environments provide explicit representations of the domain knowledge required for the realization of CAML's cloud library and profiles. How can the abstractions over the cloud environments be mapped into UML-based extensions? Some guidelines [TMC99, KKP+09] and patterns [MHS05] exist how to accomplish this mapping for developing a domain-specific language in general. For instance, Mernik *et al.* [MHS05] demonstrate the FODA approach by means of a concrete language and points out that the variabilities of modeled features indicate the information required to create instances of language concepts. In recent work, mappings for the unification of feature and class modeling have been proposed [BDA+14]. They are of particular relevance for accomplishing the transition of the captured domain abstractions from the problem space to the implementation space [CE00, AK09] because the key meta-classes of UML's profile and deployment language, *e. g.,* `stereotype` and `node`, respectively, are grounded in the classifier and relationship meta-classes. However, instead

of seeking for a unification, we aim at developing UML-based extensions from gathered feature models as depicted in Figure 4.15.

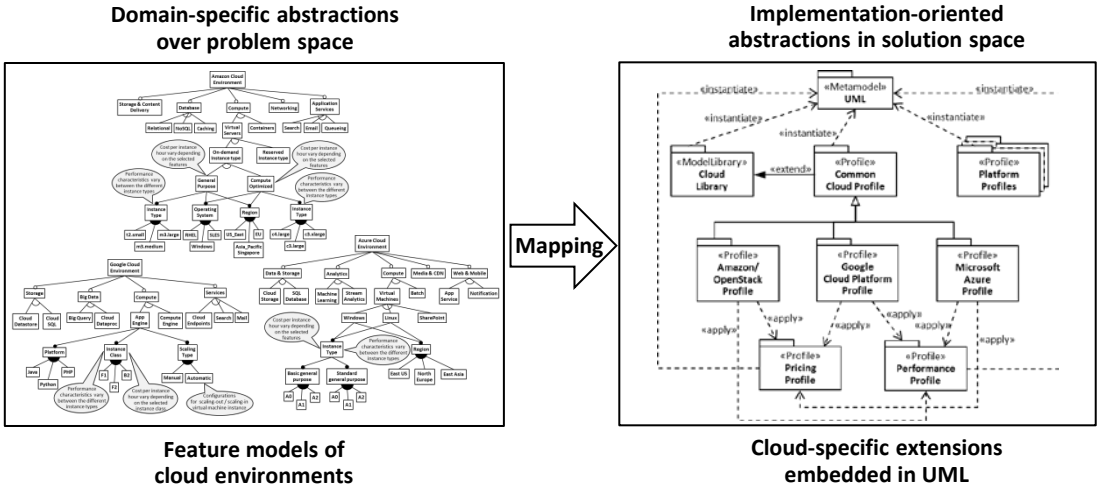

Figure 4.15: Developing UML-based extensions from gathered feature models (based on [CE00, AK09])

In essence, high-level categories of cloud services, *e. g.,* compute and storage, are potential candidates to be captured by CAML's cloud library because they are common to the analyzed cloud environments and hence independent of them. In the feature models, these categories are at the upper level close to the root concept. On the other hand, cloud services at the lower level of the feature models tend to be specific to the cloud environments. They are useful in the refinement of environment-independent deployment topologies towards the target cloud environment and hence composed into environment-specific profiles.

To make this mapping task more concrete, Figure 4.16 gives some examples of how UML-based extensions have been developed from the feature model referring to Google's cloud environment. The features referring to storage and compute services are mapped to modeling concepts of the cloud library. This design decision is a result of seeking for environment-independent modeling concepts that belong to all three analyzed cloud environments. In contrast to this cross environment decision, the profile corresponding to the Google's cloud environment has been developed independent of other environment-specific profiles. Obviously, the root concept of the feature model denotes the profile itself. Considering the feature referring to the App Engine and the features underneath it, they are mapped to either a `Stereotype` or a `Property` of type `Enumeration` covering selectable features in terms of `EnumerationLiterals`. As the *App Engine* feature is composed of other features that in turn are composed of other features, it is mapped to a `stereotype`. Otherwise, it would not be possible to relate properties or stereotypes to it. Selectable platforms are mapped to `EnumerationLiterals`. They are covered by the `Enumeration` which is assigned to the `Property` derived from the platform feature of the App Engine. The multiplicity of the `Property` is 1 because selecting a platform is required. In contrast to this "feature-to-property" approach for features that are composed of leaves in the feature

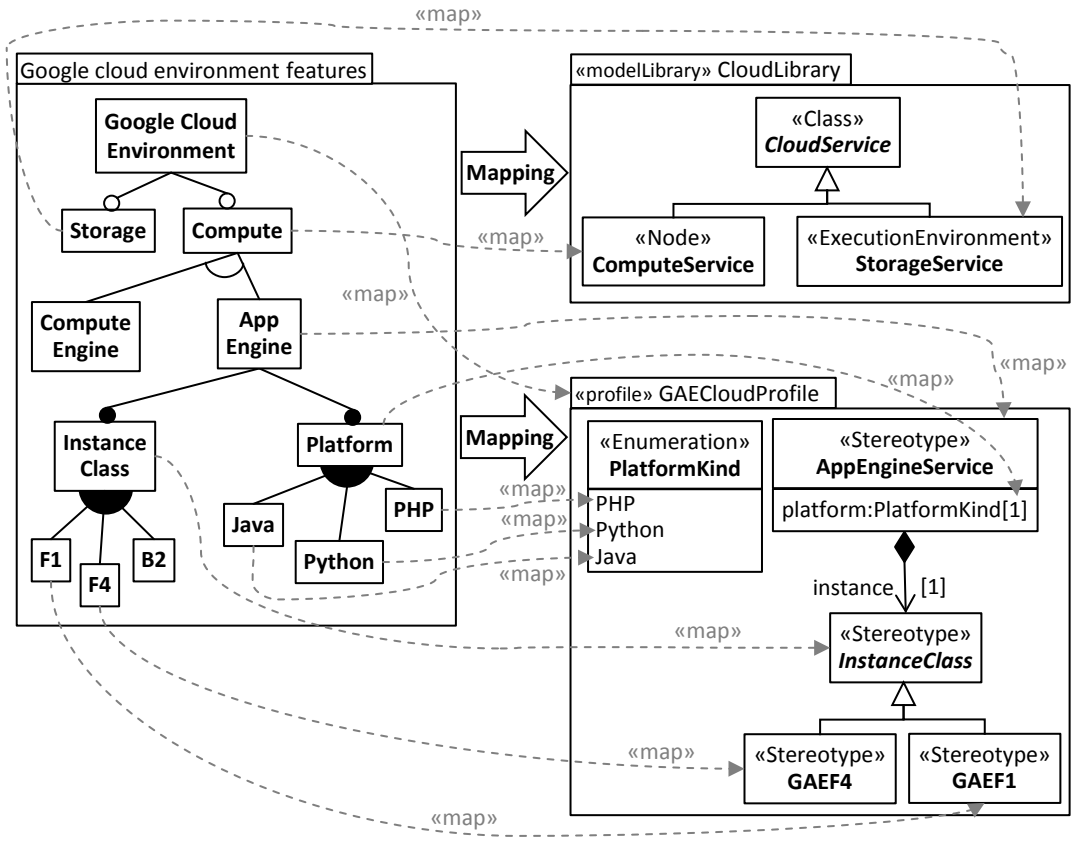

Figure 4.16: UML-based extensions from the feature model referring to Google's cloud environment

model, there are cases in which a `Stereotype` is preferable over a `Property`. Considering the virtual machine instance classes offered by the App Engine, they are mapped to stereotypes. This design decision is grounded in the fact that all the selectable virtual machine instances have different performance characteristics and operational costs which need to be captured separately and associated to the corresponding modeling concepts. As a result, it appears natural to map them to a first-class entity, *i. e.,* a `Stereotype` in the context of a profile.

4.4.3 Model library for cloud deployment topologies

The *cloud library* of CAML aims at enabling engineers to create deployment topologies independent of a target cloud environment by common cloud modeling concepts. As presented in Figure 4.17, the CAML's cloud library is built around the concept of *cloud service*. From a cloud environment perspective, a cloud service is considered as a virtual resource that is expected to be supported by a cloud environment. Cloud users can consume them over the network usually based on a pay-per-use cost model. Often the cost model includes free quotas for cloud services.

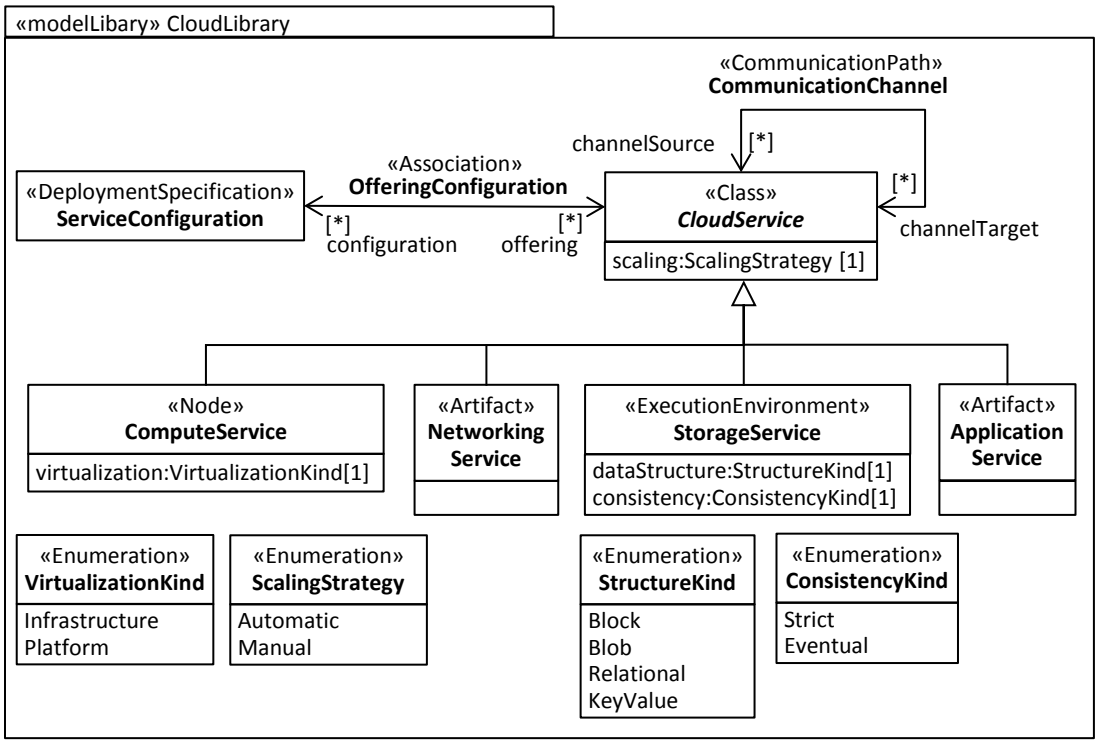

Figure 4.17: Cloud library of CAML

Once those quotas are exceeded the cloud service is charged based on measurable units, *e. g.,* service instance hours, amount of network traffic over the service, or number of operation calls against it. Modeled cloud service instances are intended to be refined towards concrete services offered by cloud environment. This refinement step is achieved by applying environment-specific stereotypes to cloud service instances. Considering the deployment topology in Figure 4.2, the modeled cloud service instances are refined towards concrete services offered by Google's App Engine.

As cloud environments are inherently elastic in the sense that provisioned resources can be scaled on-demand, *scalability strategies* are usually defined to control the provisioning of new resources and their release. The provisioning and releasing of cloud services is either manually controlled by the cloud user or automatically carried out by the target cloud environment. For instance, in case of the latter, a possible scalability strategy defines that a cloud service must be provisioned if the number of queued requests exceeds a certain threshold or released if the number of idle service instances reaches a defined limit. For that reason, CAML's cloud library allows engineers to associate a *service configuration* to a cloud service. A configuration is intended to be specified in the course of the refinement step. Similar to the refinement of cloud service instances, configurations associated to services are refined via dedicated stereotypes covered by environment-specific profiles. Currently, the cloud library does not provide configuration-related abstractions over existing cloud services, which explains why possible configurations features are solely provided by the applied stereotypes. Considering the cloud configuration in Figure 4.2, a

maximum number of idle cloud service instances is defined to ensure that instances are not only scaled-out but also scaled-in.

Based on the analysis of the service categories of existing cloud environments, the cloud library introduces four specific service types. A *compute service* provides computational capacity and operates at a certain level of *virtualization* [BGPCV12]. From an infrastructure-level perspective, a compute service comes along with an operating system, while when turning this perspective to the platform-level it also provides a platform required to execute a cloud application. For instance, a Java platform is required to execute a certain cloud application. Hence, Google's App Engine is categorized as a platform-level compute service. A NETWORKING SERVICE enables engineers to model logical network information over which a cloud environment grants control. For instance, a load balancer that distributes requests to compute services is a concrete networking service. The *storage service* refers to the diverse data persistence solutions provided by todays cloud environments. They are categorized according to how application data must be structured [FLR+14] and the possibility to increase their availability by relaxing consistency [Vog09]. Finally, an *application service* is considered as ready-to-use software components or libraries provisioned and fully managed by a service provider [CC06]. For instance, Google's "cloud endpoints service" and the "translate API" fall into this service type.

4.4.4 Profiles for refining cloud deployment topologies

The set of profiles provided by CAML aims mainly at refining environment-independent deployment topologies modeled by the cloud library towards a target cloud environment. Stereotypes covered by environment-specific profiles embody the concrete cloud services on the model level and capture their features in terms of properties. Hence, the refinement step is driven by applying stereotypes to the cloud services constituting the deployment topology and assigning values to the properties offered by those stereotypes. Considering the deployment topology in Figure 4.2, the selected platform of the modeled App Engine compute services refer to Java which is one platform out of several provided ones. The respective property is intended to be manipulated by the engineer in the course of the refinement step.

Figure 4.18 gives an overview of some stereotypes covered by the profiles for Google's App Engine service (GAE) and Amazon Web Services (AWS). The two presented "front end" virtual machine instance classes of the App Engine service as well as the "medium" and "large" offerings by the Amazon environment are concrete environment-specific realizations of a compute service. To ensure that they can be applied to deployment topologies, the *common cloud profile* establishes the link between the cloud library and the cloud environment profiles. From a technical perspective, it is important to note that the extension relationship between the cloud library's `ComputeService` modeling concept and the corresponding `ComputeService` stereotype of the common cloud profile is treated as explained in Figure 4.4. Hence, the abstract `ComputeService` stereotype hides not only the technical details for extending custom types in UML but also acts as the entry point for introducing concrete compute services offered by cloud environments. Newly introduced compute services certainly have different performance characteristics and are typically offered at a price that varies depending on those characteristics. Obviously, this technical and non-technical information over the compute services is directly associated to them and not intended to be manipulated by engineers at the time a deployment topology is modeled. Instead,

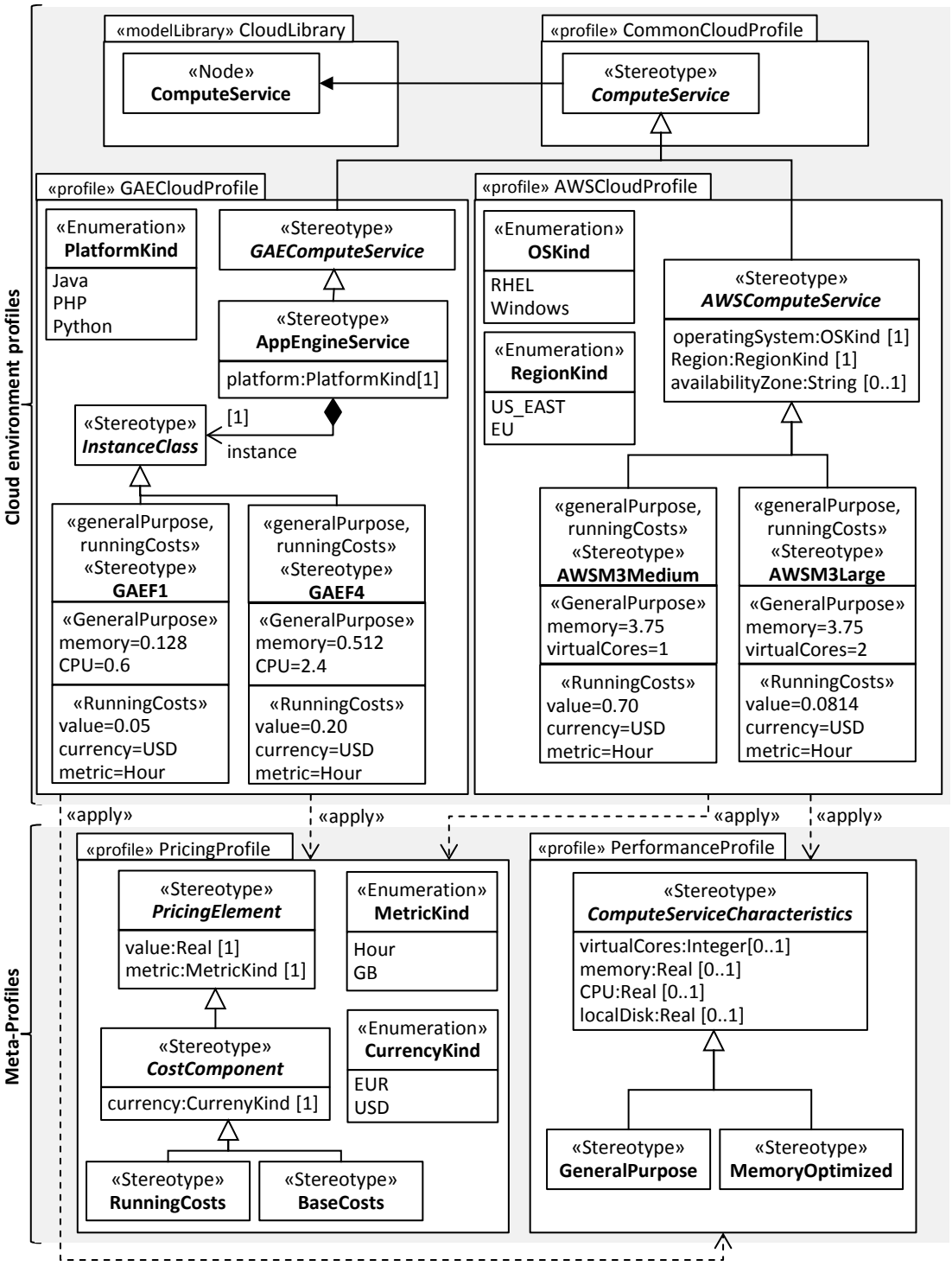

Figure 4.18: CAML profiles and meta-profiles

this additional meta-information is part of the compute services. It allows engineers to compare them. For instance, their comparison is required for the cloud service selection process.

Considering the excerpt of the *pricing profile* (see Figure B.1 of Appendix B for details about defined extensions), the `CostComponent` stereotype provides properties to capture the operational costs of cloud services. Costs may rely on various components, especially `BaseCosts` and `RunningCosts`. For instance, some of the compute services offered by Amazon have up-front costs as well as hourly costs when those services are consumed. The costs may vary depending on the performance characteristics of cloud services. Capturing those characteristics is supported by the *performance profile* (see Figure B.2 of Appendix B for details about defined extensions). It is mainly build around the `ComputeCharacteristcs` stereotype which provides a variety of properties to represent, for instance, the number and speed of virtual cores, and the amount of memory and local disk space. Several concrete stereotypes, *i. e.,* `GeneralPurpose` and `MemoryOptimized` inherit from this abstract stereotype mainly for the purpose of introducing a categorization of existing compute services. For instance memory optimized compute services have a relatively high amount of memory compared to general purposed ones. All the stereotypes covered by the pricing and performance profile are intended to be applied to cloud services of the cloud environment profiles. Hence, they extend the meta-class `Stereotype` of the UML metamodel. As the profiles are solely applied to other profiles we call them *meta-profiles*.

4.4.5 Reusable deployment topologies as UML templates

As templates in UML promote reuse, they appear to be relevant also for capturing common deployment topologies. In particular, UML's reuse mechanism can be exploited for providing frequently occurring deployment patterns as predefined UML templates. For instance, Amazon provides architectural guidelines[4] in order to build deployment topologies for cloud applications that take the full advantage of its cloud environment. If Amazon's cloud environment is selected as the target for an application deployment, these guidelines appear to be a relevant source. Ideally, the proposed topologies by Amazon can be represented by CAML and provided in terms of reusable deployment templates.

To demonstrate how CAML can be used to model a deployment template, we select the "web application hosting" scenario of Amazon's reference architecture collection. Furthermore, we assume a 2-tier Java-based web application architecture [FLR+14] in order to specify how the application components manifested by deployable artifacts are distributed in the deployment topology. In fact, the deployable artifacts and the platform required for their execution are considered as the formal template parameters. They must be substituted when the template is used. The parameter substitution is specified in terms of a binding. Figure 4.19 gives an overview of the modeled deployment topology constituting the template. It consists of two compute services that both refer to the `AWSM3Medium` virtual machine offering of the Amazon environment. They must be located in Europe and operated by a Linux operation system. For reliability reasons, they are placed in different availability zones. Requests that arrive at the compute services are first handled by a load balancing service, which enables a higher fault tolerance of the web application. The number of running compute services is automatically managed by the Amazon environment

[4] Amazon Architecture Center: https://aws.amazon.com/architecture

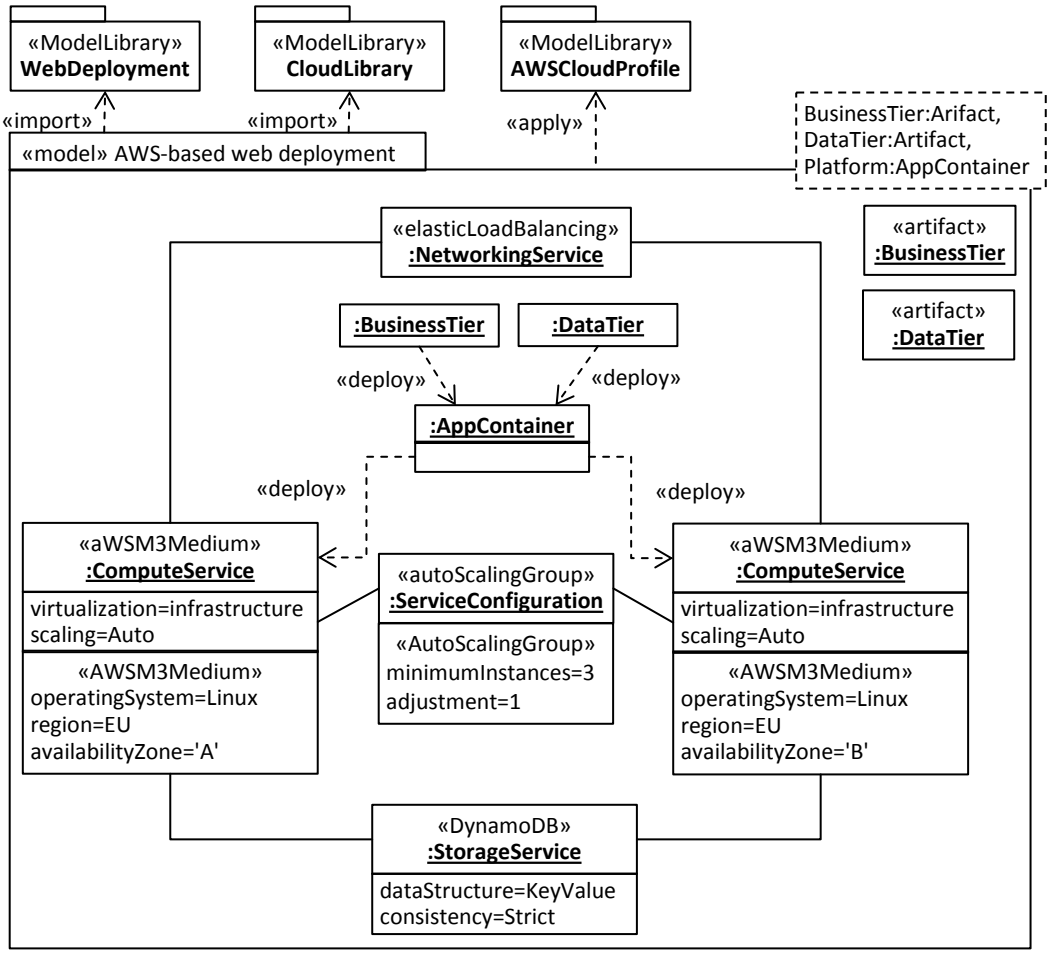

Figure 4.19: Reusable deployment template for AWS web deployment

as expressed by the scalability strategy directly associated to the compute services. Only the minimum number of running compute services and their adjustment is configured. Both compute services are connected to a storage service that in turn is replicated to improve data availability. As Amazon's compute services operate at the infrastructure-level, the platform required to execute the deployable artifacts is defined. This is in contrast to platform-level compute services, such as Google's App Engine, which provide already a fully managed platform. In case of our deployment template, the platform is embodied by a Java-based web container. A variety of different web containers is provided by the *web deployment library*. The deployable artifacts as well as the modeled platform are exposed by the deployment template in terms of formal parameters that must be substituted in terms of a template binding.

Now that we have explained how deployment templates can be produced with CAML, Figure 4.20 demonstrates a concrete binding. We take the PetApp and manifest its components into two deployable artifacts: `PetData` and `PetBusiness`. It is important to note that the `Pet-`

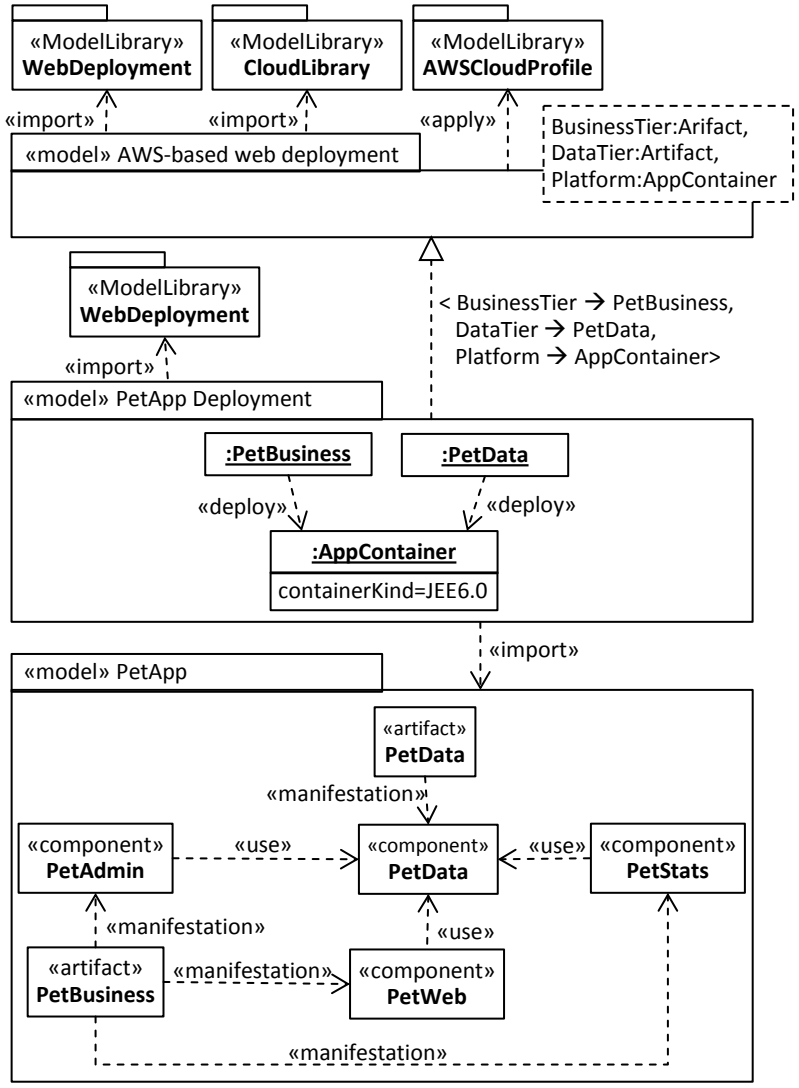

Figure 4.20: PetApp deployed based on AWS web deployment template

`Data` artifact must be implemented in a way that the persistable entities can be stored in Amazon's DynamoDB. Considering the annotation-based refinement in Figure 4.1, the DynamoDB profile must be applied instead of the Objectify profile to ensure that application code specific to Amazon's datastore solution is generated. Interestingly, in case of this scenario, the selected deployment template affects the refinement of the high-level domain model towards the platform that is supported by the target cloud environment. This further emphasizes that deployment topologies are a vital part of (cloud) architecture modeling as also pointed out by Figure 1.1. Coming back to the template binding, an application container is required for enabling the execution of the PetApp's application code. In case of our scenario, an application container that supports version 6 of Java's enterprise edition is used. The required custom types are provided

by the imported web deployment library. In a final step, the binding needs to be specified to reuse the deployment topology captured by the template in Figure 4.19. Basically, all the formal parameters exposed by the template are substituted by the deployable artifacts and the platform modeled for the PetApp. As the deployment topology of the template is already composed of cloud services of the Amazon environment, not only the reuse of cloud environment profiles is further promoted but also the refinement from an environment-independent topology towards a target cloud environment is completely reused. The latter is carried out only once in the course of creating the template.

4.5 Summary

Developing cloud-specific extensions to UML is hard in particular because cloud computing is a highly diverse domain that constantly progresses. At the same time, such extensions are just as indispensable as are libraries for programming languages since they provide core concepts of the target platform and environment. Due to the generic nature of UML, it does however neither provide cloud-specific modeling concepts by default nor guidance how the existing modeling concepts can be used to represent cloud application architectures.

In this chapter, we introduced CAML to support architecture modeling of cloud applications in UML. It supports a flexible refinement process from high-level PIMs over possibly several PSMs down to a concrete implementation of them. The accompanied tools render it useful in practice. CAML consists of a collection of UML profiles dedicated to libraries of the Java platform. Considering the large number of Java libraries, we presented a fully automatic transformation chain to generate UML profiles from Java libraries that embrace annotations. In addition to library-specific profiles, CAML provides a set of profiles that capture cloud services offered by modern cloud environments. They enable the refinement of environment-independent deployment configurations towards a selected cloud environment. The former is represented by means of CAML's cloud library. It provides common cloud modeling types which are abstractions over services of the current major cloud environments, *i. e.,* Amazon AWS, Google Cloud Platform, and Microsoft Azure. Combining the library approach with the notion of environment-specific UML profiles results in a powerful architecture style for cloud application modeling.

UML-based language for cloud application modeling. CAML extends UML in a standard compliant way, such that current modeling tools that support UML are capable to adopt its cloud-specific extensions. As a result, all the building blocks constituting CAML are instances of the UML metamodel. Since CAML exploits both libraries and profiles to contribute cloud-specific extension to UML, we provided insights into how custom UML types captured by a library on the model level can be extended via stereotypes even though UML's profile mechanism enables the extension of meta-classes only. For that reason, we introduced a pattern for extending custom types in UML via stereotypes. It allows engineers to develop concise UML profile definitions for custom UML types without requiring changes to existing UML tools because the provided generative approach ensures that extension relationships to those types are appropriately redefined if necessary.

Extensions to UML for target platforms in the cloud. UML profiles can be exploited in terms of a general injection mechanism. Their capabilities to capture platform-specific information are an enabler for both reverse engineering and forward engineering processes in architecture modeling. CAML provides a conceptual mapping between the Java platform and UML with a particular emphasis on annotations and profiles. For that reason, we present a conceptual mapping between Java's annotation language and UML's profile language as a basis to automate the generation of UML profiles from annotation-based Java libraries used in general and in the context of cloud application development in particular, *e. g.,*. Amazon's DynamoDB library and Objectify for the Google's App Engine service. Since Java's annotation mechanism has been improved with the release of version 8, we also discussed several different solutions including their pros and cons how the newly introduced repeating annotations can be supported in UML.

Extensions to UML for target cloud environments. CAML provides cloud-specific extensions to UML's deployment modeling language for modeling deployment configurations and refining them towards a target cloud environment. In order to gain insights into the cloud service landscape and identify useful abstractions over the offered services, we carried out a feature-oriented domain analysis. The results obtained from this analysis are captured in form of feature models from which we distilled the modeling concepts comprised by CAML. Clearly, these modeling concepts are also inspired from existing CMLs including the relatively recently adopted TOSCA standard. As CAML provides both environment-independent as well as environment-dependent cloud modeling concepts, it is not only capable of creating deployment configurations but also closing the gap between those configurations on the modeling level and the concrete services offered by the selected target cloud environment. To promote the reuse of deployment configurations, CAML advocates UML's template mechanism. Templates are aimed to capture deployment configurations that are already refined towards a cloud environment. Thus, not only the reuse of profiles dedicated to cloud environments is further promoted but also the refinement process must only be carried out once in the course of creating the deployment template.

In the evaluation of CAML, we investigate the quality of automatically generated UML profiles from Java libraries compared to existing UML profiles used in practice. In general, automatically generated UML profiles comprise a more comprehensive set of stereotypes and features compared to currently existing profiles that support Java libraries. Moreover, we demonstrate that CAML's transformation chain to generate UML profiles scales for large libraries and applications. Finally, we show the practical relevance of CAML by means of a modernization scenario to the cloud. Further details on CAML's evaluation are presented in Sections 7.2 to 7.4.

4.6 Related work

To differentiate CAML from existing approaches, we first discuss related work in the area of bridging modeling and programming languages with a particular emphasis on UML and Java. Thereafter, we compare CAML to existing CMLs and categorize it according to the classification and comparison framework used to review these CMLs.

4.6.1 UML profile generation

With respect to the contribution of this paper, namely to generate UML profiles from Java libraries, we consider three threads of related work. First, we discuss mappings between Java and UML because CAML builds on existing efforts in this respect. Thereafter, generative approaches dealing with UML profiles and Java Annotation Types are discussed. Finally, we consider approaches that support metamodel generation from programming libraries.

Mapping Java and UML

The elaboration on the mapping between Java and UML has a long tradition in software engineering research [EHSW99, HBR00, NNZ00, KSS+02]. Round-trip engineering for UML and Java has been extensively studied in the context of the development of FUJABA [NNZ00]. One particular concept of UML that received much attention in the context of Java code generation is the association concept [Ges08, AHMM07, GdCL03]. However, none of these mentioned approaches consider the transformation of annotation types and their applications from Java to UML. The only exception is the mTurnpike approach [WS05] that considers Java annotations on the model level. Thereby, round-trip transformations between UML models and Java code are realized by considering stereotypes and annotations in the transformations. In contrast, CAML sets the focus on the automated generation of UML profiles that facilitate round-trip transformations or transformations in general. Besides academic efforts, today's modeling tools support the transformation of Java code to UML models, and vice versa. Their current capabilities and limitations with respect to CAML are discussed in Section 7.1.

Generating UML profiles and Java annotation types

The only approaches we are aware of that deal with automated generation of profiles fall into the research area concerned with bridging the gap between MOF-based metamodels and UML's profile mechanism, which is also related to the discussion of an external domain-specific modeling languages (DSML) compared to an internal DSML where the host language is UML [Fow10]. Considering the latter, they are internal in the sense that they are embedded in a host language [MHS05] providing the base elements for which extensions and constraints are developed. In contrast, external DSMLs are built from scratch and have their own custom concepts without explicit relationships to any existing language. Mernik *et al.* [MHS05] discuss when and how to develop internal and external domain-specific (modeling) languages. Several papers discuss the pros and cons of these approaches, *e. g.,* [Sel12] and their combination, *e. g.,* [WS07].

Visualizing domain-specific models in UML with profiles is discussed in [GvD07]. Abouzahra *et al.* [ABFJ05] present an approach for interoperability of UML models and DSML models based on mappings between the DSML metamodel and the UML profile. Brucker and Doser [BD07] go one step further and propose an approach for extending a DSML metamodel for deriving model transformations able to transform DSML models into UML models that are automatically annotated with stereotypes. A related approach is presented by Wimmer [Wim09], where mappings between the UML metamodel and a DSML metamodel are defined and processed to generate UML profiles for the given DSMLs.

Considering the generation of Java annotation types from DSML models, Ann [CdL15] is a recent approach for modeling Java annotation types. It provides code generation facilities to produce the Java code of modeled annotation types as well as respective annotation processors that implement validation rules for annotations applied to program element. For instance, the `Entity` annotation type of the JPA requires that one or several attributes of the annotated Java class define the primary key. One possibility to define it is to apply the `Id` annotation type to an attribute of the respective Java class. Validating invariants can also be achieved for UML models by associating OCL constraints with stereotypes. In this case, the validation would be carried out before the Java code is actually generated from the UML model.

Generating metamodels

To the best of our knowledge, there is only one automated approach for generating modeling languages from programming libraries. All other automated approaches that deal with exploring libraries, such as the approach of Bruneliere *et al.* [BCJM10], set their focus on the generation of domain models rather than a language.

