Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diploma Thesis Nr. 3504

# An Extensible Application Topology Definition and Annotation Framework

Anja Reuter

| | |
|---|---|
| **Course of Study:** | Computer Science |
| **Examiner:** | Prof. Dr. Frank Leymann |
| **Supervisor:** | Dr. Vasilios Andrikopoulos |
| **Commenced:** | 2013-05-29 |
| **Completed:** | 2013-11-28 |
| **CR-Classification:** | D.2.2; D.2.7; H.4.2 |

# Abstract

This thesis introduces a framework for decision support during the design of applications for the cloud, or migration of existing applications to a cloud environment. For this purpose, a GENeralized Topology Language (GENTL) is introduced and mappings from existing languages to GENTL are provided. An annotation scheme for GENTL, which can capture annotations to topologies and topology elements is designed and instantiations for different annotation types are given. A framework implementing import functionalities for the topology languages Blueprint and TOSCA is presented. The framework enables the annotation of topologies with documentation annotations, references to external resources and incorporates a series of annotations which can be used to retrieve cost calculations from the external decision support system Nefolog.

# Contents

# List of Figures

# List of Tables

# List of Listings

# List of Abbreviations

| | |
|---|---|
| API | Application Program Interface |
| AWS | Amazon Web Services |
| BPEL | Business Process Execution Language |
| BPMN | Business Process Model and Notation |
| CaaS | Composite as a Service |
| Cafe | Composite Application Framework |
| DMM | Descartes Meta-Model |
| DOM | Document Object Model |
| DSL | Domain-Specific Language |
| EBNF | Extended Backus-Naur Form |
| EMF | Eclipse Modeling Framework |
| GEF | Graphical Editing Framework |
| GENTL | GENeralized Topology Language |
| GMP | Graphical Modeling Project |
| HTML | Hypertext Markup Language |
| IaaS | Infrastructure as a Service |
| JSON | JavaScript Object Notation |
| MOCCA | MOve to Clouds for Composite Applications |
| MVC | Model View Controler |
| OMG | Object Management Group |
| OWL | Web Ontology Language |
| PaaS | Platform as a Service |
| QoS | Quality of Service |
| SaaS | Software as a Service |
| SAWSDL | Semantic Annotations for WSDL and XML Schema |
| SBA | Service Based Application |
| SLA | Service Level Aggreement |
| SVG | Scalable Vector Graphics |
| TOSCA | Topology and Orchestration Specification for Cloud Applications |
| UML | Unified Modeling Language |
| URI | Uniform Resource Identifier |
| URL | Uniform Resource Locator |
| USDL | Unified Service Description Language |

| | |
|---|---|
| W3C | World Wide Web Consortium |
| WSDL | Web Services Description Language |
| WSLA | Web Service Level Agreement |
| XHTML | Extensible Hypertext Markup Language |
| XML | Extensible Markup Language |
| XSL | Extensible Stylesheet Language |
| XSLT | XSL Transformation |

# 1 Introduction

The design of composite, multi-tier applications for the cloud, as well as the migration of existing composite applications to the cloud requires various architectural decisions (cf. [ASL13]). Decisions to be made include partitioning of the applications into sets of components to be distributed to different clouds, the cloud provider to be chosen to host a specific component as well as choosing the implementation settings and options for a component which are best suited to fulfill the requirements on the component and on the application as a whole.

Architectural decisions have impact on the functional properties of the application as well as on the business aspects and Quality of Service attributes. To estimate the influence of design decisions on the characteristics of the resulting application, the application topology has to be formalized to capture the structure of the application.

Based on the formalized representation of the application and additional information on the architectural decisions, future application properties can be estimated. An example is the calculation of the cost for a specific application component, based on hosting information, like the desired service type, the intended cloud provider and the configuration parameters of service.

Many different formalized languages and frameworks exist to describe application topologies. They are designed for different purposes and focus on different areas of application, but the basic elements of those languages usually describe a graph consisting of application components and connections between them.

This thesis investigates the major existing application topology languages and frameworks and identifies the common concepts in different languages. It provides a categorization for application topology annotations into Discovery, Provision and Management and Design Support Annotations, as well as a distinction between different processing modes for annotations and between static and dynamic annotations. Existing annotations from different categories are presented.

Based on the common concepts in the presented application topology languages and frameworks, a new GENeralized Topology Language (GENTL) is designed to capture topologies from different languages. Mappings from existing languages to the new language are described. An annotation scheme for the designed language is provided, which is extensible and supports different kinds of annotations. Instantiations for static and dynamic annotations are given.

GENTL is implemented as an extensible application topology framework, which offers the functionality to import topologies from the two application topology languages Blueprint and TOSCA. Topologies are visualized as graphs and annotations following the designed annotation scheme can be attached to the topology elements.

Topologies as well as annotations on topologies and topology elements can be exported in a serialized representation of the designed application topology language. The framework is extensible to allow for new topology language imports and new topology annotations to be included into the framework.

The thesis is structured as follows: Chapter 2 describes the fundamental concepts and related work, including existing topology languages and topology annotations. In Chapter 3, the GENeralized Topology Language (GENTL) is introduced, and Chapter 4 specifies the annotation scheme for GENTL. Chapter 5 depicts the implementation of the topology annotation framework and Chapter 6 evaluates the framework. Chapter 7 concludes the thesis.

# 2 Fundamentals and Related Work

This section is structured as follows: In Section 2.1 some basic terms in cloud computing and web services are described. Section 2.2 presents exemplary application topology languages and frameworks and Section 2.3 provides an overview on annotations on application topologies.

## 2.1 Basic Terms in Cloud Computing and Web Services

This section describes the basic terms and standards in cloud computing and web services necessary for the comprehension of the remainder of this thesis. A comprehensive view on cloud computing is given e.g. in [Ley09, AFG$^+$09, BGPCV12] and detailed information about standards and technologies in web services can be found in [WCL05, wsa04].

### 2.1.1 Cloud Computing Terms

The general term “cloud” or “cloud system” comprises different possible deployment models. The following four deployment models exist for a cloud or a cloud system (c.f. [BGPCV12]):

- **Private Cloud:** Provisioned for the exclusive use by a single organization, including multiple consumers (e.g., business units). It may exist on or off premises and can be owned, managed and operated either by the organization, a third party or a combination of both.
- **Community Cloud:** Provisioned to be used by a specific community of consumers from multiple organizations. It may exist on or off premises and can be owned, managed and operated by one or more organizations in the community, a third party or a combination of them.
- **Public Cloud:** Provisioned for open and public use. It exists on the premises of the cloud provider, who owns, manages and operates the public cloud. The cloud provider may be a business, academic or government organization, or a combination of them.

- **Hybrid Cloud:** Two or more different cloud infrastructures (private, community, or public clouds) are composed and bound together to enable data and application portablity

Considering the services offered in the cloud, the following three service models are usually distinguished (c.f. [BGPCV12]):

- **Cloud Infrastructure as a Service (IaaS):** Consumers can provision processing, storage, networks and other fundamental computing resources, without managing or controlling the underlying cloud infrastructure. The consumer can deploy and control operating systems, storage and applications on the provisioned resources.
- **Cloud Platform as a Service (PaaS):** Consumers can deploy and control applications in supported programming languages onto the provided cloud infrastructure. The underlying cloud infrastructure including network, servers, operating systems or storage lies not within the control of the consumer.
- **Cloud Software as a Service (SaaS):** Consumers can access and run applications from the cloud provider running on the provider's infrastructure, through a thin client interface or a program interface. Aside from user-specific application configuration settings, the user maintains no access to management or control issues of the underlying cloud infrastructure.

The three service models form a stack, with SaaS resting on top of PaaS residing on IaaS. In [Ley09], a fourth layer called **Composite as a Service (CaaS)** is suggested on top of this stack to represent compositions of services from the IaaS, PaaS and SaaS layers of the stack.

### 2.1.2 Web Services Standards and Technologies

The World Wide Web Consortium (W3C) uses the following definition of a **Web Service:** "A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."[wsa04]

The **Web Services Description Language (WSDL)** provides an XML vocabulary for the description of web services. It consists of an abstract part, describing the operational behavior of a service through the in- and outgoing messages of the service, and a concrete part, which defines how and where to access the service implementation. The operational behavior is described in syntactic and structural terms, the semantics of a service are not in the scope of WSDL (cf. [WCL05, WSD07]).

**SOAP** is a messaging framework, providing a standardized XML based message structure, a processing model and a binding mechanism to different network protocols, as well as a way to attach non-XML encoded information to SOAP messages (cf. [WCL05]).

The **Business Process Execution Language (BPEL)** is an extensible workflow-based language that provides a way to create service choreographies. A BPEL process creates an arrangement of service interactions on the basis of the service interfaces described in WSDL. The process itself exposes WSDL interfaces and the corresponding service can be used in other choreographies (cf. [WCL05]).

## 2.2 Application Topology Languages and Frameworks

Distributed applications and applications in the cloud usually consist of a set of components and relationships between components. An application topology is the graph that is formed out of these components and relationships. It defines how the different components interact and depend on each other.

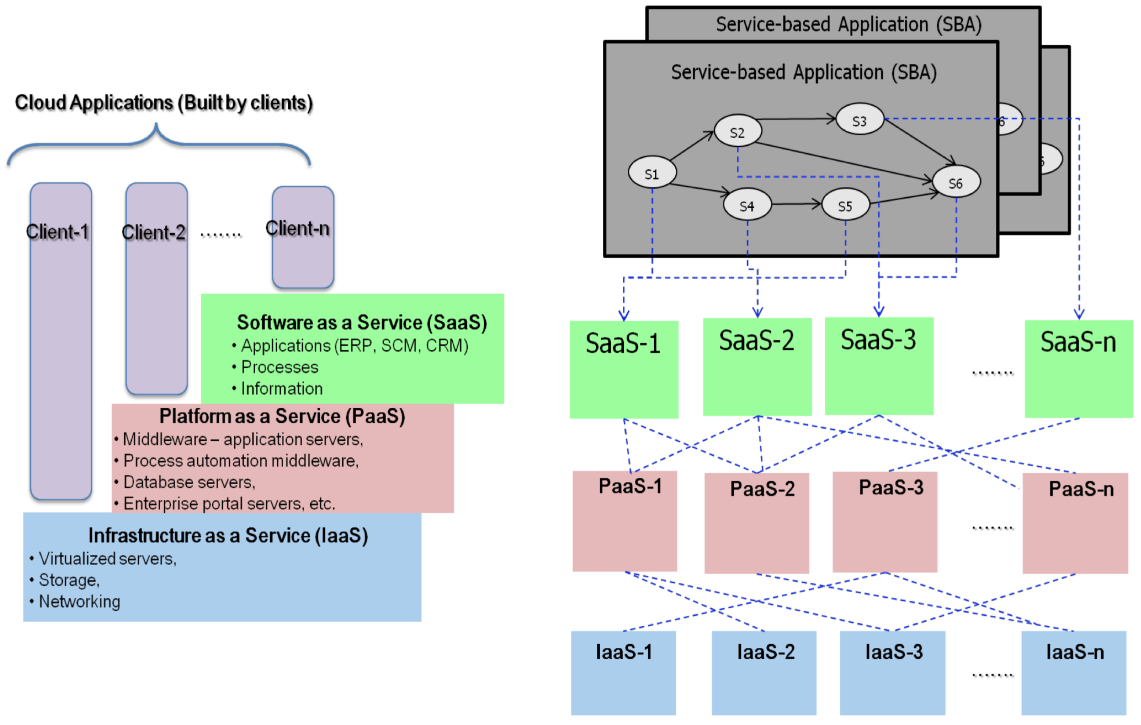
This section discusses existing application topology languages and frameworks with a focus on topology concepts for applications in the cloud. In the beginning, two general concepts for the design of Service Based Applications (SBA) in the cloud, namely the Blueprint Approach and TOSCA will be discussed followed by a description of Cafe and MOCCA, a framework for the automatic deployment of composite applications and a metamodel offering support for moving composite applications to the cloud.

In Section 2.2.4, the realization of Application Topologies in AWS CloudFormation, Flexiant Cloud Orchestrators Bento Boxes, OpenStack and OpenNebula are presented as examples for Application Topology Frameworks in Cloud Management Tools. Section 2.2.5 introduces a proposed abstraction layer for resource provision and a corresponding metamodel for application deployment in the cloud. The description of application topology languages and frameworks concludes with the discussion of the two general architecture modeling languages DMM and UML, followed by the service description language USDL. A comparison of the different approaches is given in Section 2.2.9.

### 2.2.1 Blueprint

The Blueprinting Approach as described in [NLPH12, PH11] introduces a syndicated multi-channel cloud delivery model, which is contrasted to the monolithic cloud stack solutions usually provided by cloud service vendors.