API2MoL [CJCM12] deals with generating metamodels based on Ecore from Java APIs as well as models conforming to the generated metamodels for Java objects instantiated from the Java APIs, and vice versa. As a result, an external DSML is generated from a Java API. While the general idea and motivation of the API2MoL approach is comparable to CAML, there is a significant difference on how the DSML is realized. CAML targets UML modelers that are familiar with UML class diagrams and generates internal DSMLs by exploiting the language-inherent extension mechanism of UML, *i. e., UML Profiles*. Furthermore, annotations are not explicitly considered in the metamodel generation process of API2MoL. One possible reason for neglecting them is that standard versions of current meta-modeling languages, such as Ecore, do not support language-inherent extension mechanisms out-of-the-box [LWWC12].

Antkiewicz *et al.* [ACS09] present a methodology for creating framework-specific modeling languages. While we aim for an automated approach, Antkiewicz *et al.* use a manual one to create the metamodel and the transformations between model instances and instantiated objects of the frameworks. Again, annotations are not captured by the created languages.

Finally, Noguera *et al.* [ND08] propose the extraction of annotation models in terms of class diagrams from a set of annotation types with the purpose to define validation constraints for the consistent use of annotations. They mention the study of the relationship of stereotypes and annotation types as interesting subject for future work, only.

Research of related fields consider ontologies as a kind of (meta-)model [GDD09]. In particular, research on ontology extraction from different artifacts is subsumed under ontology learning [DG08]. We are aware of only one approach for extracting ontologies from APIs [RFJ08]. It neglects however annotations. Furthermore, most of current ontology learning approaches focus on the extraction of concepts and their taxonomic relationships. Finding non-taxonomic relationships (*e. g.,* associations between classes) and intrinsic attributes are the least considered problems [KMS04] in this field.

Synopsis

To summarize, CAML is – to the best of our knowledge – the first approach to generate standard-compliant UML profiles from Java libraries that exploit annotations. While other existing approaches are capable of producing (meta-)models from Java code, the annotation concept has not received much attention. This is however in contradiction with the frequent use and ever-growing importance of the annotation concept on the programming level. Therefore, support for annotations on the model level has to be provided. We applied an internal DSML approach by exploiting the language-inherent extension mechanism of UML. It perfectly suits the annotation mechanism of Java. As a result, we close an important gap between programming and modeling.

4.6.2 Cloud-specific extensions to UML

Recently, several CMLs that address the diversity of today's cloud environments and their services have emerged. They pursue different goals, provide complementary modeling concepts, and thus differ in scope. For that reason, we conducted a systematic literature review on existing CMLs (see Chapter 3). We reviewed them according to a classification and comparison framework (see Section 3.1) and collected our findings along with detailed descriptions (see Tables 3.7 to 3.11 for a concise overview of our results). Also, we have investigated existing CMLs including UML in the light of the PetApp modernization scenario and drew up those phases of the modernization process in which standard UML requires extensions to support cloud application modeling.

Inspired from common cloud computing literature [AFG+10, BGPCV12, FLR+14], major cloud environments, and existing CMLs (see Table 3.6), we have developed CAML's cloud library and profile on top of UML. Modeling concepts of existing CMLs are thus reflected by CAML's on a level of abstraction that supports engineers in the design, deployment and provisioning of cloud applications. As a result, modeling concepts required to, *e. g.,* achieve the optimization of an application deployment (*cf. e. g.,* [LFM+11, FH11, ARSL14]), express elasticity rules (*cf. e. g.,* [CEM+12]), or exploit workload models (*cf. e. g.,* [FRC+13, HT15]) are not completely captured by CAML's. However, it enables deployment models to be specified in such a way that they are seamlessly applicable on UML models usually created throughout application modeling activities, *e. g.,* class models to specify the realization of components, because CAML is based on UML. Consequently, well-connected model-based views on cloud applications are supported. Those views can be refined towards a selected target platform and environment by CAML's profiles, which can be dynamically applied and un-applied. This additional flexible typing dimension and the benefits of a multi-viewpoint language exploited by CAML differentiates it from existing CMLs.

Cloud modeling support based on UML is also proposed by MULTICLAPP [GMMC13b], which presents a UML profile for the purpose of representing applications components that are expected to be deployed on a cloud environment. While MULTICLAPP solely relies on UML's profile mechanism to annotate components that are expected to be deployed onto a cloud environment, CAML exploits both the library concept of UML as well as profiles. The modeling concepts of CAML's cloud library are generic in the sense that they are independent of a cloud environment, whereas profiles are used to capture services offered by cloud environments. This is in contrast to MULTICLAPP where generic cloud modeling concepts are captured by means

Language scope

Pragmatics	Target
Cloud application architecture description and refinement of deployment configurations towards target cloud environment	XaaS

Language characteristics

Syntax			Semantics	Realization	Typing
Abstract	Concrete	Serialization			
UML	graphical	XMI	translational	internal UML	linguistic ontological

Modeling concerns

Application structure	Cloud environment services	Elasticity	Service level
Class Component Deployment	ontological types	Multiplicity	✗

Component and deployment viewpoint

Component		Deployment			
Component	Connector	Service	Link	Networking	Artifact
Component[C]	Prov./Req. Interfaces	CloudService[X]	Communication- Channel	Addressing	Artifact

Tool support

Modeling	Analysis	Refinement	Generation	Provisioning
Arbitrary UML model editor	✗	enrichment by UML profiles	m2t: UML-Java t2m: Java-UML m2m: CAML-TOSCA	declarative[T]

[C] Component is composable
[X] Services of all three layers can be modeled
[T] Based on OpenTOSCA as CAML provides mapping to TOSCA (see Chapter 5)

Table 4.4: CAML categorized according to the CML comparison and classification framework (see Figure 3.1)

of a profile. As MULTICLAPP's profile does not support to refine application components towards cloud services provided by a certain cloud environment, CAML is different in the sense that its cloud profile is applied to achieve exactly this platform and environment-specific refinement. In this respect, MULTICLAPP advocates feature models for capturing existing cloud services. Both CAML and MULTICLAPP provide a mapping to Java for the purpose of generating application code from models. In addition, CAML also provides a mapping to TOSCA for reasons of exploiting existing TOSCA runtime containers (*e. g.,* OpenTOSCA) that enable automatic provisioning of cloud applications and services. Table 4.4 categorizes CAML according to the classification and comparison framework used to review existing CMLs.

CHAPTER 5

Cloud application provisioning

The adopted TOSCA specification [OAS13a, BBKL14a] aims at standardizing the representation of portable cloud applications mainly for automating their provisioning and supporting life-cycle management. Cloud applications are described in terms of a deployment model that captures all the deployment artifacts and management operations (*e. g.,* "install", "start", or "shut down") required for enacting the provisioning of a cloud application on the defined deployment targets. Portability of cloud applications is accomplished by providing a standard container format to represent cloud applications and a binding mechanism for associating target-specific implementations to management operations. They are invoked in the course of a provisioning process that is described by a deployment plan. It orchestrates management operations exposed by the deployment model.

At the same time, UML supports architecture modeling from different viewpoints, including the class, component, and deployment viewpoint. As the deployment viewpoint is provided by both modeling standards, they seem to share commonalities for representing deployment artifacts and their targets. Hence, combining them appears to be beneficial from the perspective of both UML and TOSCA because they are also diverse in the sense of the modeling viewpoints and toolset that are provided for them. Moreover, TOSCA can be considered as a cloud modeling language, whereas UML is a general purpose modeling language that is applicable to a variety of domains and hence used in the broader context of architecture modeling [MLM+13, LMM+15] and software modeling [FL08, HWRK11, TTR+11].

Combining UML and TOSCA would allow engineers to exploit the full expressive power of UML for modeling applications along with the capabilities of TOSCA to enact their provisioning on a certain target cloud environment and possibly port them between cloud environments. From a UML perspective this is beneficial as it allows the application provisioning for UML deployment models by means of TOSCA compliant runtime container such as OpenTOSCA [BBH+13]. On the other hand, TOSCA deployment models can be considered in the light of UML, thereby gaining insights into the application components manifested by deployment artifacts and how they are realized from a structural viewpoint.

However, manually translating between UML and TOSCA is only achievable if engineers are familiar with the peculiarities of both languages and capable to identify the correspondences between them at both levels intensional and extensional [Küh06]. While at the intensional level common aspects of cloud environments are captured in terms of types, they are instantiated at the extensional level by assigning concrete values to their features. Moreover, due to the generic nature of UML's deployment language, it does not natively support deployment models for cloud applications, which hampers the translation between the two languages.

To support an automated translation from UML to TOSCA, we present a conceptual mapping as a basis for realizing an effective model transformer between them. The deployment viewpoint is selected to accomplish the conceptual mapping, as it is shared by both UML and TOSCA. Moreover, the cloud-specific extensions to UML provided by CAML are addressed in the mapping process. They enable the generation of meaningful TOSCA models from models represented in UML as the cloud-specific features are injected via CAML's cloud library and profile. As CAML captures cloud-specific features in terms of custom (stereo)types, both the intensional and extensional level of deployment models can be addressed in the mapping process. In this sense, CAML can also be considered as a bridge between UML and TOSCA, where our model transformer – CAML2TOSCA – provides the glue between architecture modeling and application provisioning [JAP13]. The main characteristics of CAML2TOSCA are summarized in the following.

Generic in the sense that any UML model can be translated into a TOSCA model. The conceptual mapping between UML and TOSCA is defined between their metamodels, which enables the translation of any deployment model expressed in UML into a corresponding TOSCA model. As CAML provides standard compliant extensions to UML in terms of custom types, they can be considered as supplementaries to UML's metamodel. In this way, they can be treated in the mapping process similar to UML standard meta-classes.

Comprehensive as both levels intensional and extensional are addressed. As CAML captures abstractions over cloud environments and services offered by them in terms of custom types, *i. e.,* at the intensional level, they must also be declared in the context of TOSCA to ensure that the type information is preserved throughout the translation of concrete deployment models. For that purpose, the conceptual mapping covers correspondences between language concepts not only to deal with the extensional level but also with the intensional level. The latter is achievable as both UML and TOSCA can be extended via custom types.

Automated as the mapping is realized as model-to-model transformation. The model transformation language ATL has been used to realize the model transformer, which comes as an Eclipse plug-in. It is capable to produce a standard compliant TOSCA model serialized in XML from a UML deployment model refined by CAML extensions.

The remainder of this chapter is structured as follows. In Section 5.1, we motivate the practical value of CAML2TOSCA by means of the application scenario introduced in Section 1.4. A brief introduction to TOSCA with a particular emphasis on its metamodel is presented in Section 5.2. In Section 5.3 we clarify how the intensional and extensional level introduced by UML and

TOSCA relate to each other as this is essential for providing a useful mapping between them. The conceptual mapping from UML to TOSCA is presented in Section 5.4, whereas in Section 5.5 we show how CAML2TOSCA can be integrated into a comprehensive framework for application deployment and provisioning where the entry point is a high-level architecture model represented in UML. A summary of this chapter is given in Section 5.6 before work related to CAML2TOSCA is discussed in Section 5.7.

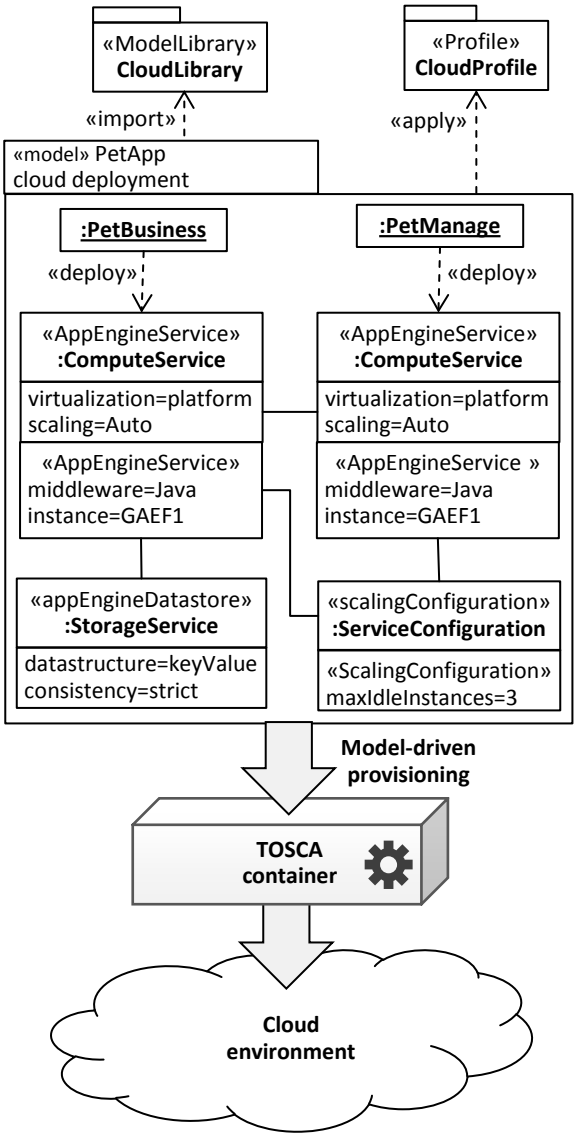

Figure 5.1: Direct use of a UML deployment model for cloud application provisioning

5.1 Motivation

To show the practical value of CAML2TOSCA, we consider again the application scenario as introduced in Section 1.4, where the main emphasis is placed on the application provisioning. Figure 5.1 shows the PetApp's deployment model refined towards the Google App Engine. The deployment model captures the two deployable artifacts of the PetApp and allocates them to platform-level compute services that offer a Java runtime environment. One of the compute services requires access to Google's cloud datastore for the purpose of persisting application data. Since the application data are exposed via a REST-based API, the `PetManage` artifact does not directly access the cloud datastore but instead invokes the respective service operations. The service configuration associated to the compute services specifies the upper limit of idle instances. It ensures that provisioned compute service instances must be released once the specified threshold is exceeded. Considering the topology defined by the deployment model, it captures the desired initial[1] state of the application provisioning. Ideally, the deployment model can directly be passed to a provisioning engine capable to enact the application provisioning. As TOSCA-compliant runtime containers offer those capabilities, it appears beneficial to exploit them for the provisioning of a UML deployment model refined towards a target cloud environment via CAML extensions. This is exactly the motivation behind a conceptual mapping between UML and TOSCA because it would allow the realization of a model transformer to automate the generation of the required TOSCA deployment model from the architectural models represented in UML. Moreover, it would allow engineers to exploit existing tools for UML as well as TOSCA without re-creating any of UML architectural models, thereby the investments in creating them is retained even though the application provisioning is carried out based on a TOSCA model.

5.2 TOSCA metamodel

The main concepts of the TOSCA metamodel relevant for the CAML2TOSCA approach are depicted in Figure 5.2. Some concepts are simplified for the sake of comprehension. The complete metamodel is covered by the TOSCA specification [OAS13a]. An introduction to it is given in the TOSCA primer [OAS13b], whereas Binz *et al.* [BBKL14a] provide a compact overview of TOSCA. Conceptually, TOSCA consists of two parts. The `TopologyTemplate` is used to describe the structure of cloud applications, whereas a management `Plan` is basically a workflow model that can be invoked to execute management tasks, *e. g.,* the provisioning of a compute service instance.

Considering the `TopologyTemplate` in more detail, it is a directed graph that consists of `NodeTemplates` and `RelationshipTemplates`. The former represent artifacts related to cloud applications and environments, while the latter define dependencies between those artifacts, *e. g.,* the PetBusiness artifact is deployed on an application server which in turn is deployed on a compute service. Both kinds of templates are typed. The type of a `NodeTemplate` is defined by a `NodeType`, similarly is a `RelationshipType` used to define the type of a `RelationshipTemplate`.

The respective types can be used to specify a `PropertiesDefinition` which covers all

[1] At run-time several instances of the modeled compute services may be provisioned on demand.

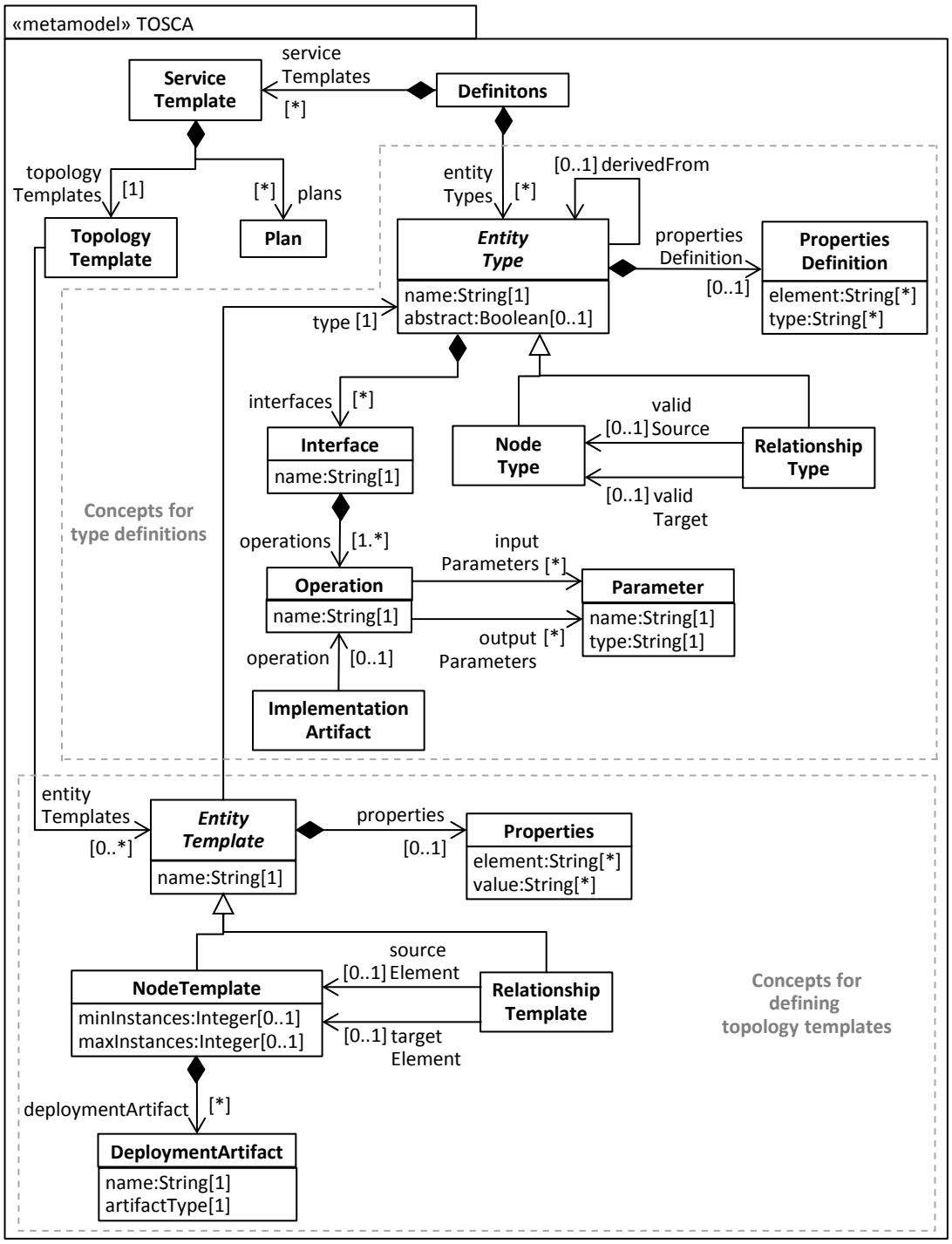

Figure 5.2: TOSCA metamodel (XML-based language [OAS13a])

the properties of a type. This explains the zero-to-one multiplicity between `EntityType` and `PropertiesDefinition`. For example, a compute service type may define two properties, "public_address" and "private_address" for internal and external communication. A template of this type enables the configuration of both addresses that are used to access the compute service instance from remote services and services within the cloud environment, respectively. A `PropertiesDefinition` specifies "element/type" pairs that embody the set of properties of a type. From a technical perspective, a `PropertiesDefinition` is realized in terms of an XML Schema. It allows the validation of the properties used by templates against their definition at type-level.

Furthermore, types in TOSCA can define management `Interfaces` including `Operations` with `Input` and `OutputParameters`. For example, a compute service type may provide two operations, "start" and "shut down". They can then be invoked via the templates of this type to start the compute service instance and shut it down. An `ImplementationArtifact` is used to provide a concrete implementation for an operation. In theory, any programming language can be used to implement an operation. From a technical perspective, a TOSCA runtime container must be capable of invoking the implementation of an operation.

Generally, types can inherit from each other, which fosters its reuse: compute services specific to a cloud environment, *e. g.,* an Amazon EC2 compute service, may inherit from a generic compute service type that provides common properties and operations.

Finally, a `DeploymentArtifact` refers to a concrete implementation of a `NodeTemplate`. For example, a Java-based web application front-end compressed in a certain format is considered as a deployment artifact. Also, binaries of an application container are `DeploymentArtifacts`. They are required to actually install the application container.

5.3 Intensional and extensional deployment modeling

Clarifying how the intensional and extensional level introduced by UML and TOSCA relate to each other is essential for combining them. Figure 5.3 depicts the core concepts of UML and TOSCA to create intensional and extensional deployment models. While in UML the various sub-meta-classes of `Classifier` are employed to model application architectures from an intensional perspective, the corresponding meta-class in TOSCA is `Type`. For instance, the compute service with the `region` property is considered as a concrete UML classifier or TOSCA type, respectively. At the extensional level, the meta-classes `InstanceSpecification` of UML and `Template` of TOSCA correspond to each other. Indeed, the stereotype applied to an instance specification needs to be taken into consideration to infer the corresponding type of a produced template because TOSCA does not directly support stereotypes. Still, a stereotype can be considered as a type in TOSCA, which inherits the properties of the base class extended by the stereotype. For instance, the extension between the "AWSM3Medium" stereotype and the compute service is represented as a generalization between the corresponding TOSCA types. This solution implies that a stereotyped artifact of an extensional deployment model needs to be translated into a TOSCA template that is typed by the type of the corresponding stereotype instead of its direct classifier. As a result, the stereotyped compute service instance at the extensional level is represented as a TOSCA template of type "AWSM3Medium" instead of *compute service*.

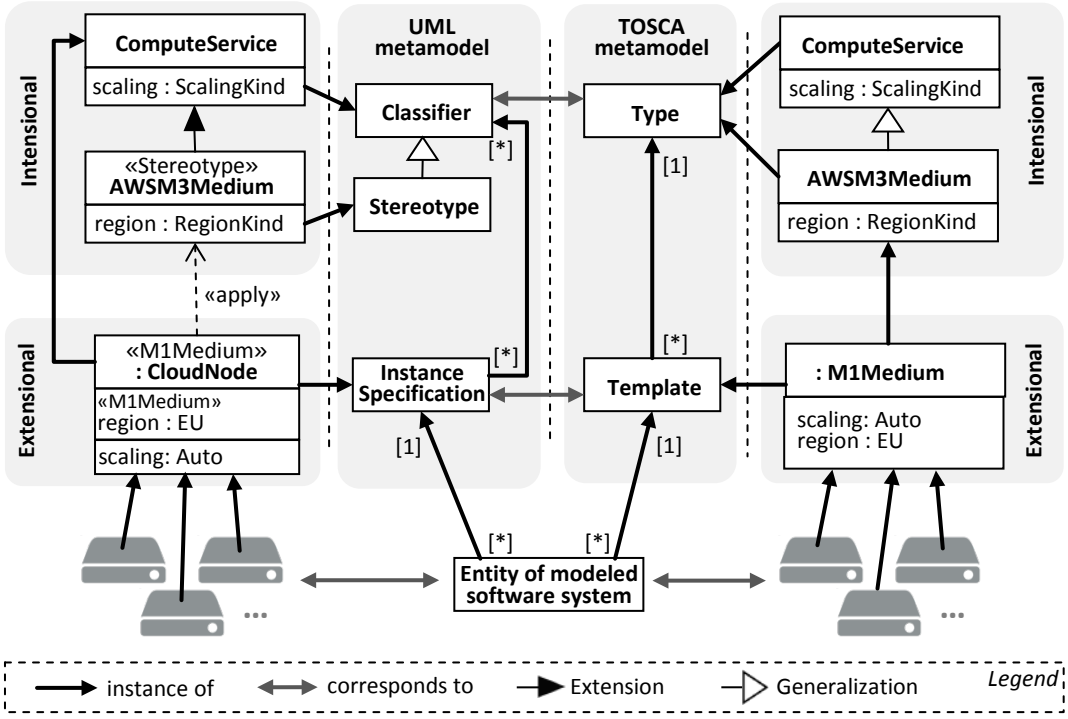

Figure 5.3: Comparison of TOSCA's and UML's intensional level and extensional level

Generally, elements modeled at the extensional level are employed to designate elements of a (software) system in the form of a one-to-one mapping concerning their individual features [Küh06]. Considering the instance of the "AWSM3Medium" compute service, it designates Amazon's M3 medium compute service located in the EU. Obviously, in the context of deployment modeling, several compute services may be comprised by a running cloud application, which requires a multiplicity concept not only for elements of intensional models but also for extensional models. Regarding the former, a one-to-many or many-to-many multiplicity is typically supported by default, *i. e.,* it is not explicitly defined by engineers but integral part of a metamodel. Multiplicities for elements at the extensional level determine the lower-bounds and upper-bounds of real-world entities that are considered as instances of these elements. This is useful for cloud applications where cloud services are provisioned and released on-demand. Clearly, to specify more sophisticated custom rules that trigger the provisioning of new cloud services or force to release them, dedicated language support seems to be required (*cf. e. g.,* [KDR14]).

5.4 Bridging UML and TOSCA

Bridging the gap between UML and TOSCA leverages not only continuous modeling support for cloud applications but also allows engineers to carry out the application provisioning, where a cloud environment is considered as the deployment target. UML provides capabilities to model application architectures from different viewpoints, whereas TOSCA enables the provisioning,

management, and termination of cloud services and applications. In the following, we give first a high-level overview of the underlying approach to combine UML and TOSCA. Thereafter, we propose an effective conceptual mapping between UML and TOSCA where CAML takes the role of capturing cloud-specific features from the perspective of UML.

5.4.1 Conceptual overview

Considering the high-level overview presented in Figure 5.4, the entry point to the CAML2TOSCA approach is a deployment model capturing the desired state of the cloud application provisioning. Creating the deployment model is considered as part of architecture modeling which usually includes to produce a variety of models (or views on models), each of which addressing a concern from a certain viewpoint. In particular, CAML deals with the class, component, and deployment viewpoint. A deployment model refined towards a target cloud environment is considered as input to a tool chain capable of producing a standard-compliant executable cloud service archive (CSAR). We collectively refer to the set of tools comprised by this chain as TOSCA tools. They allow engineers to automatically generate a TOSCA-based representation consisting of type definitions and the topology template that corresponds to the injected UML deployment model refined by CAML. Based on those type definitions, the implementations of management operations required for the application provisioning are automatically injected [KBBL13]. To enable their execution in an appropriate order, a management plan is orchestrated [BBK+14]. All the produced artifacts by the TOSCA tools constitute the CSAR. It can be executed by TOSCA-compliant runtime containers. The presented conceptual mapping between UML and TOSCA provides the basis for automating the generation of a typed TOSCA topology template from a CAML-based deployment model.

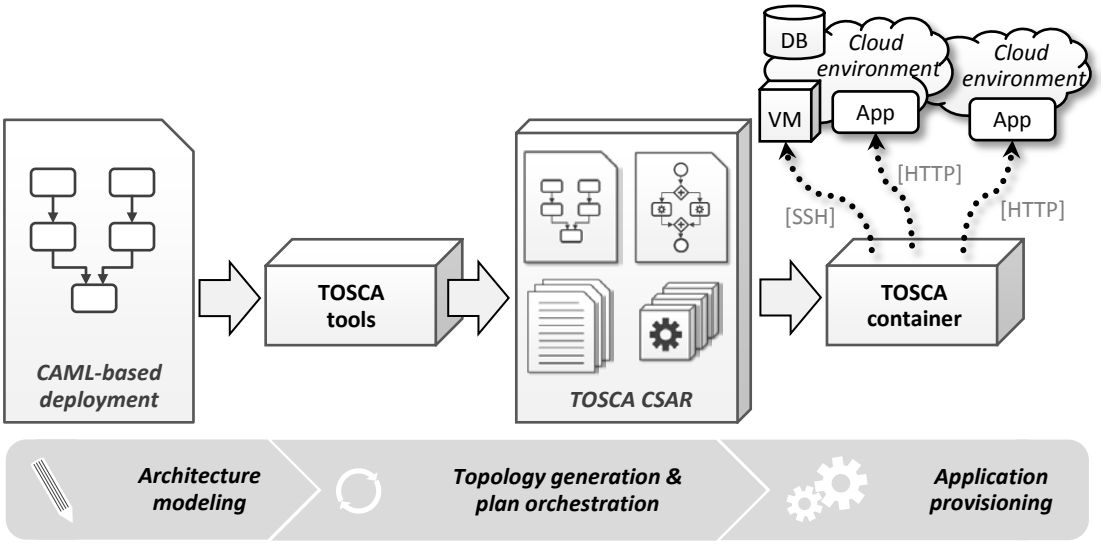

Figure 5.4: Overview of the CAML2TOSCA approach

5.4.2 Conceptual mapping

The proposed conceptual mapping is generic in the sense that any UML deployment model can be translated into a corresponding TOSCA model. As CAML provides standard compliant extensions to UML in terms of custom types, they can be considered as supplementaries to UML's metamodel. In this way, those types can be treated in the mapping process similar to UML standard meta-classes. Considering CAML's defined custom types is vital to produce typed TOSCA topology templates. The types comprise in fact all the properties and operations that are provided to the templates. In CAML, concrete classifiers of UML, *e. g.,* `Node`, `ExecutionEnvironment`, and `Artifact`, are employed for creating such types. In addition, stereotypes are exploited to capture features of existing cloud environments possibly selected as deployment target.

Generally, a concrete `Classifier` of CAML can be represented as a `NodeType` in TOSCA, whereas their use at the extensional level can be represented in terms of a `NodeTemplate`. A specific case are stereotypes that required special treatment for inferring the type information assigned to produced node templates. Moreover, standard relationships of UML typically applied for deployment modeling, *i. e.,* `Deployment` and `Dependency` are addressed in the conceptual mapping. They are usually applied directly at the extensional level. As a result, they are represented as instances of the corresponding meta-classes at the extensional level instead of an `InstanceSpecification`. Both relationship types can certainly be stereotyped if additional features are required or a certain vocabulary with specific semantics needs to be introduced.

Intensional level

The mapping presented in Table 5.1 acts as the basis to generate TOSCA type definitions. In fact, `NodeTypes` can be generated from concrete `Classifiers` that constitute CAML's cloud library and profiles. Basically, the signature of a classifier including its name and whether it is abstract or concrete can straightforwardly be mapped to a `NodeType`. If the concrete classifier is a `Stereotype`, we need to distinguish whether they extend a base element or inherit from a `Stereotype`. In the former case, the generated `NodeType` specializes the corresponding `NodeType` of the stereotype's base element, whereas in the latter case, it inherits from the node type of the super stereotype. For instance, the stereotype `AppEngineService` covered by the Google cloud profile inherits from the `GAEComputeService` stereotype (see fig:cam:cloudproviderprofile). Concrete compute services of the different cloud profiles extend the `ComputeService` concept provided by CAML's cloud library. The latter is considered as an abstraction over the concrete compute services and hence extended by them. If a concrete `Classifier` comprises properties, a `PropertiesDefinition` is created and added to the corresponding `NodeType`. The `PropertiesDefinition` can be considered as the container of the properties provided by a type. An `Operation` including its input and output `Parameters` can be mapped to an `Operation` of a node type and added to the exposed management `Interface`.

Finally, an `Association` can be mapped to a `RelationshipType` where the first `memberEnd` of the `Association` corresponds to the `validSource` and the second one to the `validTarget` of the `RelationshipType`.

UML/CAML	→	TOSCA
uml:Model m		**add** Definitions d
uml:Classifier c		**if**(c.oclIsTypeOf(uml:Class)) **add** NodeType nt nt.name = c.name nt.abstract = c.isAbstract nt.derivedFrom = **if**(c.oclIsTypeOf(Stereotype) **and** c.extension.oclIsDefined()) c.getBaseElement()[1] **else** c.general **if**(c.attribute.notEmpty()) **add** PropertiesDefinition pd **if**(c.ownedOperation.notEmpty()) **add** Interface i
uml:Property p		**add** Element e, Type t **for each** uml:Property p **in** c.attribute e = p.createElementDefinition()[2] t = p.obtainElementType()[3] pd.element = e pd.type = t
uml:Operation uo		**add** Operation o **for each** uml:Operation uo **in** c.ownedOperation o.name = uo.name **add** Parameter p **for each** uml:Parameter up in uo.ownedParamer p.name = up.name p.type = up.type o.inputParameters = uo.ownedParameter **where** p.direction = in o.outputParameters = uo.ownedParameter **where** p.direction = out **add** ImplementationArtifact ia **for each** uml:Behavior b in uo.method b.provideImplementation()[4] nt.interfaces.operations = o
uml:Association a		**add** NodeRelationship nr nr.name = a.name nr.validSource = a.memberEnd.at(1) nr.validTarget = a.memberEnd.at(2)

[1] Retrieves the (ontological) base element that is extended by the stereotype
[2] Creates an element definition from the name of the property and assigns it to the element
[3] Obtains the type of the property and assigns it to the element's type
[4] Provides the concrete implementation for an operation (*e. g.,* in form of Java)

Table 5.1: Conceptual mapping for the intensional level

Extensional level

The mapping presented in Table 5.2 provides the basis to generate a TOSCA `TopologyTemplate` from a UML deployment model refined towards a cloud environment. Thus, the emphasis is now placed on `InstanceSpecifications` and the generation of corresponding `NodeTemplates` and `RelationshipTemplates` from them. If a stereotype is applied to an instance specification the inferred type refers to the node type of the stereotype instead of the type assigned to the instance specification. Indeed, we need to consider the properties of both the type assigned and the stereotype applied to the instance specification. If the assigned type refers to an `Artifact`, a TOSCA `DeploymentArtifact` must be created additionally. It describes the type of an artifact that is actually deployed, *e. g.,* a web application implemented in Java compressed as JAR.

The `Deployment` relationship in UML is directly mapped to the predefined TOSCA relationship template `hosted on`. A collection of base types is defined by the simple profile for TOSCA [OAS15]. In the given mapping between UML and TOSCA, those based types are

UML/CAML	→	TOSCA

uml:Model m		**add** Definitions d
		  add ServiceTemplate st
		  st.name = m.name
		  add TopologyTemplate tt

uml:InstanceSpecification is **switch**(is.classifier.oclType())		

case: uml:Classifier c **using**
rst = is.getAppliedSubstereotype
  (cpp:CAMLElement)
mst = is.getAppliedStereotype
  (cpp:CAMLMultiplicity)
		add NodeTemplate nt
		  nt.name = is.name
		  nt.type =
		  if(rst.oclIsUndefined()) is.classifier
		  else rst.getToscaType()[1]
		  nt.minInstances = is.getValue(mst, "lower")
		  nt.maxInstances = is.getValue(mst, "upper")
		  add Properties
		  add Element e, Value v **for each** uml:Slot sl
		  in is.slots
		  e = sl.definingFeature.name
		  v = sl.getValues()
		  add Element e, Value v **for each** uml:Property p
		  in rst.attribute
		  e = p.name
		  v = is.getValue(rst, p.name)
		  if(c = uml:Artifact)
		  add DeploymentArtifact da
		  da.name = is.name
		  da.artifactType = c

case: uml:Association a **using**
rst = is.getAppliedSubstereotype
  (cpp:CAMLElement)
		add RelationshipTemplate rt
		  rt.name = is.name
		  rt.type =
		  if(rst.oclIsUndefined()) is.classifier
		  else rst.getToscaType()[1]
		  add Properties
		  add Element e, Value v **for each** uml:Property p
		  in rst.attribute
		  e = p.name
		  v = is.getValue(rst, p.name)

uml:Deployment d **using**		**add** RelationshipTemplate rt
		  rt.name = "HostedOn"
		  rt.type = ttl:HostedOn[2]
		  rt.sourceElement = d.source
		  rt.targetElement = d.target

uml:Dependency d **using**
  rst = is.getAppliedSubstereotype
  (cpp:CAMLElement)
		add RelationshipTemplate rt
		  name =
		  if(rst.oclIsUndefined()) "DependsOn"
		  else rst.name
		  type =
		  if(rst.oclIsUndefined()) ttl:DependsOn
		  else rst.getToscaType()
		  add Properties
		  add Element e, Value v **for each** uml:Property p
		  in rst.attribute
		  e = p.name
		  v = is.getValue(rst, p.name)
		  rt.sourceElement = d.source
		  rt.targetElement = d.target

[1] Retrieves the corresponding TOSCA type, *e. g.,* `WebServer` in case of a node or `connectsTo` in case of a relationship (see [OAS15])
[2] ttl: TOSCA type library

Table 5.2: Conceptual mapping for the extensional level

prefixed with a dedicated namespace: *ttl* (TOSA type library). Even though there is also a predefined TOSCA relationship template `depends on` that fits well to the `Dependency` relationships in UML, we need to consider the fact that a dependency may be stereotyped to specialize its semantics. As a result, a possibly applied stereotype determines on the type of the produced relationship template.

5.5 Framework for architecture modeling and application provisioning

The overall framework for architecture modeling and application provisioning to the cloud is presented in Figure 5.5. It distinguishes between two kinds of users. *Application engineers* who model cloud applications in UML and refine them with CAML. Those applications are automatically provisioned to be employed by *business users*.

To create the high-level architecture of a cloud application including its desired deployment on a cloud environment, engineers can employ the Papyrus UML modeling tool for which CAML plug-ins are available. In a first step, the deployment model is refined towards the selected target cloud environment by applying the CAML cloud profile, see ①. For instance, considering the deployment model in Figure 5.1, the Google Cloud Platform was selected as the deployment target. Clearly, any other cloud environment is conceivable as a target as well provided that an appropriate profile is available. Applying environment-specific profiles ensures that a properly typed TOSCA-based representation can be generated from a deployment model created in UML and refined by CAML. To support the application provisioning, the CAML deployment model is translated into a functionally equivalent TOSCA topology, see ②.

This second step is accomplished by the Caml2Tosca transformer. It is grounded in the presented conceptual mapping between UML and TOSCA (see Section 5.4.2). From a technical perspective, the Caml2Tosca model transformer comes as an Eclipse plug-in where the conceptual mapping between UML and TOSCA is implemented by means of the proven model transformation language ATL. To employ ATL, the source and target metamodels must be available in a compatible format. Metamodels defined in EMF's Ecore are directly supported by ATL. However, language definitions expressed in terms of an XML Schema are not compatible with it. For that purpose, we automatically reverse-engineered an Ecore-based representation from the XML-based metamodel of the TOSCA standard [NBM+15]. To produce an XML-based representation of the TOSCA models generated by the Caml2Tosca transformer, the standard serialization mechanisms provided by EMF are exploited. EMF provides dedicated model converters to translate between the serialization formats used by Ecore and XML Schema. The former employs XMI [OMG11c] while the latter relies obviously on XML.

The TOSCA-based representation generated by the Caml2Tosca transformer solely describes the typed topology template of the CAML deployment model. However, for the purpose of enacting the application provisioning process, executable artifacts are required. They are injected into the TOSCA topology via Winery [KBBL13], which is part of OpenTOSCA to model TOSCA-based cloud applications. Winery injects implementations of all management operations required for the provisioning by looking up the corresponding node types and relationship types in a local repository and embedding the required artifacts and type definitions into the TOSCA

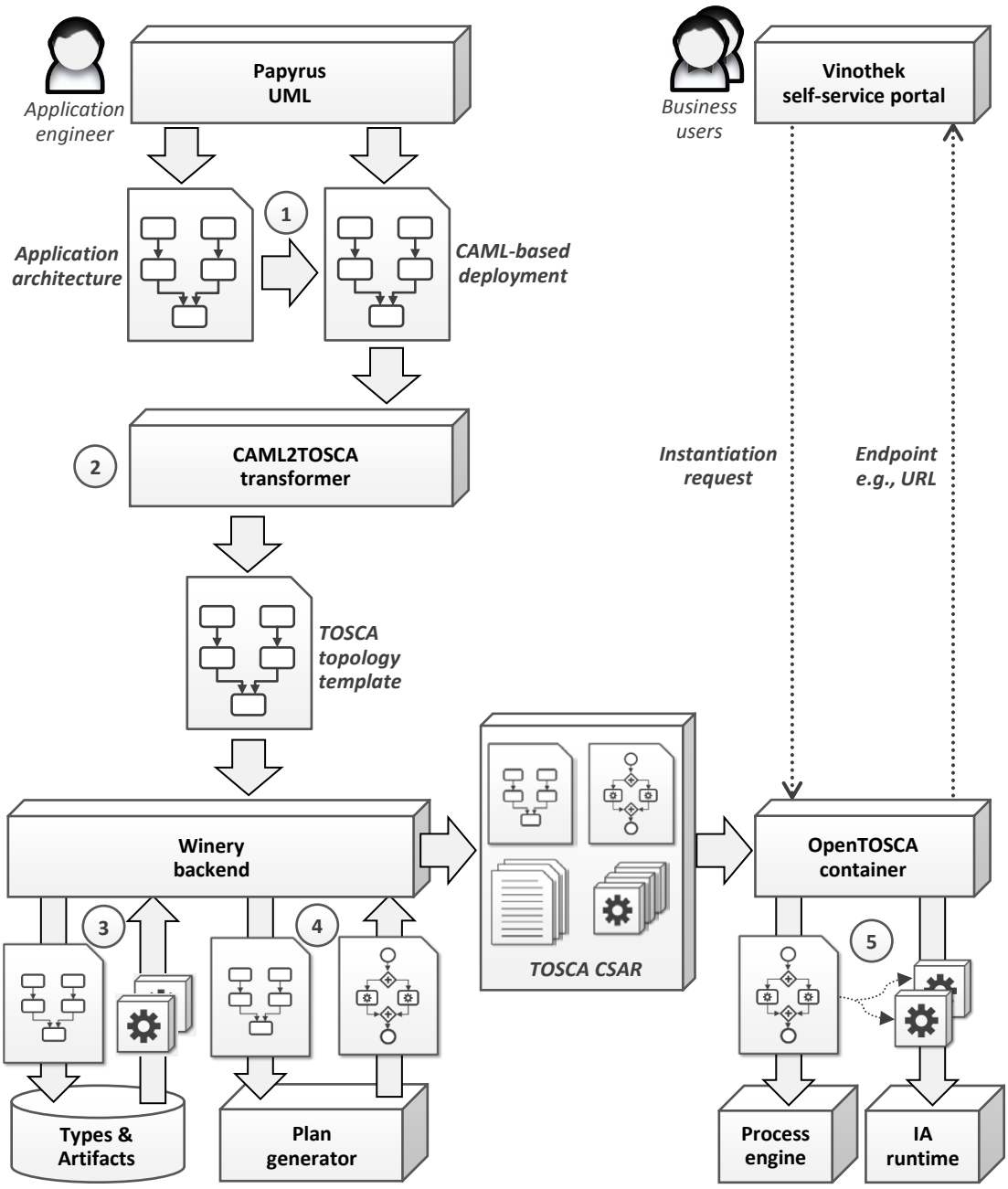

Figure 5.5: Overall framework for architecture modeling and application provisioning to the cloud

topology, see ③. To ensure that the appropriate type definitions are available, we semantically aligned CAML's cloud profile with the respective TOSCA types. As a result, the CAML deployment model can be transformed into a functionally equivalent TOSCA-based representation

without changing its overall semantics. In addition, behavioral aspects, *i. e.,* the implementations of operations required for the provisioning, can be embedded seamlessly without additional manual effort when creating the CAML deployment model. However, very specific stereotypes or TOSCA types, respectively, may require further manual adaptation of the resulting TOSCA topology template or deployment plan. For example, a special script may need to be added to the TOSCA model in order to enable automatically establishing a connection between two custom business components.

To automatically execute the required operations, a BPEL-based deployment plan is automatically generated [BBK+14]. It orchestrates the operation implementations in an appropriate order, see ④. The generated deployment plan along with the topology template and all required artifacts and type definitions are packaged as portable CSAR.

The OpenTOSCA container consumes the CSAR for installing it. The CSAR enables the container to provision the modeled application. In fact, the generated deployment plan is executed on a local workflow engine, see ⑤. As TOSCA allows implementations of management operations using arbitrary technologies, operation implementations that are not executed in the application's target cloud environment, *e. g.,* local services that wrap cloud environment APIs, are deployed on a local IA runtime (Implementation artifact runtime) and bound to the deployment plan [WBB+14b, WBB+14a].

Finally, business users can trigger the provisioning of cloud applications via the Vinothek [BBKL14b], which is a self-service portal that offers those applications.

5.6 Summary

Realizing an effective mapping between UML and TOSCA requires a deep understanding of their peculiarities and the identification of language correspondences at both modeling levels intensional and extensional. The former level must be considered to ensure that appropriately typed TOSCA topology templates can be generated from deployment models expressed in UML and refined by CAML towards the selected cloud environment. As a result, CAML takes the role of capturing cloud-specific features from the perspective of UML.

In this chapter, we presented our approach Caml2Tosca for bridging the gap between UML and TOSCA. As a result, engineers are capable to combine the capabilities of both languages where the deployment viewpoint is exploited for realizing the conceptual mapping between them. CAML's cloud-specific extensions to UML are exploited to accomplish the bridge towards TOSCA. To automate the translation from UML to TOSCA, we implemented the Caml2Tosca transformer. It is grounded in the proposed conceptual mapping between the two languages and provides the necessary glue to leverage a framework for architecture modeling and application provisioning based on UML and TOSCA. To accomplish this framework, we first clarified how the intensional and extensional modeling levels introduced by UML and TOSCA relate to each other as this is essential for providing an effective mapping between them. Thereafter, we identified the language correspondences between UML's deployment language extended by CAML concepts and TOSCA. This paves the way for a continuous cloud modeling support and allows engineers to carry out the provisioning of cloud applications.

Intensional and extensional deployment modeling. As the deployment viewpoint is exploited to realize the bridge from UML to TOSCA, we discussed how their core concepts to model custom types and instances of them can be related in a useful way. At the intensional level, the meta-classes `Classifier` of UML and `Type` in TOSCA correspond to each other. Similarly at the extensional level, a direct language correspondence can be identified: UML's meta-class `InstanceSpecification` corresponds to TOSCA's `Template`. Special treatment is required if stereotypes are applied to instance specifications because TOSCA does not directly support this additional typing dimension of UML. However, we showed how stereotypes in UML can be treated as types in TOSCA. Our solution implies that a stereotyped instance specification is translated into a TOSCA template that is typed by the type of the corresponding stereotype instead of its direct classifier. As a result, the way is paved for generating appropriately typed TOSCA topology templates from UML deployment models refined by CAML's cloud profile.

Conceptual mapping for bridging UML and TOSCA. We proposed a conceptual mapping between UML and TOSCA that is generic in the sense that any UML deployment model refined by CAML can be translated into a corresponding TOSCA model. This is possible because CAML provides standard compliant extensions to UML in terms of custom types which can be considered as supplementaries to UML's metamodel and treated in the mapping process similar to UML standard meta-classes. As a result, the type information is preserved in the translation from UML to TOSCA. Moreover, as we semantically aligned CAML's custom types with possibly existing TOSCA types, *e. g.,* `Hosted on` or `Deployed on`, the CAML deployment model can be transformed into a functionally equivalent TOSCA-based representation without changing its overall semantics. The presented conceptual mapping provides the grounding for the developed Caml2Tosca transformer which is capable to automatically produce typed TOSCA topology templates from CAML deployment models.

Framework for architecture modeling and application provisioning. The Caml2Tosca transformer provides the glue between UML-based architecture modeling and application provisioning based on TOSCA. We presented the overall framework for combining UML and TOSCA. It suggests as entry point a deployment model capturing the desired state of the cloud application provisioning. Creating the deployment model is considered as part of architecture modeling which usually includes producing a variety of models. Based on the deployment model refined by CAML a standard-compliant executable cloud service archive is generated by a set of TOSCA tools: (*i*) the Caml2Tosca transformer generates a typed TOSCA topology template, (*ii*) Winery [KBBL13] injects implementations of management operations based on the type definitions that come with the generated topology template and (*iii*) OpenTOSCA [BBK+14] orchestrates a deployment plan which prescribes the execution order of the application provisioning. The latter is a TOSCA-compliant runtime container and hence capable to execute the application provisioning.

As the entry point to the framework for architecture modeling and application provisioning is a deployment model created in UML and refined by CAML, we investigated the methods of current industrial and open-source UML tools for deployment modeling in general and support

for cloud-based deployment targets in particular. Our findings show that current UML tools lack cloud-based refinement support for deployment models. However, they are capable of adopting CAML, which shows the practical value of CAML2TOSCA for engineers. Further details on this evaluation are presented in Section 7.1.

5.7 Related work

Considering work related to CAML2TOSCA, we discuss existing mapping efforts where TOSCA is involved in process.

Andrikopoulos *et al.* [ARSL14] investigate existing CMLs for representing deployment topologies of cloud applications with the goal to identify a set common modeling concepts shared among them. Based on the identified modeling concepts the Generalized Topology Language (GENTL) has been developed. It aims at providing generic modeling concepts with the goal to establish a pivot language that promotes reuse of deployment topologies represented in different CMLs and enables the composition of those topologies in the long term. As a key aspect of GENTL is to preserve the generic nature of its modeling concepts, it provides an annotation mechanism to support the representation of the various different aspects such as QoS properties, pricing information, performance characteristics, or management operations of cloud services that are addressed by current CMLs. GENTL distinguishes between static and dynamic annotations, where the latter is considered as a service invocation (*e. g.,* a management operation to provision a compute service or release it). For that reason, GENTL provides not only a language definition for representing deployment topologies but also for capturing annotations. Currently, conceptual mappings from TOSCA and Blueprint to GENTL are available. The GENTL environment provides the respective tool support to transform models represented in TOSCA or Blueprint into GENTL models.

Carrasco *et al.* [CCP15] propose to combine TOSCA and CAMP [OAS12] for the purpose of automating the application provisioning to the cloud. For that reason, a provisioning engine called Brooklyn [2] is employed. It complies with the CAMP specification and exposes many of its REST-based API endpoints for managing cloud services at the PaaS layer[3]. To represent deployment topologies which are considered as input to the Brooklyn engine, Carrasco *et al.* propose to use TOSCA. Their approach intends that deployment topologies are modeled using TOSCA in a first step, whereas the application provisioning is carried out based on a CAMP-compliant runtime environment instead of a TOSCA-compliant one. Clearly, this requires a translation of a TOSCA deployment topology into a CAMP blueprint plan, a specification of typed deployment artifacts and their targets which is interpreted by a CAMP-compliant runtime environment. The required conceptual mapping along with the transformation to automate it is currently developed in the context of the SeaClouds project [BCC+15].

In contrast to our work, Andrikopoulos *et al.* and Carrasco *et al.* consider TOSCA as the source language in their mapping process, whereas in our approach UML is mapped towards

[2] Apache Brooklyn: https://brooklyn.apache.org

[3] Even though Brooklyn complies with the CAMP specification, it uses jclouds for connecting to cloud environments. jclouds is not restricted to a particular service layer but rather provides a variety of connectors to different cloud services and environments mainly operated at the IaaS and PaaS layer

TOSCA. Hence, it is considered as the target language in the mapping proposed by CAML2TOSCA. Moreover, the goal of CAML2TOSCA is not only to exploit TOSCA-based application provisioning for deployment models created in UML and refined by CAML but also to bridge architecture modeling and application provisioning to the cloud based on adopted standards in software engineering and cloud computing. This differentiates the CAML2TOSCA approach from the work of Carrasco *et al.* where the main goal is to automate the provisioning of multi-cloud applications. The latter can also be achieved by OpenTOSCA and hence by the provisioning process supported by the CAML2TOSCA approach.

CHAPTER 6

Cloud model patching

Model transformation is a key technique to automate software engineering tasks and hence probably the most penetrating aspect of model-based software development [SK03, BCW12]. It enables reverse engineering processes where lower level implementation artifacts are lifted to higher level models as well as processes where those implementation artifacts are generated from models in a forward engineering context. Moreover it facilitates exchanging models between tools [ADL+12]. Transformations are often implemented *out-place* [MG06]. This means that the model produced by a transformation is built from scratch based on properties of the input model. Output models are newly produced whenever an out-place transformation is (re-)executed. Out-place transformations are typically used when the input and output models are expressed using different languages, *i. e.,* they are conform to different metamodels.

In this thesis, several out-place transformations have been realized. For instance, the generation of UML profiles from annotation-based Java libraries relies on an out-place transformation where the two involved languages are Java and UML. In this scenario, the input model conforms to the Java metamodel, whereas the produced output model, *i. e.,* a profile, conforms to the UML metamodel. As another example, the CAML2TOSCA transformer is realized based on an out-place style because the source metamodel refers to UML, whereas the target metamodel is TOSCA. In both examples the output models are produced from the properties of the input models. A variety of other transformation examples are collected by the "ATL Transformation Zoo"[1].

Like any other piece of software, transformations change over time [WKS+09, vAvdB11, RNHR13]. Modifications to transformations can obviously invalidate previously produced models possibly maintained in a repository. For instance, the platform-specific UML profiles generated for Java libraries are collected in the Eclipse UML Profiles Repository. If the respective *Java2UMLProfile* transformation is changed, the performed modifications must be propagated to existing output models. This process is often called change propagation. A straightforward approach to propagate changes of a transformation to existing output models is to re-execute it entirely for each of them. This ensures that existing output models are newly produced from input

[1] Transformation Zoo: http://www.eclipse.org/atl/atlTransformations

models by taking into consideration the changes to the transformation. However, this approach induces an unnecessary overhead, particularly when computation-intensive transformations are marginally revised. Moreover, possible manual updates to existing models prior the transformation re-execution get discarded in the newly produced models. Furthermore, identifier-based inter-model references involving those existing models are also affected because re-creating the output models from scratch can break them. For instance, an inter-model reference is created as a result of applying a UML profile to a model. Re-creating UML profiles is hence an apparent issue, especially if they are already applied to a variety of models. Consequently, a less invasive approach that avoids re-creating existing models from scratch is required. We refer to this challenge as *transformation evolution/output model co-evolution*, where changes to a transformation imply an evolution and output models must co-evolve according to them.

To overcome the challenge of co-evolving output models with changes in a transformation, we present *model patching*, a non-invasive approach that enables the propagation of changes to existing output models without re-creating them from scratch. It infers *in-place* patch transformations from evolved out-place transformations for existing output models. An in-place execution strategy implies that all changes to a model are updates to it [MG06], which is in contrast to an out-place strategy where the model is always newly produced. Applying an in-place execution strategy requires that the input and output models of a transformation are the same. A patch transformation is capable to satisfy this requirement because it solely operates on the existing output models to which changes must be propagated. It captures only the affected parts of a transformation evolution in terms of model patches. A model patch reformulates a change in an out-place transformation as in-place transformation that is capable of propagating those changes to the respective output models. It is inferred from the type of change to a transformation, *e. g.,* deletion of a transformation rule, and a taxonomy of transformation change types which provides the appropriate co-change, *e. g.,* deletion of model elements produced by this transformation rule. In order to elaborate this taxonomy, we investigated ATL [JABK08], a hybrid model transformation language that supports both styles in-place and out-place. Model patching fills the gap between approaches for propagating changes from either input models to output models based on incremental transformation execution [JE04, HLR06, RBÖV08, JT10, RK12, EKK+13] or metamodels to transformations [LBNK09, GDA12, IPM12, RIP13]. The main characteristics of *model patching* are summarized in the following.

Complete with respect to change types and co-changes. The taxonomy of change types and corresponding co-changes is key to model patching. We have systematically analyzed ATL's metamodel with respect to possible instantiations of its meta-classes and modifications of their features. In a second step, we generalized change types from the analysis results and elaborated corresponding co-changes for those change types in a last step. Given a change in an ATL transformation, the taxonomy is capable to provide the co-change that must be applied to existing output models of this transformation for accomplishing their co-evolution with it.

Non-invasive with respective to change propagation. Model patches realize co-changes in terms of executable in-place transformations that can be applied to existing models. They reflect exactly the intended effects of the changes performed to the a transformation. As model patches

only update the output models based on an in-place execution strategy, elements including their features that need not to be patched are completely preserved.

Incremental with respect to transformation re-execution. Incremental transformation execution shifts the runtime complexity of a transformation from the size of input models to the size of changes performed on them. As a result, only those parts of a transformation are re-executed that are required for propagating the changes in an input model to a corresponding output model. This benefit of incremental transformations is brought to propagating changes in transformations to previously produced output models by model patching.

The remainder of this chapter is structured as follows. In Section 6.1, we motivate model patching by means of the *Java2UMLProfile* transformation developed for generating UML profiles from annotation-based Java libraries. We introduce different dimensions of model transformation evolution and potential co-evolution tasks in Section 6.2. In Section 6.3, we present the taxonomy of change types and corresponding co-changes for the ATL model transformation language. Thereafter, we provide the conceptual foundation for generating patch transformations from evolved out-place transformations implemented in ATL. Finally, we give a summary of this chapter in Section 6.5 before work related to model patching is discussed in Section 6.6.

6.1 Motivation

To motivate the need of model patching, we consider an excerpt of the *Java2UMLProfile* transformation, where the emphasis is placed on its evolution. Listing 6.1 shows two transformation rules for generating stereotypes including properties from declared annotations and their members. For the purpose of demonstrating an evolution, we assume that the third binding of the second rule (see line 19 of Listing 6.1) were added after the first release of the transformation. The binding ensures that the default value of a declared member of an annotation is assigned to the property produced from it.

Listing 6.1: Evolution of Java2UML transformation in ATL (out-place style).

```
1  create Profile : UML from Library : JAVA;
2
3  rule Annotation2Stereotype {
4    from
5      s1: JAVA!AnnotationTypeDeclaration()
6    to
7      t1: UML!Stereotype (
8        name <- s1.name,
9        ownedAttribute <- s1.bodyDeclarations -> select(e | e.oclIsTypeOf
10                       (JAVA!AnnotationTypeMemberDeclaration)))}
11
12 rule Member2Property {
13   from
14     s1: JAVA!AnnotationTypeMemberDeclaration()
15   to
16     t1: UML!Property (
17       name <- s1.name,
18       type <- s1.type.getUMLType()
19       default <- s1.default.getUMLRepresentation())}
```

Obviously, this modification to the transformation should ideally also be propagated to existing UML profiles which were produced by the first release of the transformation, *i. e.,* before the evolution has been carried out. The model patch for the added binding is shown in Listing 6.2. The transformation is in-place because it takes as input a UML profile and provides as output the patched version of that profile (see line 1 Listing 6.1). Furthermore, the model patch requires in addition to the profile the library from which it was produced and the trace model that records the executed transformation rules along with the input and output elements in terms of trace links. The library provides the default values of the annotation members, whereas the trace model reveals to which property a certain default value must be assigned. To obtain the relevant properties, two auxiliary functions are used (see line 4 and 7 Listing 6.1). Considering the evolved transformation rule and the corresponding model patch, the assignment of the default value to the property is identical (see line 19 of Listing 6.1 and line 18 of Listing 6.2). Hence, the model patch propagates exactly the change in the evolved transformation to those properties that must in fact be patched. They are only updated because the change is reformulated, such that an in-place instead of an out-place execution strategy can be applied (see line 12 and 16 of Listing 6.2). As a result, a complete computation intensive re-execution of the evolved transformation which would newly create all stereotypes and properties, *i. e.,* invasive modifications to existing profiles, can be avoided.

Listing 6.2: Model patch for the added binding (in-place style).

```
1  create PatchedProfile : UML refining Profile : UML, Library : JAVA Trace : TRACE;
2
3  -- trace links refering to the generation of properties from annotation members
4  helper def : tracelinks : TRACE!TraceLink = TRACE!TraceLink.allInstancesFrom('TM'
       ) -> select(e | e.ruleName = 'Member2Property' and e.targetElements -> exists
       (f | f.oclIsTypeOf(UML!Property)));
5
6  -- properties which must be patched
7  helper def : outElements : OclAny = (thisModule.tracelinks -> collect(e | e.
       targetElements)) -> flatten();
8
9  rule PatchAddedBinding {
10   from
11     -- the model patch is applied to properties
12     ps1 : UML!Property in Profile (thisModule.outElements -> includes(ps1))
13     using {s1 : JAVA!AnnotationTypeMemberDeclaration = thisModule.tracelinks
14                  -> any(e | e.targetElements -> includes(ps1)).sourceElements ->
                         first();}
15   to
16     -- default values are assigned to selected properties
17     t1 : UML!Property (
18       default <- s1.default.getUMLRepresentation())}
```

Figure 6.1 shows the result of applying the patch transformation to the Objectify profile produced by the *Java2UMLProfile* transformation before it has been modified to support also default values of annotation members. Comparing it to the Objectify profile in Figure 4.11, all the properties of the generated stereotypes provide yet default values.

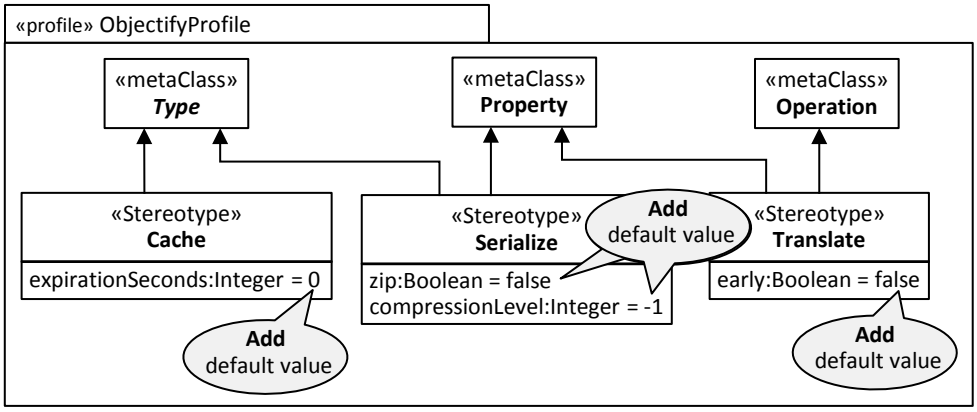

Figure 6.1: Evolved Objectify profile

6.2 Model transformation evolution

Considering the model transformation pattern (see Figure 2.5), a variety of artifacts at different model levels are involved, each of them may subject to evolution. If a certain artifact is modified, others may also require modifications mainly due to the dependencies, *e. g.,* a transformation is *defined upon* a (or several) metamodel(s), between the artifacts of the model transformation pattern. Consequently, the evolution of an artifact usually requires co-evolution tasks. Two evolution dimensions have gained attention in the last decade which led to several approaches that aim at overcoming evolution challenges in the field of model transformation engineering. In the following, we give a brief overview of these two evolution dimensions and discuss how *transformation evolution/output model co-evolution* relates to them. The latter introduces in fact a third evolution dimension for which we provide a solution in this chapter.

The main distinguishing features of the evolution dimensions refer to the type of artifact, *e. g.,* input model or metamodel, that is assumed to evolve and the type of artifact to which a corresponding co-evolution task must be applied. As a result, each evolution dimension is characterized by an initial evolution step and a suggested co-evolution step. Figure 6.2 provides an overview of currently investigated evolution dimensions in the context of model transformation engineering.

Since a metamodel contributes the type information upon which a transformation is defined, modifications to it may also require propagating changes to existing transformations. Considering the ATL transformation in Listing 6.1, the type information is provided by the Java metamodel as well as the metamodel of UML. For instance, Java's language definition changed with version 8 released early 2014 and so its metamodel must evolve to support the newly introduced language features. This in turn may require co-evolving transformations that are defined upon Java's metamodel. As a result, this evolution dimension is often referred to as *metamodel evolution/-transformation co-evolution* (see Figure 6.2(a)) and addressed by several proposed approaches (*cf. e. g.,* [LBNK09, MEMC10, RIP13, IPM12, GDA12]). The main goal of those approaches is to (semi-)automatically adapt transformations to new metamodel versions while preserving their overall behavior.

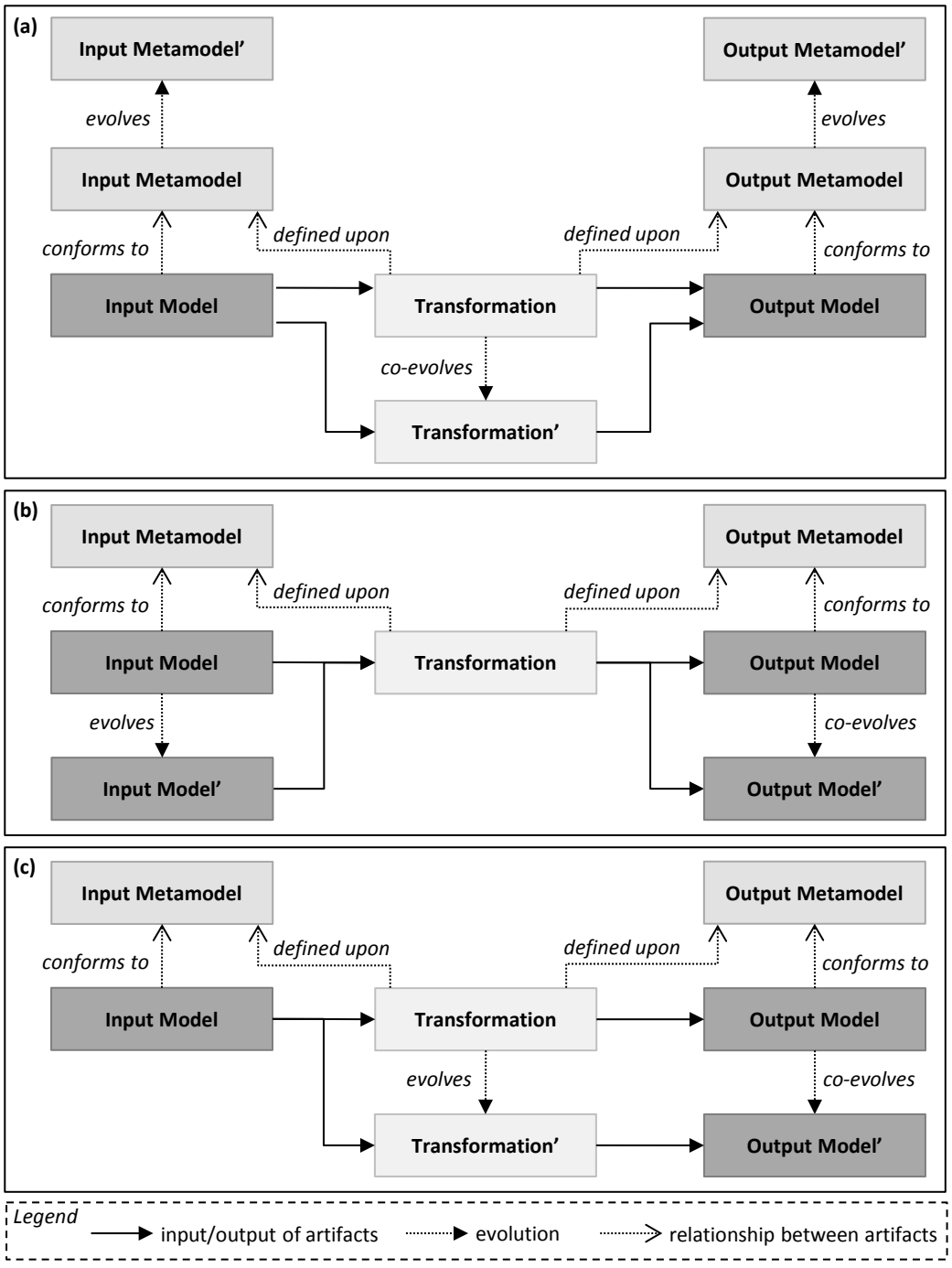

Figure 6.2: Dimensions of model transformation evolution: (a) metamodel evolution/transformation co-evolution, (b) input model evolution/output model co-evolution, (c) transformation evolution/output model co-evolution

While some changes to a metamodel can automatically be propagated to a transformation, others are hardly or even impossible to automate. For instance, if an element of a metamodel is renamed all the occurrences of this element in a transformation can automatically be replaced by the new name [RIP13]. On the other hand, if a new element is added to a metamodel automatically deriving a transformation rule for it is hard [LBNK09]. In this case, only some default rule skeletons can be provided, while the behavior of the rule must be implemented by engineers [RIP13].

The evolution at model-level poses another evolution challenge: *input model evolution/output model co-evolution* (see Figure 6.2(b)). If input models evolve, the respective output models often must co-evolve as well. The co-evolution of output models can straightforwardly be accomplished by re-executing the entire transformation in batch mode for new versions of input models. However, due to several reasons an incremental execution of the transformation is preferred over a complete re-execution. For instance, in case of minor changes on large models the execution time of a transformation can be reduced if the affected parts of it are re-executed only. Another reason is that incremental transformation execution is capable to preserve possible manual updates to output models prior the re-execution because only changes in input models are propagated by the transformation to output models. There are several approaches that allow incremental execution of model transformations with respect to changes in input models (*cf. e. g.,* [JE04, HLR06, RBÖV08, JT10, RK12]). A review of existing approaches for incremental transformation execution has been carried out by Etzlstorfer *et al.* [EKK+13]. The main idea of incremental transformation is that the runtime complexity of a transformation is no longer proportional to the size of input models but instead to the size of changes performed on them.

Beside the two introduced evolution dimensions, *transformation evolution/output model co-evolution* (see Figure 6.2(c)) can be considered as the intersection of the former two dimensions. This third dimension assumes that a transformation based on which output models have already been produced from input models evolves and all transformation executions must be reproduced to turn existing output models into valid transformation results. Obviously, the same benefits incremental transformations provide for propagating changes in input models to output models are also desirable for reflecting changes in transformations to previously produced output models.

6.3 Model patches for out-place transformations in ATL

Model patches are the main constituents of a patch transformation. Depending on the change to a transformation, they provide a co-change suggested to be applied to existing output models produced by the transformation. In the following, we first introduce ATL's language elements for which we present model patches. Thereafter, we present a taxonomy of change types and co-changes for ATL.

6.3.1 Transformation language elements

ATL is a hybrid model transformation language supporting a mixture of declarative and imperative constructs. Considering the patch transformation in Listing 6.2, it comprises two auxiliary functions for computing relevant trace links and elements that need to be patched. These functions are imperative constructs, whereas the two rules of the patch transformation are considered as

declarative constructs. In the remaining chapter, we place emphasis on ATL's declarative part. Transformations implemented in ATL are unidirectional, operate on read-only input models and produce write-only output models. This is an important feature of ATL to facilitate patch transformations as it ensures that input models are immutable throughout the execution of a transformation. Its metamodel is depicted in Figure 6.3.

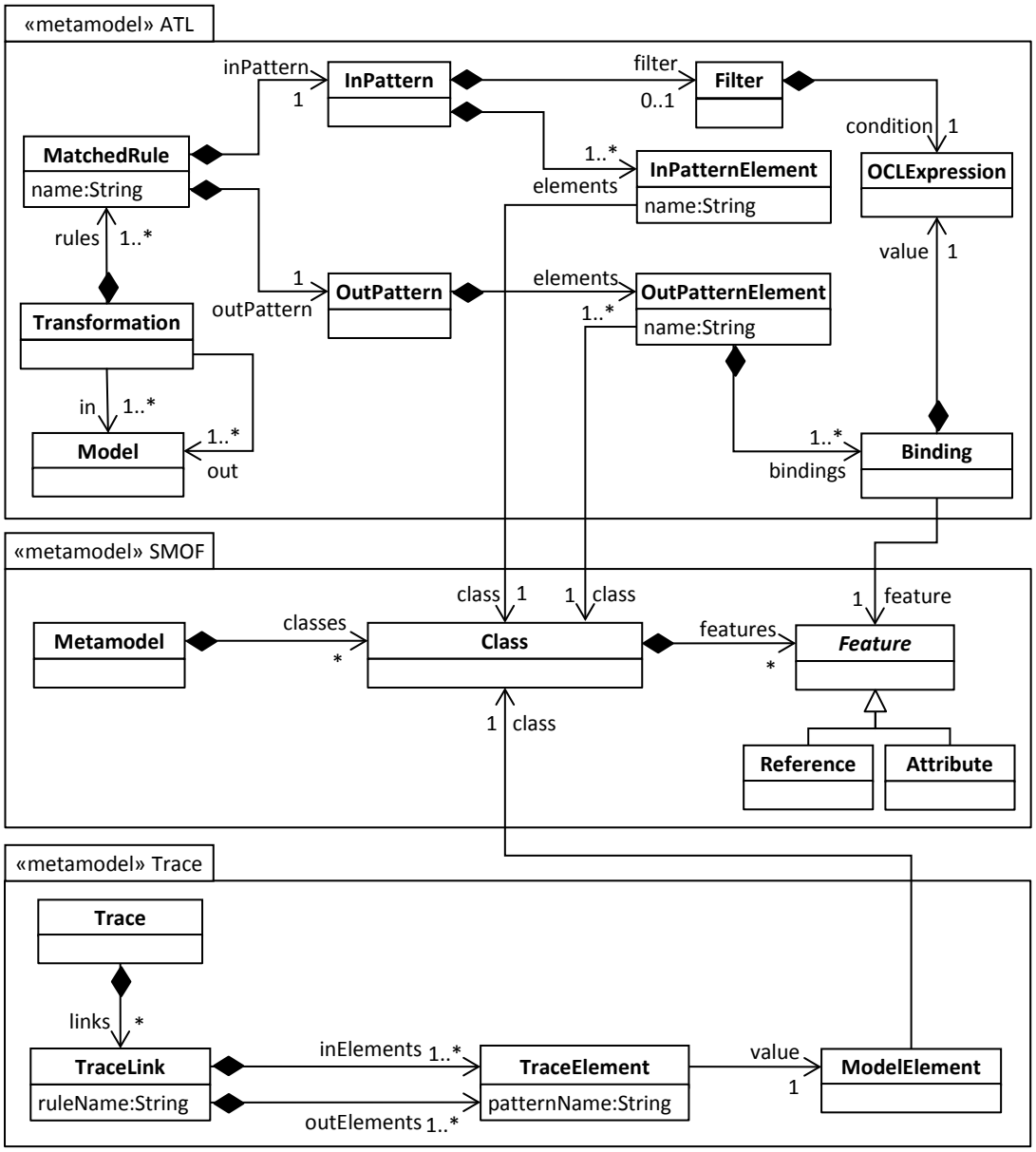

Figure 6.3: Metamodel excerpts: (a) transformation language, (b) metamodeling language, and (c) trace language

A `Transformation` is composed of declarative `MatchedRules`. It gets `Models` as input and produces output `Models`. A `MatchedRule` contains one `InPattern` and one `OutPattern`. The former is a query on the input model and gathers the set of `InPatternElements` that represent the input model elements of the rule. They are defined upon elements of the source metamodel(s). In the context of MOF, `Classes` that may comprise `Features` are used to develop a metamodel. We use the abbreviation SMOF to refer to a *simplified* version of MOF which is however sufficient for the purpose of introducing model patches. `InPatternElements` can also contain a `Filter`. If the conditions of a `Filter` are satisfied by the `InPatternElements`, the respective rule is applied. `Filters` are expressed by means of a constraint language. For that purpose, ATL relies on OCL [OMG14b]. `OutPatterns` describe the creation of elements in the output model. Those elements are typed by the `Class` of the `OutPatternElements`. Each `OutPatternElement` is composed of a set of `Bindings`. Their values are expressed and computed by OCL expressions that are used to initialize the features of output model elements.