The standard monolithic cloud stack solution as illustrated in Figure 2.1a is composed of the Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as


Cloud Applications (Built by clients)
Client-1
Client-2
.......
Client-n
Software as a Service (SaaS)
• Applications (ERP, SCM, CRM)
• Processes
• Information
Platform as a Service (PaaS)
• Middleware – application servers,
• Process automation middleware,
• Database servers,
• Enterprise portal servers, etc.
Infrastructure as a Service (IaaS)
• Virtualized servers,
• Storage,
• Networking
Service-based Application (SBA)
Service-based Application (SBA)
S1
S2
S3
S4
S5
S6
SaaS-1
SaaS-2
SaaS-3
SaaS-n
PaaS-1
PaaS-2
PaaS-3
PaaS-n
IaaS-1
IaaS-2
IaaS-3
IaaS-n

(a) Monolithic Delivery Models (b) Syndicated Multi-channel Delivery Model

**Figure 2.1:** Monolithic Cloud Stack Solutions vs. Syndicated Multi-channel Cloud Delivery Model Introduced in the Blueprint Approach [NLPH12].

a Service (SaaS) offerings from a single cloud service provider. Customers can either use the IaaS from the provider, or the PaaS which is deployed on the vendors IaaS, or the SaaS which runs on top of the vendors PaaS and IaaS.

The Blueprint approach aims at a more flexible architecture for SBAs. The objective is an architecture model which is able to combine IaaS, PaaS and SaaS offerings from multiple vendors as illustrated in Figure 2.1b, to enable the development of customized, flexible and agile SBAs. To capture and formalize such an architecture, the Blueprint Template is introduced.

A Blueprint gives an abstracted description of a cloud service offering and is composed out of the following six parts as (depicted in Figure 2.2):

- **Basic Properties:** General properties describing the blueprint (e.g. ID, ownership, release date etc.)

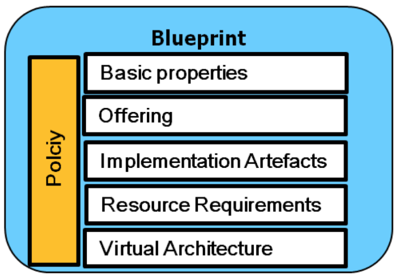


**Figure 2.2:** Fundamental Blueprint Structure [NLPH12].

- **Offering:** The specification of one or many cloud service offerings. This specification of a cloud service offering includes the names, functionalities, signature interfaces, interaction protocols, elasticity offerings and Quality of Service (QoS) offerings.
- **Implementation Artifacts:** Description of the elements necessary to implement the offerings (may include binary files, configuration files or deployable web packages).
- **Resource Requirements:** The description of the cloud resources that are necessary for the deployment of the implementation artifacts. The description includes the QoS requirements of the respective cloud resources. A Resource Requirement can be fulfilled by discovering a Blueprint of a matching cloud resource.
- **Virtual Architecture:** Depiction of the virtual architecture desired by the application developers. The virtual architecture is given by the interdependencies and interconnections between the offerings, implementation artifacts and resource requirements across blueprints. These interdependencies have to be specified by the application developer.
- **Policy:** The specification of the policy constraints that all elements of the blueprint (offering, implementation artifacts, resource requirements) have to comply with.

The application topology depicted by a blueprint is the graph representation of the virtual architecture, that links the specified elements (offerings, implementation artifacts or resource requirements) of the blueprint.

The reusability is part of the core concept in the Blueprinting Approach and the development process relies on marketplace repositories for the publication, purchase and reuse of Blueprints. A Blueprint under development is called Target Blueprint and a Blueprint that has been submitted to a marketplace repository to be purchased and reused by other developers is called a Source Blueprint.

To create a deployable Source Blueprint out of a Target Blueprint, the developer has to complete the first five phases of the Blueprint lifecycle:

1. In the **Target Blueprint Design** phase, the developer of a new SBA in the cloud uses the Blueprint Template to define a Target Blueprint. The Target Blueprint contains a description of the functionalities offered by the SBA, a definition of the required resources, the necessary policy constraints and the desired architecture topology.
2. In the **Target Blueprint Resolution** phase, the specified resource requirements are resolved with Source Blueprints of matching SaaS, PaaS or IaaS offerings which fulfill the requirements. This step results in a set of alternatives to resolve the required resources in the Target Blueprint with different cloud service offerings.
3. Choosing which of the alternative solutions found in the previous phase should be used for the SBA is done in the **Target Blueprint Customization** phase.
4. In the **Target Blueprint Checking & Testing** phase, the resolved Blueprint is tested. If the tested Blueprint satisfies all criteria specified for the final cloud application, the application can be deployed, or the Blueprint can be stored in a marketplace/repository to be reused by other developers. If any faults or false behavior is found during testing, the developer can either go back to the Target Blueprint Customization phase and choose different service offerings as resources, or go back to the first step and redesign the Target Blueprint.
5. The next step is the **Target Blueprint Deployment**, in which the application is deployed following a deployment plan that ensures SLAs for the correct QoS specified by the developer.
6. In the **Target Blueprint Monitoring** phase, the operational performance of the deployed application must be monitored. If an SLA constraint is violated, the developer should be notified. To reestablish the fulfillment of all SLAs, the developer can choose between a redeployment, a new Target Blueprint resolution or a completely new design of the cloud application.

### 2.2.2 TOSCA

The Topology and Orchestration Specification for Cloud Applications (TOSCA) defined in [tos13] provides a language specification for a portable description of service components, their relationships and management procedures. TOSCA's objective is to make the semi-automatic creation and management of application layer services portable across alternative cloud implementation environments while the services remain interoperable.

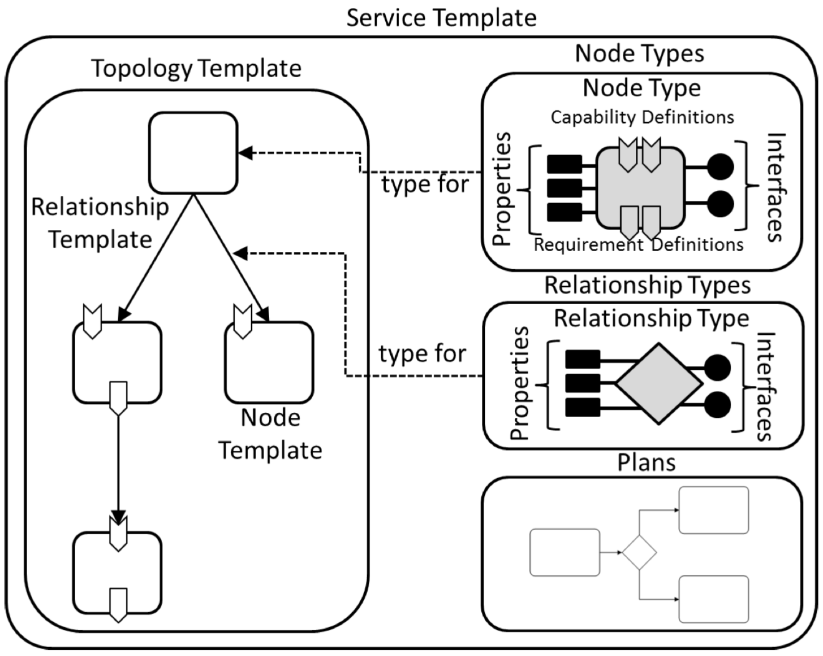


**Figure 2.3:** Structural Elements of a TOSCA Service Template and their Relations [tos13].

To describe everything that has to be preserved across service deployments in different environments, the TOSCA Service Template uses a combination of a service topology and orchestration processes. The Service Template provides a foundation for the interoperable deployment of cloud services, as well as for the service management throughout the lifecycle and for the porting of services over alternative cloud environments.

The structure of the Service Template is depicted in Figure 2.3 and contains the following elements:

- **Node Type:** Definition of the properties and interfaces of a certain type of component. A property of a node type for a server may for example be the IP address of a server instance. The interfaces describe the operations available to manipulate the component, for example an operation to start or shut down a server instance. The Requirement and Capability Definitions depicted in the Node Type can be used to express a components requirements against other components or the hosting environment and the functionalities the component offers to other components.
- **Node Template:** Representation of a specific component as a reference to a defined node type with added usage constraints, such as the number of times a component can occur.

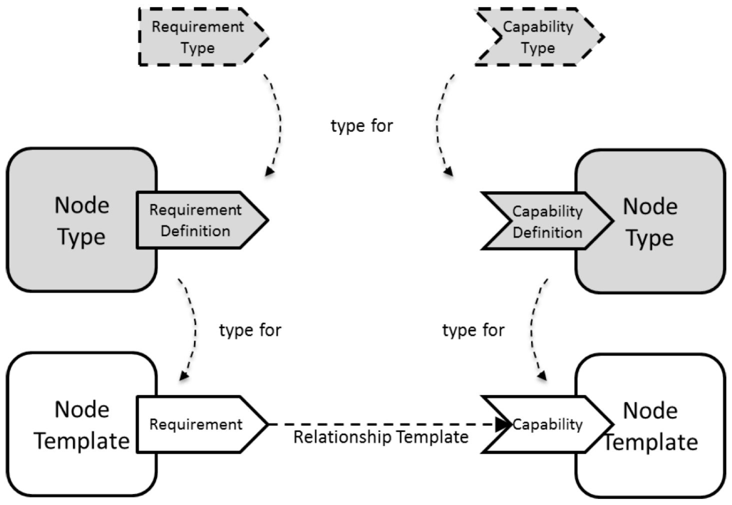


**Figure 2.4:** Requirements and Capabilities for Node Types and Templates in TOSCA [tos13].

- **Relationship Type:** Definition of the semantics and properties of a type of relationship between node templates, such as “hosted by” or “deployed on”.
- **Relationship Template:** Specification of a relationship between two node templates. Each relationship template refers to a relationship type and indicates the direction of the relationship by representing the connected components in a nested source and target element.
- **Topology Template:** The service components and their relationships form a graph with components as nodes and relationships between components as edges of the graph. The Topology Template represents this graph as a set of node and relationship templates.
- **Plans:** Description of the management aspects of service instances like the creation and termination of service instances. The management issues are described in process models which can either be included as part of the plan or as a reference to a separate process model. Process models can be defined in any existing language (like BPMN or BPEL) and can contain tasks referring to operations of Interfaces of Node Templates/ Types or Relationship Templates/ Types.

As illustrated in Figure 2.4, Node Types can be annotated with Requirement and Capability Definitions which are of a certain Requirement/Capability Type. If a Requirement Type for a database connection is defined, different Node Types can have a Requirement Definition of this Type.

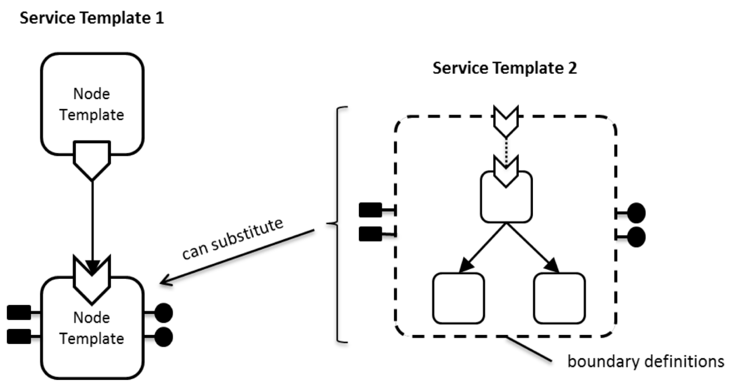


**Figure 2.5:** Service Template Composition in TOSCA [tos13].

A Node Template which is of a Node Type with a Requirement Definition, has a corresponding Requirement with concrete property values specific to the Node Template. A property value for a database connection requirement could be for example the minimum capacity or the type of database that is required.

The structure of Capability Definitions, Capability Types and Capabilities corresponds to the structure for Requirement Definitions, Requirement Types and Requirements. In the Topology Template, a Requirement of one Node Template can be connected to a Capability of another Node Template to indicate that the particular Requirement is fulfilled by the chosen Capability.

Non-functional behaviors and QoS can be expressed via Policy Types and Policy Templates in TOSCA. A Policy Type expresses a certain kind of non-functional behavior or QoS which can be provided by a Node Type (e.g. high availability). Policy Templates define a particular non-functional behavior or QoS of a Node Template. Each Policy Template refers to a specific Policy Type.

TOSCA also offers the possibility to compose different Service Templates as shown in Figure 2.5. A Node Template can be substituted by a Service Template if the Service Template has the same boundary definitions (i.e. properties, interfaces, requirements, capabilities) as the Node Template.

Deployment and implementation artifacts can be defined with Artifact Types and Templates. An Artifact Template represents a deployment or implementation artifact which can be referenced by objects in a Service Template. Each Artifact Template refers to an Artifact Type, which specifies the type and structure of the artifact.

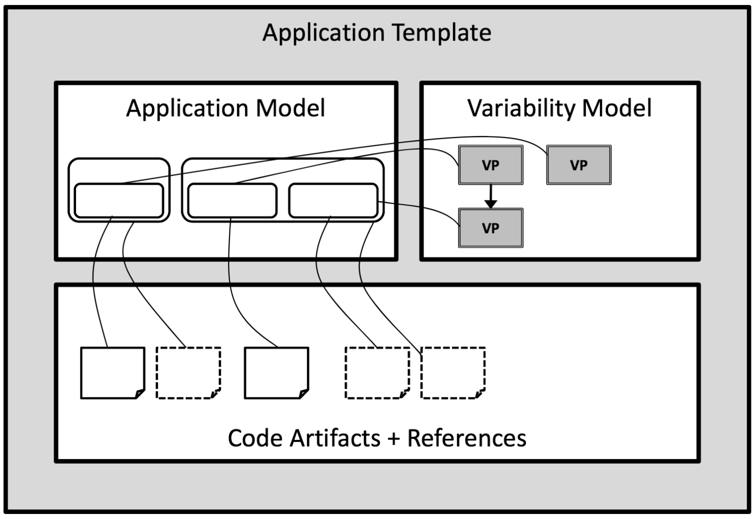


**Figure 2.6:** Ingredients of a Cafe Application Template [Mie10].

### 2.2.3 Cafe and MOCCA

#### Cafe

The Composite Application Framework (Cafe) framework presented in [Mie10, MUL09] provides the means to describe composite service-oriented applications and provision them automatically across different providers. Figure 2.6 shows the ingredients of a Cafe application template.

A Cafe application template consists of an application model, a variability model and code artifacts and references. The variability model captures variability points for the parameterization of the application. The variability points are used in the deployment process along with the code artifacts and references.

The topology of a Cafe application is captured in the application metamodel depicted in Figure 2.7. An Application Model is a Component, which consists of a set of Components and can also be used as Component in other Application Models. Components can be deployed on other Components and are implemented by an Implementation. An Implementation can be realized by a set of files or supplied by the provider. Components have an associated Component Type and Implementations are of a specific Implementation Type.

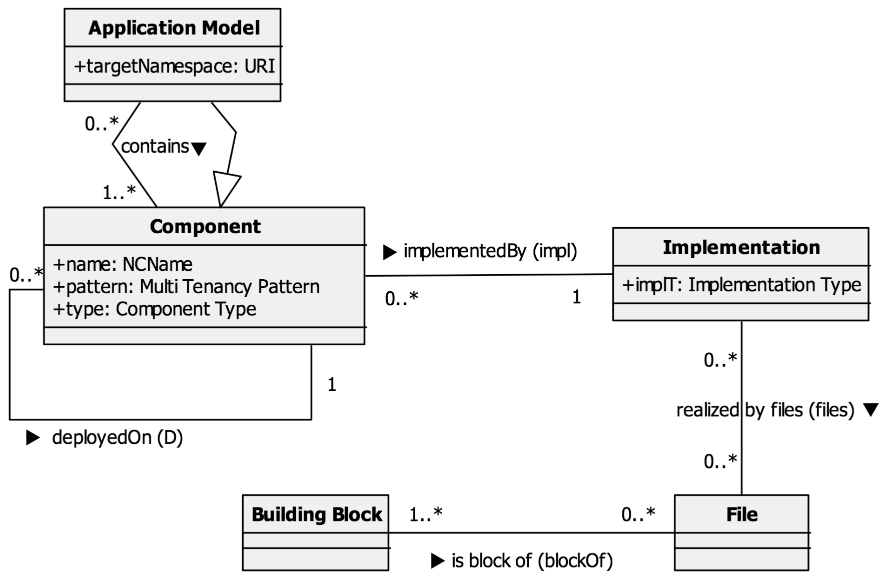


**Figure 2.7:** Cafe Application Metamodel [Mie10].

## MOCCA

The MOCCA Method (MOve to Clouds for Composite Applications) presented in [LFM+11] formalizes the necessary steps to be done when moving an application to the cloud. It introduces a metamodel which supports to split an application and move parts of it to different clouds, including automatic provisioning in the different target clouds.

MOCCA requires an architecture model of the application, a deployment model of the application and implementation artifacts such as virtual images of (parts of) the application to be provided.

To rearrange and provision an application in the cloud, the MOCCA method proposes to produce and combine artifacts in the following steps:

1. Provide an architecture model of the application to be moved to the cloud. The architecture model describes the architecture of the application by giving the components and their relations similar to the application topologies discussed.
2. Provide a deployment model of the application. This can be done as an enrichment of the architecture model with deployment information. The deployment model defines the required runtime containers for the application and the distribution of the architecture components across the containers.

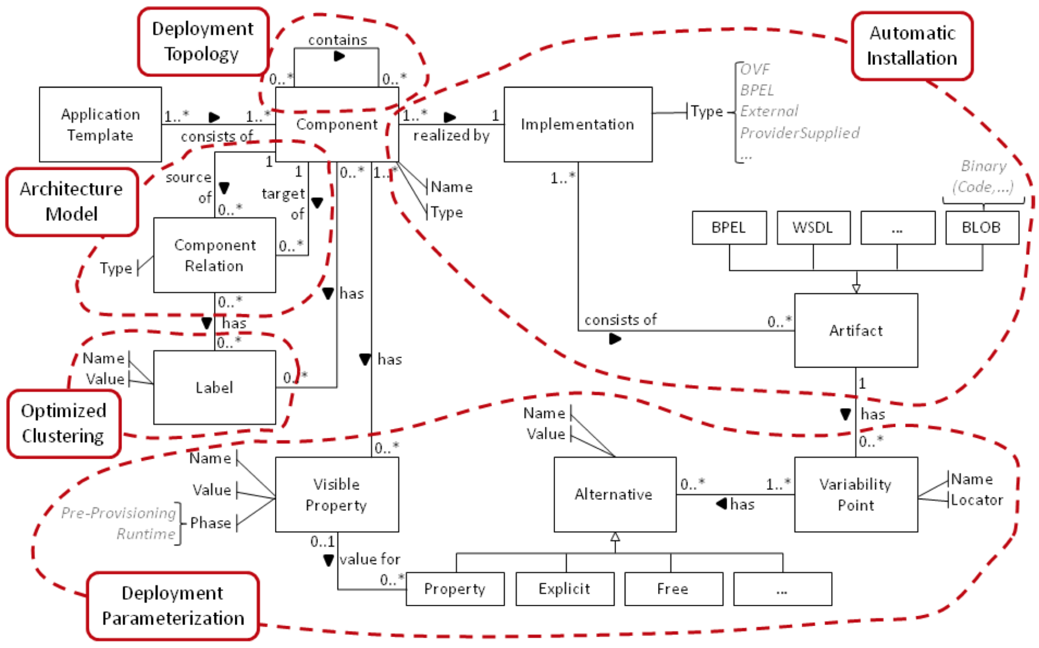


**Figure 2.8:** The Model Types and Metamodel used for MOCCA [LFM+11] .

3. Rearrange the architecture model into groups of components that belong into the same cloud. The resulting segmentation is called a cloud distribution
4. Provide all implementation units required for the application, like executables or virtual images of (parts of) the application. This supports automatic provisioning.
5. Form a provision cluster by combining the cloud distribution and the combined architecture/deployment model annotated with the required implementation units. The provision cluster represents all the information needed to provision the rearranged application into its target clouds.

Figure 2.8 illustrates how the MOCCA metamodel represents the artifacts used in the MOCCA method. The metamodel is an adaption of the Cafe application metamodel and variability model.

An application is represented by an Application Template which consists of one or more Components. Each Component may contain other Components and it is source as well as target of zero or more Component Relations. The Components and Component Relations formalize the architecture model of the application. The deployment model of the application is represented by Components containing other Components. This models runtime containers and the distribution of architecture components across the containers.

Every Component Relation and every Component has zero or more Labels and a Label is defined as a pair of a name and a value attribute. The Labels are used to provide the necessary information for the generation of the cloud distribution.

Each component is realized by an Implementation which consists of zero or more artifacts. The Implementation Artifacts are used for automatic installation of the application. To support deployment parameterization, an Implementation Artifact can have zero or more Variability Points providing zero or more alternatives for deployment.

In the context of MOCCA, the relevant types of alternatives are Explicit Alternatives, Free Alternatives and Property Alternatives. Explicit Alternatives provide a choice between a set of pre-defined values, Free Alternatives take an arbitrary value as input and Property Alternatives point to a Visible Property of a Component. Visible Properties are made visible to the outside for the purpose of overwriting.

### 2.2.4 Topology Frameworks in Cloud Management Tools

The main focus of cloud management tools lies on the deployment and management functionalities as well as security and billing issues. In consequence, the representations of application topologies aim at easy deployment and management of the used resources.

In this section, the cloud management tools AWS CloudFormation, Flexiant Cloud Orchestrator, OpenStack and OpenNebula are discussed as examples for cloud provision and management tools.

#### AWS CloudFormation

AWS CloudFormation [clo13] provides the functionality to create and manage collections of related Amazon Web Services resources. CloudFormation stacks are described in a JSON based template format. To create a new CloudFormation stack, the user can either choose one of the sample templates provided by AWS CloudFormation, or create a custom template.

Figure 2.9 depicts the basic structure of an AWS CloudFormation template. Templates can be used repeatedly to create multiple instances of the same stack. Parameters can be used to customize a template, or create stacks with the same basic structure, which differ in certain settings (e.g. different settings according to the amount of expected traffic).

The templates represent the application topology of an AWS CloudFormation stack. Application topology reusability is supported by the reuse, parameterization and customization of existing templates.

```
{
   "Description" :  "A text description for the template usage",
   "Parameters":  {
            // A set of inputs used to customize the template per deployment
            },
   "Resources" :  {
            // The set of AWS resources and relationships between them
            },
   "Outputs" :  {
            // A set of values to be made visible to the stack creator
            },
   "AWSTemplateFormatVersion" :  "2010-09-09"
}
```

**Figure 2.9:** Structure of AWS CloudFormation Templates [clo13].

### Flexiant Cloud Orchestrator

Flexiant Cloud Orchestrator [ben13] is a software solution for cloud management developed by Flexiant Limited. It provides functionality for Resource Management, Access Control, Metering and Billing and since the Flexiant Cloud Orchestrator V3, Flexiant added a feature called Bento Boxes.

Bento Boxes offers a drag and drop user interface that allows service providers and their customers to build, deploy and share software templates graphically. The interface is intended to provide a way for non-IT-Experts to deploy and customize pre-packaged, configured and sophisticated application stacks.

The service providers can enhance the preconfigured application stacks by adding so called Deployment Questions, which are presented to the customers in the graphical interface each time a template is deployed. The obtained information can then be used for runtime configuration and customization of a standard template or it can be combined with tools like Chef or Puppet for more advanced use cases.

For the customer, this results in a deployment process which only requires the customer to enter necessary information (like e.g. user names and passwords) once in the beginning to start an otherwise completely automated deployment process. Through the graphical Interface the reuse of an application topology by a customer is possible without extensive knowledge about the technical details of the application.

### Open Stack

The open source software OpenStack [ope13b] is a cloud computing platform for public and private clouds. It is developed and maintained by a global collaboration of software developers and is composed out of the following parts as illustrated in Figure 2.10:

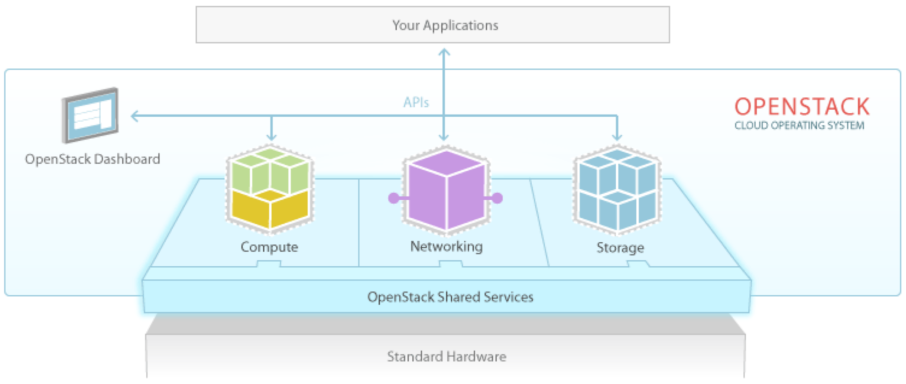


**Figure 2.10:** Structure of the OpenStack Software [ope13b].

- **Compute** enables the provision and management of large networks of virtual machines.
- **Storage** provides object and block storage for servers and applications.
- **Networking** supplies a pluggable, scalable, API-driven network and IP management.
- **Dashboard** is a graphical interface for users and administrators to access, provision and automate cloud-based resources.
- **Shared Services** integrates the OpenStack components with each other as well as with external systems through services like identity and image management and a web interface to provide a unified experience for the user interacting with different cloud resources.