Concerning the semantics of ATL, the order in which the rules are defined does not affect the computation of output models. ATL applies a two-phase process. In the first phase, matching conditions of rules, *i. e.,* `InPatterns`, are evaluated. As a result, the set of output model elements that correspond to `OutPatterns` declared in evaluated rules can be determined. In the second phase, those elements are initialized by feature values obtained by the respective `Bindings`.

Finally, we use in our approach an explicit trace metamodel. In fact, a trace model conforming to this metamodel is automatically obtained from a transformation execution, *e. g.,* by exploiting Jouault's "TraceAdder" [Jou05] for standard ATL. A more advanced ATL VM developed by

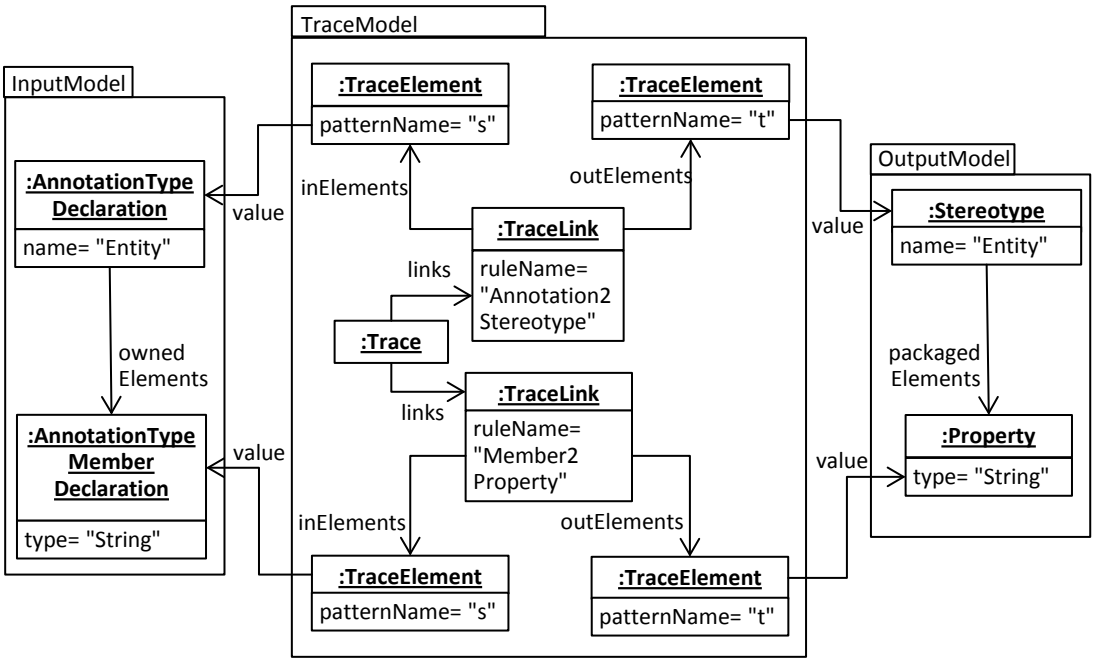

Figure 6.4: Example trace model fragment of Java2UMLProfile transformation

Wagelaar [WTCJ11] has built-in functionality for producing trace models. A `Trace` is composed of `TraceLinks`. A `TraceLink` captures the name of applied `MatchedRule` and contains `TraceElements`. Those elements capture the `name` of the corresponding `InPatternElement` or `OutPatternElement` and reference to the input model elements or output model elements that have been queried or generated, respectively. To sum up, trace models explicitly capture transformation rule executions and information about input model elements that contributed to the generation of output model elements. An example trace model for a possible execution of the *Java2UMLProfile* transformation is shown in Figure 6.4.

6.3.2 Change types and co-changes

In order to establish a taxonomy of change types and co-changes, we have investigated ATL's language definition. In fact, we systematically analyzed the addition and deletion of instances for any meta-class in the transformation metamodel (see Figure 6.3) and modifications of their features. The complete set of distilled change types for which co-changes are highly desirable are summarized in Table 6.1. The collected language concepts refer to the metamodel of ATL (see Figure 6.3).

Table 6.1: Change types and co-changes for ATL

Concept	Change type	Co-change in output model
MatchedRule	Addition	Execution of MatchedRule
	Deletion	Deletion of previously produced elements
	Modification (*name* feature)	Propagation of name change to trace model (*ruleName* feature)
InPatternElement	Addition Deletion	1. Deletion of previously produced elements 2. Execution of the MatchedRule
	Modification (*class* feature)	Considered as Addition and Deletion of InPatternElement
	Modification (*name* feature)	Propagation of name change to trace model (*patternName* feature)
Filter	Addition Deletion Modification (*condition* feature)	1. Deletion of elements that do not satisfy the Filter 2. Creation of elements that satisfy the Filter 3. Execution of Bindings
OutPatternElement	Addition	1. Creation of elements 2. Execution of its Bindings
	Deletion	Deletion of previously produced elements
	Modification (*class* feature)	Considered as Addition and Deletion of OutPatternElement
	Modification (*name* feature)	Propagation of name change to trace model (*patternName* feature)
Binding	Addition	Execution of added Binding
	Deletion	Deletion of feature values
	Modification (*value* feature)	Re-execution of changed Binding
	Modification (*feature* feature)	Considered as Addition and Deletion of Binding

MatchedRule

Adding or deleting a `MatchedRule` implies to add or delete the elements that the rule creates. A change of the rule name is propagated to the trace model, to keep it properly updated.

InPatternElement

If an `InPatternElement` is added or deleted, the matches of a rule for a given input model may change as well. For instance, if we had only one `InPatternElement` and we add another one, the match is realized now with the cartesian product of both element types. Contrarily, if we remove an `InPatternElement`, the number of matches for a rule may decrease. Furthermore, the addition or deletion of an `InPatternElement` may lead to a change of the `OutPattern` in the rule provided that the variable referring to the new/old `InPatternElement` is used in one or several `Bindings`. As a result, we delete all the changes produced by the rule and execute it. Similarly, when the `class` feature of an `InPatternElement` changes, we consider it as an addition and a deletion. As for the modification of its `name` feature, we need to propagate the change to the trace model.

Filter

The effect of adding, deleting, or modifying a `Filter` is equally treated. Even if a `Filter` is not defined, we can still consider one whose `condition` is set to *true*. Similarly, if a `Filter` is removed, it is the same as changing it to *true*. Consequently, we consider the three cases as if the `Filter` is modified. In a first step, elements are added to the output model that are created from elements in the input model that now satisfy the `Filter`, whereas, in a second step, elements are deleted in the output model that correspond to elements in the input model that do not satisfy `Filter` anymore. Finally, the corresponding `Bindings` are executed.

OutPatternElement

Adding or deleting an `OutPatternElement` implies to add or delete the respective elements in the output model and to re-execute corresponding `Bindings`. If the `class` feature is changed, we consider it as addition and deletion of the `OutPatternElement`, whereas the modification of the `name` feature is propagated to the trace model.

Binding

In case a `Binding` is added or deleted, the effect is to compute its value or delete the value that was previously computed. If the `value` expression of a `Binding` changes, it has to be recomputed and reassigned, whereas if the target `feature` is changed, we consider it as an addition and a deletion of the `Binding`.

Rule dependencies and execution

Apart from the co-changes described above, we also have to take into account the dependencies between bindings and rules as explained in Section 6.3. If changes occur in `MatchedRules`,

`InPatternElements`, `Filters`, and `OutPatternElements` (except for modification of `name` features), we need to check if such changes are involved in explicit dependencies. If they are, the generated patch transformations ensure that the respective bindings are recomputed.

The way our co-changes are applied for producing the evolved output models follows the same semantics ATL applies for full execution (see Section 6.2). In a first phase, the output model elements are appropriately added or deleted. Subsequently, in the second phase, feature values of the output model elements are computed if necessary. The latter implies the execution of affected bindings.

6.4 Generation of patch transformations

A patch transformation is comprised of model patches. The general idea underlying a patch transformation is to reformulate changes in out-place transformations as in-place transformations that are capable of propagating those changes to the respective output models. The latter is accomplished by co-changes expressed in terms of model patches. Figure 6.5 gives an overview of how a patch transformation can be generated for an evolved transformation.

The upper part shows the *transformation evolution/output model co-evolution* dimension already discussed in Figure 6.2. It captures the input/output of all involved artifacts and the relationships between them. In particular, the relationship between the two transformation versions indicates the evolution.

Turning the focus on the lower part, it describes the transformation evolution and captures the main steps of model patching. Based on the original transformation and the evolved transformation, a *diff model* is produced. It describes the differences between the two transformation versions and hence the changes performed on the original transformation. To compute the differences between ATL transformation versions, we first inject the transformation code into a model representation by using the ATL injector component. This model-based representation of the transformations allows us to employ EMF Compare[2] for computing the differences between them. To get a more concise diff model for the purpose model patching, we aggregate the diff elements produced by EMF Compare in a post-processing step of the comparison. The idea of this post-processing step is to derive change types as introduced in Table 6.1. As a result, a solid basis for producing a patch transformation is obtained.

In the next step, a higher order transformation (HOT) takes this diff model and the two transformation versions as input and produces a patch transformation. It defines the transformation rules required to co-evolve existing output models according to changes in the transformation that has been executed to produce these models. A patch transformation requires three input models: (*i*) an output model that has been produced from the original transformation because it is evolved according to the changes indicated by a patch transformation, (*ii*) an existing input model which provides model elements and assigned values to them possibly required for appropriately modifying output models, and (*iii*) a trace model as it captures interconnections between model elements of an input and output model. Obviously, the more information is provided as input, the more accurate the evolved output models can be. A patch transformation is capable to produce

[2] www.eclipse.org/emf/compare

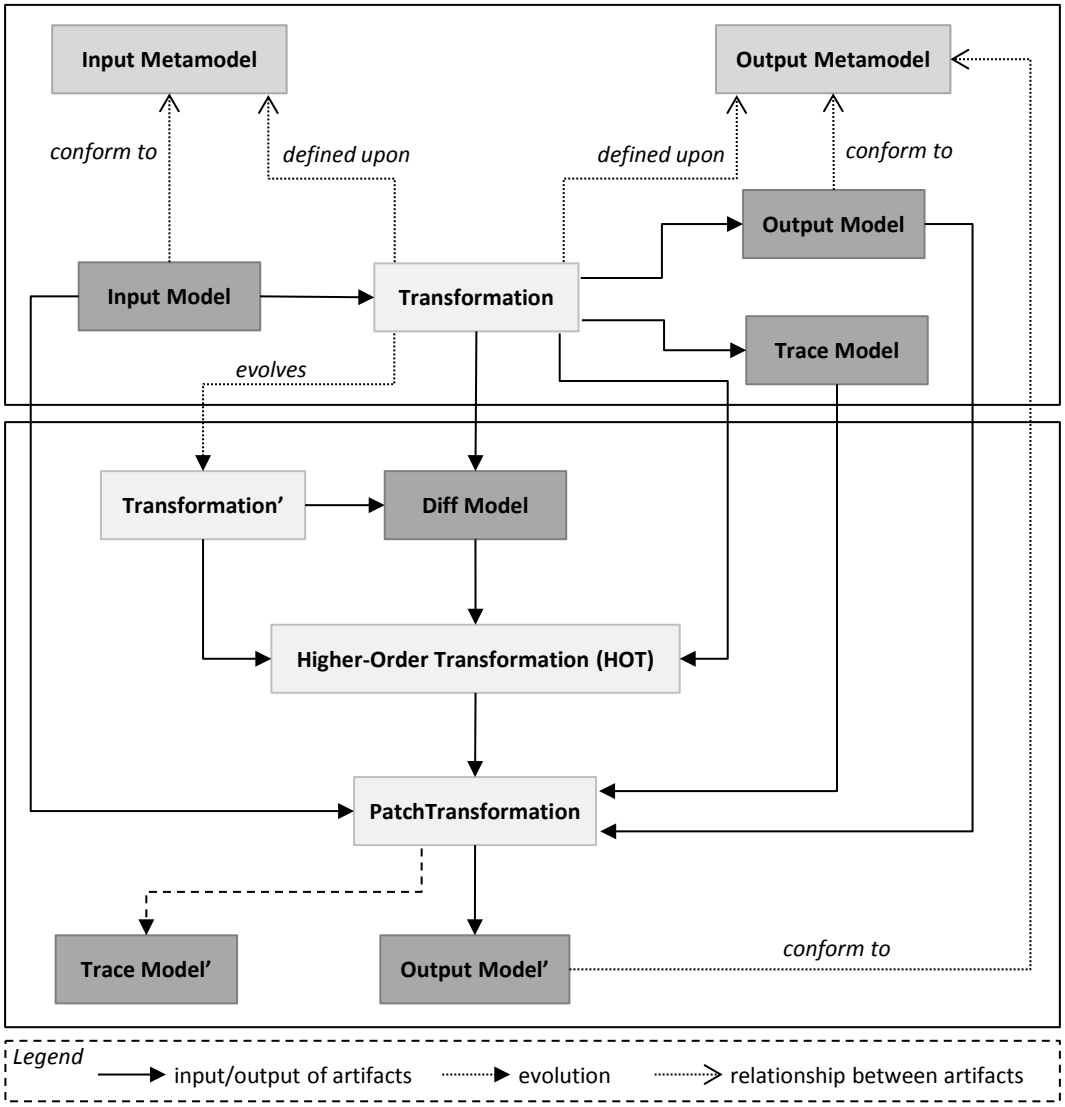

Figure 6.5: Patch transformation generation at a glance

results, *i. e.,* output models and trace models, that are equal compared to results gained from a complete re-execution of the evolved transformation for which a patch transformation has been generated. In case a trace model is not available, it can probably be re-calculated by matching techniques for out-place transformation. If the input model is not at hand, the question arise to what extent the output model can be kept conform to new transformation versions at all. In some cases, the deletion of elements in the output model can be carried out without the existence of an input model.

One important aspect of ATL transformations that affect the generation of patch transformations refers to what we call inter-rule dependencies.

Listing 6.3: Evolution of the *Java2UML* transformation.

```
1  create Model : UML from Application : JAVA;
2
3  rule Package2Package{
4    from
5      s1:JAVA!Package
6    to
7      t1:UML!Package (
8        name <- s.name,
9        packagedElement <- s.ownedElements )}
10
11 rule Class2Class{
12   from
13     s1:JAVA!ClassDeclaration(
14       ~~Set{'domain','web'}->includes(s.package.name)~~
15       Set{'domain','service'}->includes(s.package.name) )
16   to
17     t1:UML!Class(
18       name <- s.name )}
```

Considering the excerpt of the *Java2UML* transformation in Listing 6.3, it transforms Java packages and class declarations to their UML correspondences. For the purpose of demonstrating the need to consider inter-rule dependencies for generating patch transformations, we assume that only class declarations contained by certain packages, *i. e., domain* and *web*, are considered. This behavior is achieved by the defined `Filter` of the `InPattern` in the second `MatchedRule`. The `Bindings` of the two rules ensure that the names of the packages and classes as well as the references in-between are pushed from the Java (input) models to the produced UML (output) models. To introduce a possible revision of the transformation, the `Filter` condition of the second rule is modified (see line 14 of Listing 6.3). As a result, class declarations contained by the *service* package instead of the *web* package are expected in the produced UML model. In our transformation, the `return type` of the value expression `s.ownedElements` is of type `Sequence(JAVA!AbstractTypeElement)`. Thus, it may also contain classes because *JAVA!ClassDeclaration* is a subtype of *JAVA!AbstractTypeElement*. For that reason, if the binding is computed, `packagedElement` will reference, among others, those elements created in the second rule. Technically speaking, ATL performs a transparent lookup of output model elements for given input model elements when executing bindings. Hence, ATL automatically retrieves corresponding UML elements for queried Java elements. It is important that we deal with these inter-rule dependencies when creating patch transformations. In case of our transformation, a modification in the second rule may cause the re-computation of the second *Binding* in the first rule (see line 9 of Listing 6.3). To deal with such situations, we perform a static analysis on the revised transformation. It consists of using a HOT to determine the return types of value expressions of bindings. Then, we calculate the dependencies between bindings and rules.

Listing 6.4 presents an excerpt of the patch transformation inferred from the revision of the original out-place transformation in Listing 6.3. In the first rule, produced elements of existing models that do not satisfy the revised filter condition are deleted, whereas in the second rule, elements that have previously not satisfied the filter condition but satisfy the revised one are produced. In our example, these elements refer to UML classes of the original output model.

Listing 6.4: Patch transformation for evolved *Java2UML* transformation (see Listing 6.3).

```
1  create PatchedModel : UML refining Model : UML, Application : JAVA Trace : TRACE;
2
3  rule PatchFilterDelete {
4    from
5      s1:UML!Class(thisModule.elementsForDeletion->includes(s))}
6
7  rule PatchFilterAdd {
8    from
9      s1:Java!ClassDeclaration(thisModule.elementsForAddition->includes(s))
10   to
11     t1:UML!Class( name <- s1.name )}
12
13 rule Package2Package {
14   from
15     ps:UML!Package
16     using {s:JAVA!Package = thisModule.tracelinks->any(e|e.outElements->exists(f|
           f.value = ps)).inElements->any(e|e.patternName = 't1').value;}
17   to
18     t1:UML!Package(
19       packagedElement <- ps.packagedElement->union(JAVA!ClassDeclaration.
             allInstances() -> select(e|s.ownedElements->includes(e) and thisModule.
             adds->includes(e))))}
```

The UML classes are computed prior to the execution of the patch rules and provided by the respective sets, *i. e.,* `elementsForDeletion` and `elementsForAddition`. They are accomplished by executing the original and revised filter conditions against the original input model and calculating their differences. The third rule is dedicated to the re-execution of bindings affected by transformation revisions. Even though in our example the binding itself has not been explicitly revised, its re-execution is required to ensure that newly added UML classes are appropriately referenced by their containing UML packages. This shows the importance to consider inter-rule dependencies for generating patch transformations. As UML packages may already contain classes, the `union` operator needs to be applied to accomplish the expected result. In this respect, the pertinent Java packages are required as well (see line 16 of Listing 6.4). They are queried from the trace model in the `using` part of the patch transformation (see line 19 of Listing 6.4).

6.5 Summary

Transformation changes must be propagated to existing output models possibly maintained in a central repository. To accomplish this co-evolution task, an evolved transformation can entirely be re-executed. However, completely re-executing evolved out-place transformations for propagating changes to existing output models is computation intensive in particular if the models are large and the changes only marginal. Moreover, possible modifications to those models prior the transformation re-execution get discarded because an out-place transformation typically produces output models from scratch. Consequently, a less invasive and computation intensive approach is required to overcome the co-evolution of existing output models with changes to transformation.

In this chapter, we proposed model patching to tackle the challenge of *transformation evolution and output model co-evolution*. Model patching exploits the benefits of incremental transformation execution, which has already been investigated and applied for propagating

changes from input models to output models. To overcome the challenge of *input model evolution and output model co-evolution* only the relevant parts of a transformation are re-executed, such that previous transformation results are updated instead of discarded. Similarly, model patching updates existing output models by reformulating changes to out-place transformations into an in-place patch transformation. A patch transformation takes as input existing output models and updates them by an in-place execution strategy. In this way, elements in the output model that do not require modifications are completely retained. As a result, model patching is non-invasive with respect to change propagation. Furthermore, it is usually faster compared to a complete transformation re-execution because a patch transformation for a typical evolution scenario comprises only a small portion of the rules covered by an evolved transformation.

Model transformation evolution. Several artifacts at different model levels may subject to evolution in model transformation engineering. We investigated and discussed two evolution dimensions that gained attention in the last decade and introduced *transformation evolution and output model co-evolution* as a third evolution dimension. It can be considered as the intersection of the former two dimensions: *metamodel evolution and transformation co-evolution* and *input model evolution and output model co-evolution*. In particular, the investigation of the latter revealed that the benefits incremental transformation provides for propagating changes in input models to output models are also desirable for reflecting transformation changes to previously produced output models.

Model patches for out-place transformations in ATL. Following the idea of incremental transformation, we introduced model patches that provide output model co-changes for changes to transformations. To establish a set of effective co-changes for models produced by an ATL transformation, we investigated ATL's metamodel with an emphasis on possible transformation changes. In fact, we systematically analyzed the addition and deletion of instances for any meta-class of ATL's metamodel and the modifications of features comprised by those instances. Based on the results of this analysis, a taxonomy of change types and corresponding co-changes has been developed. This taxonomy is key to provide a model patch for a certain change to a transformation. Technically speaking, a model patch is implemented by a transformation rule that updates models according to transformation changes.

Generation of patch transformations. A patch transformation reformulates changes to an out-place transformation in terms of an in-place transformation that is suggested to be applied to existing output models. The change propagation is accomplished by executing the model patches comprised by a patch transformation. We introduced a framework for generating a patch transformation from the *diff model* of two versions of a transformation. A higher order transformation is at the core of this framework. It takes the *diff model* and the two transformation versions as input and generates a patch transformation capable to propagate the identified transformation changes to existing output models. In this respect, a patch transformation considers not only co-changes directly derived from the taxonomy of change types but also possible inter-rule dependencies that may affect the update of output models, *e. g.,* a binding which depends on the results of a rule invocation. To appropriately deal with inter-rule dependencies, we perform a static analysis on

the revised transformation for determining dependencies between bindings and rules.

To investigate whether model patching is equally effective compared to completely re-executing evolved transformations, we carried out a case study based on two versions of the *Java2UML* transformation. In general, our findings show that the updates to the output models by model patching reflect exactly the intended effects of the changes in the evolved transformation. When comparing the execution times of patch transformations and evolved transformations for co-evolving existing output models, our results show a speed-up for the majority of patch transformations. More details on this evaluation are presented in Sections 7.2 and 7.3.

6.6 Related work

To the best of our knowledge, there is currently no other approach that deals with *transformation evolution and output model co-evolution*. Since model patching exploits incremental transformation execution, we discuss it according to existing approaches that also exploit this technique for propagating the changes in an input model to a corresponding output model, thereby the emphasis of those approaches is placed on *input model evolution/output model co-evolution*. A detailed comparison of existing incremental model transformation approaches is presented by Etzlstorfer *et al.* [EKK+13].

Johann and Egyed present a framework for incremental transformation where a notified change to an input model is instantly propagated to existing output models [JE04]. The framework provides generic functions that need to be implemented for a given transformation. They ensure that existing output model are kept synchronized with changes to input models [XLH+07]. As a result, the change propagation is accomplished by executing those implemented functions instead of certain parts of the original transformation. This is in contrast to model patching, whereby certain parts of the original transformation are re-executed instead of auxiliary functions that need to be re-implemented for each transformation separately.

The change propagation from input models to output models as proposed by Hearnden *et al.* [HLR06] is based on *live transformation*, whereby the transformation context of the original transformation is preserved, such that the effects of changes to input models can be reflected to output models. A live transformation does not terminate but instead reacts to changes in input models. The approach of Hearnden *et al.* is applicable to declarative rule-based transformation languages such as Tefkat [LS06]. Jouault and Tisi present an implementation of live transformation for ATL [JT10]. They focus on the declarative part of ATL, which is also considered by model patching. However, model patching does currently not support live transformation in the sense of dynamically re-executing rules of a model transformation in memory, but rather generates a new patch transformation for a changed transformation. The scope of the live transformation approach proposed by Ráth *et al.* [RBÖV08] also comprises partially declarative and imperative implementations of transformations. Existing results on live transformation have been generalized by Bergmann *et al.* [BRVV12] in terms of a classification of change scenarios and a formalism for change processing.

Razavi and Kontogiannis turn the focus on the imperative implementation style of model transformation languages [RK12]. Their approach derives incremental model transformations from existing transformations to improve their performance in succeeding executions. They exploit partial evaluation techniques known from the area of program optimization. Partial evaluation is a program optimization technique which is based on the premise that parts of a program can be pre-evaluated for static input data with the goal to avoid the re-computation of those parts in subsequent executions of the program [Jon96]. In the approach of Razavi and Kontogiannis static input data refer to elements of the input model that are expected to remain unchanged. This information is lifted to the meta-level and passed to the transformation via user-defined annotations. They are applied to elements of the input metamodel. The parts of the transformation that are expected to be pre-evaluated are determined based on the annotated elements. The results of the pre-evaluated transformation parts are either cached in a local store or in-lined in the transformation under subject.

CHAPTER 7

Evaluation

As UML plays an important role in the *architecture style* proposed by this thesis, we investigate the methods of current UML modeling tools to deal with platform and environment-specific extensions and compare them to CAML's methodological approach. In particular, we place emphasis on their representational capabilities to declare and apply annotation types of the Java platform at model level and their support for deployment modeling where cloud environments are considered as the target. Considering the former, CAML provides annotation types in terms of dedicated UML profiles. To investigate the quality of UML profiles automatically generated from Java libraries, we compare them with profiles offered by current modeling tools. In fact, we selected four established Java libraries for which a corresponding UML profile is available. As some existing Java libraries are obviously relatively large (more than 700K reverse-engineered elements), we also report on the performance of CAML's tool for UML profile and profiled UML model generation.

Both the quality and performance of *model patching* is examined by means of model transformations developed in the course of the ARTIST project and applied to its use cases. Regarding the quality of model patching, we investigate whether it is equally effective compared to completely re-executing evolved transformations on existing output models. For that reason, we selected six different revisions of the *Java2UML* transformation to cover the introduced change types and the core effects of their co-changes. In order to assess whether model patching is faster compared to a re-execution of an evolved transformation, we carry out a performance analysis. We compare the execution times of revised transformations and their respective patch transformations mainly to provide an impression of the performance improvements that can be expected by applying model patching.

Finally, in order to demonstrate the practical relevance of the proposed architecture style for cloud application modeling, we report on its application in the context of a modernization scenario to the cloud. The modernization scenario builds on the PetApp introduced in Section 1.4. The main highlights of our findings from the conducted investigations are summarized in the following.

Cloud application modeling based on libraries and profiles can be adopted by current UML modeling tools. Current commercial and open-source modeling tools that support UML are also capable of adopting cloud-specific extensions based on libraries and profiles provided by CAML.

Lack of cloud-based refinement for UML deployment models emphasizes practical value of CAML. The majority of current UML modeling tools do not provide dedicated modeling concepts for refining deployment models towards a select target cloud environment.

Automatically generated UML profiles are higher in quality compared to profiles used in practice. With a fully automated approach, the quality of current profiles can be improved by providing more complete stereotypes that precisely capture the intention of the original annotation types in terms of target definitions, member declarations and return values of such members.

Model patching is less invasive and equally effective compared to complete transformation re-execution. A Patch transformation exactly reflects the intended effects of a transformation revision to existing output models based on an in-place execution strategy and thus only updates existing output models.

A speed-up of model patching can be observed for the majority of transformation change scenarios. Generated patch transformations execute in general less rules compared to the revised transformations and are thus faster compared to a complete transformation re-execution for propagating transformation changes to existing output models.

Annotation-based modeling leverages reverse and forward engineering scenarios. Annotations on the model level enable higher quality results produced from reverse engineering processes and richer application code generated in course of forward engineering processes.

Cloud-specific refinements to architecture models close the gap between deployment configurations and target cloud environments. Architecture models created in UML and refined by CAML are interpretable by a TOSCA-based runtime container for cloud application and service provisioning thanks to the fully-automated Caml2Tosca transformer.

The remainder of this chapter is structured as follows. In Section 7.1, we evaluate the methodological approach underlying CAML compared to the methods of current commercial and open-source modeling tools that support UML. Thereafter, in Section 7.2, we investigate the quality of (*i*) CAML's generative techniques for UML profiles and (*ii*) model patching for updating existing output models. Both of them are subject to a performance evaluation in Section 7.3. Finally, in Section 7.4, we demonstrate the practical relevance of the architecture style for cloud application modeling.

7.1 Methodological evaluation

Today, several commercial and open-source modeling tools support UML. We investigate their methods for dealing with platform-specific information captured by Java annotation types. Thereafter, we place emphasis on the methods of those tools for deployment modeling in general and support for cloud-based deployment targets in particular. In total, we selected seven major industrial modeling tools as summarized in Table 7.1.

Name	Version	Availability	Source
Altova UML	2015	commercial and free for academic use	www.altova.com/umodel.html
ArgoUML	0.34	open-source	argouml.tigris.org
Enterprise Architect	9.3	commerical and free for academice use	www.sparxsystems.com
Magic Draw	18.0	commerical and free for academice use	www.nomagic.com
Rational Software Architect	8.5.1	commerical and free for academice use	www.ibm.com/developerworks/rational/products/rsa
Papyrus	1.0.0	open-source	www.eclipse.org/papyrus
Visual Paradigm	12.1	commercial and free community edition	www.visual-paradigm.com

Table 7.1: Selected modeling tools that support UML

7.1.1 Declaration and application of Java annotation types in UML

The aim of this study is to investigate the methods of current UML tools for dealing with the declaration and application of Java annotations types in UML. For that reason, we set the focus on a Java-based reverse engineering example that includes annotations and their declarations with the goal to answer the following research question.

Methodological approach – Research question 1: *What are the methods of current modeling tools to represent Java annotation types and their applications in UML and what are the practical implications?*

In order to answer this research question, we defined a set of comparison criteria that mainly address (*i*) how the conceptual mapping between Java and UML for annotations is achieved by current modeling tools and (*ii*) the generative capabilities of these tools regarding profiles. Based on the defined criteria, we evaluate six representative modeling tools and CAML.

Comparison criteria

As there are different approaches on how annotation types and their applications are represented on the model level, the first and the second criteria ($CC1$ and $CC2$) refer to these extensional capabilities. The third criterion ($CC3$) refers to the support of generative capabilities regarding profiles.

- *CC*1: How are Java annotations applied to UML models?
- *CC*2: How are Java annotation type declarations represented in UML?
- *CC*3: Is the generation of UML profiles from Java code supported?

Selected tools

We selected six out of the seven major industrial modeling tools that claim to support reverse engineering capabilities for Java and UML. Papyrus does not yet directly support automatic reverse engineering. The selected tools are summarized in Table 7.2.

	Mapping (Java -> UML)		UML Profile Generation
Modeling Tool	**Annotation Application**	**Annotation Declaration**	
Altova UML	Generic Java Profile	Interface	-
ArgoUML	Generic Java Profile	Interface	-
Enterprise Architect	Built-in Tool Feature	Interface	-
Magic Draw	Generic Java Profile	Interface	-
Rational Software Architect	Specific Profiles	Stereotype	-
Visual Paradigm	Built-in Tool Feature	Class	-
CAML	Specific Profiles	Stereotype	+

Table 7.2: Comparison results

Evaluation procedure

We defined a simple reference model that declares a Java class to which we applied an annotation type from the JPA (see Figure 2.8). For the purpose of importing the application, we activated the offered functionality of the modeling tools required for a reverse engineering scenario from Java to UML. While some of the modeling tools are delivered with standard configurations, other modeling tools allow configurations to change the reverse engineering capabilities by using specific wizards. Moreover, some modeling tools go one step further and allow modifications on the transformation scripts used for the import of Java code. We evaluated the capabilities of the modeling tools offered in the standard settings and explored the different wizard configurations if supported, but we restrained from modifying transformation scripts.

Results

The results of our comparison are summarized in Table 7.2. It shows that the investigated tools apply one of three significantly different approaches to represent Java annotations in UML: (*i*) the *built-in* annotation feature of modeling tools is used, (*ii*) a *generic* profile for Java is provided, which enables capturing annotations and their type declarations, or (*iii*) profiles are offered, which

are *specific* to a Java library or even an application with custom annotation type declarations. The first solution is certainly the most generic one as it goes beyond Java and UML. Clearly, it facilitates capturing Java annotations, though the type declaration of an annotation in terms of a UML element and its application are not connected. A generic profile for Java emulates the representational capabilities of Java's annotation language. Although with this approach, the connection of annotation type declarations and their applications can be ensured, the native support of UML for annotating elements with stereotypes is still neglected. However, stereotypes specifically defined for annotation types would facilitate their application in a controlled UML standard-compliant way as they extend only the required UML meta-classes. From a language engineering perspective, such stereotypes facilitate defining constraints and model operations, such as model analysis or transformations, because they can directly be used in terms of explicit types similar to a meta-class in UML. CAML is based on a conceptual mapping between Java's annotation language and UML's profile language, which enables the generation of specific stereotypes for corresponding annotation types that in turn leverage platform-specific profiles. While all evaluated modeling tools support the generation of annotated UML class diagrams from Java applications, none of them is capable of generating profiles dedicated to Java libraries. Only the Rational Software Architect also exploits the powerful capabilities of stereotypes and profiles for capturing declared Java annotation types.

7.1.2 Representation of deployment models and their cloud-based refinement

In order to investigate the methods of current UML tools for representing deployment models and refining them towards a cloud environment, we created the deployment model of the PetApp (see Figure 4.2) in each of the selected tools. The aim of this study is to answer the research question as follows.

Methodological approach – Research question 2: *What are the methods of current UML modeling tools to represent cloud-based deployment models and what are the practical implications?*

In order to answer this research question, we address (*i*) the levels at which deployment models are represented, (*ii*) the support for multiplicities not only at type level but also at instance level, and (*iii*) the offered possibilities to refine environment-independent deployment models towards a selected cloud environment. Regarding the second criterion, the support for multiplicities at the instance level is not directly supported by the UML standard. However, defining them for modeled application artifacts and cloud services appears to be of particular importance. The multiplicities determine the lower bound of running application artifacts and cloud services as well as their upper bound since in a highly scalable cloud environment [VRB11] they are provisioned as their demand increases but also released once their demand decreases.

Comparison criteria

As deployment models are specified by exploiting the intensional and extensional level, the first criterion ($CC1$) refers exactly to the capability of UML modeling tools to support both levels.

The second criterion ($CC2$) is dedicated to the support of multiplicities at the extensional level because this seems of particular interest for modeling cloud applications. Finally, to investigate the need of UML libraries and UML profiles covering cloud-specific domain concepts, the third criterion ($CC3$) addresses the support of current UML modeling tools for refining deployment models towards a cloud environment.

- $CC1$: Is deployment modeling supported at both levels intensional and extensional?
- $CC2$: Is the definition of multiplicities supported for elements at the extensional level?
- $CC3$: Is the refinement of deployment models towards a cloud environment supported?

Selected tools

The selected set of commercial and open-source UML modeling tools that claim to support deployment modeling are summarized in Table 7.3.

	UML				
	Deployment		Multiplicities		Cloud Support
Modeling Tool	**Intensional level**	**Extensional level**	**Intensional level**	**Extensional level**	
Altova UML	supported	supported via Object Diagram	supported	not supported	no support
ArgoUML	supported	directly supported	supported	not supported	no support
Enterprise Architect	supported	supported via Object Diagram	supported	supported	SOMF-based
Magic Draw	supported	directly supported	supported	not supported	no support
Rational Software Architect	supported	directly supported	supported	not supported	partial support
Papyrus	supported	supported via Object Diagram	supported	not supported	no support
Visual Paradigm	supported	supported via Object Diagram	supported	not supported	no support

Table 7.3: Comparison results

Evaluation procedure

We imported the deployment model of Figure 4.2 including CAML's cloud library and profile into the modeling tools. We evaluated the capabilities of the modeling tools offered in the standard settings and explored the different wizard configurations if supported.