The structure of clouds managed with OpenStack is represented in the Compute, Storage and Networking part. The Dashboard is an entry point to the underlying structure of the cloud. As a graphical interface it provides an overview about the components from the Compute and Storage part of OpenStack as well as the connections managed by the Networking part.

### OpenNebula

OpenNebula [ope13a] is an open-source industry standard for data center virtualization. The primary use case for OpenNebula is the management of virtualized infrastructures in data centers or clusters. It also offers support for combining local infrastructure with public cloud-based infrastructure and it provides cloud interfaces to expose its functionality for virtual machine, storage and network management.

As key features for the management and orchestration of the virtual infrastructure, OpenNebula lists:

- Virtual infrastructure management adjusted to enterprise data centers
- Complete life-cycle management of virtual resources
- Powerful hooking system
- Full control, monitoring and accounting of virtual infrastructure resources
- Fine-grained multi-tenancy

These features in combination with the provided interfaces are the control mechanisms for the underlying application topology. The application topology in OpenNebula consists of Virtual Machines, Data Stores, Networking Components and Hosts and Clusters.

OpenNebula also provides an appliance marketplace offering virtual appliances ready to run in OpenNebula environments. The appliance marketplace provides a way to reuse existing application topologies and integrate them into new applications or build new applications on top of existing ones.

### 2.2.5 CloudML

The process of provisioning resources and deploying applications in the cloud varies between the different cloud providers. An application which was implemented to be deployed on a certain cloud may have to be redesigned to be deployable on another cloud. To avoid the resulting vendor lock-in, in [BMM12b] an abstraction layer is proposed to model available resources in the cloud. Figure 2.11 depicts the CloudML metamodel.

The Domain-Specific Language (DSL) CloudML provides a platform-independent model for the specification of the needed resources for an application. The CloudML engine uses this model to create a run-time model of the provisioned resources. The run-time model handles the interaction with the provisioned resources and the deployment of the application.

In [BMM12a], a domain-specific language for the deployment of applications in the cloud is proposed. The language Pim4Cloud DSL provides a component based metamodel, which is depicted in Figure 2.12. A component can be a scalar or a composite, which contains sub-components. Components can offer or require deployment services which are used to deploy one component onto another and connectors are used to describe links between components. The properties offered and expected by components are used at runtime to establish RuntimeBindings for information transfer. Defined components can be reused in other deployments (i.e. architectural patterns can be expressed as components).

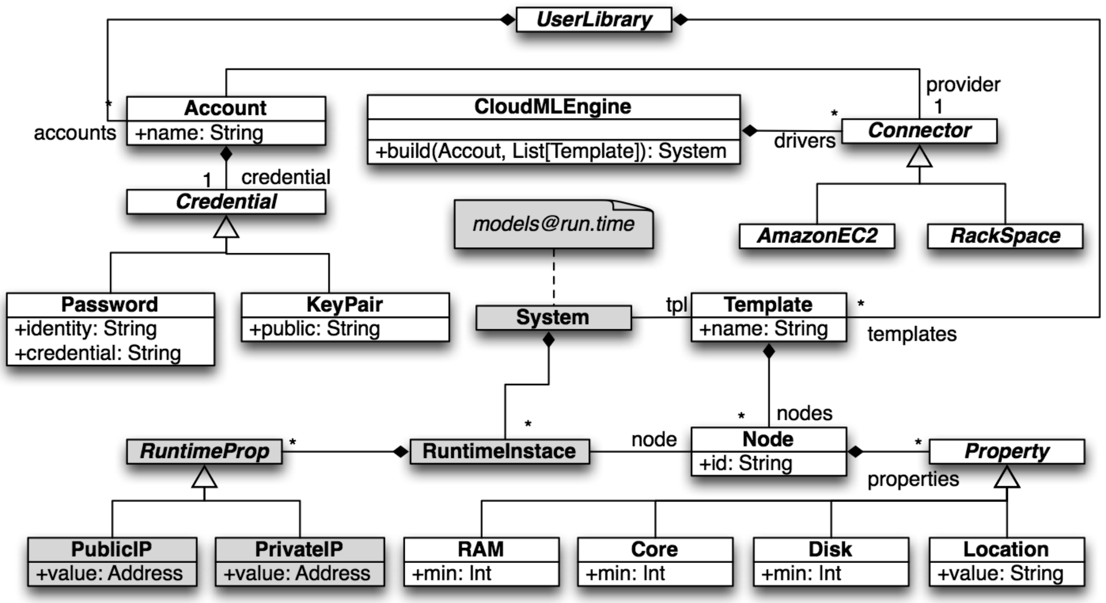


**Figure 2.11:** Architecture of CloudML [BMM12b] .

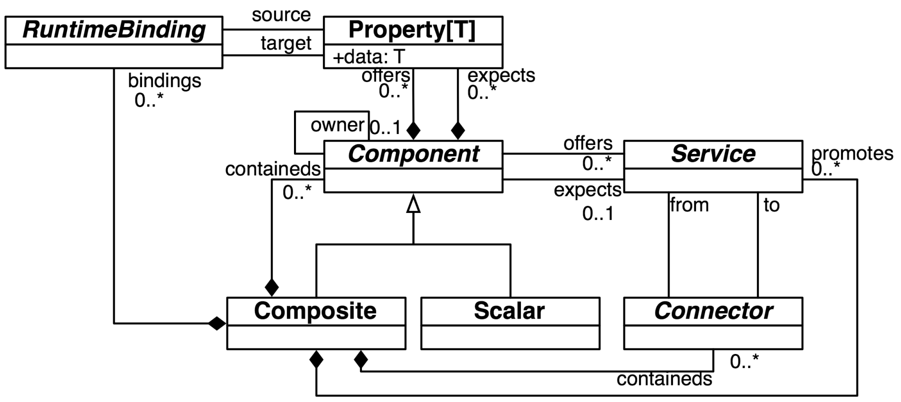


**Figure 2.12:** Metamodel of Pim4Cloud DSL [BMM12a] .

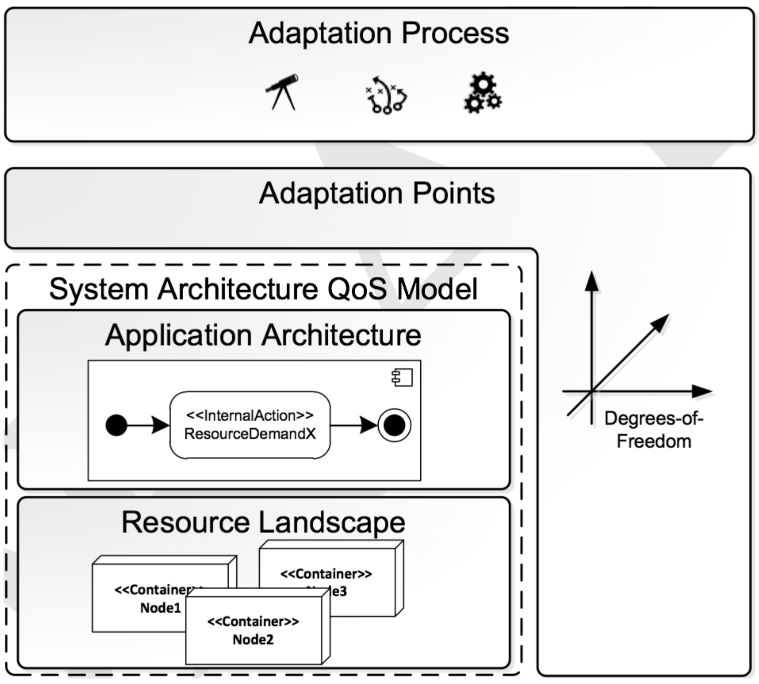


**Figure 2.13:** Structure of the Descartes Meta-Model [KBH12].

### 2.2.6 Descartes Meta-Model

The Descartes Meta-Model (DMM) defined in [KBH12] is an architecture-level modeling language for modern dynamic IT systems, infrastructures and services. DMM model instances are meant to be used for self-aware system adaption at runtime. It focuses on online performance prediction to ensure compliance to SLAs while optimizing resource utilization. The Descartes Meta-Model consists of the following four sub-meta-models depicted in Figure 2.13:

**Resource Landscape:** Describes the resources and their structural order. It provides the means to describe the computing infrastructure and its physical resources as well as the different layers within the system which also provide logical resources. Additionally, the layered execution environment can be described, which increases the flexibility and reuse of reoccurring container types.

To enable the prediction of system performance changes at runtime, the influences of individual layers on the systems performance, the dependencies among these influences and the resource allocations at each layer can also be captured as part of the model.

**Application Architecture:** The application architecture is depicted as a component-based software system. To capture the behavior and resource consumption of a component, the implementation of the component, existing dependencies on external services and their performance as well the usage profile and the execution environment in which the component is running can be taken into account. The performance behavior of a system is the performance behavior assembled over all components.

**Adaptation Points:** The adaptation points meta-model captures the variable parts of the resource landscape and application architecture which can be adapted during runtime. It reflects the possible valid states of the system architecture without the specification of the actual changes on the model instance or the system.

**Adaptation Process:** The adaptation process meta-model uses the adaptation points model to specify the adaptations to changes in the system. It provides means to describe system adaptation processes at the system architecture level with a set of modeling abstractions to specify strategies, tactics and actions in a generic, human-understandable and reusable way.

The application architecture and the resource landscape meta-model form the system architecture QoS model. It specifies the application topology enhanced by information on performance influences and dependencies.

### 2.2.7 Unified Modeling Language

The Unified Modeling Language (UML) specified by the Object Management Group (OMG) in [Gro11] is widely used to model application structures, behaviors and architectures as well as business processes and data structures.

UML 2.0 defines thirteen types of diagrams divided into the three categories of Structure Diagrams, Behavior Diagrams and Interaction Diagrams. The modeling of application topologies is captured in Structure Diagrams. Structure Diagrams are the Class Diagram, Object Diagram, Component Diagram, Composite Structure Diagram, Package Diagram, and Deployment Diagram.

The Class Diagram models the static elements of a system and the characteristics of and relationships between those elements. The classes in the diagram can contain attributes and operations and a class can realize or require a specific interface. Abstract classes and templates can be defined and stereotypes can be used to specify the purpose or type of an element. Relationships can be expressed as associations, aggregations, compositions or generalizations. (cf. [Kec05, p. 29-112])

The Object Diagram can be regarded as a special kind of class diagram, which captures a snap shot of the objects, attribute values and relationships in a limited timespan during runtime. It captures objects which are class realizations and links which specify the relationships between the objects. (cf. [Kec05, p. 113-126])

The Component Diagram captures the organization and the dependencies of components in a system. It consists of components and artifacts which can be nested in components. The components communicate with each other over ports and interfaces. (cf. [Kec05, p. 149-164])

The Composite Structure Diagram models the internal structure of components and the possible collaborations of components. The internal structure is represented by parts, ports which enable interaction between parts, classes and their surroundings and connectors between parts, ports or classes. A collaboration is an abstract description of a functionality offered by interconnected elements and a composite structure is a concrete application of a collaboration. (cf. [Kec05, p. 127-148])

The Package Diagram captures the organization of elements in packages. A package groups classes or other packages and defines namespaces. It can access or import other packages and the contents of two packages can be merged via package-merge. (cf. [Kec05, p. 175-198])

The Deployment Diagram models the architecture of distributed systems at runtime. Nodes in the diagram represent computing resources and artifacts in the nodes model software components running on the computing resource. Communication paths are depicted by connections between the nodes. (cf. [Kec05, p. 165-173])

### 2.2.8 Unified Service Description Language

The Unified Service Description Language (USDL) described in [USD11] provides a platform-neutral, UML based language for service description. USDL merges information from the business, operational, and technical level of a service into one language. This is achieved by the definition of a set of interconnected UML class models.

Figure 2.14 depicts the different modules of USDL and their dependencies. The service module is the center of USDL and describes the fundamental concepts of the service as well as the relations to the other service description modules. Concepts which are pertaining to several aspects of the service description, like e.g. naming conventions, or which are independent of the service itself, like persons or organizations can be captured in the foundations module.

The technical module captures the available access methods of a service like interface and access protocols and the functional module describes the offered functionality of a service (e.g. function, parameter and fault).

Information about the participating actors in provisioning, delivery and consumption of a service can be captured in the participants module. Participants include for example the provider, intermediary, stakeholder or consumer of a service.

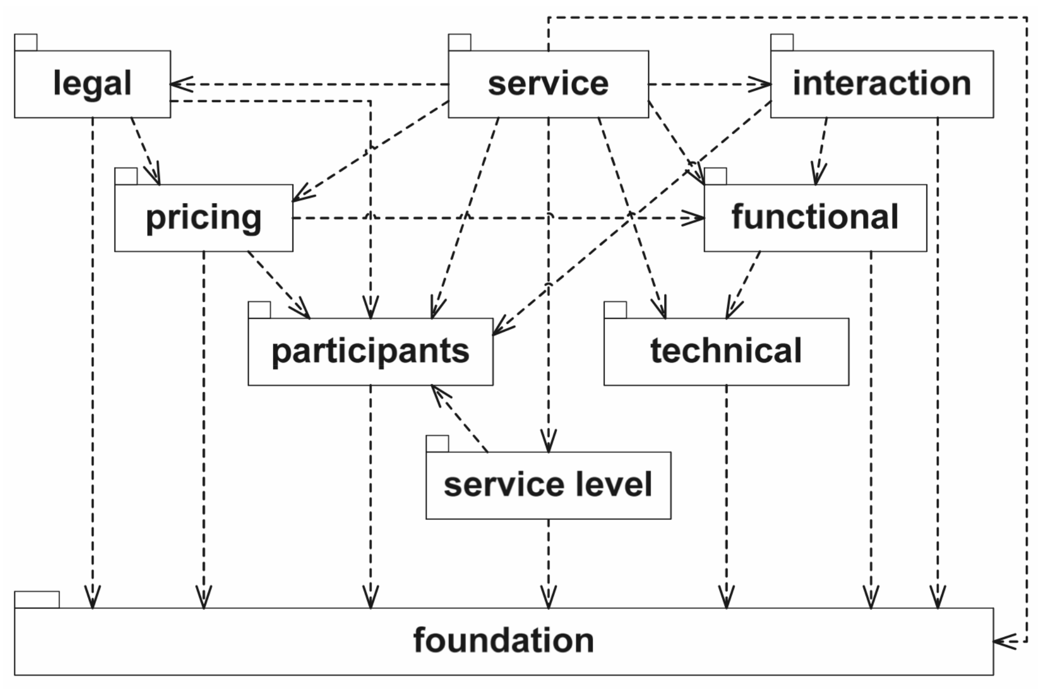


**Figure 2.14:** UML Package Diagram of USDL [USD11].

The interaction module captures the necessary interaction sequences for service execution. It has dependencies on the functional and the participants module to capture the functionalities and the actors involved in the interaction sequences.

The pricing structure of a service (e.g. price plan, price component and price level) is captured in the pricing module which has dependencies on the participants and on the functional module to capture price structures depending on used functionalities and involved actors.

The legal module captures licensing and copy right information and the agreed QoS guarantees for the provisioning, delivery and consumption of a service are captured in the service level module.

## 2.2.9 Comparison

Extensibility, reusability and composability are essential concepts in the development of applications in the cloud. Table 2.1 presents a summary of the extension points, the features supporting reuse of application topologies, and the possibilities to compose new applications with existing topologies given by the cloud related topologies and frameworks discussed in this section.

| | **Extension Points** | **Reusability Support** | **Composability** |
|---|---|---|---|
| Blueprint | extensible template | source blueprint concept and marketplace repository | resolve resource requirements |
| TOSCA | all elements are extensible | portable and composable core concept | substitute node templates with service templates |
| Cafe and MOCCA | variability model automatic installation clustering | variability model | contains-relationship on containers |
| AWS CloudFormation | | reuse, parameterization and customization of templates | |
| Flexiant Cloud Orchestrator | integration with existing systems | Bento Boxes and Deployment Questions | |
| OpenStack | OpenStack Extensions | | |
| OpenNebula | plugins for integration with third party tools | appliance marketplace | reuse of virtual appliances for compositions |

**Table 2.1:** Extension Points, Reusability and Composability of cloud related Topology Languages and Frameworks

Blueprint and TOSCA have a clear focus on the development of application topologies. The extensibility, reusability and composability of application topologies is part of the core concept in both approaches.

Cafe aims at the automatic provisioning of composite service-oriented applications. Deployment parameterization is enabled through a variability model, supporting the reusability of application topologies. Existing application models can be used as components in other application models, providing composability of application topologies. MOCCA provides an extension of the Cafe framework. It formalizes the necessary steps to move an application to the cloud and provides the means to split applications into provision clusters which can be deployed on different clouds automatically.

The presented Cloud Management Tools have their primary focus on the deployment and management of cloud applications. The extension points focus on the connection to or integration with existing systems. Underlying topologies are represented by the different network, storage and computing resources used. The reuse of application topologies aims at customized deployment options and ready to use appliances.

CloudML introduces an abstraction layer to prevent vendor lock-in in the provisioning of cloud resources. Pim4Cloud DSL provides a matching model for application deployment. The Descartes Meta-Model focuses on self-aware system adaptation at runtime. The depicted application topologies are enhanced by information on system performance influences and dependencies and possible adaptations are captured in adaptation points and corresponding processes. UML provides a whole collection of topology models for different purposes and different phases in application development. USDL aims at a global service description model, capturing all aspects of a service.

## 2.3 Annotation Approaches for Application Topologies

Additional information on topologies can be useful for several purposes, like interoperability, management, runtime system adaption, discoverability, metering and billing. Some of the topology languages and frameworks discussed in Section 2.2 include different kinds of additional information as part of the core language or enable the developer to include references to externally defined annotations on the topology.

The remainder of this section is structured as follows: A classification of topology annotations is given in Section 2.3.1. Section 2.3.2 gathers the annotations included in the discussed topology languages and frameworks. Exemplary annotations concerning discovery are given in Section 2.3.3, annotations used for provision and management are presented in Section 2.3.4 and Section 2.3.5 discusses annotations as design support.

### 2.3.1 Classification of Annotations

According to their intended usage, topology annotations can be grouped into the following, sometimes overlapping categories:

- **Discovery:** Annotations in this category describe the offerings or requirements of topologies and topology elements. They are used to determine compatible offerings for a set of defined requirements. The possible descriptions of offerings and requirements range from functional interface descriptions over desired QoS characteristics to semantic annotations.
- **Provision and Management:** Annotations in this category cover the necessary steps to be taken for the provision of resources, the deployment of an application or for the execution of management tasks during the lifecycle of an application. They can be used to automate the provisioning and management of applications. Management tasks to be automated range from simply installing or removing components of a topology to complex system adaptations at runtime.

- **Design Support:** Annotations in this category are designed to provide some kind of decision support during the design of a new application or an adaptation of an existing topology (like the migration of an application to the cloud). They can be used to provide a prediction of characteristics of the designed application (e.g. performance or cost calculations), to identify possible flaws in the design (e.g. performance bottlenecks) or to compare the characteristics imposed by different design options. Annotations can also be used to identify applicable patterns for the application design and to capture the decisions made during the design process.

Annotations can be *static*, meaning that they are interpreted "as is" during processing, or *dynamic* annotations which provide the basis for a calculation of the actual annotation value and may require additional input from a user.

Based on the level of automation in the processing of the information captured in annotations, the application topology annotations can be grouped in three classes:

- **automatic processing:** annotations which are processed entirely by machines (e.g. annotations for automated discovery)
- **human processing:** annotations designed to be processed by humans (e.g. annotations in natural language)
- **hybrid processing:** annotations which are processed by machines but require additional input from the user (e.g. customization options for automated provisioning)

Annotations used for discovery are usually static annotations which are processed automatically, while provision and management annotations combine static and dynamic annotations and use hybrid or automatic processing methods. Design supporting annotations can be static or dynamic and may use human or hybrid processing.

### 2.3.2 Annotations in Presented Topology Languages and Frameworks

#### Blueprint

In the Blueprinting Approach, each blueprint includes a policy section to describe constraints that all elements of the blueprint have to comply with. This policy is attached to the blueprint via a global policy constraint attribute in the blueprint properties section, which references an external policy definition file.

Blueprints also include QoS information in offerings and resource requirements, which are used in the blueprint resolution phase to discover matching offerings for resource requirements. The QoS characteristics of offerings and resource requirements are expressed via policies, attached to them via policy attributes in the offering and resource requirement sections of a blueprint.

### TOSCA

The TOSCA Service Template includes a container for plans. Plans are used to describe management aspects of service instances in the topology. The management issues are described in process models, which are workflows of one or more steps. TOSCA does not define a language for the description of process models. Existing languages like BPEL or BPMN can be used instead.

TOSCA offers the concept of Policy Types and Policy Templates to include non-functional and QoS characteristics in the topology description. A Policy Type specifies a certain kind of non-functional or QoS characteristic and the Node Types it can be applied to. The Policy Templates refer to a Node Type and specify the actual values for the invariant properties defined in the referred Node Type. Policy Templates can then be used to define a Policy within a Node Template.

### Cafe

The basic topology model in Cafe is enhanced with a variability points model which is used for deployment parameterization. A variability point can be associated with a concrete artifact via a locator. (cf. [Mie10, p. 92])

In MOCCA, the initial application topology of Cafe is enriched by additional information on the possible distribution of components to different clouds. This information is captured via labels on components and component relations. The result is a segmentation of the topology into a cloud distribution which is used for the provisioning of the rearranged application to the cloud.

### DMM

The Descartes Meta-Model is designed to enable the prediction of system performance changes at runtime. This results in a model which is enriched to capture the influences of individual layers on the systems performance, the dependencies among these influences and the resource allocations at each layer. Performance information is captured as part of the core model in DMM.

Additional annotations capture the valid states of the system architecture and system adaption processes used to adapt to system performance changes at runtime. This is reflected in the adaptation point and adaptation process model on top of the system architecture QoS model.

#### USDL

USDL provides a service description language which combines information from the business, operational and technical level of a service. The different modules of USDL can be divided into a set of modules representing the service topology and a set of modules to capture additional information.

The union of the information captured in service module, technical module and functional module roughly resembles the information captured in the other topology languages and frameworks. Annotations on the service topology are captured in the interaction, service level, participants, pricing and legal module. The different modules reflect the different kinds of information captured about the service.

### 2.3.3 Discovery Annotations

#### WS-Policies

The Web Services Policy Framework defined in [WSP07] provides a framework and a model for the declaration of policies referring to domain-specific capabilities, requirements and general characteristics of entities in a web service based system. Policies are used to negotiate the configuration options for service interactions.

A policy consists of zero or more policy alternatives. The alternatives indicate choices in the requirements or capabilities reflected through the policy. Each alternative represents a valid combination of constraints and requirements, governing the interaction with a service or the access to a resource. A policy with zero alternatives contains no choices. (cf. [WSP07], [WCL05, p. 131])

Each policy alternative is a collection of zero or more policy assertions. A policy assertion reflects a behavior specifying a constraint or requirement for the interaction with a service or the access to a resource. An alternative containing one or more assertions denotes exclusively those behaviors implied by the assertions. Alternatives without assertions indicate no behaviors. (cf. [WSP07], [WCL05, p. 131])

Each policy assertion reflects a requirement, a capability, or other property of a behavior of a policy subject. Policy subjects are entities (e.g. endpoints, messages, resources, operations) to which a policy can be associated - the policy itself does not specify its subject. Policies can be attached to specific subjects via a generic policy annotation which can be used in arbitrary XML documents or via an external attachment mechanism. (cf. [WSP07], [WCL05, p. 131,139])

Assertions in a policy alternative can be grouped into valid combinations with the ExactlyOne operator and the All operator. The ExactlyOne operator specifies a choice of assertions from which exactly one can be a part of the alternative at any one time. All assertions which are combined with the All operator are part of the behavior of the policy alternative. To indicate that the inclusion of an assertion is optional, an attribute optional=“true” can be set. This is essentially a shortcut for two policy alternatives, one that includes the assertion and one that does not. If the optional attribute is not specified, the interpretation defaults to optional=“false”. (cf. [WCL05, p. 131ff.])