Results

The results of our study are summarized in Table 7.3. As expected, all evaluated UML modeling tools support both the intensional and extensional level to create deployment models. Support for the extensional level slightly differs between the tools because some of them offer to directly create instances of deployment artifacts, where the respective instance specification assigned with a classifier is generated automatically, *i. e.,* without additional user interaction. Regarding multiplicities at the extensional level, only Enterprise Architect supports them by default. Most of the tools lack cloud-based refinement support for UML deployment models. Only Rational Software Architect introduces a cloud node concept which is resembled by CAML's cloud node. This emphasizes the value of CAML not only to leverage the refinement of deployment models towards a cloud environment but also as a bridge from UML to TOSCA. Finally, even though this evaluation focuses on UML, it is worth noting that Enterprise Architect supports describing and analyzing cloud environment topologies as part of the service-oriented modeling framework (SOMF)[1].

7.2 Quality evaluation

As UML profiles are already offered by current modeling tools, we investigate their quality in comparison with profiles automatically generated by CAML. Then, we turn the focus on the quality of model patching. In fact, we investigate whether it is equally effective compared to completely re-executing evolved transformations on existing output models. In order to conduct the case studies, we follow the guidelines of Roneson and Hörst [RH09].

7.2.1 Comparison of UML profiles

As UML profiles are already offered by current modeling tools, the aim of this case study is to investigate their quality in comparison with profiles automatically generated by CAML. For that reason, we conduct a positivist case study [Lee89] based on real-world Java libraries. We evaluate the commonalities and differences between generated profiles and profiles used in practice with the goal to answer the following research question.

Quality – Research question 1: *How is the quality of UML profiles automatically generated from annotation-based Java libraries compared to UML profiles used in practice?*

In order to answer this research question, we define the requirements of the case study, briefly mention the used Java libraries, and specify the measures based on which the comparison is conducted. Then, we discuss the results of our study not only from a syntactic perspective, but also from a semantic one. The rationale behind this two-step approach is that even though a syntactical matching process for comparing the profiles provides already valuable results, some interesting correspondences may still be uncovered because of potential syntactical and structural

[1] SOMF: http://www.sparxsystems.com/somf

heterogeneities [WKK+10] between the compared profiles and the conservative matching strategy applied for the syntactical comparison.

Case-study design

To conduct this study, the source code of Java libraries that exploit annotations is required. Furthermore, we require existing profiles that claim to support the selected Java libraries on the model level. To accomplish an appropriate coverage of different scenarios, the selected Java libraries ideally comprise different intrinsic properties with respect to the design complexity and exploited language elements. Unfortunately, profiles specific to Java libraries in reasonable quality are rarely available. Consequently, in the process of selecting the Java libraries for this study, we were also confronted with the actual offering of modeling tools. IBM's Rational Software Architect (RSA) is obviously close to CAML and offers several profiles of well-known Java libraries mainly for code generation purposes. Thus, we conducted this study by relying on profiles of RSA in version 8.5.1. We selected four established Java libraries for which the source code is available and a corresponding RSA profile in the same major version is offered: JPA, EJB, Struts[2] and Hibernate[3]. RSA offers them in a UML standard-compliant way. Consequently, we could directly compare them without an intermediate conversion step.

Case-study measures

The measures used in the case study are based on model comparison techniques [KDRPP09]. Thus, we are interested in equivalent elements that reside in our generated profiles and in the RSA profiles, elements that reside in both solutions but still show differences in their features, and elements that are only available in one of the compared solutions. The measures for estimating the quality of the generated profiles are collected in a two-step matching process. While the first step automatically collects measures based on syntactic model comparison, the second step relies on manually processing differences produced in the first step to deal with semantic aspects.

In the syntactic model comparison, we compute the following measures for certain model elements. To determine element correspondences, we employ as matching heuristic name equivalence, *i. e.,* only if two elements have completely the same name, they are considered to be corresponding. If an element has no name, such as the `Extension` relationship, it is considered that the elements are corresponding if their source and target elements correspond. Finally, fine grained comparison of the feature values for the given elements is performed. Regarding model elements, we set the focus on (*i*) `Stereotypes` that are common to both and unique either to CAML or RSA, (*ii*) differences regarding the `Extensions` of common `Stereotypes`, and (*iii*) differences regarding the `Properties` such `Stereotypes` cover.

In the semantic model comparison, we take the syntactical differences as input and aim at finding additional correspondences between elements which are hardly explored by a pure syntactic comparison due to the conservative matching strategy. We investigate unmatched elements, especially stereotypes, in our generated profiles and in the RSA profiles, and reason about possible element correspondences beyond String equivalences. Finally, in the semantic

[2]Struts: http://struts.apache.org

[3]Hibernate: http://hibernate.org/orm

processing, we further evaluate the correspondences found in the first phase due to the potential syntactical and structural heterogeneities.

Results

We now present the results of applying JUMP to the selected Java libraries and compare them to the profiles offered by RSA. The absolute number of generated stereotypes by JUMP and the provided ones by RSA are depicted in Figure 7.1.

Figure 7.2 summarizes (*i*) the number of stereotypes generated by JUMP but not covered by the RSA profiles, (*ii*) the number of stereotypes that are exclusively covered by the RSA profiles, and (*iii*) the number of stereotypes that are common to both. These results include correspondences between stereotypes detected throughout the syntactic and semantic comparison. For instance, the EJB profile of RSA covers stereotypes that refer to the `@Local` and `@Remote` annotations of

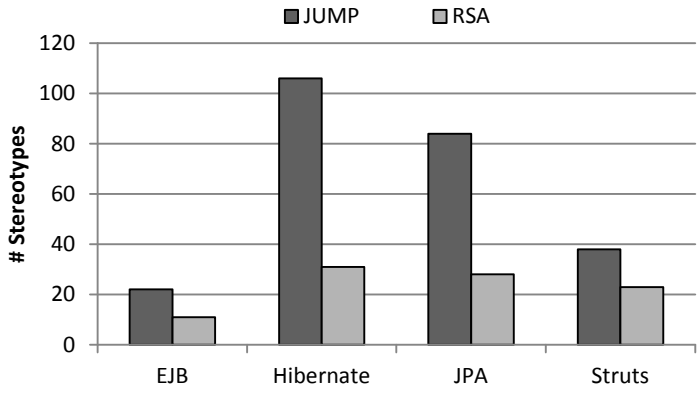

Figure 7.1: Absolute number of `Stereotypes`

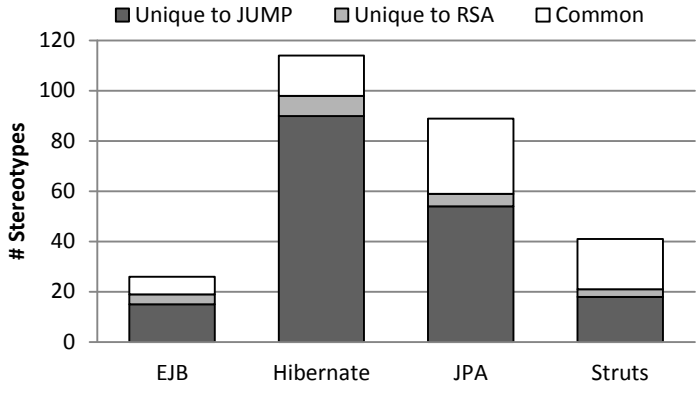

Figure 7.2: Comparison of `Stereotypes`

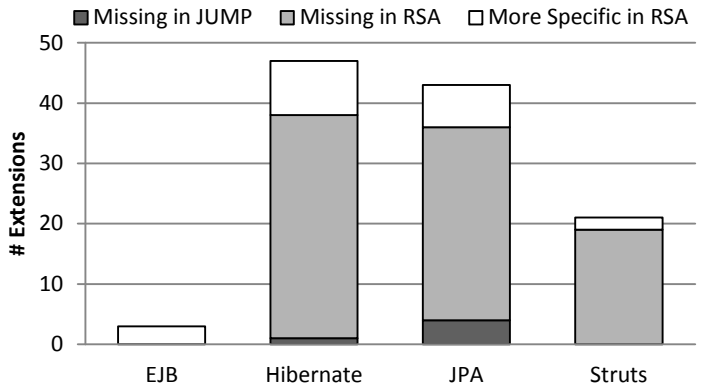

Figure 7.3: Comparison of `Extensions`

the EJB library, though their signature additionally contains the substring "`Interface`". Another example refers to the class `QueryHint` in the JPA profile of RSA, which is in fact an annotation type in the JPA library. In our solution, the `QueryHint` is represented by a stereotype even though it is also valid to use a class instead, because the `QueryHint` cannot actually be applied, but can rather only be used inside of another annotation. Although some stereotypes in the set of common ones show differences regarding the meta-classes they extend, we granted them to be equal if the extended meta-classes are related by a generalization relationship. We encountered this case in the EJB and the JPA library with respect to extensions of the meta-classes `Type` and `Class`. Stereotypes generated by CAML extend the more general meta-class `Type` because the scope of Java's element type `Type` also covers `Enumeration`, `Interface` and `AnnotationType` in addition to `Class`.

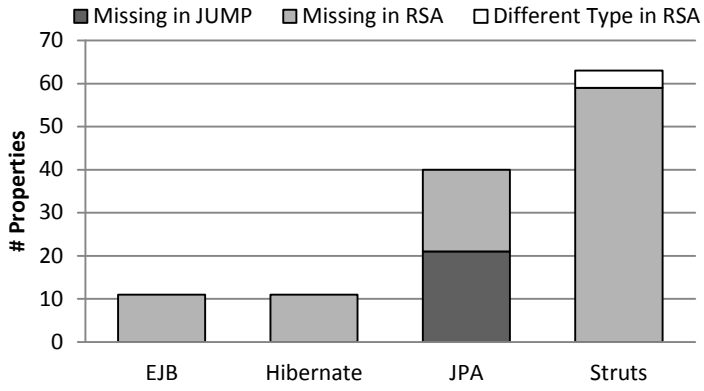

Figure 7.4: Comparison of `Properties`

The comparison regarding extensions of stereotypes common to both Jump and RSA is summarized in Figure 7.3. In a few cases, the RSA profiles comprise extensions to the UML meta-class `Association` to allow stereotypes on associations between elements rather than on properties contained by associations. Although both modeling variants are valid, we adhere to the second one as it is more accurate with respect to the target specifications of the original annotation type declarations.

Finally, in Figure 7.4, the differences regarding the properties of common stereotypes are presented. Except for the JPA profile, we cover all stereotype properties of the RSA profiles. Consequently, our profiles are more complete. The main reason for missing properties in our JPA profile seems to be that RSA provides additional properties for code generation purposes, but these properties are not covered by the JPA library.

Discussion

In this study, we have demonstrated that automatically generated UML profiles from Java libraries comprise a more comprehensive set of stereotypes and features compared to profiles used in practice for the purpose of supporting such libraries. Clearly, the purpose of the developed profiles plays an important role. From a forward engineering perspective, one may argue that the set of stereotypes, which is actually supported by the accompanying code generators is reasonable to capture on the model level. In fact, RSA offers code generation capabilities specific to the profiles we have evaluated in this study. However, for unsupported annotations, which have no corresponding stereotypes, code generators may only produce program code by conventions without allowing engineers to intervene in this generation process at the modeling level. From a reverse engineering perspective, we would lose relevant information on the model level if offered profiles provide less capabilities compared to the programming level, which is, however, the case for RSA profiles. Hence, with a fully automated approach, the quality of current profiles can be improved by providing more complete stereotypes that precisely capture the intention of the original annotation types in terms of target definitions, member declarations and return values of such members.

Threats to validity

There are two main threats that may jeopardize the internal validity of this study. First, we consider only profiles from RSA. The main reason for this procedure is that RSA applies a similar approach as Jump and offers specific UML profiles for Java libraries. Furthermore, RSA offers standard-compliant UML profiles that conform to the same UML 2 metamodel implementation as used in CAML. Second, it may be possible that we missed correspondences between elements of the profiles involved in the study. Several kinds of heterogeneities [WKK+10] exist that are real challenges for model matching algorithms and, thus, may affect the results of our study. However, by applying a two-step matching process which includes a syntactic as well as semantic comparison phase, we tried to minimize the possibility of missing correspondences as a result of different naming conventions and modeling styles. While in the first phase we used a quite conservative matching strategy to avoid false positives, we applied a rather liberal strategy in the second phase to avoid losing potential correspondences. Concerning external validity, CAML

sets the focus on Java annotations. Many libraries embrace them and real-world cases provide validity for annotated Java code [PBMH12]. However, we cannot claim any results outside of Java.

7.2.2 Effectiveness of model patching

To evaluate model patching, we investigate the *Java2UML* model transformation for which several well-documented evolutions were performed in the context of the ARTIST project. The revised transformations serve as basis to generate patch transformations. Both of them are executed for propagating the respective transformation changes to output models. They are compared with the goal to answer the following research question.

Quality – Research question 2: *Is a patch transformation equally effective as a revised transformation for which it was generated?*

In order to answer this research question, we briefly introduce the main artifacts involved in the case study and specify the measures based on which the comparison is conducted. Thereafter, we discuss the results of our study. In contrast to the previous case study on the quality of automatically generated UML profiles, a syntactical matching process for comparing the produced output models of the patch transformations and revised transformations is sufficient as syntactical and structural heterogeneities are not expected.

Case-study design

To conduct this case study, input models for the *Java2UML* transformation are required. We selected the PetApp and a framework that was of high interest in the context of the ARTIST framework, EclipseLink[4]. To generate the respective Java models for the PetApp and EclipseLink, we employed MoDisco. We selected six different revisions of the Java2UML transformation that have been performed throughout its development to cover the presented change types and the core effects of patch transformations. Based on these revisions, we generated the corresponding patch transformations. In a first step, we executed both the revised transformations and the generated patch transformations. The produced output models of the transformations are the basis to investigate the effectiveness of patch transformations compared to their corresponding revised transformations. Then, in a second step, we passed the respective pairs of output models to EMF Compare for automating the comparison task. Clearly, the diff model computed by EMF Compare needs to be empty to show that the produced output models are equal. It is important to note that the element identifiers can be different in the output models as patch transformations preserve them while they are newly produced if revised out-place transformations are re-executed.

Case-study measures

The measures used in the case study are once again based on syntactic model comparison techniques. Also in this case study, we employ name equivalence to determine whether two

[4]EclipseLink: www.eclipse.org/eclipselink

elements correspond to each other. Elements without a name are deemed to equivalent if their source and target elements correspond to each other. In contrast to the previous quality evaluation, we do not focus on particular model elements in the comparison. Instead the emphasis is placed on the equality of the output models produced by a revised transformation and the corresponding patch transformation.

Results and discussion

Considering the output models of the revised transformations and the patch transformations, their comparison shows that our approach produces effective results. In fact, the updates of our patch transformations to the output models reflect exactly the intended effects of the revisions performed to the original transformation. Clearly, as patch transformations only update the output models based on an in-place execution strategy, identifiers of existing elements and possible manual changes to elements that need not to be patched are preserved. As a result, the benefit of model patching to guarantee a non-invasive update to the output models is in our case study always given.

Threats to validity

We focused on the *Java2UML* case and applied patch transformations on small to large models that represent real-world applications and frameworks. Concerning internal validity, we need to further explore different combinations of changes and investigate if they can be correctly detected and efficiently propagated. Concerning external validity, we cannot claim any results outside of our performed case study concerning other transformation languages or model transformations. The latter may be considered as subject to future work.

7.3 Performance evaluation

To report on the scalability of the CAML's reverse-engineering capabilities, we measured the execution times of applying the *JavaCode2UMLProfile* and *JavaCode2ProfiledUML* transformations to several libraries used in practice and real-world applications. Also, we report on the speed-up that can be expected by applying model patching instead of completely re-executing evolved transformation for propagating transformation changes to existing output models. Both performance studies were conducted in Eclipse Luna 4.4.2 with Java 1.8 on commodity hardware: Intel Core i5-2520M CPU, 2.50 GHz, 8,00 GB RAM, Windows 7 Professional 64 Bit.

7.3.1 Scalability of UML profile and profiled UML model generation

As some existing Java libraries and applications are relatively large, the goal of this performance evaluation is to investigate whether CAML's reverse-engineering capabilities scales for those libraries and applications. In particular, we aim to answer the following research question.

Performance – Research question 1: *Do CAML's reverse-engineering capabilities scale for Java libraries and applications?*

In order to answer this research question, CAML's transformation chains for UML-based reverse engineering from Java artifacts were executed on a variety of libraries and applications. The rationale behind our selection of libraries and applications is to consider small-sized to large-sized libraries and applications with varying number of declared and applied stereotypes. We collected all measured execution times along with some characteristics of the input and output models.

Library	Size of input / output model	Applied stereotypes	Execution time in sec
EJB	10K / 1.5K	32	1.302
JPA	20K / 4K	84	2.165
Objectify	40K / 0,6K	20	1.442
Struts	90K / 2.5K	38	3.672
Hibernate	300K / 5K	108	12.042
Spring[1]	500K / 3K	63	9.463
EclipseLink	700K / 6K	127	19.193

[1] Spring: http://projects.spring.io/spring-framework

Table 7.4: Performance measures: UML profile generation

Application	Size of input / output model	Declared stereotypes	Execution time in sec
EJB	10K / 0.6K	5 (1 Profile)	1.647
Petstore (PS)[1]	10K / 1.5K	287 (12 Profiles)	3.977
DEWS-core[2]	30K / 3K	253 (2 Profiles)	2.179
Struts	90K / 20K	753 (2 Profiles)	8.447
Findbugs[3]	100K / 50K	1808 (3 Profiles)	22.267
Spring	500K / 90K	7973 (3 Profiles)	50.909
EclipseLink	700K / 200K	7117 (3 Profiles)	177.978

[1] Petstore: http://oracle.com/technetwork/java/index-136650.html
[2] DEWS-core: A component of the Distant Early Warning System (DEWS), a use case of the ARTIST project
[3] Findbugs: http://findbugs.sourceforge.net

Table 7.5: Performance measures: Profiled UML model generation

Results

Tables 7.4 and 7.5 summarize our obtained results. We focus on the involved artifacts and their characteristics mainly for the purpose of providing an explanation for the increasing execution times.

1. The number of *elements* in the intermediate Java model, *i. e.,* the input of the transformations, and the produced UML profile / model, *i. e.,* the output of the transformations
2. The number of *declared* and *applied stereotypes*
3. The measured *execution times*

Two results are accompanied with scatter plots, see Figures 7.5 and 7.6, which show the ratio of model size and execution time for UML profile generation and profiled UML model generation,

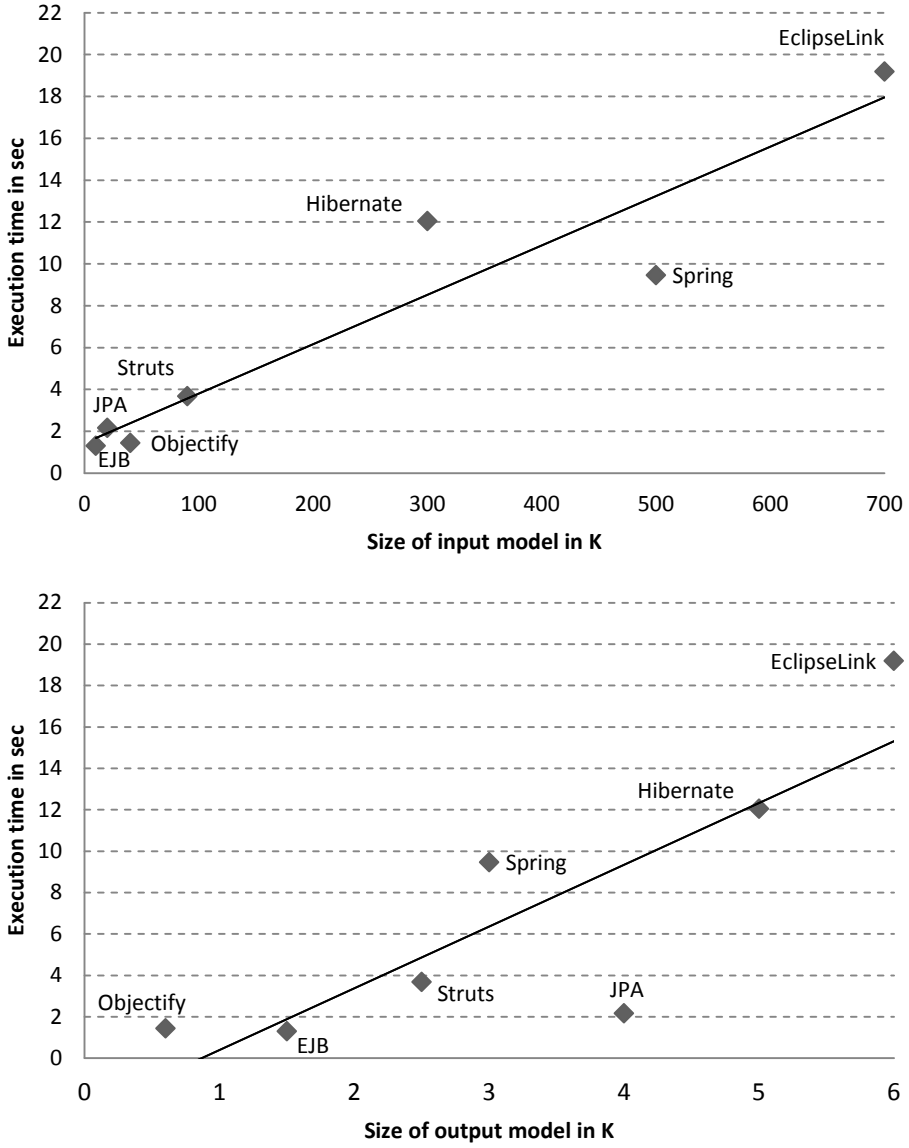

Figure 7.5: Ratio of model size and execution time for UML profile generation

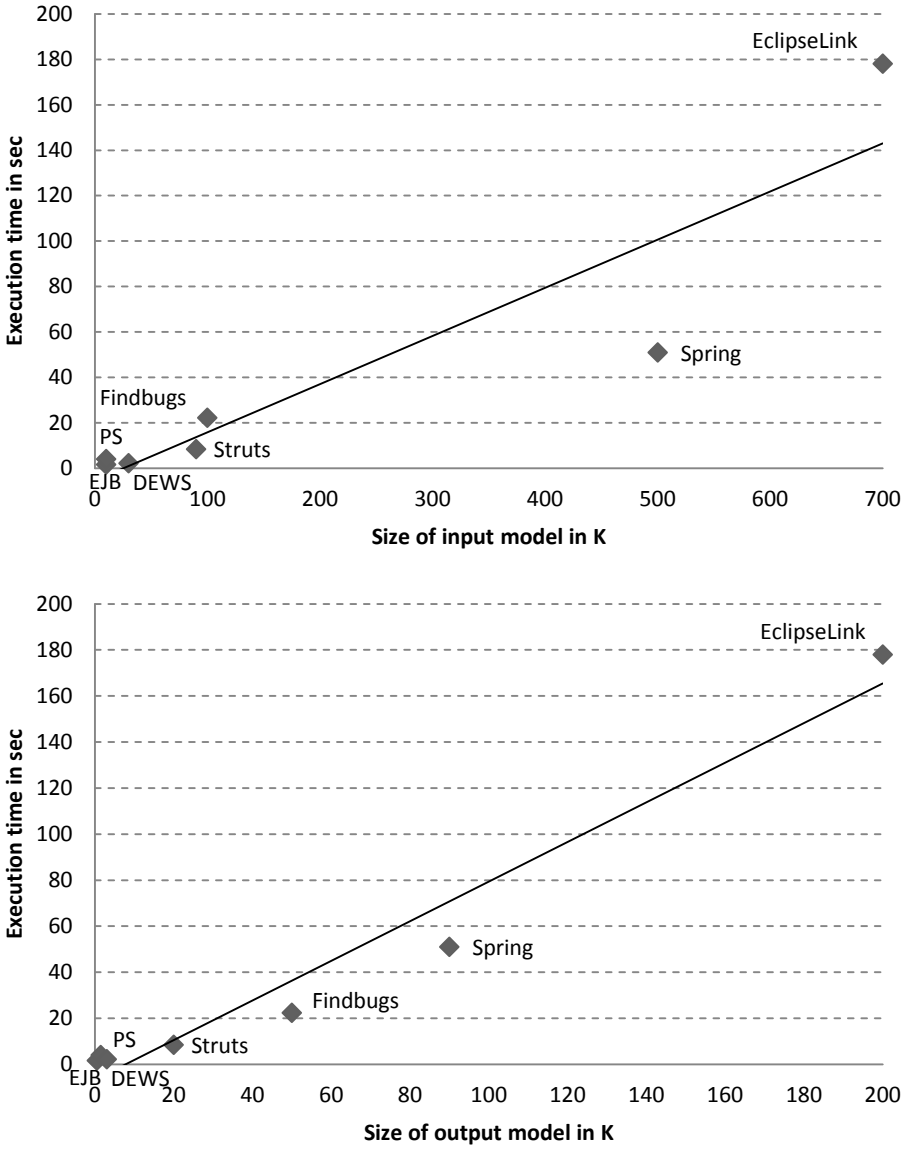

Figure 7.6: Ratio of model size and execution time for profiled UML model generation

respectively. The scatter plots include a linear regression curve to show the trend of increasing execution time with respect to growing model size by considering both input and output model.

Discussion

Clearly, the size of the input models passed to the transformations as well as the size of the produced output models has a strong impact on the execution time of the CAML tool as they are traversed throughout the generation of profiles and profiled models. Regarding the UML

profile generation, the number of generated stereotypes is another main factor that impacts on the execution time. Generally, the more stereotypes are generated, the more extensions to UML meta-classes and features of these stereotypes need to be created. As a result, the number of produced stereotypes has a strong impact on the size of the generated UML profile. For instance, even though the JPA is compared to Objectify smaller in size, the execution time is higher because a lot more transformation rules are applied when considering the number of declared stereotypes. Similarly, the execution time of generating a profile for EclipseLink is twice as high as it is for Spring, which can be explained by the major difference of the number of declared stereotypes, *i. e.,* 127 vs. 63.

Regarding the generation of profiled UML models, the more stereotypes from different UML profiles are applied, the higher is the execution time. Similarly, the number of applied stereotypes and their respective profiles influences the execution time. For instance, in the Petstore (PS) application, stereotypes are applied from 12 different profiles, which explains the higher execution time compared to DEWS-Core, even though the input model of the latter is larger in size.

Considering the measured results for Struts and Findbugs show the strong impact of the size of the generated output model and the number of applied stereotypes on the execution time. Even though the input models of Struts and Findbugs are almost similar in size, the execution time of the latter is more than twice as high as of the former. This can be explained by the fact that the output model and the number of applied stereotypes in the case of Findbugs is double in size compared to Struts.

If the size of both input and output models is as large as it is in case of EclipseLink, the high memory consumption and as a result the excessive use of garbage collection may also be an additional factor that influences execution time. This is one reason why the execution time for EclipseLink is above the overall estimated trend, see Figure 7.6.

Considering the execution time of the profiled UML model generation, it is generally higher compared to profiles because the class structure of the former is much larger in size compared to the latter. For instance, considering EclipseLink and the number of generated stereotypes compared to classes the factor is almost 30.

7.3.2 Speed-up of model patching

To evaluate the performance of model patching, we measured the execution times of all the revised and patch transformations involved in the quality evaluation (see Section 7.2.2) with the aim to answer the research question as follows.

Performance – Research question 2: *How is the speed-up of executing a patch transformation compared to re-executing the respective revised transformation?*

In order to answer this research question, we collected the respective execution times obtained by conducting the quality evaluation of model patching. In the course of the quality evaluation, we selected a small-sized application and a large-sized framework mainly for reasons of reusing them also in the context of a performance evaluation.

Results

Table 7.6 summarizes the collected execution times of the revised and patch transformations. It also provides the speed-up of model patching compared to a complete re-execution of revised transformations. We placed the emphasis on the involved artifacts and their size as the latter obviously influences the execution times.

	Reference Appliction (> 1000 Elements)			**EclipseLink (> 100.000 Elements)**		
	Transformation		Speed-Up	Transformation		Speed-Up
Change Type	Revised	Patch		Revised	Patch	
MatchedRuleAddition	0,076	0,067	1,134	6,698	4,766	1,405
MatchedRuleDeletion	0,057	0,014	4,071	6,114	1,417	4,315
FilterModification	0,066	0,079	0,835	5,854	4,987	1,174
OutPatternElementAddition	0,066	0,058	1,138	7,531	3,149	2,392
BindingAddition	0,063	0,010	6,300	6,687	0,104	64,298
BindingDeletion	0,064	0,009	7,111	6,882	0,058	118,655

Table 7.6: Re-execution vs. patch execution (time measures in sec.)

Discussion

Considering the runtime efficiency of model patching, generally, the generated patch transformations execute less rules compared to the revised transformations. The number of required rules of a patch transformation slightly varies depending on the considered change type. In our case study, one up to six rules were required to build-up a patch transformation. Clearly, this number increases if certain rules need to be re-executed as a result of revising another rule, *e. g.,* inter-rule dependencies. Such dependencies lead to patch transformations covering not only revised rules but also rules that are affected by the performed revisions, *e. g.,* bindings, as shown in Listing 6.3. Finally, the adaptation of the trace model requires also additional rules in the patch transformation, *e. g.,* when matched rules are added. While the number of rules has certainly an impact on the execution time of patch transformations, our results show that their need to traverse and query the traces of the original transformation produces an overhead compared to the revised transformations. Still, in our case study, for the majority of patch transformations a speed-up can be observed, as summarized in Table 7.6. In fact, only in one case, such a speed-up could not be achieved as the input and output models of the reference application are rather small and the inferred patch transformation for this case is more complex compared to other ones. However, the benefit of patch transformations to guarantee a non-invasive update to the output models is in our case study always given.

7.4 Practical relevance

To demonstrate the practical relevance of the proposed architecture style, we report on our experiences of applying it in the context of an application modernization to the cloud that involves both a reverse engineering and a forward engineering process. In doing so, we elaborate on the application scenario motivated in Section 1.4, where a change of the data access platform is discussed and provide insights into the transition of a JPA-based solution to an Objectify-based solution. The Google Cloud Platform is considered as the deployment target of the modernized PetApp. It allows entities to be retrieved not only by plain service classes but also via a REST-based client, which basically resembles Google's Cloud Endpoints service[5] (*cf. e. g.,* [EIG+15]).

To carry out the transition towards the cloud-based solution we apply Kazman's "horseshoe" in light of model-based engineering and cloud-oriented software modernization. As a result of applying advanced techniques of model-based engineering, we reverse engineered a platform and environment-independent domain model in a quality that allowed us to directly refine it towards the target platform and environment. The refined domain model is passed to a code generation facility capable to automatically produce the complete cloud-based solution. In this modernization scenario we particularly emphasize the benefit of annotation-based modeling to improve the quality of the reverse engineered domain model and to generate model artifacts that could have hardly been generated otherwise. Based on our insights gained from the outlined modernization scenario, we aim to answer the research question as follows.

Practical relevance – Research question 1: *How can engineers benefit from CAML's architecture style by applying it in a modernization scenario?*

In order to answer this research question, we apply the modernization process introduced in Figure 3.3. Most of the activities in this process are supported by dedicated transformations. They come with CAML. To *get an understanding* of the PetApp's domain model, several transformations are applied to the application code and intermediate models to ultimately obtain a UML-based representation of the domain model. As a target cloud environment, we *select* Google's Cloud Platform, which obviously influences the *adaptations* we must perform. In our modernization scenario, the PetApp domain model is adapted towards an Objectify-based solution. In addition, a REST-based client is provided to allow accessing entities via service endpoints. To *prepare the deployment* of the modernized PetApp CAML's cloud library and profile is employed. As a result, deployment model of the PetApp can directly be refined towards the Google Cloud Platform by applying stereotypes of the respective profile. The deployment model basically describes the desired state of the application provisioning, which is *executed* in the course of the last activity of the modernization process. This activity is supported by a transformation of the deployment towards a TOSCA topology template that in turn is passed to a TOSCA-compliant provisioning engine. A high-level overview of the modernization roadmap is given in Figure 7.7. It relates all the code, model, and transformation artifacts involved in our modernization scenario.

[5] Google Cloud Endpoints: https://cloud.google.com/appengine/docs/java/endpoints/

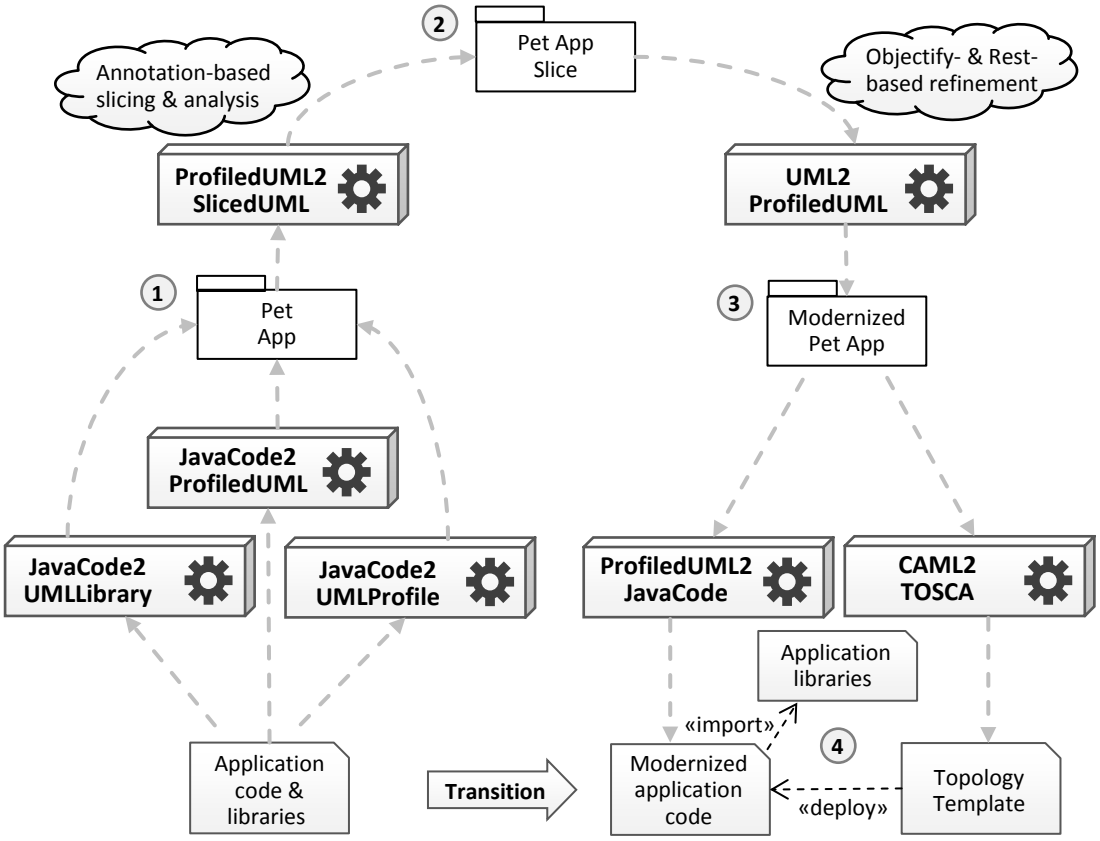

Figure 7.7: Modernization roadmap to the cloud

7.4.1 Reverse engineering process

Considering the first step in the reverse engineering process (see ① of Figure 7.7), a UML model that captures the complete PetApp from a structural perspective is generated. Moreover, Java libraries are reverse engineered into corresponding UML libraries and UML profiles as they enable succeeding transformations to exploit information that is specific for the current environment or platform. They can be exploited to provide abstractions over the initially generated models that are usually specific to a platform, *e. g.,* Java and libraries provided for it such as JPA. Moreover, we have also discussed their usefulness for improving the quality of a reverse engineered domain model.

Listing 7.1: Relationship between `Category` and `Product`

```
/* Category */
package domain;

import javax.persistence.*;
import java.util.*;

@Entity(name = "Category")
public class Category {
```

```
  @Id private Long id;
  @OneToMany(mappedBy = "category", cascade = CascadeType.ALL) private List<
      ↪Product> products;
}

/* Product */
package domain;

import javax.persistence.*;

@Entity(name = "Product")
public class Product {

  @Id private Long id;
  @ManyToOne private Category category;
}

/* Catalog Service */
package service;

import javax.persistence.*;
import java.util.*;
import domain.*;

public class CatalogService

  private EntityManager em;

  public Category findCategory(Long categoryId) {
    if (categoryId == null) throw new ValidationException("Invalid␣category␣id");
    return em.find(Category.class, categoryId);
  }

  public Product findProduct(Long productId) {
    if (productId == null) throw new ValidationException("Invalid␣product␣id");
    return em.find(Product.class, productId);
  }
}
```

For instance, the profiled model represented in Figure 7.8 captures the two entities and the service class to retrieve them of Listing 7.1. This reverse-engineered model is specific to the Java platform as the `products` property and both methods `findAllCategories` and `findAllProducts` are of type `java.util.List`. Furthermore, it is specific to JPA because all the applied stereotypes refer to annotations types captured by the JPA.

In the second step of the reverse engineering process (see ② of Figure 7.7), the domain model is sliced from the previously produced model of the PetApp. All model elements that do not denote entities are withdrawn, thereby only persistable entities are retained. Furthermore, Java-specific types are turned into corresponding UML concepts. Considering the slicing part, we have implemented an annotation-based slicer where the point of interest is a set of stereotypes, *e. g.,* `Entity` and `Embeddable`. They are captured by a slicing criterion according to which model elements are either withdrawn or retained. In case of our modernization scenario, model elements to which at least one of the stereotypes is applied are considered as part of the computed model slice. Furthermore, structural features comprised by model elements of the slice are also completely retained. The computed model slice is presented in Figure 7.9 (see Figure B.3 of

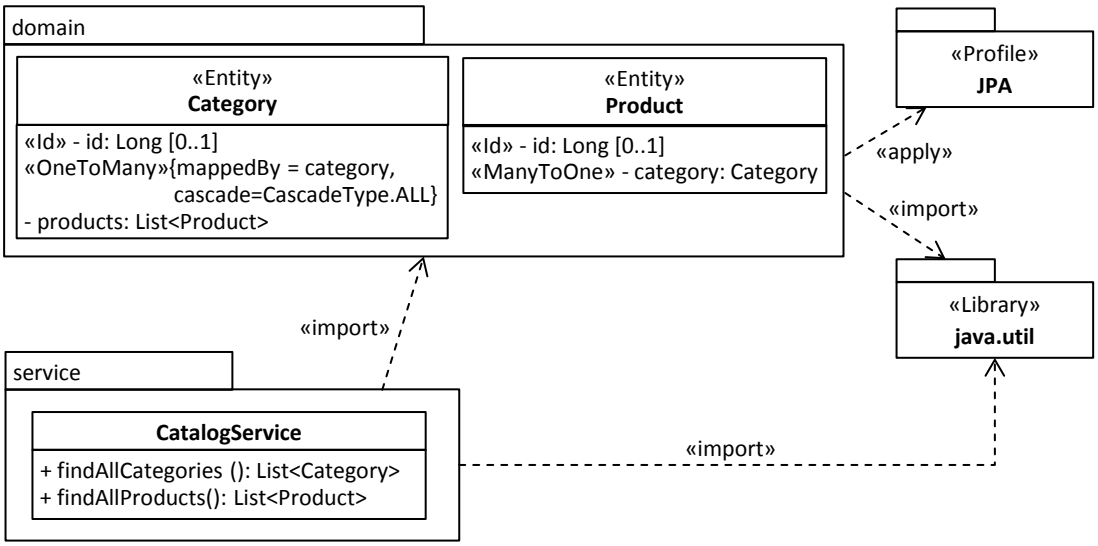

Figure 7.8: Reverse engineered PetApp model specific to the Java and JPA platform

Appendix B for the complete domain model of the PetApp). To improve the quality of the sliced platform-independent domain model, stereotypes applied to persistable elements are analyzed before they are withdrawn.

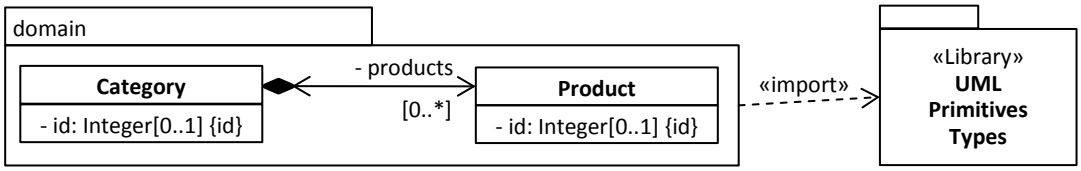

Figure 7.9: Platform-independent domain model of the PetApp

For instance, the composition relationship between `Category` and `Product` of the sliced domain model has been generated on the basis of the `@OneToMany` stereotype applied to the `products` property of the `Category` class in the platform-specific model (see Figure 7.8). The selected `CascadeType` allows the composition relationship to be derived where its member ends are determined by the property to which the `@OneToMany` stereotype is applied and the property assigned to the `mappedBy` element. Without this detailed consideration of the `@OneToMany` stereotype, we would at best be able to generate properties with the respective types, *i. e.,* `Category` and `Product`. Concerning Java-specific types, we mainly turned collection types of properties into multi-valued properties and primitive types known from Java into primitive types offered by UML.

7.4.2 Forward engineering process

Based on the sliced domain model, we started the refinement towards the target platform and environment, which is part of the forward engineering process. In fact, we produced two different platform-specific models in the course of this process, one of which capturing the "front-end". It describes the API that is used by the REST-based clients. They manipulate entity instances of the domain model. The second model refers to the "back-end". It is specific to Objectify. The role of Objectify is to manage the entity instances in cloud datastore of the Google Cloud Platform. Generally, client requests are delegated from the front-end to the back-end. In addition to the front-end and back-end models, we also created the deployment model for the PetApp. We started from an environment-independent deployment model and refined it towards the Google Cloud Platform by applying the respective profile.

Generating front-end and back-end model

How the front-end and back-end model is generated from the platform-independent domain model is outlined in Figure 7.10.

It shows the process and the main models, profiles, and libraries involved in the refinement. The *UML2ProfiledUML* artifact in Figure 7.7 is in fact realized by several in-place transformations, each of which supporting an activity of the refinement process. In the first step, service interfaces and service methods are generated for the entities of the domain model to create, read, update, and delete them. For instance, the `CategoryService` interface in Figures 7.11 and 7.12 is a result of this first step.

Depending on whether the front-end CESM or the back-end CESM is generated, getter/setter methods are generated either in a standard way or specific for Objectify. In case of generating the back-end CESM, the EIDM is annotated with stereotypes of the Objectify profile and the Jackson profile before concrete service classes for the service interfaces are generated. For instance, an explicit composition relationship between two entities where the contained one cannot be identified by a dedicated property indicates that the latter entity needs to be embedded by the former entity. This embedding of entities can be expressed in Objectify via the `Embed` stereotype, *e. g.,* the `Address` domain class in Figure 7.11. Moreover, the `Id` stereotype is required for generating the behavior of service methods where the identifier of an entity needs to be accessed, *e. g.,* the method `findCategory(long entityId)` of the service class `CategoryService` in Figure 7.11. In cases of cyclic or bi-directional relationships between entities the stereotype `JsonIdentityInfo` is required for instructing the serialization and de-serialization process as part of a REST-based solution to turn bi-directional relationships into cross-references of a tree-based structure as determined by JSON. Annotating the service interfaces with stereotypes of the JAX-RS[6] profile is a prerequisite for exposing them to clients. JAX-RS is a Java API for RESTful web services. Moreover, these stereotypes are required to generate the behavior of the service proxies as part of the front-end CESM. For instance, to correctly deal with the `HttpResponse` object of the `findCategory(long entityId)` method in the `CategoryServiceProxy`, the `Path` stereotypes applied to the service interfaces and service operations, and the stereotypes determining the REST method need to be accessed.

[6]JAX-RS: https://jax-rs-spec.java.net

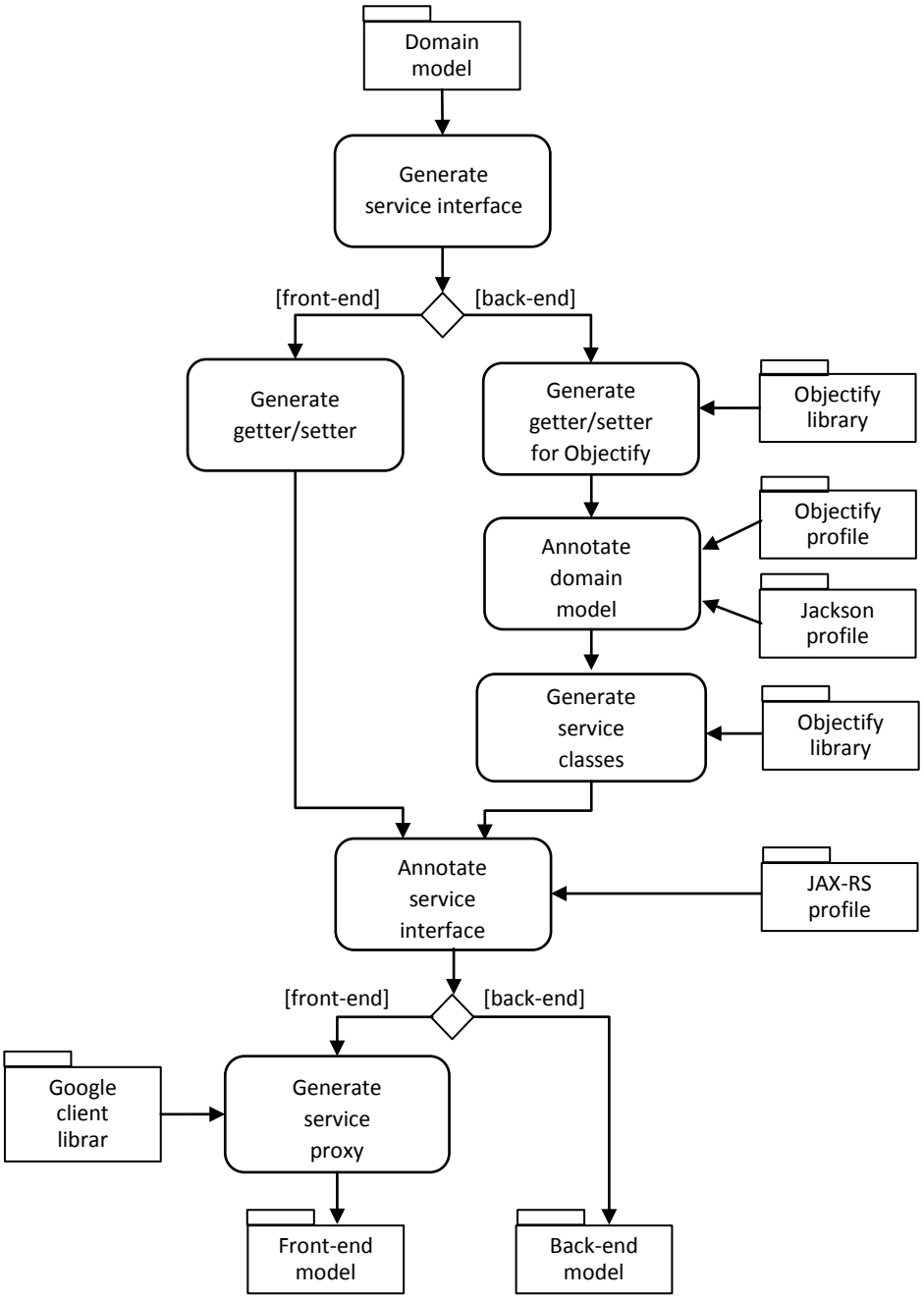

Figure 7.10: Refinement of EIDM towards CESM

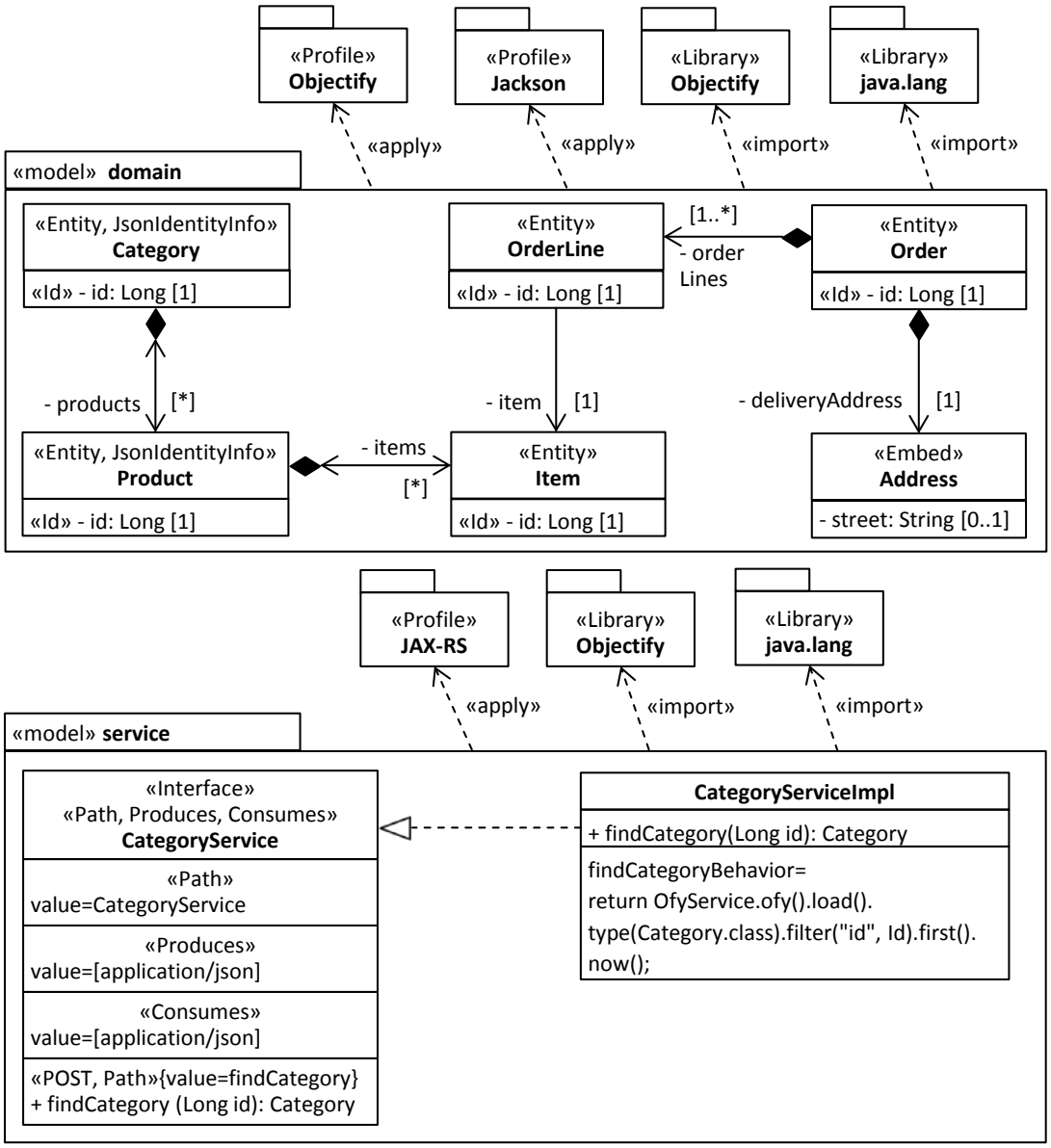

Figure 7.11: Modernized back-end model specific to the target cloud environment

Having generated both the front-end CESM and the back-end CESM as a result of the above refinement steps, the corresponding Java code can be produced from them. For that reason, we have adapted the Java-based model transformer provided by Obeo Network mainly to support (*i*) `Stereotypes` for which the corresponding annotations need to be produced, (*ii*) `OpaqueBehaviors` that is used to generate method bodies, and (*iii*) `ElementImports` as they indicate the required import statements of the Java code. Listing 7.2 shows the generated back-end Java code for the `Category` domain class whereas in Listing 7.3 its front-end Java code is given.

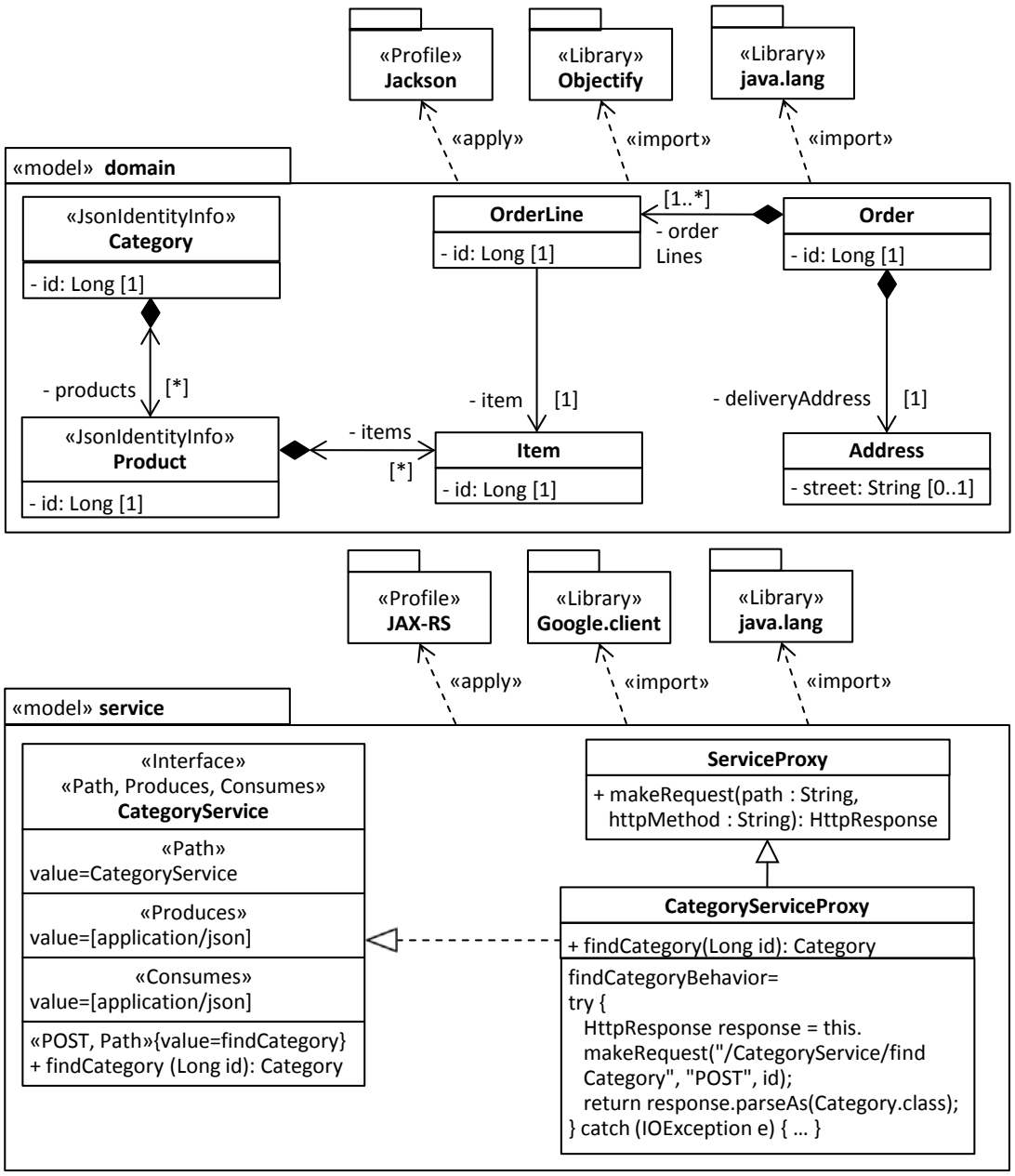

Figure 7.12: Modernized front-end model specific to the target cloud environment

Listing 7.2: Generated back-end code for `Category`

```
/* Category */
package domain;

import com.googlecode.objectify.*;
import com.fasterxml.jackson.annotation.*;
```

```
import java.util.*;
import domain.Product;

@Entity
@JsonIdentityInfo(generator = IntSequenceGenerator.class,property = "@id")
public class Category {

  @Id private Long id;
  private List<Ref<Product>> products;
}

/* CategoryService */
package service

import javax.ws.rs.*
import domain.Category;

@Path(value = "CategoryService")
@Produces(value = {"application/json"})
@Consumes(value = {"application/json"})
public interface CategoryService {

  @POST
  @Path(value = "findCategory")
  public Category findCategory(long id);
}

/* CategoryServiceImpl */
package service

import domain.Category;
import service.CategoryService;
import service.OfyService;

public class CategoryServiceImpl implements CategoryService {

  public Category findCategory(long id) {
    return OfyService.ofy().load().type(Category.class).filter("id", id).first().
        ↪now();
  }
}
```

Listing 7.3: Generated front-end code for `Category`

```
/* Category */
package domain;

import com.fasterxml.jackson.annotation.*;
import java.util.*;
import domain.Product;

@JsonIdentityInfo(generator = IntSequenceGenerator.class,property = "@id")
public class Category {

  private Long id;
  private List<Product> products;
}

/* CategoryService */
package service
```

```
import javax.ws.rs.*
import domain.Category;

@Path(value = "CategoryService")
@Produces(value = {"application/json"})
@Consumes(value = {"application/json"})
public interface CategoryService {

  @POST
  @Path(value = "findCategory")
  public Category findCategory(long id);
}

/* CategoryServiceProxy */
package proxy

import com.google.api.client.http.HttpResponse;
import java.io.IOException;
import domain.Category;
import proxy.ServiceProxy;
import service.CategoryService;

public class CategoryServiceProxy extends ServiceProxy implements CategoryService
    ↪ {

  public Category findCategory(long id) {
    try {
      HttpResponse response = this.makeRequest("/CategoryService/findCategory", "
          ↪POST", id);
        return response.parseAs(Category.class);
    } catch (IOException e) {
      // log error message
    }
    return null;
  }
}
```

Creating the deployment model

In addition to adapting the domain model of the PetApp towards Google's key-value cloud datastore, the components constituting the PetApp must be re-deployed on the cloud environment of Google. The latter has been selected as the deployment target. We assume that the PetApp's components are currently deployed on an on-premise environment. For that reason, we take the viewpoint of the application components and their deployment as depicted in Figure 7.13. It shows the three main components of the PetApp and the manifestation of them by deployable artifacts. The PetAdmin and PetStats components retrieve and manipulate data from the PetWeb component via the generated REST-based client API. Furthermore, a possible on-premise deployment model for the PetApp is presented. As expected, they are deployed on a Java-based platform and a relational DBMS, which are in turn deployed on nodes with certain (virtual) machine characteristics. The model elements of the PetApp on-premise deployment are instances of the custom types defined in the context of the component viewpoint and the web deployment library, respectively. Both the component model and the model library providing custom types for deploying web applications are imported by the PetApp on-premise deployment topology. Details about how the model library is actually defined are presented in Figure 2.2.

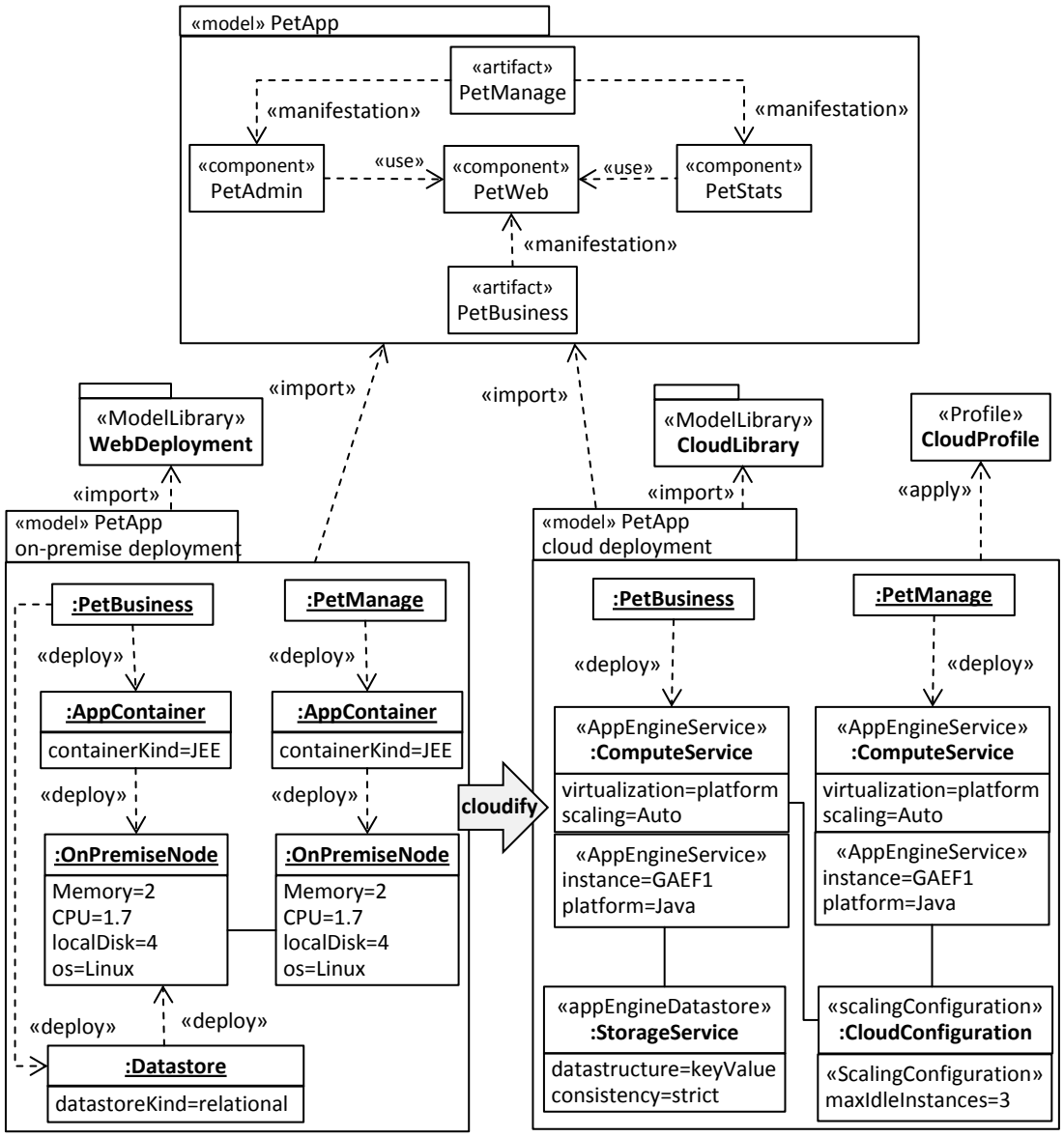

Figure 7.13: On-premise and cloud deployment model for PetApp

Now with the emergence of cloud services and the demand to exploit them, deployment models must be expressive enough to capture those services. This is exactly the idea of CAML's cloud library and profiles. It is precisely because CAML is realized in terms of lightweight extensions to UML. Its library and profiles are directly applicable to UML models and so to the modeled components of the PetApp and the respective deployable artifacts. Considering them in the context of PetApp's cloud deployment topology, they are deployed on automatically scaled compute services, whereas one of those compute services is capable to access the key-value cloud storage

for managing the application data in a strictly consistent way. The artifacts are directly deployed on the compute services because they operate at the platform-level. Google's App Engine service provides compute services with fully managed platforms. By applying CAML's cloud profile, environment-specific information can be expressed on the model level. The applied stereotypes basically enable the refinement of an environment-independent deployment model towards an environment-specific model. In case of our modernization scenario, stereotypes are applied to refine the deployment model towards concrete cloud services offered by Google. As a result, both the modeled compute services refer to a Java-based `AppEngineService` with `F1` instance types. The configuration attached to these compute services constrains the maximum number of idle

CSAR

Definitions

```
<NodeTemplate name="App Engine Service" type="ns2: AppEngineService">
 <Properties>
  <ns2:AESProperties xmlns:ns2="..." xmlns="…">
   <platform>Java</platform>
  </ns2:AESPropertiess>
 </Properties>
</NodeTemplate>

<NodeType name="AppEngineService">
 <DerivedFrom typeRef="ns1:GAEComputeService"/>
 <PropertiesDefinition element="tns:AESProperties"/>
  <Interfaces>
   <Interface name="http://docs.oasis-open.org/tosca/ns/interfaces/lifecycle">
    <Operation name="start"/>
    <Operation name="stop"/>
   </Interface>
  </Interfaces>
</NodeType>
```

Types

```
<xs:complexType name="tAESProperties">
 <xs:element default="Java" name="platform" type="PlatformKind"/>
</xs:complexType>

<xs:simpleType name="PlatformKind">
  <xs:restriction base="xs:string">
   <xs:enumeration value="Java"/>
   <xs:enumeration value="PHP"/>
   <xs:enumeration value="Python"/>
  </xs:restriction>
</xs:simpleType>

<xs:element name="AESProperties" type="tAESProperties"/>
```

Figure 7.14: CSAR for PetApp

compute services. The UML deployment model refined by CAML can now be translated into a functionally equivalent TOSCA topology template. This topology is enriched by Winery and packaged into a CSAR, which can be consumed by the OpenTOSCA container. An excerpt of the CSAR is depicted in Figure 7.14. It shows the definition of the `AppEngineService` type assigned to a template. Properties of the `AppEngineService` type are captured by an XML schema to support their validation when they are used.

7.4.3 Synopsis

We have shown the transition of a non-cloud software into a cloud software with a particular focus on modernizing the data access layer. As a result of the reverse engineering process we obtained an environment-independent domain model from which we generated environment-specific models for the back-end as well as the front-end of the REST-based solution hosted on the Google Cloud Platform, where Objectify manages the access to the cloud datastore. Table 7.7 gives some quantitative characteristics of the PetApp we dealt with in the model refinement and code generation.

Model element types	Number of model elements					
	PSM[1]	PIM[2]	CPSM[3]			
			Front-end		Back-end	
			reused	generated	reused	generated
Classes / Interfaces / Enumerations	39	9	9	13	-	22
Properties	157	49	40	5	-	40
Operations	263	-	85*	74*	-	153*
Annotations	287	-	-	78	-	95

[1]PSM …Platform-specific model
[2]PIM … Platform-independent model
[3]CPSM … Cloud platform-specific model
*including behavior

Table 7.7: Quantitative characteristics of the generated UML models

The PSM reflects its structural elements, whereas the PIM captures the essence of the entities without operations to access their properties as these operations may not fit the requirements of the target environment anyhow. Finally, the CPSM's produced for the front-end and the back-end represent the structure as well as the behavior of the cloud-based solution. We distinguish model elements that can be reused from the original implementation from the model elements that are generated as part of the forward engineering process. In fact, the domain classes of the original implementation can obviously be reused for the front-end. Service classes are newly generated as they need to be appropriately annotated and the service proxies have not existed in the original implementation. Regarding the back-end, all the artifacts are newly generated as the change from JPA to Objectify requires modifying the domain classes and the service classes to access

them. Finally, Table 7.8 summarizes the benefits of annotation-based modeling in the context of our modernization scenario. While the presented stereotypes were automatically applied in our modernization scenario by model transformations, other stereotypes may directly be applied by engineers in a manual refinement step. For instance, Objectify provides annotations to index properties of entities or cache retrieved entity instances. As all the annotations of Objectify are captured by respective stereotypes on the model level, the engineers have full control over such platform-specific decisions at any phase of the forward engineering process.

Profile	Stereotype	Benefit
Objectify	Entity	Indicates entities that need to be persisted Allows the entity registry to be generated
	Id	Indicates properties that idendity domain classes Allows the behavior of service classes to be generated
Jackson	JsonIdentityInfo	Allows cross-references to be produced
JAX-RS	Path	Allows the URL of the service request to be generated
	Post, Put, Delete	Indicates the employed REST method Allows the service request to be completed
CAML	AppEngineService, AppEngineDatastore, ScalingConfiguration	Allows the refinement towards a target cloud environment Allows generation of useful TOSCA topology template

Table 7.8: Benefits of annotated models in the modernization scenario

CHAPTER 8

Conclusion

In this thesis, we presented four major contributions to realize a UML-based architecture style for cloud application modeling. In the following, we summarize the contributions presented in this thesis along with some conclusions drawn from their evaluation. Thereafter, we give an outlook on possible future directions that build upon the research conducted in the course of this thesis.

8.1 Summary

Scientific contribution 1: Systematic review of cloud modeling languages. Several languages for modeling cloud applications have been proposed since 2010. As the cloud computing domain is highly diverse, those languages pursue different goals and provide a complementary set of modeling concepts to represent certain aspects of cloud applications. Existing cloud modeling concepts are in fact highly relevant for realizing cloud-specific extensions to UML. Thus, we systematically reviewed existing cloud modeling languages (CMLs). For that reason, we introduced a common classification and comparison framework not only to support cloud users in selecting the CML which fits the needs of their application scenario but also to investigate characteristics and concepts of current CMLs. The results of this systematic review and the investigation of major cloud environments provided the necessary basis to derive a core set of features inherent to the cloud computing domain. The presented cloud-specific extensions to UML are grounded in this set of features. Furthermore, the contributions presented in this thesis address some of the interesting findings of the conducted CML review. We observed that current CMLs pay only little attention to standard modeling languages and the exchange of models between them requires manual effort for replicating those models in the different languages. In the course of this thesis, we selected UML as the base language on top of which we developed cloud-specific modeling concepts and proposed an automated approach that allows engineers to combine standards emerged in the area of cloud computing and software engineering: TOSCA and UML. Another observation shows that existing CMLs have placed the greatest emphasis on modeling, generation, and provisioning of cloud applications and the least on analysis and refinement. In this thesis,

we exploited TOSCA-based tools for cloud application provisioning and placed emphasis on realizing flexible cloud-specific refinement support for high-level architecture models.

Scientific contribution 2: Cloud application modeling. In order to support architecture modeling for cloud applications, we presented the cloud application modeling language CAML, a collection of libraries and profiles realized on top of UML. CAML supports a flexible semi-automatic refinement process from high-level architecture models down to a concrete implementation of them. In this respect, we placed emphasis on the Java platform for which libraries and profiles can be generated automatically by a transformation chain of CAML toolset. In addition to platform-specific libraries and profiles, CAML provides a set of profiles that capture cloud services offered by modern cloud environments. They enable the refinement of environment-independent deployment configurations towards a selected target cloud environment. Environment-independent deployment configurations can be represented by means of CAML's cloud library, which provides common cloud modeling types. They are abstractions over services of the current major cloud environments, *i. e.,* Amazon AWS, Google Cloud Platform, and Microsoft Azure. Combining the library approach with the notion of platform and environment-specific UML profiles results in a powerful architecture style for cloud application modeling. In the evaluation of CAML, we investigated the quality of automatically generated UML profiles from Java libraries compared to existing UML profiles used in practice. Our findings show that automatically generated UML profiles comprise in general a more comprehensive set of stereotypes and features compared to currently existing profiles for Java libraries. We demonstrated the practical relevance of CAML for engineers by means of a modernization scenario to the cloud. A considerable part of the modernized application code and a complete TOSCA-based representation of the respective application deployment configuration could be automatically generated from architecture models expressed in UML and refined by CAML.

Scientific contribution 3: Cloud application provisioning. Deployment configurations based on CAML specify the desired state of the application provisioning. As automated cloud application provisioning and management is supported by TOSCA-compliant runtime containers such as OpenTOSCA, we elaborated an effective mapping between UML and TOSCA where CAML provides the cloud-specific features required to produce appropriately typed TOSCA topology templates. To automate the translation from UML to TOSCA, we implemented the Caml2Tosca model transformer. It is grounded in the proposed conceptual mapping between the two languages and leverages a framework for architecture modeling and application provisioning based on UML and TOSCA. It paves the way for a continuous cloud modeling support and allows engineers to carry out the provisioning of cloud applications. As the entry point to the framework is an architecture model refined by CAML, we investigated current industrial and open-source UML tools and found that they lack cloud-specific refinement support for deployment configurations. At the same time, they are capable of adopting CAML, which shows the practical value of Caml2Tosca for engineers.

Scientific contribution 4: Cloud model patching. As model transformations play an important role to automate refinement and translation processes, we proposed an approach for effectively maintaining transformations and their produced artifacts. In particular, we placed emphasis on the co-evolution of existing models with changes to transformations. To accomplish this co-evolution, we proposed model patching. It exploits the benefits of incremental transformation execution to propagate transformation changes to existing models, such that previous transformation results are updated instead of discarded. As a result, model patching is non-invasive with respect to change propagation. Furthermore, it is usually faster compared to a complete transformation re-execution because model patches for a typical evolution scenario comprise only a small portion of the rules covered by an evolved transformation. The evaluation of model patching shows that it is equally effective compared to completely re-executing evolved transformations. In general, our findings show that the updates to the output models by model patching reflect exactly the intended effects of the changes in the evolved transformation. When comparing the execution times of the model patching approach and the complete re-execution of an evolved transformation, our results show a speed-up for the majority of studied cases. In recent work, we have exploited techniques of model patching for mutation-based testing of model transformations [TBBW15]. To reduce the computational costs of executing model transformation mutants, we generated patch transformations for the mutated parts of the model transformation. In this way, we could avoid re-executing complete model transformations that were mutated on a certain statement.

8.2 Outlook

Improving the integration between programming and modeling. With CAML's support for platform-specific libraries and profiles, we proposed an approach to close the gap between programming and modeling concerning annotation mechanisms. Still, a number of future challenges remain to further integrate programming and modeling. For instance, some interesting differences between Java annotations and UML profiles remain to be explored. On the UML side, inheritance between stereotypes is possible, a concept that is not supported by Java for annotation types. Thus, the design quality of automatically generated UML profiles can be enhanced by exploiting inheritance. On the Java side, retention policies determine at which stages annotations are accessible. UML stereotypes are considered only at design-time. Therefore, an interesting line of future work is to support stereotype applications also during run-time, which becomes especially interesting for executable models, a research area that is currently experiencing its renaissance by the emergence of the FUML standard [OMG16] and recent work in this context. Other programming languages such as C# and Python also support annotations to program elements. Investigating how their concepts, *i. e.,* attributes and decorators, correspond to UML profiles appears also desirable.