The described operators can be nested inside each other. A policy is in normal form, if each policy alternative consist of an All element (containing the assertions) and all alternatives are contained in a single exactly one operator. In other words, only one alternative can be chosen and the behavior of every assertion contained in the chosen alternative must be applied. (cf. [WCL05, p. 133])

The compatibility between two policies is usually established by the policy intersection mechanism followed up by discipline-specific engines. Policy intersection inspects the structure of the two policies and results in a set of policy alternatives which appear in both policies. It only matches the element names of the XML representation of the assertions which helps to discard clear nonmatches. To ensure technical compatibility, the policy intersection has to be validated with discipline-specific engines. (cf. [WCL05, p. 135f.])

#### Semantic Annotations

Semantic annotations in web services can enhance the discoverability of services by adding semantic information to the service descriptions and include this information in the service retrieval process.

An example for semantic annotations in web services are the Semantic Annotations for WSDL and XML Schema (SAWSDL) proposed by the W3C in [FL07]. SAWSDL specifies how to extend WSDL and XML Schema definitions to capture additional semantics of WSDL components or XML elements.

Semantic information is included through references to semantic models like ontologies. Instead of a language for the representation of semantic models, SAWSDL provides the means to reference semantic models from within WSDL and XML Schema components.

SAWSDL provides three extension attributes named modelReference, liftingSchemaMapping and loweringSchemaMapping. The modelReference attribute is used in XML Schema type, element and attribute declarations as well as WSDL interfaces, operations and faults. It indicates the association of a WSDL or XML Schema component with a concept in a semantic model. The liftingSchemaMapping and loweringSchemaMapping is used in XML Schema element declarations and type definitions. A liftingSchemaMapping defines a mapping that transforms data from XML to a semantic model, while a loweringSchemaMapping translates data from a semantic model into XML.

The model references and schema mappings may contain multiple references to different semantic models. In schema mappings, the different models are treated as alternatives while multiple model references are all applied.

The Semantic Annotations for WS-Policy (SA WS-Policy) proposed in [Spe10] are inspired by SAWSDL and provide means to annotate policy assertions and match requirements and capabilities not only on a syntactical but also on a semantical level.

The semantic model is captured in the Web Ontology Language (OWL) [HKP$^+$09]. An ontology in OWL can describe individuals, classes referring to groups of individuals and relations between individuals or between classes. Each individual, class and relation in OWL is labeled with an URI.

In SA WS-Policy, the assertions in policies are annotated with a modelReference attribute, a liftingSchema attribute or both. A modelReference attribute in SA WS-Policy contains an URI referencing an OWL class and a liftingSchema attribute contains a link to an XSL transformation which is used to generate an OWL class definition from the assertions XML node and its children.

The matching of offering and requesting policies is achieved with ontology reasoning. Policy assertions are represented by classes and an offering matches a request, if the offering is a subclass of the request.

### 2.3.4 Management and Provisioning Annotations

#### Web Service Level Agreement

The Web Service Level Agreement (WSLA) defined in [LKD$^+$02] specifies a language for the formal description of service level agreements. It is designed to enable automatic deployment, monitoring and enforcement of SLAs.

Figure 2.15 depicts the role of a WSLA in a service interaction. The WSLA specifies the agreed performance characteristics and provides information about the metrics to be measured, and the performance guaranties that have to be fulfilled. Metrics may be measured from various sources, enabling for example to capture server-side metrics from the provider and client-side metrics from the customer.

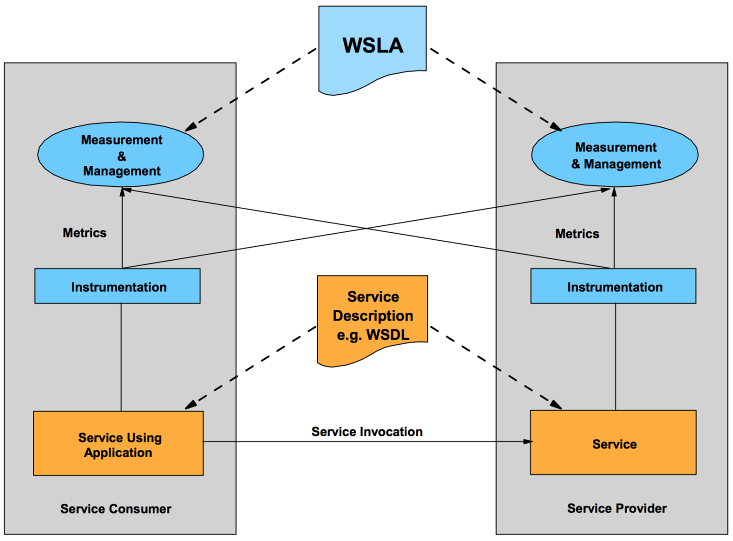


**Figure 2.15:** Role of a Web Service Level Agreement in Service Interactions [LKD+02].

The management of a WSLA during runtime is achieved by retrieving the measured metrics from the systems instrumentation, evaluating the conditions to meet the guarantied performance and trigger a management action in case of a guarantee violation. The measurement and condition evaluation functionalities may be outsourced to supporting parties which offer according services. Those supporting parties do not have access to the whole WSLA but receive only relevant information from the party which outsources a functionality to them.

The WSLA language consists of a parties section, a service definition section and an obligations section as depicted in Figure 2.16. Parties can be either signatory parties, who are assumed to “sign” the SLA, or supporting parties which offer measurement and condition evaluation services and are sponsored by one or both signatory parties.

Each service definition contains one or more service objects which are abstractions of services that are relevant for the definition of SLA parameters. An SLA parameter is defined by a metric and a metric specifies either a measurement directive or a function which computes the metric. A measurement directive specifies how a value is measured from a source. Metrics retrieved by a measurement directive are called resource metrics and computed metrics are referred to as composite metrics.

Obligations can be either a service level objective or an action guarantee and every obligation has an obliged party. A service level objective guarantees a particular state of SLA parameters in a certain time period and is usually the obligation of the service provider. An action guarantee is the assurance to execute a defined action in a specific situation. This can be obliged to any party, including the supporting parties.

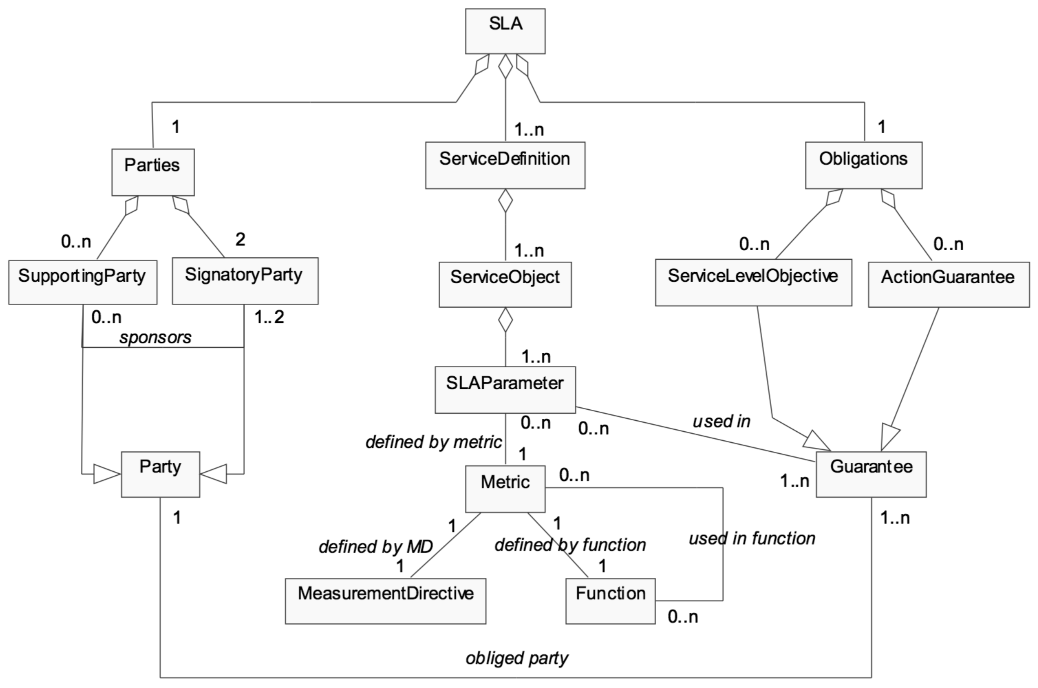


**Figure 2.16:** Overview of main WSLA Concepts [LKD+02].

## Policy-Aware Provisioning

The management framework presented in [BBKL13] uses management planlets and management annotations to generate management plans for an application. Management planlets provide small management tasks like installing or removing components. A management plan defines an orchestration of management planlets to achieve a desired state of the topology. Management plans can also be used for provisioning of an application. The desired state of the topology is described with management annotations. They are attached to topology elements and specify low-level management tasks which have to be performed on the corresponding elements.

In [BBK+13] the framework is extended to support policy-aware provisioning of applications. The general concept is depicted in Figure 2.17: Topology elements are annotated with policies describing non-functional requirements of the respecting element. Elements in planlets are annotated with policies describing the non-functional capabilities that may be provided by the planlet for the element in the executed task.

During the generation of a provisioning plan, the policies attached to the topology are checked for compatibility with the available planlets. A planlet is applicable, if all policies attached to topology elements which are contained in the planlet are fulfilled.

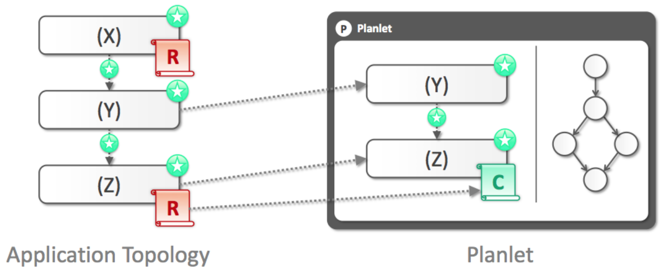


**Figure 2.17:** Concept for policy-aware Provisioning of Cloud Applications [BBK+13].

### 2.3.5 Design Support Annotations

#### Architectural Patterns

In [FLR+11] an architectural pattern language of cloud-based applications is introduced and a catalog of identified patterns is provided. A method for pattern-driven application development based on the introduced language is defined and a framework for the design process is proposed.

The architectural patterns are described in a uniform format which captures the context in which the pattern applies, the challenges which are handled, the details of the proposed solution with corresponding instructions for the realization, possible variations in the pattern, known uses of the pattern and relations to other patterns.

Based on the pattern catalog, the interrelations between the different patterns are determined and a decision table is constructed which matches each combination of two patterns to one of the following three types of relations:

- **Strong Cohesion:** the two patterns are likely to be combined
- **Exclusion:** the two patterns cannot be combined
- **Undetermined:** the two patterns neither form a strong cohesion relation nor an exclusion relation

The resulting decision recommendation table is used to identify applicable patterns in the design process, which incorporates the following steps:

1. The developer selects patterns describing the chosen environment for application deployment
2. Based on the selected patterns, a set of possible patterns are recommended for the implementation
3. The developer selects the patterns to be used
4. Based on the selection, possible conflicts are detected and displayed

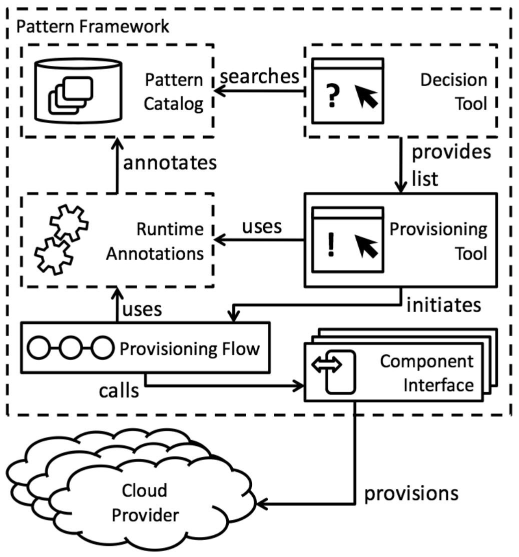


**Figure 2.18:** Cloud Pattern Framework [FLR+11].

5. The developer resolves the conflicts by deciding which patterns are more important

Each set of patterns which are chosen by the developer results in three different sets of patterns which can be derived from the pattern interrelations in the decision recommendation table. Patterns with a strong cohesion relationship to the set of chosen patterns are likely to be applicable as well, while patterns with an exclusion relation to the set of chosen patterns cannot be used. The last set contains patterns for which the applicability is undetermined. To obtain the effective set of patterns for a defined use case, the developer iteratively refines the list of applicable patterns.

Figure 2.18 depicts the envisioned cloud pattern framework. The pattern format and annotations to patterns are supposed to form the basis for a pattern catalog component and a runtime annotations component. A decision tool component, based on the decision recommendation table shall be used by the application developer to identify the required patterns.