From technology-specific to technology-independent profiles. All the profiles provided by CAML for the Java platform are specific to libraries and their technology-specific concepts. For instance, a UML profile is available for JPA which supports relational datastore solutions. Another example is Objectify which supports key-value datastore solutions. Obviously, both profiles deal with data persistence even though for different technologies. While this is beneficial to

better understand the various technologies for data persistence, it also enables technology-specific refinement for architecture models. This paves the way for generating highly effective platform and technology-specific application code. Moreover, full control over platform and technology-specific decisions can be given to engineers as those decisions are explicitly manifested by the refined model. Still, the need for technology and platform-independent profiles appears obvious as they would allow engineers to express design decisions by means of more generic concepts. They would be retained even if the technology and platform is changed. Generalizing technology and platform-independent profiles from the existing set Java-based profiles provided by CAML is hence of high interest. Ideally, they provide potential mappings to existing technology and platform-specific profiles as a basis for implementing dedicated transformations.

Maintaining environment-specific profiles. CAML enables automated generation of platform-specific profiles from programming libraries. Whenever those libraries change, the corresponding UML profile can automatically be adapted, *e. g.,* by model patching. In contrast to the platform-specific profiles, CAML's profiles dedicated to cloud environments have been developed largely in a manual process. As a result, the environment-specific profiles provided by CAML need to be manually maintained. For instance, if a new cloud service is offered by a cloud environment, it needs to be added to the respective profile. Similarly, if existing cloud services change, the respective profiles need to be updated according to those changes. Automating the manual processes required for maintaining CAML's environment-specific profiles would thus highly desirable given the fact that some aspects of cloud services may change quite frequently, *e. g.,* their costs. One possibility to automatically gather these cloud service offerings is to apply web information extraction techniques. Another possibility is to exploit APIs available for cloud environments or existing cloud programming libraries. For instance, Amazon provides service endpoints for programmatically retrieving prices of cloud services. Furthermore, jclouds could be used to collect available services of various cloud environments in a programmatic way.

Simulation of deployment configurations. A deployment configuration created in UML and refined by CAML captures design-time artifacts that prescribe the desired state of the application provisioning. It is primarily used as input to a model-based provisioning process supported by TOSCA and respective tools. Cloud services are provisioned according to the modeled deployment configuration and application components manifested in form of deployable artifacts are distributed on top of those services as specified by deployment relationships. However, predicting non-functional properties such as costs and performance before the actual application provisioning is carried out seems of high interest. For instance, evaluating the cost-effectiveness of moving an on-premise application to the cloud before possible adaptations to application components are performed appears desirable. The simulation of deployment configurations refined towards a cloud environment could contribute to predict operational costs of certain cloud services. Moreover, it could support engineers to evaluate the effectiveness of defined elasticity rules before they are applied to the running application. Providing simulation support for CAML-based deployment configurations requires an operational semantics for UML's deployment language in general and the extensions provided by CAML in particular. A first promising step could be to explore fUML as it can be used to define the operational semantics of MOF-based languages [MLWK13].

Model-based reactive provisioning. As a CAML-based deployment configuration captures the desired state of the application provisioning, changes to it need to be propagated to the provisioned cloud services and the application components deployed on top of them. For instance, if a modeled compute service is replaced by another one for some purpose, *e. g.,* to save costs or increase performance, the corresponding running service instances need to be adapted according to the changes in deployment configuration. Such an adaptation may require releasing certain running service instances and enact the provisioning of other service instances. Re-executing the complete provisioning process to accomplish the required adaptations appears however unfavorable in particular if only some parts of the deployment configuration are concerned by a change. Exploiting techniques from reactive programming could be promising to improve the presented provisioning process based on UML/CAML and TOSCA. They could be used to realize a non-intrusive propagation of deployment configuration changes to the services of the target cloud environment. The latter would be the system that needs to react to events triggered by changes to a deployment configuration. In addition to design-time aspects, the consideration of run-time aspects could also be promising, as it would allow the representation and manipulation of individual service instances on the model level.

Live model patching. Model patching exploits the benefits of incremental transformation which has already been investigated and applied for propagating changes from input models to output models, such that previous transformation results are updated instead of discarded. Live transformation improves the change propagation between models because the effects of changes to input models are instantly mapped to respective output models. This is accomplished by preserving the transformation context and considering the execution of a transformation as phases that are triggered whenever a model change is notified. In this sense, a live transformation does not terminate but instead reacts to changes in input models. Model patching does currently not support live transformation in the sense of dynamically re-executing rules of a model transformation in memory, but rather generates a new patch transformation based on the diff of two transformation versions. Producing this diff certainly leads to a computational overhead when generating patch transformation. Moreover, in case the produced diff is not accurate enough, a patch transformation is unnecessarily generated. Thus, avoiding the computation of the diff is of high interest. Applying the idea of live transformation to model patching appears worthwhile in several respects. Changes to transformations could be notified with the same mechanisms used to notify changes of models because in MBE all artifacts can be considered as models. The incremental transformation rule execution mechanisms could also be reused. However, if a rule needs to be (re-)executed, adaptations according to our proposed taxonomy of change types and co-changes are required.

APPENDIX A

Research prototypes

The research prototypes implemented in the course of this thesis are available under the Eclipse Public License (EPL)[1]. They can be found at public github repositories.

CAML. Platform-specific libraries and profiles as well as the CAML tools can be found at:
`https://github.com/alexander-bergmayr/jump`
The CAML library and profiles for cloud environments can be found at:
`https://github.com/alexander-bergmayr/caml`

CAML2TOSCA. The model transformer can be found at:
`https://github.com/alexander-bergmayr/caml2tosca`

Model patching. Patch transformations realized for the *Java2UML* case can be found at:
`https://github.com/alexander-bergmayr/modelpatching`

[1] EPL: http://www.eclipse.org/legal/epl-v10.html

APPENDIX B

CAML and PetApp artifacts

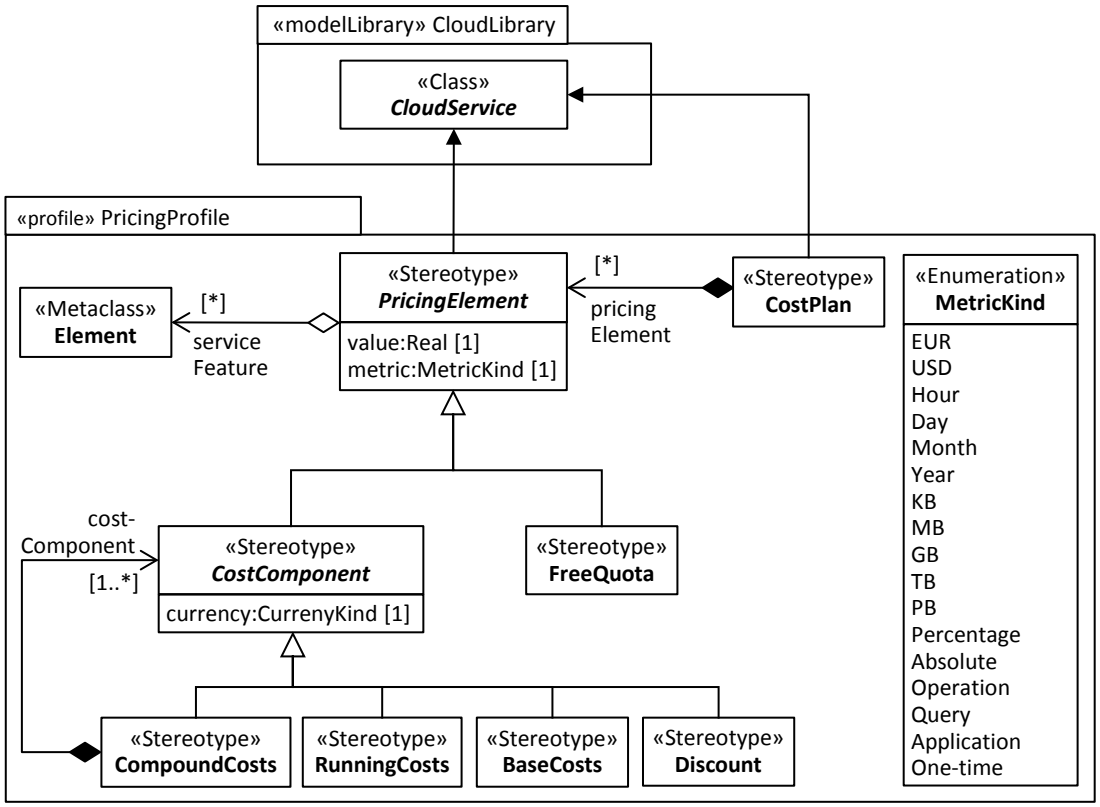

Figure B.1: Pricing profile

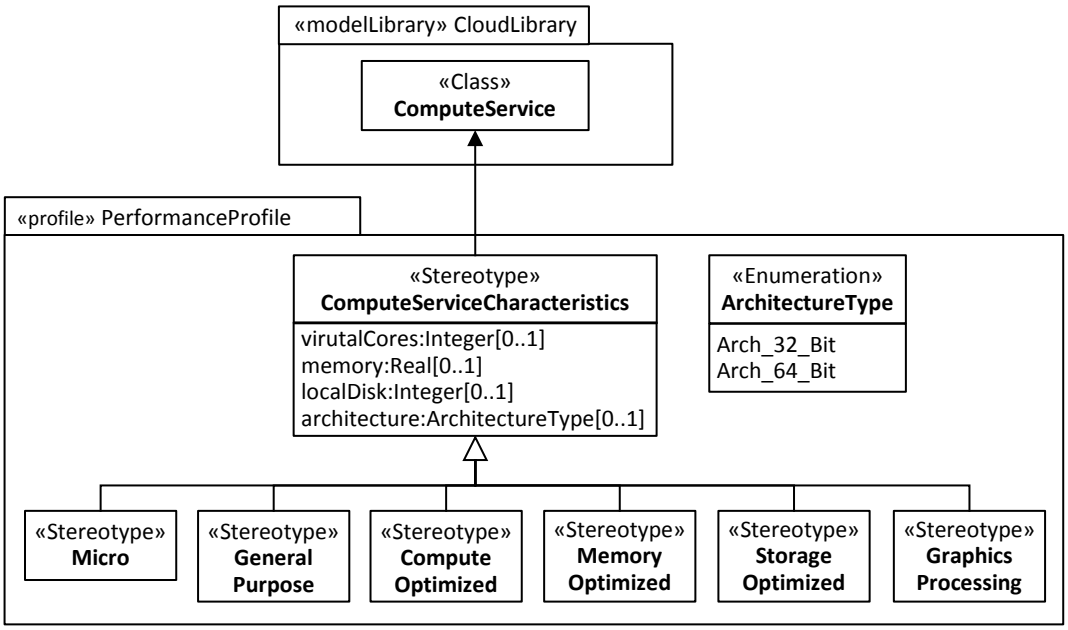

Figure B.2: Performance profile

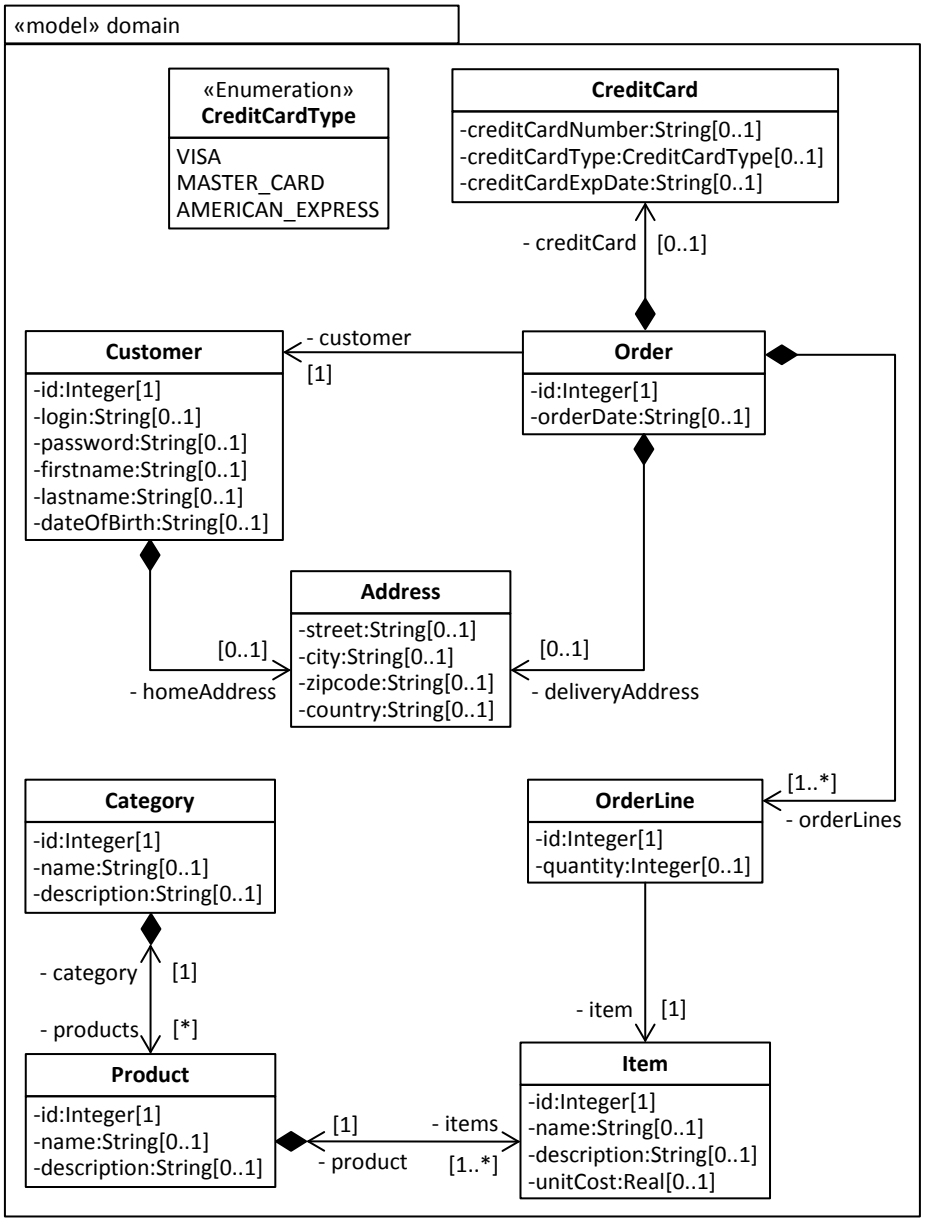

Figure B.3: Sliced domain model of the PetApp

Bibliography

[ABFJ05] Anas Abouzahra, Jean Bézivin, Marcos Didonet Del Fabro, and Frédéric Jouault. A Practical Approach to Bridging Domain Specific Languages with UML Profiles. In *Proc. of Intl. Workshop on Best Practices for Model-Driven Software Development*, pages 1–8, 2005.

[ABLS13] Vasilios Andrikopoulos, Tobias Binz, Frank Leymann, and Steve Strauch. How to Adapt Applications for the Cloud Environment. *Computing*, 95(6):493–535, 2013.

[ACB05] Paolo Atzeni, Paolo Cappellari, and Philip A. Bernstein. ModelGen: Model Independent Schema Translation. In *Proc. of Intl. Conf. on Data Engineering (ICDE)*, pages 1111–1112, 2005.

[ACS09] Michal Antkiewicz, Krzysztof Czarnecki, and Matthew Stephan. Engineering of Framework-Specific Modeling Languages. *IEEE Trans. Software Eng.*, 35(6):795–824, 2009.

[ADL+12] Moussa Amrani, Jürgen Dingel, Leen Lambers, Levi Lúcio, Rick Salay, Gehan Selim, Eugene Syriani, and Manuel Wimmer. Towards a Model Transformation Intent Catalog. In *Proc. of Intl. Workshop. on Analysis of Model Transformations (AMT)*, pages 3–8, 2012.

[AFG+09] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, 2009. https://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf.

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Commun. ACM*, 53(4):50–58, 2010.

[AGK09] Colin Atkinson, Matthias Gutheil, and Bastian Kennel. A Flexible Infrastructure for Multilevel Language Engineering. *IEEE Trans. Software Eng.*, 35(6):742–755, 2009.

[AHMM07] David H. Akehurst, W. Gareth J. Howells, and Klaus D. McDonald-Maier. Implementing Associations: UML 2.0 to Java 5. *Software and Systems Modeling*, 6(1):3–35, 2007.

[AK03] Colin Atkinson and Thomas Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, 2003.

[AK05] Colin Atkinson and Thomas Kühne. Concepts for Comparing Modeling Tool Architectures. In *Proc. of Intl. Conf. on Model Driven Engineering Languages and Systems (MoDELS)*, pages 398–413, 2005.

[AK07] Colin Atkinson and Thomas Kühne. A Tour of Language Customization Concepts. *Advances in Computers*, 70:105–161, 2007.

[AK09] Sven Apel and Christian Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology*, 8(5):49–84, 2009.

[AKHS03] Colin Atkinson, Thomas Kühne, and Brian Henderson-Sellers. Systematic Stereotype Usage. *Software and Systems Modeling*, 2(3):153–163, 2003.

[ALS08] Carsten Amelunxen, Elodie Legros, and Andy Schürr. Generic and Reflective Graph Transformations for the Checking and Enforcement of Modeling Guidelines. In *Proc. of Intl. Symposium of Visual Languages and Human-Centric Computing (VL/HCC)*, pages 211–218, 2008.

[ANM+12] Danilo Ardagna, Elisabetta Di Nitto, Parastoo Mohagheghi, Sébastien Mosser, Cyril Ballagny, Francesco D'Andria, Giuliano Casale, Peter Matthews, Cosmin-Septimiu Nechifor, Dana Petcu, Anke Gericke, and Craig Sheridan. MODAClouds: A Model-Driven Approach for the Design and Execution of Applications on Multiple Clouds. In *Proc. of Intl. Workshop on Modeling in Software Engineering (MISE)*, pages 50–56, 2012.

[ARSL14] Vasilios Andrikopoulos, Anja Reuter, Santiago Gomez Saez, and Frank Leymann. A GENTL Approach for Cloud Application Topologies. In *Proc. of European Conf. on Service-Oriented and Cloud Computing (ESOCC)*, pages 148–159, 2014.

[ARXL14] Vasilios Andrikopoulos, Anja Reuter, Mingzhu Xiu, and Frank Leymann. Design Support for Cost-Efficient Application Distribution in the Cloud. In *Proc. of Intl. Conf. on Cloud Computing (CLOUD)*, pages 697–704, 2014.

[ASLW14] Vasilios Andrikopoulos, Santiago Gomez Saez, Frank Leymann, and Johannes Wettinger. Optimal Distribution of Applications in the Cloud. In *Proc. of Intl. Conf. on Advanced Information Systems Engineering (CAiSE)*, pages 75–90, 2014.

[Bad00] Greg J. Badros. JavaML: A Markup Language for Java Source Code. *Computer Networks*, 33(1-6):159–177, 2000.

[BASS11] Theophilus Benson, Aditya Akella, Anees Shaikh, and Sambit Sahu. CloudNaaS: A Cloud Networking Platform for Enterprise Applications. In *Proc. of Intl. Symposium on Cloud Computing (SOCC)*, pages 1–13, 2011.

[BBC+13] Alexander Bergmayr, Hugo Bruneliere, Javier Cánovas, Jesús Gorroñogoitia, George Kousiouris, Dimosthenis Kyriazis, Philip Langer, Andreas Menychtas, Leire Orue-Echevarria, Clara Pezuela, and Manuel Wimmer. Migrating Legacy Software to the Cloud with ARTIST. In *Proc. of European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 465–468, 2013.

[BBF09] Gordon S. Blair, Nelly Bencomo, and Robert B. France. Models@run.time. *IEEE Computer*, 42(10):22–27, 2009.

[BBH+13] Tobias Binz, Uwe Breitenbücher, Florian Haupt, Oliver Kopp, Frank Leymann, Alexander Nowak, and Sebastian Wagner. OpenTOSCA – A Runtime for TOSCA-Based Cloud Applications. In *Proc. of Intl. Conf on Service-Oriented Computing (ICSOC)*, pages 692–695, 2013.

[BBK+12] Uwe Breitenbücher, Tobias Binz, Oliver Kopp, Frank Leymann, and David Schumm. Vino4TOSCA: A Visual Notation for Application Topologies based on TOSCA. In *Proc. of Intl. Conf. on On the Move to Meaningful Internet Systems (OTM)*, pages 416–424, 2012.

[BBK+14] Uwe Breitenbücher, Tobias Binz, Kálmán Képes, Oliver Kopp, Frank Leymann, and Johannes Wettinger. Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA. In *Proc. of Intl. Conf. on Cloud Engineering (IC2E)*, pages 87–96, 2014.

[BBK+16] Alexander Bergmayr, Uwe Breitenbücher, Oliver Kopp, Manuel Wimmer, Gerti Kappel, and Frank Leymann. From Architecture Modeling to Application Provisioning for the Cloud by Combining UML and TOSCA. In *Proc. of Intl. Conf. on Cloud Computing and Services Science (CLOSER)*, 2016. to appear.

[BBKL14a] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. TOSCA: Portable Automated Deployment and Management of Cloud Applications. In Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel, editors, *Advanced Web Services*. Springer, 2014.

[BBKL14b] Uwe Breitenbücher, Tobias Binz, Oliver Kopp, and Frank Leymann. Vinothek – A Self-Service Portal for TOSCA. In *Proc. of Central-European Workshop on Services and their Composition (ZEUS)*, pages 69–72, 2014.

[BBLS12] Tobias Binz, Gerd Breiter, Frank Leymann, and Thomas Spatzier. Portable Cloud Services Using TOSCA. *IEEE Internet Computing*, 16(3):80–85, 2012.

[BCBB15] Arnaud Blouin, Benoît Combemale, Benoit Baudry, and Olivier Beaudoux. Kompren: Modeling and Generating Model Slicers. *Software and Systems Modeling*, 14(1):321–337, 2015.

[BCC+15] Antonio Brogi, José Carrasco, Javier Cubo, Elisabetta Di Nitto, Francisco Durán, Michela Fazzolari, Ahmad Ibrahim, Ernesto Pimentel, Jacopo Soldani, PengWei Wang, and Francesco D'Andria. Adaptive management of applications across multiple clouds: The SeaClouds Approach. *CLEI Electron. J.*, 18(1), 2015.

[BCDM14] Hugo Bruneliere, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. MoDisco: A Model Driven Reverse Engineering Framework. *Information & Software Technology*, 56(8):1012–1032, 2014.

[BCJM10] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering. In *Proc. of Intl. Conf. on Automated Software Engineering (ASE)*, pages 173–174, 2010.

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, 2012.

[BD07] Achim D. Brucker and Jürgen Doser. Metamodel-based UML Notations for Domain-specific Languages. In *Proc. of Intl. Workshop on Software Language Engineering (ATEM)*, pages 1–15, 2007.

[BDA+14] Kacper Bąk, Zinovy Diskin, Michał Antkiewicz, Krzysztof Czarnecki, and Andrzej Wąsowski. Clafer: Unifying Class and Feature Modeling. *Software and Systems Modeling*, pages 1–35, 2014.

[Béz05a] Jean Bézivin. Model Driven Engineering: An Emerging Technical Space. In *Proc. of Intl. Summer School of Generative and Transformational Techniques in Software Engineering (GTTSE)*, pages 36–64, 2005.

[Béz05b] Jean Bézivin. On the Unification Power of Models. *Software and Systems Modeling*, 4(2):171–188, 2005.

[BGPCV12] Mark Lee Badger, Timothy Grance, Robert Patt-Corner, and Jeffery M. Voas. Cloud Computing Synopsis and Recommendations. Technical Report USP 800-146, National Institute of Standards and Technology (NIST), 2012. http://csrc.nist.gov/publications/nistpubs/800-146/sp800-146.pdf.

[BGWK14a] Alexander Bergmayr, Michael Grossniklaus, Manuel Wimmer, and Gerti Kappel. Bridging Java Annotations and UML Profiles with JUMP. In *Proc. of Demo Track of Intl. Conf. on Model Driven Engineering Languages and Systems (MoDELS)*, pages 1–5, 2014.

[BGWK14b] Alexander Bergmayr, Michael Grossniklaus, Manuel Wimmer, and Gerti Kappel. JUMP – From Java Annotations to UML Profiles. In *Proc. of Intl. Conf. on Model Driven Engineering Languages and Systems (MoDELS)*, pages 552–568, 2014.

[BGWK15] Alexander Bergmayr, Michael Grossniklaus, Manuel Wimmer, and Gerti Kappel. UML Profile Generation for Annotation-based Modeling. In *Proc. of Software*

Engineering & Management: Multikonferenz der GI-Fachbereiche Softwaretechnik (SWT) und Wirtschaftsinformatik (WI), pages 101–102, 2015.

[BGWK16] Alexander Bergmayr, Michael Grossniklaus, Manuel Wimmer, and Gerti Kappel. Leveraging Annotation-based Modeling with JUMP. *Software and Systems Modeling*, 2016. to appear.

[BKL+10] Petra Brosch, Horst Kargl, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, and Gerti Kappel. Conflicts as First-Class Entities: A UML Profile for Model Versioning. In *Proc. of Intl. Workshops and Symposia on Models in Software Engineering*, pages 184–193, 2010.

[BPBB14] Younes Benslimane, Michel Plaisent, Prosper Bernard, and Bouchaib Bahli. Key Challenges and Opportunities in Cloud Computing and Implications on Service Requirements: Evidence from a Systematic Literature Review. In *Proc. of Intl. Conf. on Cloud Computing Technology and Science (CloudCom)*, pages 114–121, 2014.

[BRF+15] Alexander Bergmayr, Alessandro Rossini, Nicolas Ferry, Geir Horn, Leire Orue-Echevarria, Arnor Solberg, and Manuel Wimmer. The Evolution of CloudML and its Manifestations. In *Proc. of Intl. Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE)*, pages 13–18, 2015.

[BRJ05] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide*. Addison-Wesley, 2nd edition, 2005.

[BRVV12] Gábor Bergmann, István Ráth, Gergely Varró, and Dániel Varró. Change-Driven Model Transformations - Change (in) the Rule to Rule the Change. *Software and Systems Modeling*, 11(3):431–461, 2012.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.

[BTN+14] Alexander Bergmayr, Javier Troya, Patrick Neubauer, Manuel Wimmer, and Gerti Kappel. UML-based Cloud Application Modeling with Libraries, Profiles, and Templates. In *Proc. of Intl. Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE)*, pages 56–65, 2014.

[BTW14] Alexander Bergmayr, Javier Troya, and Manuel Wimmer. From Out-Place Transformation Evolution to In-Place Model Patching. In *Proc. of Intl. Conf. on Automated Software Engineering (ASE)*, pages 647–652, 2014.

[BW13] Alexander Bergmayr and Manuel Wimmer. Generating Metamodels from Grammars by Chaining Translational and By-Example Techniques. In *Proc. of Intl. Workshop on Model-driven Engineering By Example (MDEBE)*, pages 22–31, 2013.

[BWKG14] Alexander Bergmayr, Manuel Wimmer, Gerti Kappel, and Michael Grossniklaus. Cloud Modeling Languages by Example. In *Proc. of Intl. Conf. on Service-Oriented Computing and Applications (SOCA)*, pages 137–146, 2014.

[BWRZ13] Alexander Bergmayr, Manuel Wimmer, Werner Retschitzegger, and Uwe Zdun. Taking the Pick out of the Bunch – Type-Safe Shrinking of Metamodels. In *Proc. of Software Engineering: Fachtagung des GI-Fachbereichs Softwaretechnik (SWT)*, pages 85–98, 2013.

[CASG13] Faruk Caglar, Kyoungho An, Shashank Shekhar, and Aniruddha Gokhale. Model-driven Performance Estimation, Deployment, and Resource Management for Cloud-hosted Services. In *Proc. of Intl. Workshop on Domain-Specific Modeling (DSM)*, pages 21–26, 2013.

[CBMK10] Jorge Cardoso, Alistair Barros, Norman May, and Uwe Kylau. Towards a Unified Service Description Language for the Internet of Services: Requirements and First Developments. In *Proc. of Intl. Conf. on Services Computing (SCC)*, pages 602–609, 2010.

[CC06] Frederick Chong and Gianpaolo Carraro. Architecture Strategies for Catching the Long Tail. Technical report, Microsoft Corporation, 2006. https://msdn.microsoft.com/en-us/library/aa479069.aspx.

[CCP15] Jose Carrasco, Javier Cubo, and Ernesto Pimentel. Towards a Flexible Deployment of Multi-cloud Applications Based on TOSCA and CAMP. In *Proc. of the Workshops of European Conf. on Service-Oriented and Cloud Computing (ESOCC)*, pages 278–286, 2015.

[CdL15] Irene Córdoba and Juan de Lara. A Modelling Language for the Effective Design of Java Annotations. In *Proc. of Intl. Symposium on Applied Computing (SAC)*, pages 2087–2092, 2015.

[CDPC11] Gerardo Canfora, Massimiliano Di Penta, and Luigi Cerulo. Achievements and Challenges in Software Reverse Engineering. *Commun. ACM*, 54(4):142–151, 2011.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CEM+12] Clovis Chapman, Wolfgang Emmerich, Fermín Galán Márquez, Stuart Clayman, and Alex Gallis. Software Architecture Definition for On-Demand Cloud Provisioning. *Cluster Comput.*, 15(2):79–100, 2012.

[CGL+03] Paul Clements, David Garlan, Reed Little, Robert Nord, and Judith Stafford. Documenting Software Architectures: Views and Beyond. In *Proc. of Intl. Conf. on Software Engineering (ICSE)*, pages 740–741, 2003.

[CH06a] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[CH06b] Krzysztof Czarnecki and Simon Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[CI90] Elliot J. Chikofsky and James H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7:13–17, Jan 1990.

[CJCM12] Javier Cánovas, Frédéric Jouault, Jordi Cabot, and Jesús García Molina. API2MoL: Automating the Building of Bridges between APIs and Model-Driven Engineering. *Information & Software Technology*, 54(3):257–273, 2012.

[CRB+11] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, César A. F. De Rose, and Rajkumar Buyya. CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms. *Softw. Pract. Exper.*, 41(1):23–50, 2011.

[DG08] Lucas Drumond and Rosario Girardi. A Survey of Ontology Learning Procedures. In *Proc. of Intl. Workshop on Ontologies and their Applications WONTO*, pages 13–24, 2008.

[DWC10] Tharam S. Dillon, Chen Wu, and Elizabeth Chang. Cloud Computing: Issues and Challenges. In *Proc. of Intl. Conf. on Advanced Information Networking and Applications (AINA)*, pages 27–33, 2010.

[DWS11] Brian Dougherty, Jules White, and Douglas C. Schmidt. Model-Driven Auto-Scaling of Green Cloud Computing Infrastructure. *Future Generation Computer Systems*, 28(2):371–378, 2011.

[ECB+11] Xavier Etchevers, Thierry Coupaye, Fabienne Boyer, Noel de Palma, and Gwen Salaun. Automated Configuration of Legacy Applications in the Cloud. In *Proc. of Intl. Conf. on Utility and Cloud Computing (UCC)*, pages 170–177, 2011.

[ECBP11] Xavier Etchevers, Thierry Coupaye, Fabienne Boyer, and Noel De Palma. Self-Configuration of Distributed Applications in the Cloud. In *Proc. of Intl. Conf. on Cloud Computing (CLOUD)*, pages 668–675, 2011.

[EHSW99] Gregor Engels, Roland Hücking, Stefan Sauer, and Annika Wagner. UML Collaboration Diagrams and their Transformation to Java. In *Proc. of Intl. Conf. on The Unified Modeling Language – Beyond the Standard (UML)*, pages 473–488, 1999.

[EIG+15] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, Abel Gómez, Massimo Tisi, and Jordi Cabot. EMF-REST: Generation of RESTful APIs from Models. *CoRR*, abs/1504.03498, 2015.

[EKK+06] Tamar Eilam, Michael H. Kalantar, Alexander V. Konstantinou, Giovanni Pacifici, John Pershing, and Aditya Agrawal. Managing the Configuration Complexity of Distributed Applications in Internet Data Centers. *IEEE Communications Magazine*, 44(3):166–177, 2006.

[EKK+13] Juergen Etzlstorfer, Angelika Kusel, Elisabeth Kapsammer, Philip Langer, Werner Retschitzegger, Johannes Schoenboeck, Wieland Schwinger, and Manuel Wimmer. A

Survey on Incremental Model Transformation Approaches. In *Proc. of Intl. Workshop on Models and Evolution (ME)*, pages 4–13, 2013.

[ESM05] Michael Eichberg, Thorsten Schäfer, and Mira Mezini. Using Annotations to Check Structural Properties of Classes. In *Proc. of Intl. Conf. on Fundamental Approaches to Software Engineering (FASE)*, pages 237–252, 2005.

[FFH13] Sören Frey, Florian Fittkau, and Wilhelm Hasselbring. Search-based Genetic Optimization for Deployment and Reconfiguration of Software in the Cloud. In *Proc. of Intl. Conf. on Software Engineering (ICSE)*, pages 512–521, 2013.

[FFV04] Lidia Fuentes-Fernández and Antonio Vallecillo. An Introduction to UML Profiles. *Europ. Journal for the Informatics Professional*, 5(2):5–13, 2004.

[FH10] Sören Frey and Wilhelm Hasselbring. Model-Based Migration of Legacy Software Systems into the Cloud: The CloudMIG Approach. *Softwaretechnik-Trends*, 30(2):1–2, 2010.

[FH11] Sören Frey and Wilhelm Hasselbring. The CloudMIG Approach: Model-Based Migration of Software Systems to Cloud-Optimized Applications. *Intl. J. Advances in Software*, 4(3&4):342–353, 2011.

[FHS13] Sören Frey, Wilhelm Hasselbring, and Benjamin Schnoor. Automatic Conformance Checking for Migrating Software Systems to Cloud Infrastructures and Platforms. *Journal of Software: Evolution and Process*, 25(10):1089–1115, 2013.

[FL08] Andrew Forward and Timothy C. Lethbridge. Problems and Opportunities for Model-centric Versus Code-centric Software Development: A Survey of Software Professionals. In *Proc. of Intl. Workshop on Models in Software Engineering (MiSE)*, pages 27–32, 2008.

[FLR+14] Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. *Cloud Computing Patterns – Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014.

[Fow10] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley, 2010.

[FR10] Robert France and Bernhard Rumpe. Modeling for the Cloud. *Software and Systems Modeling*, 9(2):139–140, 2010.

[FR13] Robert B. France and Bernhard Rumpe. The Evolution of Modeling Research Challenges. *Software and Systems Modeling*, 12(2):223–225, 2013.

[FRC+13] Nicolas Ferry, Alessandro Rossini, Franck Chauvel, Brice Morin, and Arnor Solberg. Towards Model-Driven Provisioning, Deployment, Monitoring, and Adaptation of Multi-cloud Systems. In *Proc. of Intl. Conf. on Cloud Computing (CLOUD)*, pages 887–894, 2013.

[FSR+14] Nicolas Ferry, Hui Song, Alessandro Rossini, Franck Chauvel, and Arnor Solberg. Cloud MF: Applying MDE to Tame the Complexity of Managing Multi-cloud Applications. In *Proc. of Intl. Conf. on Utility and Cloud Computing (UCC)*, pages 269–277, 2014.

[GBI+10] David Garlan, Felix Bachmann, James Ivers, Judith Stafford, Len Bass, Paul Clements, and Paulo Merson. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2nd edition, 2010.

[GDA12] Jokin García, Oscar Díaz, and Maider Azanza. Model Transformation Co-evolution: A Semi-automatic Approach. In *Proc. of Intl. Conf. on Software Language Engineering (SLE)*, pages 144–163, 2012.

[GdCL03] Gonzalo Génova, Carlos Ruiz del Castillo, and Juan Lloréns. Mapping UML Associations into Java Code. *Journal of Object Technology*, 2(5):135–162, 2003.

[GDD09] Dragan Gasevic, Dragan Djuric, and Vladan Devedzic. *Model Driven Engineering and Ontology Development*. Springer, 2nd edition, 2009.

[Ges08] Dominik Gessenharter. Mapping the UML2 Semantics of Associations to a Java Code Generation Model. In *Proc. of Intl. Conf. on Model Driven Engineering Languages and Systems (MoDELS)*, pages 813–827, 2008.

[GES+11] Glauco Gonçalves, Patricia Endo, Marcelos Santos, Djamel Sadok, Judith Kelner, Bob Merlander, and Jan-Erik Mångs. CloudML: An Integrated Language for Resource, Service and Request Description for D-Clouds. In *Proc. of Intl. Conf. on Cloud Computing Technologies and Science (CloudCom)*, pages 399–406, 2011.

[Gli07] Martin Glinz. On Non-Functional Requirements. In *Proc. of Intl. Conf. on Requirements Engineering (RE)*, pages 21–26, 2007.

[GMMC13a] Joaquín Guillén, Javier Miranda, Juan Manuel Murillo, and Carlos Canal. A Service-oriented Framework for Developing Cross Cloud Migratable Software. *J. Syst. Softw.*, 86(9):2294–2308, 2013.

[GMMC13b] Joaquín Guillén, Javier Miranda, Juan Manuel Murillo, and Carlos Canal. A UML Profile for Modeling Multicloud Applications. In *Proc. of European Conf. on Service-Oriented and Cloud Computing (ESOCC)*, pages 180–187, 2013.

[GS03] Jack Greenfield and Keith Short. Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools. In *Proc. of Intl. Conf on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 16–27, 2003.

[GvD07] Bas Graaf and Arie van Deursen. Visualisation of Domain-Specific Modelling Languages Using UML. In *Proc. of Intl. Conf. on Engineering of Computer-Based Systems (ECBS)*, pages 586–595, 2007.

[HBR00] William Harrison, Charles Barton, and Mukund Raghavachari. Mapping UML Designs to Java. In *Proc. of Intl. Conf. on Object-Oriented Programming Systems, Languages & Applications (OOPSLA)*, pages 178–187, 2000.

[HJSW10] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. Closing the Gap between Modelling and Java. In *Proc. of Intl. Conf. on Software Language Engineering (SLE)*, pages 374–383, 2010.

[HLR06] David Hearnden, Michael Lawley, and Kerry Raymond. Incremental Model Transformation for the Evolution of Model-Driven Systems. In *Proc. of Intl. Conf. on Model Driven Engineering Languages and Systems (MoDELS)*, pages 321–335, 2006.

[HLS+13] Mark Harman, Kiran Lakhotia, Jeremy Singer, David Robert White, and Shin Yoo. Cloud Engineering is Search Based Software Engineering too. *Journal of Systems and Software*, 86(9):2225–2241, 2013.

[HLT11] Mohammad Hamdaqa, Tassos Livogiannis, and Ladan Tahvildari. A Reference Model for Developing Cloud Applications. In *Proc. of Intl. Conf. on Cloud Computing and Services Science (CLOSER)*, pages 98–103, 2011.

[HMPR04] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design Science in Information Systems Research. *MIS Quarterly*, 28(1):75–105, 2004.

[Hol14] Ta'id Holmes. Automated Provisioning of Customized Cloud Service Stacks using Domain-Specific Languages. In *Proc. of Intl. Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE)*, pages 46–55, 2014.

[Hol15] Ta'id Holmes. Facilitating Migration of Cloud Infrastructure Services: A Model-Based Approach. In *Proc. of Intl. Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE)*, pages 7–12, 2015.

[HPR14] Lars Hermerschmidt, Antonio Navarro Perez, and Bernhard Rumpe. A Model-based Software Development Kit for the SensorCloud Platform. In Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe, editors, *Trusted Cloud Computing*, pages 125–140. Springer, 2014.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, 2004.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical report, Software Engineering Lab, RWTH Aachen University, Aachen, 2012. http://webdoc.sub.gwdg.de/ebook/serien/ah/AIB/2012-03.pdf.

[HRW11] John Hutchinson, Mark Rouncefield, and Jon Whittle. Model-Driven Engineering Practices in Industry. In *Proc. of Intl. Conf. on Software Engineering (ICSE)*, pages 633–642, 2011.

[HT14] Mohammad Hamdaqa and Ladan Tahvildari. The (5+1) Architectural View Model for Cloud Applications. In *Proc. of Intl. Conf. on Computer Science and Software Engineering (CASCON)*, pages 46–60, 2014.

[HT15] Mohammad Hamdaqa and Ladan Tahvildari. StratusML: A Layered Cloud Modeling Framework. In *Proc. of Intl. Conf. on Cloud Engineering (IC2E)*, pages 96–105, 2015.

[HWRK11] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of MDE in industry. In *Proc. of Intl. Conference on Software Engineering (ICSE)*, pages 471–480, 2011.

[INS+14] Christian Inzinger, Stefan Nastic, Sanjin Sehic, Michael Vögler, Fei Li, and Schahram Dustdar. MADCAT: A Methodology for Architecture and Deployment of Cloud Application Topologies. In *Proc. of Intl. Symposium on Service Oriented System Engineering (SOSE)*, pages 13–22, 2014.

[IPM12] Ludovico Iovino, Alfonso Pierantonio, and Ivano Malavolta. On the Impact Significance of Metamodel Evolution in MDE. *Journal of Object Technology*, 11(3):1–33, 2012.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A Model Transformation Tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008.

[JAP13] Pooyan Jamshidi, Aakash Ahmad, and Claus Pahl. Cloud Migration Research: A Systematic Review. *IEEE Trans. Cloud Computing*, 1(2):142–157, 2013.

[JCD+12] Jean-Marc Jézéquel, Benoit Combemale, Steven Derrien, Clément Guy, and Sanjay Rajopadhye. Bridging the Chasm between MDE and the World of Compilation. *Software and Systems Modeling*, 11(4):581–597, 2012.

[JE04] Sven Johann and Alexander Egyed. Instant and Incremental Transformation of Models. In *Proc. of Intl. Conf. on Automated Software Engineering (ASE)*, pages 362–365, 2004.

[JHS13] Keith Jeffery, Geir Horn, and Lutz Schubert. A Vision for Better Cloud Applications. In *Proc. of Intl. Workshop on Multi-cloud Applications and Federated Clouds (MultiCloud)*, pages 7–12, 2013.

[Jon96] Neil D. Jones. An Introduction to Partial Evaluation. *Comput. Surv.*, 28(3):480–503, 1996.

[Jou05] Frédéric Jouault. Loosely Coupled Traceability for ATL. In *Proc. of European Workshop on Traceability (TW)*, pages 29–37, 2005.

[JT10] Frédéric Jouault and Massimo Tisi. Towards Incremental Execution of ATL Transformations. In *Proc. of Intl. Conf. on Model Transformation (ICMT)*, pages 123–137, 2010.

[KBA02] Ivan Kurtev, Jean Bézivin, and M Akşit. Technological Spaces: An Initial Appraisal. In *Proc. of Intl. Conf. on Cooperative Information Systems (CoopIS)*, pages 1–6, 2002.

[KBB+09] Barbara Kitchenham, Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen G. Linkman. Systematic Literature Reviews in Software Engineering - A Systematic Literature Review. *Information & Software Technology*, 51(1):7–15, 2009.

[KBBL13] Oliver Kopp, Tobias Binz, Uwe Breitenbücher, and Frank Leymann. Winery – A Modeling Tool for TOSCA-Based Cloud Applications. In *Proc. of Intl. Conf. on Service-Oriented Computing (ICSOC)*, pages 700–704, 2013.

[KC07] Barbara Kitchenham and Stuart Charters. Guidelines for performing Systematic Literature Reviews in Software Engineering. Technical report, Keele University and Durham University Joint Report, EBSE-2007-01, 2007.

[KCH+90] Kang Kyo, Sholom Cohen, James Hess, William Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 1990.

[KDR14] Kyriakos Kritikos, Jörg Domaschka, and Alessandro Rossini. SRL: A Scalability Rule Language for Multi-cloud Environments. In *Proc. of Intl. Conf. on Cloud Computing Technology and Science (CloudCom)*, pages 1–9, 2014.

[KDRPP09] Dimitrios Kolovos, Davide Di Ruscio, Alfonso Pierantonio, and Richard Paige. Different Models for Model Matching: An Analysis of Approaches to Support Model Differencing. In *Proc. of Intl. Workshop on Comparison and Versioning of Software Models (CVSM)*, pages 1–6, 2009.

[Ken02] Stuart Kent. Model Driven Engineering. In *Proc. of Intl. Conf. on Integrated Formal Methods (IFM)*, pages 286–298, 2002.

[KKK+06] Gerti Kappel, Elisabeth Kapsammer, Horst Kargl, Gerhard Kramler, Thomas Reiter, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. Lifting Metamodels to Ontologies: A Step to the Semantic Integration of Modeling Languages. In *Proc. of Intl. Conf. on Model Driven Engineering Languages and Systems (MoDELS)*, pages 528–542, 2006.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Proc. of Intl. Workshop on Domain-Specific Modeling (DSM)*, pages 7–13, 2009.

[KLV05] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an Engineering Discipline for Grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14(3):331–380, 2005.

[KMS04] Martin Kavalec, Alexander Maedche, and Vojtěch Svátek. Discovery of Lexical Entries for Non-taxonomic Relations in Ontology Learning. In *Proc. of Intl. Conf. on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, pages 249–256, 2004.

[KPB+10] Barbara Kitchenham, Rialette Pretorius, David Budgen, Pearl Brereton, Mark Turner, Mahmood Niazi, and Stephen G. Linkman. Systematic Literature Reviews in Software Engineering - A Tertiary Study. *Information & Software Technology*, 52(8):792–805, 2010.

[Kra07] Jeff Kramer. Is Abstraction the Key to Computing? *Commun. ACM*, 50(4):36–42, 2007.

[KRV10] Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: A Framework for Compositional Development of Domain Specific Languages. *Intl. Journal on Software Tools for Technology Transfer*, 12(5):353–372, 2010.

[KSS+02] Ralf Kollman, Petri Selonen, Eleni Stroulia, Tarja Systä, and Albert Zündorf. A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering. In *Proc. of Intl. Working Conf. on Reverse Engineering (WCRE)*, pages 22–32, 2002.

[KSW+15] Angelika Kusel, Johannes Schönböck, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. Reuse in Model-to-Model Transformation Languages: Are we there yet? *Software and Systems Modeling*, 14(2):537–572, 2015.

[Küh06] Thomas Kühne. Matters of (Meta-)Modeling. *Software and Systems Modeling*, 5(4):369–385, 2006.

[KWC98] Rick Kazman, Steven G. Woods, and S. Jeromy Carrière. Requirements for Integrating Software Architecture and Reengineering Models: CORUM II. In *Proc. of Intl. Working Conf. on Reverse Engineering (WCRE)*, pages 154–163, 1998.

[LBNK09] Tihamer Levendovszky, Daniel Balasubramanian, Anantha Narayanan, and Gabor Karsai. A Novel Approach to Semi-automated Evolution of DSML Model Transformation. In *Proc. of Intl. Conf. on Software Language Engineering (SLE)*, pages 23–41, 2009.

[Lee89] Allen S. Lee. A Scientific Methodology for MIS Case Studies. *MIS Quarterly*, 13(1):33–50, 1989.

[Ley11] Frank Leymann. Cloud Computing. *it - Information Technology*, 53(4):163–164, 2011.

[LFM+11] Frank Leymann, Christoph Fehling, Ralph Mietzner, Alexander Nowak, and Schahram Dustdar. Moving Applications to the Cloud: An Approach Based on Application Model Enrichment. *Int. J. Cooperative Inf. Syst.*, 20(3):307–356, 2011.

[LMM+15] Patricia Lago, Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, and Antony Tang. The Road Ahead for Architectural Languages. *IEEE Software*, 32(1):98–105, 2015.

[LS06] Michael Lawley and Jim Steel. Practical Declarative Model Transformation with Tefkat. In *Proc. of Satellite Events at Intl. Conf. on Model Driven Engineering Languages and Systems (MoDELS)*, pages 139–150, 2006.

[LSS+13] Hongbin Lu, Mark Shtern, Bradley Simmons, Michael Smit, and Marin Litoiu. Pattern-Based Deployment Service for Next Generation Clouds. In *Proc. of World Congress on Services (SERVICES)*, pages 464–471, 2013.

[LWWC12] Philip Langer, Konrad Wieland, Manuel Wimmer, and Jordi Cabot. EMF Profiles: A Lightweight Extension Approach for EMF Models. *Journal of Object Technology*, 11(1):1–29, 2012.

[LZ11] Dongxi Liu and John Zic. Cloud#: A Specification Language for Modeling Cloud. In *Proc. of Intl. Conf. on Cloud Computing (CLOUD)*, pages 533–540, 2011.

[MEMC10] David Méndez, Anne Etien, Alexis Muller, and Rubby Casallas. Towards Transformation Migration After Metamodel Evolution. In *Proc. of Intl. Workshop on Models and Evolution (ME)*, pages 84–89, 2010.

[MFBC12] Pierre-Alain Muller, Frédéric Fondement, Benoit Baudry, and Benoît Combemale. Modeling Modeling Modeling. *Software and Systems Modeling*, 11(3):347–359, 2012.

[MG06] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electron. Notes Theor. Comput. Sci*, 152:125–142, 2006.

[MG11] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. Technical Report NIST SP 800-145, National Institute of Standards and Technology (NIST), 2011.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[MJS+00] Hausi A. Müller, Jens H. Jahnke, Dennis B. Smith, Margaret-Anne Storey, Scott R. Tilley, and Kenny Wong. Reverse Engineering: A Roadmap. In *Proc. of Intl. Conf. on Software Engineering (ICSE)*, pages 47–60, 2000.

[MLM+13] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. What Industry Needs from Architectural Languages: A Survey. *IEEE Trans. Software Eng.*, 39(6):869–891, 2013.

[MLP08] Ralph Mietzner, Frank Leymann, and Mike P. Papazoglou. Defining Composite Configurable SaaS Application Packages Using SCA, Variability Descriptors and Multi-tenancy Patterns. In *Proc. of Intl. Conf. on Internet and Web Applications and Services (ICIW)*, pages 156–161, 2008.

[MLWK13] Tanja Mayerhofer, Philip Langer, Manuel Wimmer, and Gerti Kappel. xMOF: Executable DSMLs based on fUML. In *Proc. of Intl. Conf on Software Language Engineering (SLE)*, pages 56–75, 2013.

[MMP08] Ivano Malavolta, Henry Muccini, and Patrizio Pelliccione. DUALLY: A Framework for Architectural Languages and Tools Interoperability. In *Proc. of Intl. Conf. on Automated Software Engineering (ASE)*, pages 483–484, 2008.

[MMPT10] Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, and Damien A. Tamburri. Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies. *IEEE Trans. Software Eng.*, 36(1):119–140, 2010.

[Moo09] Daniel L. Moody. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Trans. Software Eng.*, 35(6):756–779, 2009.

[MRRR02] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling Software Architectures in the Unified Modeling Language. *ACM Trans. Softw. Eng. Methodol.*, 11(1):2–57, 2002.

[MT00] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Software Eng.*, 26(1):70–93, 2000.

[NBM+15] Patrick Neubauer, Alexander Bergmayr, Tanja Mayerhofer, Javier Troya, and Manuel Wimmer. XMLText: From XML Schema to Xtext. In *Proc. of Intl. Conf. on Software Language Engineering (SLE)*, pages 71–76, 2015.

[ND08] Carlos Noguera and Laurence Duchien. Annotation Framework Validation Using Domain Models. In *Proc. European Conf. on Model Driven Architecture – Foundations and Applications (ECMDA-FA)*, volume 5095 of *LNCS*, pages 48–62. Springer, 2008.

[NLPvdH12] Dinh Khoa Nguyen, Francesco Lelli, Mike P. Papazoglou, and Willem-Jan van den Heuvel. Blueprinting Approach in Support of Cloud Computing. *Future Internet*, 4(1):322–346, 2012.

[NLT+11] Dinh Khoa Nguyen, Francesco Lelli, Yehia Taher, Michael Parkin, Mike P. Papazoglou, and Willem-Jan van den Heuvel. Blueprint Template Support for Engineering Cloud-Based Services. In *Proc. of European Conf. on Towards a Service-Based Internet (ServiceWave)*, pages 26–37, 2011.

[NNZ00] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA Environment. In *Proc. of Intl. Conf. on Software Engineering (ICSE)*, pages 742–745, 2000.

[NP07] Carlos Noguera and Renaud Pawlak. AVal: An Extensible Attribute-Oriented Programming Validator for Java. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(4):253–275, 2007.

[NSJ12] Tam Le Nhan, Gerson Sunyé, and Jean-Marc Jézéquel. A Model-driven Approach for Virtual Machine Image Provisioning in Cloud Computing. In *Proc. of European Conf. on Service-Oriented and Cloud Computing (ESOCC)*, pages 107–121, 2012.

[OAS12] OASIS. Cloud Application Management for Platforms (CAMP), 2012. https://www.oasis-open.org/committees/camp/.

[OAS13a] OASIS. Topology and Orchestration Specification for Cloud Applications (TOSCA), 2013. https://www.oasis-open.org/committees/tosca.

[OAS13b] OASIS. Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer, 2013. http://docs.oasis-open.org/tosca/tosca-primer/v1.0/tosca-primer-v1.0.html.

[OAS15] OASIS. TOSCA Simple Profile in YAML, 2015. http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0.

[OMG11a] OMG. Business Process Model and Notation (BPMN), 2011. http://www.omg.org/spec/BPMN/2.0.

[OMG11b] OMG. Meta Object Facility (MOF), 2011. http://www.omg.org/spec/MOF.

[OMG11c] OMG. XML Metadata Interchange (XMI), 2011. http://www.omg.org/spec/XMI.

[OMG12] OMG. Service oriented architecture Modeling Language (SoaML), 2012. http://www.omg.org/spec/SoaML.

[OMG14a] OMG. Catalog of UML Profile Specifications, 2014. http://www.omg.org/spec/#Profile.

[OMG14b] OMG. Object Constraint Language (OCL), 2014. http://www.omg.org/spec/OCL/.

[OMG15] OMG. Unified Modeling Language (UML), 2015. http://www.omg.org/spec/UML.

[OMG16] OMG. Semantics of a Foundational Subset for Executable UML Models (FUML), 2016. http://www.omg.org/spec/FUML/.

[Ora15] Oracle. JLS8, 2015. http://docs.oracle.com/javase/specs.

[Owe10] Dustin Owens. Securing Elasticity in the Cloud. *Commun. ACM*, 53(6):46–51, 2010.

[Par10] Jesús Pardillo. A Systematic Review on the Definition of UML Profiles. In *Proc. of Intl. Conf. on Model Driven Engineering Languages and Systems (MoDELS)*, pages 407–422, 2010.

[PBMH12] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. Adoption and Use of Java Generics. *Empirical Software Engineering*, 18(6):1–43, 2012.

[Pet14] Dana Petcu. Consuming Resources and Services from Multiple Clouds. *J. Grid Comput.*, 12(2):321–345, 2014.

[PFMM08] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. Systematic Mapping Studies in Software Engineering. In *Proc. of Intl. Conf. on Evaluation and Assessment in Software Engineering (EASE)*, pages 68–77, 2008.

[PR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Proc. of Intl. Workshop on Model-Driven Engineering for High Performance and CLoud computing (MDHPCL)*, pages 15–24, 2013.

[PV12] Michael P. Papazoglou and Luis M. Vaquero. Knowledge-Intensive Cloud Services: Transforming the Cloud Delivery Stack. In Jussi Kantola and Waldemar Karwowski, editors, *Knowledge Service Engineering Handbook*, pages 447–492. CRC Press, 2012.

[RBÖV08] István Ráth, Gábor Bergmann, András Ökrös, and Dániel Varró. Live Model Transformations Driven by Incremental Pattern Matching. In *Proc. of Intl. Conf. on Model Transformation (ICMT)*, pages 107–121, 2008.

[RdLGN15] Alessandro Rossini, Juan de Lara, Esther Guerra, and Nikolay Nikolov. A Comparison of Two-Level and Multi-level Modelling for Cloud-Based Applications. In *Proc. of European Conf. on Modeling Foundations and Applications (ECMFA)*, pages 18–32, 2015.

[RFJ08] Daniel Ratiu, Martin Feilkas, and Jan Jürjens. Extracting Domain Ontologies from Domain Specific APIs. In *Proc. of European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 203–212, 2008.

[RH09] Per Runeson and Martin Höst. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

[RIP13] Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. A Methodological Approach for the Coupled Evolution of Metamodels and ATL Transformations. In *Proc. of Intl. Conf. on Model Transformation (ICMT)*, pages 60–75, 2013.

[RK12] Ali Razavi and Kostas Kontogiannis. Partial Evaluation of Model Transformations. In *Proc. of Intl. Conf. on Software Engineering (ICSE)*, pages 562–572, 2012.

[RNHR13] Andreas Rentschler, Qais Noorshams, Lucia Happe, and Ralf Reussner. Interactive Visual Analytics for Efficient Maintenance of Model Transformations. In *Proc. of Intl. Conf. on Model Transformation (ICMT)*, pages 141–157, 2013.

[RR11] Jan Oliver Ringert and Bernhard Rumpe. A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. *International Journal of Software and Informatics (IJSI)*, 5(1-2):29–53, 2011.

[Sch06] Douglas C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[SDA12] Le Sun, Hai Dong, and Jamshaid Ashraf. Survey of Service Description Languages and Their Issues in Cloud Computing. In *Proc. of Intl. Conf. on Semantics, Knowledge and Grids (SKG)*, pages 128–135, 2012.

[Sel07] Bran Selic. A Systematic Approach to Domain-Specific Language Design Using UML. In *Proc. of Intl. Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 2–9, 2007.

[Sel08] Bran Selic. MDA Manifestations. *UPGRADE: The European Journal for the Informatics Professional*, 9(2):12–16, 2008.

[Sel12] Bran Selic. The Less Well Known UML – A Short User Guide. In *Proc. of Intl. School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM)*, pages 1–20, 2012.

[Sha02] Mary Shaw. What Makes Good Research in Software Engineering? *Journal on Software Tools for Technology Transfer (STTT)*, 4(1):1–7, 2002.

[Sha03] Mary Shaw. Writing Good Software Engineering Research Paper. In *Proc. of Intl. Conf. on Software Engineering (ICSE)*, pages 726–737, 2003.

[SHSW12] Yih Leong Sun, Terence Harmer, Alan Stewart, and Peter Wright. Mapping Application Requirements to Cloud Resources. In *Proc. of European Parallel Processing Workshops (Eur-Par)*, pages 104–112, 2012.

[SK03] Shane Sendall and Wojtek Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, 2003.

[SR10] Amit Sheth and Ajith Ranabahu. Semantic Modeling for Cloud Computing, Part 2. *IEEE Internet Computing*, 14(4):81–84, 2010.

[SRC13] Gabriel Costa Silva, Louis M. Rose, and Radu Calinescu. A Systematic Review of Cloud Lock-In Solutions. In *Proc. of Intl. Conf. on Cloud Computing Technology and Science (CLOUDCOM)*, pages 363–368, 2013.

[SRC14] Gabriel Costa Silva, Louis M. Rose, and Radu Calinescu. Cloud DSL: A Language for Supporting Cloud Portability by Describing Cloud Entities. In *Proc. of Intl. Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE)*, pages 36–45, 2014.

[Szy03] Clemens Szyperski. Component Technology: What, Where, and How? In *Proc. of Intl. Conf. on Software Engineering (ICSE)*, pages 684–693, 2003.

[TBBW15] Javier Troya, Alexander Bergmayr, Loli Burgueño, and Manuel Wimmer. Towards Systematic Mutations for and with ATL Model Transformations. In *Proc. of Intl. Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–10, 2015.

[TJF+09] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the Use of Higher-Order Model Transformations. In *Proc. of European Conf. on Model Driven Architecture – Foundations and Applications (ECMDA-FA)*, pages 18–33, 2009.

[TMC99] Scott Thibault, Renaud Marlet, and Charles Consel. Domain-Specific Languages: From Design to Implementation Application to Video Device Drivers Generation. *IEEE Trans. Software Eng.*, 25(3):363–377, 1999.

[TMW+05] Vanish Talwar, Dejan S. Milojicic, Qinyi Wu, Calton Pu, Wenchang Yan, and Gueyoung Jung. Approaches for Service Deployment. *IEEE Internet Computing*, 9(2):70–80, 2005.

[TP04] Paolo Tonella and Alessandra Potrich. *Reverse Engineering of Object Oriented Code*. Springer, 2004.

[TTR+11] Marco Torchiano, Federico Tomassetti, Filippo Ricca, Alessandro Tiso, and Gianna Reggio. Preliminary Findings from a Survey on the MD* State of the Practice. In *Proc. of Intl. Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 372–375, 2011.

[UPR15] UPR. Eclipse UML Profiles Repository, 2015. https://projects.eclipse.org/projects/modeling.upr.

[vAvdB11] Marcel van Amstel and Mark G. J. van den Brand. Model Transformation Analysis: Staying Ahead of the Maintenance Nightmare. In *Proc. of Intl. Conf. on Model Transformation (ICMT)*, pages 108–122, 2011.

[vdBCC05] Klass van den Berg, Jose Maria Conejero, and Ruzanna Chitchyan. AOSD Ontology 1.0 - Public Ontology of Aspect-Orientation. Technical report, D9 AOSD-Europe-UT-01, AOSD-Europe, 2005.

[Vog09] Werner Vogels. Eventually Consistent. *Commun. ACM*, 52(1):40–44, 2009.

[VP04] Dániel Varró and András Pataricza. Generic and Meta-transformations for Model Transformation Engineering. In *Proc. of Intl. Conf. on Unified Modeling Language (UML)*, pages 290–304, 2004.

[VRB11] Luis Miguel Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. Dynamically Scaling Applications in the Cloud. *Computer Communication Review*, 41(1):45–52, 2011.

[VW12] Will Venters and Edgar A. Whitley. A Critical Review of Cloud Computing: Researching Desires and Realities. *Journal of Information Technology*, 27(3):179–197, 2012.

[W3C04] W3C. OWL-S: Semantic Markup for Web Services, 2004. https://www.w3.org/Submission/OWL-S.

[Wal09] Edward Walker. The Real Cost of a CPU Hour. *Computer*, 42(4):35–41, 2009.

[WBB+14a] Johannes Wettinger, Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. Streamlining Cloud Management Automation by Unifying the Invocation of Scripts and Services Based on TOSCA. *Int. J. Organ. Collect. Intell.*, 4(2):45–63, 2014.

[WBB+14b] Johannes Wettinger, Tobias Binz, Uwe Breitenbücher, Oliver Kopp, Frank Leymann, and Michael Zimmermann. Unified Invocation of Scripts and Services for Provisioning, Deployment, and Management of Cloud Applications Based on TOSCA. In *Proc. of Intl. Conf. on Cloud Computing and Services Science (CLOSER)*, pages 559–568, 2014.

[WBL14] Johannes Wettinger, Uwe Breitenbücher, and Frank Leymann. Standards-Based DevOps Automation and Integration Using TOSCA. In *Proc. of Intl. Conf. on Utility and Cloud Computing (UCC)*, pages 59–68, 2014.

[WHR+13] Jon Whittle, John Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem? In *Proc. of Intl. Conf. on Model Driven Engineering Languages and Systems (MoDELS)*, pages 1–17, 2013.

[WHR14] Jon Whittle, John Hutchinson, and Mark Rouncefield. The State of Practice in Model-Driven Engineering. *IEEE Software*, 31(3):79–85, 2014.

[Wim09] Manuel Wimmer. A Semi-Automatic Approach for Bridging DSMLs with UML. *IJWIS*, 5(3):372–404, 2009.

[Wir71] Niklaus Wirth. Program Development by Stepwise Refinement. *Commun. ACM*, 14(4):221–227, 1971.

[WKK+10] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönboeck, and Wieland Schwinger. Towards an Expressivity Benchmark for Mappings based on a Systematic Classification of Heterogeneities. In *Proc. of Intl. Workshop on Model-Driven Interoperability (MDI)*, pages 32–41, 2010.

[WKS+09] Manuel Wimmer, Gerti Kappel, Johannes Schönböck, Angelika Kusel, Werner Retschitzegger, and Wieland Schwinger. A Petri Net Based Debugging Environment for QVT Relations. In *Proc. of Intl. Conf. on Automated Software Engineering (ASE)*, pages 3–14, 2009.

[WS05] Hiroshi Wada and Junichi Suzuki. Modeling Turnpike Frontend System: A Model-Driven Development Framework Leveraging UML Metamodeling and Attribute-Oriented Programming. In *Proc. of Intl. Conf. on Model Driven Engineering Languages and Systems (MoDELS)*, pages 584–600, 2005.

[WS07] Ingo Weisemöller and Andy Schürr. A Comparison of Standard Compliant Ways to Define Domain Specific Languages. In *Proc. of Intl. Workshops and Symposia on Models in Software Engineering*, pages 47–58, 2007.

[WTCJ11] Dennis Wagelaar, Massimo Tisi, Jordi Cabot, and Frédéric Jouault. Towards a General Composition Semantics for Rule-Based Model Transformation. In *Proc. of Intl. Conf. on Model Driven Engineering Languages and Systems (MoDELS)*, pages 623–637, 2011.

[WWB+13] Tim Waizenegger, Matthias Wieland, Tobias Binz, Uwe Breitenbücher, Florian Haupt, Oliver Kopp, Frank Leymann, Bernhard Mitschang, Alexander Nowak, and Sebastian Wagner. Policy4TOSCA: A Policy-Aware Cloud Service Provisioning Approach to Enable Secure Cloud Computing. In *Proc. of Intl. Conf. on the Move to Meaningful Internet Systems (OTM)*, pages 360–376, 2013.

[XLH+07] Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, and Hong Mei. Towards Automatic Model Synchronization from Model Transformations. In *Proc. of Intl. Conf. on Automated Software Engineering (ASE)*, pages 164–173, 2007.

Curriculum Vitae

Alexander Franz Bergmayr

Maria-Tusch-Straße 7/251
1220 Wien
Austria

Email: bergmayr@big.tuwien.ac.at
Web: http://www.big.tuwien.ac.at/staff/abergmayr
Date of Birth: 14-Aug-1980
Nationality: Austria

Work experience

07/2012 – present Researcher
Business Informatics Group, TU Wien, Austria

Main activities and responsibilities:
- ARTIST project, http://www.artist-project.eu, FP7 EU-funded, 317859 (09/2012 – 09/2015)
- Doctoral dissertation, “An Architecture Style for Cloud Application Modeling” (05/2016)

Master thesis co-supervision:
- Stefan Weghofer, “Moola: Model Operation Orchestration Language” (ongoing)
- Michael Kühberger, “MOF-based Metamodel Generation from (E)BNF Grammars” (ongoing)
- David Madner, “Model-based Deployment and Provisioning of Applications to the Cloud” (2014)

Bachelor thesis co-supervision:
- Michael Mittermayr, “Code generation for multiple class inheritance and enumeration inheritance” (2015)
- Alexander Altenhuber, “UML2Java – A Literature Review on Code Generation” (2015)

Research topics:
- Modeling & Metamodeling
- Software Modernization
- Reverse & Forward Engineering
- Cloud Applications & Environments

09/2008 – 07/2012 Researcher and Lecturer
Knowledge Engineering Group, University of Vienna, Austria

Main activities and responsibilities:
- PlugIT project, http://plug-it-project.eu, EU-funded 231430 (03/2009 – 08/2011)
- Teaching: Metamodeling (S2009, S2010, S2011, S2012), Modeling (S2012), IT Organisation (W2009), Large Scale IS (W2009), IS Infrastructure (W2010), IS Technology (W2010)

Research Topics:
- Modeling & Metamodeling
- Business-IT Alignment

05/2008 – 09/2008 Developer
team Communication Technology Management GmbH, Austria

Main activities and responsibilities:
Software development in the area of road traffic management

05/2008 – 07/2008 Researcher
Institute of Application Oriented Knowledge Processing, Johannes Kepler University of Linz, Austria

Main activities and responsibilities:
Contract research in the area of data migration

Research topics:
Data Migration
Software Modernization

09/2007 – 12/2007 Project assistant
Institute of Telecooperation, Johannes Kepler University Liny, Austria

Main activities and responsibilities
ModelCVS project, http://www.modelcvs.org, FFG-funded FIT-IT-810806

Research topics
Modeling & Metamodeling
Model Comparison & Versioning

Education

2002 – 2008 Bachelor/Master Studies Informatics (passed with distinction)
Johannes Kepler University Linz, Austria

01/2007 – 07/2007 Exchange semester
University of Reading, England

Publications

Journal articles

Alexander Bergmayr, Michael Grossniklaus, Manuel Wimmer, and Gerti Kappel. Leveraging Annotation-based Modeling with JUMP. To appear in Software and Systems Modeling (SoSyM), 2016

Conference papers

Alexander Bergmayr, Uwe Breitenbücher, Oliver Kopp, Manuel Wimmer, Gerti Kappel, and Frank Leymann. From Architecture Modeling to Application Provisioning for the Cloud by Combining UML and TOSCA. Accepted for publication in the Proceedings of the International Conference on Cloud Computing and Services Science (CLOSER), 2016 (**nominated as best paper candidate**)

Patrick Neubauer, Alexander Bergmayr, Tanja Mayerhofer, Javier Troya, and Manuel Wimmer. XMLText: From XML Schema to Xtext. In Proceedings of the International Conference on Software Language Engineering (SLE), 2015

Alexander Bergmayr, Michael Grossniklaus, Manuel Wimmer, and Gerti Kappel. UML Profile Generation for Annotation-based Modeling. In Proceedings of Software Engineering & Management: Multikonferenz der GI-Fachbereiche Softwaretechnik und Wirtschaftsinformatik, 2015

Alexander Bergmayr, Javier Troya, and Manuel Wimmer. From Out-Place Transformation Evolution to In-Place Model Patching. In Proceedings of the International Conference on Automated Software Engineering (ASE), 2014

Alexander Bergmayr, Michael Grossniklaus, Manuel Wimmer, and Gerti Kappel. JUMP – From Java Annotations to UML Profiles. In Proceedings of the International Conference on Model-Driven Engineering Languages and Systems (MoDELS), 2014 (**among the best papers**)

Alexander Bergmayr, Michael Grossniklaus, and Manuel Wimmer. Cloud Modeling Languages by Example. In Proceedings of the International Conference on Service Oriented Computing & Applications (SOCA), 2014

Alexander Bergmayr, Hugo Bruneliere, Javier Luis Canovas Izquierdo, Jesus Gorronogoitia, George Kousiouris, Dimosthenis Kyriazis, Philip Langer, Andreas Menychtas, Leire Orue-Echevarria, Clara Pezuela, and Manuel Wimmer. Migrating Legacy Software to the Cloud with ARTIST. In Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR), 2013

Alexander Bergmayr, Manuel Wimmer, Werner Retschitzegger, and Uwe Zdun. Taking the Pick out of the Bunch – Type-Safe Shrinking of Metamodels. In Proceedings of Software Engineering: Fachtagung des GI-Fachbereichs Softwaretechnik (SE), 2013

Florin Abazi and Alexander Bergmayr. Knowledge-Based Process Modelling for Nuclear Inspection. In Proceedings of the International Conference on Knowledge Science, Engineering and Management (KSEM), 2009

Workshop papers

Alexander Bergmayr, Hugo Bruneliere, Jordi Cabot, Jokin García, Tanja, Mayerhofer, and Manuel Wimmer. fREX: fUML-based Reverse Engineering of Executable Behavior for Software Dynamic Analysis. Accepted for publication in the Proceedings of the International Workshop on Modeling in Software Engineering (MiSE) co-located with the International Conference on Software Engineering (ICSE), 2016

Alexander Bergmayr, Alessandro Rossini, Nicolas Ferry, Geir Horn, Leire Orue-Echevarria, Arnor Solberg, and Manuel Wimmer. The Evolution of CloudML and its Manifestations. In Proceedings of the International Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE) co-located with the International Conference on Model-Driven Engineering Languages and Systems (MoDELS), 2015

Javier Troya, Alexander Bergmayr, Loli Burgueno, and Manuel Wimmer. Towards Systematic Mutations for and with ATL. In Proceedings of the International Workshop on Mutation Analysis (MUTATION) co-located with the International Conference on Software Testing, Verification and Validation (ICST), 2015

Alexander Bergmayr, Javier Troya, Patrick Neubauer, Manuel Wimmer, and Gerti Kappel. UML-based Cloud Application Modeling with Libraries, Profiles and Templates. In Proceedings of the International Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE) co-located with the International Conference on Model-Driven Engineering Languages and Systems (MoDELS), 2014

Alexander Bergmayr and Manuel Wimmer. Generating Metamodels from Grammars by Chaining Translational and By-Example Techniques. In Proceedings of the International Workshop on Model-driven Engineering By Example (MDEBE) co-located with the International Conference on Model-Driven Engineering Languages and Systems (MoDELS), 2013

Alexander Bergmayr. ReuseMe – Towards Aspect-Driven Reuse in Modelling Method Development. In Proceedings of Models in Software Engineering, Workshops and Symposia at the International Conference on Model-Driven Engineering Languages and Systems (MoDELS), 2010

Margit Schwab, Dimitris Karagiannis, and Alexander Bergmayr. i* on ADOxx®: A Case Study. In Proceedings of the International i* Workshop (iStar) co-located with the International Conference on Advanced Information Systems Engineering (CAiSE), 2010

Thomas Reiter, Kerstin Altmanninger, Gabriele Kotsis, Wieland Schwinger, and Alexander Bergmayr, Models in Conflict - Detection of Semantic Conflicts in Model-based Development. In Proceedings of the International Workshop on Model-Driven Enterprise Information Systems (MDEIS) co-located with the International Conference on Enterprise Information Systems (ICEIS), 2007

Tool demonstrations

Alexander Bergmayr, Michael Grossniklaus, Manuel Wimmer, and Gerti Kappel. Bridging Java Annotations and UML Profiles with JUMP. In Proceedings of the Demonstrations Track of the International Conference on Model-Driven Engineering Languages and Systems (MoDELS), 2014 (**received best tool demonstration award**)