For the provisioning of the application, the reuse of the existing provisioning tool Cafe [MUL09] is suggested to fulfill the functionality of the provisioning tool, provisioning flow and component interface part of the framework. The provisioning tool is used for the customization and provisioning of pattern implementations and their runtime infrastructures. Annotations to the patterns are used to determine the customization options for a pattern. The provisioning flow accesses a set of component interfaces and provisions components of the customized pattern implementations in the appropriate order.

A pattern based approach for the migration of application data to the cloud is presented in [SABL13]. Distinct cloud data migration scenarios are mapped to functional, non-functional and confidentiality cloud data patterns. Functional patterns address possible challenges related to the offered functionalities, non-functional patterns help to ensure an acceptable QoS level by means of scalability and confidentiality patterns focus on preventing disclosure of confidential data.

To migrate application data to the cloud, in the first step the relevant cloud data migration scenario has to be identified. Then the desired cloud data store has to be described and a suitable cloud data store has to be chosen based on the description. To solve incompatibilities, the applicable patterns for the refactoring of the application have to be identified and implemented by adapting the database layer and upper architectural layers. In a last step the actual data has to be migrated to the selected cloud data store.

### CAP Properties

The CAP theorem describes the relationship of three fundamental requirements in distributed systems:

- **Consistency:** if all system parts see the same data at the same time
- **Availability:** the percentage of time a system is accessible and functions properly
- **Partition-tolerance:** whether the system can tolerate network failures

Brewer [Bre00] observed, that a distributed system can only satisfy two out of those three requirements at the same time. The hypothesis was later formally proven in [GL02].

A CAP oriented methodology for the design of cloud-native applications is proposed in [ASFL13]. The CAP properties can be added as an extension to the cloud pattern framework introduced in [FLR$^+$11] to enable estimation of CAP properties based on captured design decisions.

The degree to which a distributed System fulfills each of the CAP properties can be visualized by positioning the properties at the edges of a tetrahedron as depicted in Figure 2.19 with minimum values in the intersection of the axes. Representation of the CAP properties of a specific system is done by a triangular area which cuts through the tetrahedron.

A system which satisfies availability and consistency, but is not tolerant to network failures would be positioned at the left side of the tetrahedron. With a strict interpretation of the CAP theorem, any system would be positioned at one side of the tetrahedron. In practice, system designers and developers may trade some degree of one

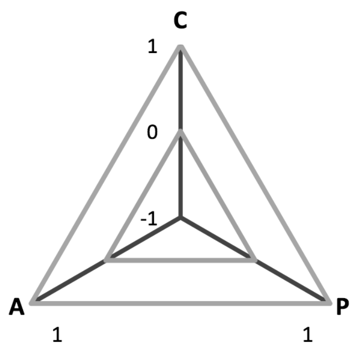


**Figure 2.19:** The CAP Properties of a Distributed System [ASFL13].

of the CAP properties to achieve a higher degree of the other properties. This results in systems with different emphasis on each of the CAP properties according to the system requirements.

The CAP-oriented design methodology takes the possible influences of architectural decisions on the CAP properties into account and consists of the following phases:

- **Identify CAP Requirements:** The application developer identifies the desired CAP properties of the application.
- **Capture Design Decisions:** The design decisions made by the application designer are recorded.
- **Select *aaS Solutions:** The abstract design decisions are translated into Software-, Platform- or Infrastructure-as-a-Service solutions.
- **Estimate CAP Properties:** An Estimate of the overall CAP properties of the application is computed based on the CAP properties of the solutions chosen in the previous phase. To enable the estimation, the different *aaS solutions have to be annotated with their CAP property values. As a representation for those annotations, a triplet (c, a, p) is suggested with c, a, p $\in [-1, 1]$. Values close to 1 signify a strong correlation to the property, while values close to -1 indicate that the solution has a degrading effect on the property. The annotated values can be aggregated to estimate the overall CAP properties.
- **Update Design and Solutions:** If the estimated CAP properties satisfy the requirements, the designer can proceed with the development, deployment and provisioning of the application. Otherwise, the design decisions may be reconsidered and changed, leading to different solutions with different CAP properties and a new estimation of the overall system CAP properties.

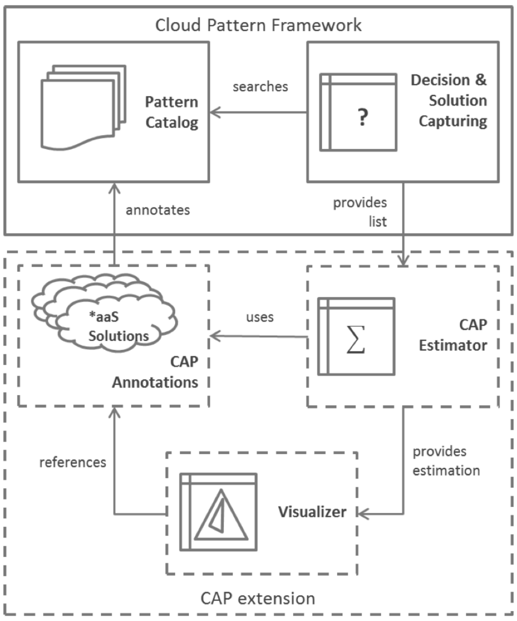


**Figure 2.20:** CAP Extension of the Cloud Pattern Framework [ASFL13].

Figure 2.20 depicts the proposed integration with the cloud pattern framework to enable the annotation with CAP properties, the estimation of the systems overall properties and the visualization in the form of a tetrahedron.

### 2.3.6 Summary

Topology annotations serve various purposes. The topology languages and frameworks discussed in Section 2.2 include annotations concerning the discovery, management and distribution of services as well as performance prediction and runtime system adaptation.

Discovery Annotations provide information which is useful to discover the right service and to establish the appropriate way of interaction between service and consumer. WS-Policies help to negotiate the configuration options for service interaction and can be used to discover a service which supports the configurations required by the consumer. Semantic annotations aim at a higher recall in service discovery by including semantic information in the retrieval process.

Management and Provision Annotations are used for the automation of application provision and managent. The purpose of WSLA lies in the automatic deployment, monitoring and enforcement of SLAs for service interactions. Policy-Aware Provisioning includes non-functional requirements in the provisioning process.

Design Support Annotations provide decision support during application design. Architectural Patterns capture applicable architectural decisions and can serve as reference during application design. Predictions of system characteristics provided during the design phase, can indicate possible flaws in the application design (e.g. performance bottlenecks). Design decisions captured in annotations to the topology can serve as documentation and ease adaption, extension and reuse of application topologies.

# 3 Generalization of Application Topology Languages

The presented topology languages and frameworks rely on a set of common fundamental concepts, which have different representations in each language. A generalized Application Topology Language has to provide a metamodel for topology languages, which can represent the information captured in different languages.

The remainder of this chapter is structured as follows: The common fundamental concepts and their representation in the discussed languages and frameworks are depicted in detail in Section 3.1. Section 3.2 lists the requirements derived from the presented common concepts and defines a generalized application topology language with appropriate mappings from selected existing topology languages and frameworks.

## 3.1 Common Fundamental Concepts in Application Topologies

The application topology languages and frameworks presented in Section ref-sec:topolLanguages are based on similar concepts. They all represent a topology graph, which consists of components (nodes) and connectors (edges). The components as well as the connectors have certain attributes, and most languages also offer a way to assemble components into groups and to form subgraphs.

The remainder of this section describes the common fundamental concepts in detail and gives the corresponding constructs for each fundamental concept in the discussed cloud related topology languages.

### 3.1.1 Components

Components are the nodes in the topology graph. The different application topology languages and frameworks provide topology descriptions in different levels of detail. This results in different types of components.

The blueprinting approach presented in Section 2.2.1 supports components of variable granularity reaching from implementation artifacts, that can be a single script to offerings and resource requirements which can be whole services.

In TOSCA (Section 2.2.2) components are node templates of specified node types. Since there are no semantic restrictions on the granularity of a node type or template, components in TOSCA can be anything from a simple script to a complex service.

Besides the component depicted in the Cafe metamodel in Section refsec:cafe, the implementations as well as the files in the application metamodel can be seen as components of the application topology.

The topology frameworks in cloud management tools presented in Section refsec:ManagementTool are focused on the management of cloud resources. This results in a topology graph with cloud resources (like servers, virtual machines, images, server instances etc.) as components.

### 3.1.2 Connectors

Connectors are the edges in the topology graph and they represent the relationship between the components in the graph. A connector is a directed link between two components. The presented languages and frameworks provide different types of connectors.

Blueprint makes a distinction between vertical, horizontal and resource links to represent different types of dependency. A vertical link from a to b specifies that the blueprint element a has to be deployed on the blueprint element b. A horizontal link from a to b indicates a functional dependency between blueprint element a and b, i.e. a reuses the functionality of b. A resource link is a connection to a virtual network resource captured in either an IaaS service offering or an IaaS resource requirement.

TOSCAs connectors are relationship templates of a specified relationship type. This enables developers to specify any type of relationship desired, resulting in a wide range of possible connections.

Connectors in Cafe are expressed through the contains and deployedOn relationships between components, the implementedBy relationship between components and their implementations and the realized by files relationship between implementations and files.

In the presented topology frameworks in cloud management tools, the explicitly shown connection is which component a is deployed on which component b. Some relationships between components can be represented indirectly through connections to the managed networks. Components with a connection to the same network may or may not communicate with each other. Functional dependencies are not reflected in the presented cloud management tools.

### 3.1.3 Attributes

The components and connectors of an application topology, as well as the topology as a whole may have several attributes providing descriptive information on the respective constructs. In contrast to topology annotations, which contain additional, external information, attributes are part of the topology and provide information which is essential for the completeness of the topology. The discussed languages and frameworks offer different representations of such attributes.

A Blueprint has basic properties giving details about the Blueprint as a whole (e.g. ID, ownership, release date etc.). Each offering in a Blueprint has attributes describing the offered functionalities and each resource requirement has attributes describing a required cloud resource. The implementation artefacts also have attributes specifying for example the artefact type or location.

The properties and interfaces of node and relationship types and templates in TOSCA are essentially attributes of components and connectors in the topology. Capability and requirement definitions of node types and templates are also attributes of the respective components.

The attributes in Cafe include the component and implementation types. Other attributes for components, implementations and files may also be present without a concrete specification in the application metamodel.

The attributes represented by the discussed topology frameworks in cloud management tools are the configuration and setting options for the managed resources and networks (e.g. size, bandwidth etc.).

### 3.1.4 Groups of Components

Some components in an application topology may form a group, because they are deployed on the same host or the same cloud, or because they form a service which offers functionalities for other parts of the application. These groups will be referred to as sub-topologies from now on. In other respects, components which share common features or behaviors may be considered part of the same group representing a certain kind of component. Those will be called component groups later on.

The Blueprinting Approach relies on the concept of resource requirements expressing the need for a resource. They can be resolved by a blueprint containing an offering that fulfills the requirements. Each Blueprint represents a topology with possible other Blueprints nested in it as sub-topologies. There are two distinct component groups in a blueprint, since the components are either resource requirements or implementation artefacts.

Every node template in TOSCA is of a defined node type, so each node type forms a component group. Since TOSCA offers the possibility to replace a node template with a service template, the node templates can also be sub-topologies.

The application model in Cafe can be used as component in other application models, which converts it into a sub-topology. Component groups in Cafe are formed by Cafe components, implementations and files.

In the presented topology frameworks in cloud management tools, the components may form sub-topologies in the sense that they are installed on the same hosts or in the same clusters or virtual data centers. Component groups may be defined by the types of the manageable cloud resources.

## 3.2 GENTL - a GEneralized Topology Language

This section first lists the requirements for an application topology metamodel in Section 3.2.1. In Section 3.2.2 the GEneralized Topology Lanuagage GENTL is introduced and a formal description of the language is given. The chapter concludes with mappings between GENTL and selected topology languages and frameworks discussed in Section 3.2.3.

### 3.2.1 Requirements for a Generalized Application Topology Language

To create a metamodel capturing the information represented in topology languages and frameworks, the described common concepts have to be modeled in a way which is generic and extensible. As stated in Section 2.2.9, extensibility, reusability and composability are key concepts in application topologies. This leads to the following requirements for GENTL:

- To support reusability and composability of topologies, GENTL should provide a topology representation which enables the reuse of existing topologies as sub-topologies and provides a way to form compositions of topologies.
- The model should be extensible and enable mappings from different existing topology languages and frameworks to GENTL. This leads to a generic modeling of the topology elements. The required elements for a GENTL topology are:
  - A component representation that can capture arbitrary attributes.
  - Representations for connectors of different types with arbitrary attributes.
  - A component group representation that can capture arbitrary attributes.
- The metamodel should capture coarse as well as fine grained topologies and provide a quick overview on the whole topology as well as more detailed information on each topology element when required.

All topology elements require a way to capture arbitrary attributes in order to support the mapping from various topology languages and frameworks to GENTL. This indicates the need of a generic model for attributes which can be applied to the topology elements. The next section defines how the different topology elements and their attributes are represented in GENTL.

### 3.2.2 Realization of Topology Concepts

GENTL relies on a generic but typed attribute system to support arbitrary kinds of attributes, while at the same time providing some information about the type of a captured attribute. The language provides simple and composite attributes.

A simple attribute consists of a name and a value. Simple attributes can be TextAttributes, IntegerAttributes, DateAttributes, TimeAttributes, ReferenceAttributes or generic attributes. The value of a generic attribute can be of any type, while the values of the other attributes are restricted to string, integer, date, time and URI values.

A composite attribute has a name and contains a sequence of attributes which can be simple or composite attributes. This enables the nesting of attributes which can be used to express characteristics which are to complex to captured in simple attributes.

The Extended Backus-Naur Form (EBNF) in Listing 3.1 provides a formal description of GENTL. A GENTL topology has a name and a unique ID, and contains zero or more components, connector classes, connectors groups. Topology attributes can be used to capture general information which applies to the topology as a whole. A topology attribute can either be a simple or a composite attribute.

GENTL components have a unique ID, a name and an optional "representsTopology" property. The "representsTopology" property can be used to link to another GENTL topology, which depicts the inner structure of the component. This enables the reuse and composition of GENTL topologies. All other aspects and properties of the component are captured as component attributes, which can be simple and composite attributes.

A Connector has a unique ID, a name, a sourceComponentID and targetComponentID value and belongs to a connectorClass. A connectorClass represents a type of connector (e.g. deployed on) and consist of a name and a unique ID. The sourceComponentID and targetComponentID serve as reference to the components which are linked through the connector. Further information on the represented relationship can be captured in connector attributes which follow the same structure as the component attributes.

A Group has a unique ID and a name. The grouped topology elements are represented as listings of component IDs. Information on groups can be captured in group attributes in the same manner as described for topologies, components and connectors. A component may be part of multiple groups.

### 3.2.3 Mappings between GENTL and existing Topology Languages

Table 3.1 provides an overview on the correlation between GENTL and the topology elements in the cloud related topology languages and frameworks. In the remainder of

**Listing 3.1** Formal Description of GENTL using EBNF

```
    topology  =  name, uuid, topolAttr, topolParts;
  topolParts  =  {component}, {connClass}, {connector}, {group};
   component  =  name, uuid, [reprTopology], compAttr;
   connClass  =  name, uuid;
   connector  =  name, uuid, source, target, class, connAttr;
       group  =  name, uuid, grAttr, compRef;
   topolAttr  =  {attribute};
    compAttr  =  {attribute};
    connAttr  =  {attribute};
      grAttr  =  {attribute};
   attribute  =  simpleAttr | composAttr;
  composAttr  =  name, attribute, {attribute};
  simpleAttr  =  textAttr | intAttr | dateAttr | timeAttr | uriAttr | genAttr;
    textAttr  =  name, stringValue;
     intAttr  =  name, integerValue;
    dateAttr  =  name, dateValue;
    timeAttr  =  name, timeValue;
     uriAttr  =  name, uriValue;
     genAttr  =  name, genericValue;
      source  =  uuid;
      target  =  uuid;
       class  =  uuid;
     compRef  =  uuid;
reprTopology  =  uuid;
```

this section the details of the mappings from Blueprint, TOSCA and Cafe to GENTL are described to demonstrate the generic nature of GENTL.

### Mapping from Blueprint to GENTL

A blueprint offering resembles a topology on GENTL. Each GENTL topology which is created from a blueprint, contains the two groups “resource requirements” and “implementation artefacts”, and the two connectorClasses “horizontal” and “vertical”.

Resource requirements and implementation artefacts from the source blueprint are mapped to components, and the components are added to the respective groups in the GENTL topology. A resource requirement which is resolved by a matched offering results in a component referencing the topology which represents the matched offering.

| | **Blueprint** | **TOSCA** | **Cafe** |
|---|---|---|---|
| Topology | Offering | Topology Template | Application Model |
| Component | Resource Requirement<br>Implementation Artefact | Node Template | Component<br>Implementation<br>File |
| Connector | Vertical Link<br>Horizontal Link<br>Resource Link | Relationship Template | delpoyedOn<br>implementedBy<br>contains |
| Group | “resourceRequirements”<br>“deploymentArtefacts” | Node Type | “CafeComponent”<br>“Implementation”<br>“File” |
| Component Attribute | Resource Requirement Property<br>Implementation Artefact Property | Node Property<br>Node Interface<br>Capability Definition<br>Requirement Definition | |
| Connector Attribute | | Relationship Property<br>Relationship Interface | |
| Group Attribute | | Node Type Property | |
| Topology Attribute | Basic Property<br>Offering Property | | |
| Connector Class | Link Type | Relationship Type | “contains”<br>“deployedOn”<br>“implementedBy” |

**Table 3.1:** Mappings between GENTL and other Topology Languages

The vertical and horizontal links captured in a blueprint are represented by connectors of the “vertical” and “horizontal” classes respectively. The “resource links” discussed in [NLPH12] are not included in the XML Schema for Blueprints, hence they are not included in the mapping from Blueprint to GENTL.

The properties of resource requirements are added as component attributes to the corresponding components and the basic properties and offering properties are added as topology attributes. A GENTL topology derived from a blueprint neither contains group nor connector attributes.

### Mapping from TOSCA to GENTL

A TOSCA service template corresponds to a topology in GENTL. The TOSCA node types are represented as groups and the relationship types as connectorClasses.

Node templates are designed as components, which belong to the group that represents their node type. The properties, interfaces, capability definitions and requirement definitions of a node template are added as component attributes.

A relationship template in TOSCA resembles a connector in GENTL and the relationship properties and interfaces are represented by connector attributes.

### Mapping from Cafe to GENTL

A Cafe application model may be represented as a GENTL topology. The Components, Implementations and Files in the application model can be mapped to components in GENTL.

GENTL groups can be formed from the component and implementation types in the application model, or the GENTL components could be grouped according to their corresponding Cafe elements into “CafeComponent”, “Implementation” and “File” groups. A combination of both group concepts may also be applied.

The relationships in a Cafe metamodel can be represented by GENTL connectors belonging to “contains”, “deployedOn” or “implementedBy” connectorClasses.

# 4 GENTL Annotations

The discussed annotations are designed for different purposes, but they follow some similar basic concepts. In this section, the common structures and concepts in topology annotations are identified, respective requirements for GENTL annotations are established and a generic annotation model for GENTL topologies is given.

The chapter is structured as follows: The structures and concepts in topology annotations are identified in Section 4.1 and the resulting requirements for GENTL annotations are given in Section 4.2. The chapter concludes with the definition, formalization and exemplary instantiation of the GENTL annotation model in Section 4.3.

## 4.1 Structures and Concepts in Topology Annotations

The discussed annotations are designed for different purposes, but they follow some similar basic concepts. Annotations like WS-Policies, WSLA, SAWSDL and SA WS-Policy, as well as the different modules of USDL are designed to be passed to specific engines for processing. All of them offer an XML representation of the captured information.

There are three different ways to connect the annotations to the topologies or topology elements:

1. Include annotations as part of the core model with specific language constructs
2. Include references to externally defined annotations in topology elements
3. Connect the annotations to the topology elements via external references to the topology elements.

The first mechanism is used by USDL while WSLA, SAWSDL and SA WS-Policy use the second mechanism and WS-Policy can be attached by the second as well as the third mechanism.

Three different levels for annotations can be identified according to which parts of the topology an annotation applies:

- The most general annotations are applied on topology level, like for example the policy section in a blueprint which applies to all parts of the blueprint.
- Annotations on group level apply to a group of topology elements, like for example the cloud distributions in MOCCA which form a segmentation of the topology.
- The finest granularity for annotations are annotations on element level which only apply to a single topology element like the discussed CAP properties for a single component or component specific performance characteristics captured in DMM. A common use of annotations on element level is to calculate some kind of aggregation on group or topology level (e.g. the CAP properties of the overall system).

As discussed in Section 2.3.1, based on the intended usage, the annotations can be grouped into annotations for discovery of applications, management and provisioning of applications and for support during the design of applications. Based on the processing of annotations, the three different modes “human processing”, “automatic processing” and “hybrid processing” can be distinguished and annotations can be classified as static or dynamic.

## 4.2 Requirements for GENTL Annotations

This section first identifies the annotations to be supported by GENTL and the ways in which they should be supported. Then the possible attachment mechanisms for annotations are discussed and the appropriate method to be used for GENTL annotations is identified. The section concludes with a listing of the overall assembled requirements for GENTL annotations.

### 4.2.1 Annotations to be supported

The purpose of GENTL annotations lies in providing an annotation framework to support the development of cloud applications. This includes the development of cloud native applications as well as migrating applications to the cloud. As a consequence, the focus of GENTL annotations lies on Design Support Annotations which facilitate design decisions. However, other types of annotations should also be supported.

Provision and Management Annotations can be used to leverage the added provision and management flexibility in cloud environments and Discovery Annotations are useful to find reusable components and encourage others to use the designed application.

GENTL annotations should provide ways to capture Provision and Management Annotations as well as Discovery Annotations. The processing of those annotations, namely implementations for the automated provision, management and the discovery of applications based on those annotations is not in the scope of GENTL annotations.

Static as well as dynamic annotations have to be provided to support the different kinds of annotations. Functionalities to enable human and hybrid processing of Design Support Annotations should also be included in GENTL annotations.

### 4.2.2 Attachment Mechanisms for Annotations

Annotations may reside on topology, on group or on element level. This requires attachment mechanisms which enable the user to specify annotations on GENTL topologies as well as on groups or elements inside a GENTL topology.

WS-Policy was created as a separate specification from WSDL to clearly separate concerns between the functional descriptions in WSDL and non-functional descriptions and QoS aspects handled by WS-Policy. Another advantage of the separation is that policy subjects are not limited to service endpoints, enabling policies to be attached to a huge variety of subjects. Due to the flexibility of the attachment mechanism, it is possible to incrementally add capabilities (e.g. a new authentication mechanisms) to an existing service description (cf. [WCL05]).

The definition of annotations separate from the topologies offers similar advantages as stated for WS-Policies. Separation from the topology enables annotations to be added, updated or removed while the topology remains unchanged. Different types of annotations might be added without interfering with each other or cluttering the core definition of the topology. An annotation might be defined once and attached to multiple topology elements, even spanning multiple topologies.

In consequence, GENTL annotations shall be defined in a separate specification. Appropriate attachment mechanisms to connect annotations with topologies and topology elements must be established.

### 4.2.3 Assembled Requirements

Based on the discussion about the annotations to be supported and the attachment mechanisms suitable for GENTL, the following overall requirements for GENTL annotations can be assembled:

- Primary focus on Design Support Annotations
- Capturing Provision and Management Annotations as well as Discovery Annotations without processing them

- Capture static annotations as well as dynamic annotations
- Enable human and hybrid processing of Design Support Annotations
- Capture annotations on topology, group and element level
- Provide a specification of GENTL annotations separate from GENTL specification
- Provide appropriate attachment mechanisms to connect annotations with topologies and topology elements

The next section specifies the structure of the GENTL annotation model and describes how the established requirements are fulfilled by the GENTL annotation model.

## 4.3 Annotation Model

This section provides the GENTL annotation model. In Section refsec:annotationModelStruct the basic structure of the annotation model is described and a formalization of GENTL annotations is provided. Exemplary instantiations of the annotation model for different kinds of annotations are provided in Section 4.3.2.

### 4.3.1 Annotation Structure

GENTL annotations are based on a generic model which can be further restricted to define the structure of a concrete annotation type. The basic annotation model specifies three different annotation types:

1. **service invocation:** a dynamic annotation containing a request endpoint and request parameters to invoke a service call
2. **simple annotation:** a static annotation containing a sequence of simple and composite GENTL attributes
3. **external reference:** a static annotation which contains a reference to an external resource containing the actual annotation

The third kind of annotation could be captured as well by an instantiation of the second annotation type containing a uriAttribute with a reference to the respective resource. The motive for an additional type in the core model is the frequent occurrence of static annotations which are best suited to be included via reference (e.g. WS-Policy, SAWSDL, WSLA).

The EBNF in Listing 4.1 gives a formal description of a GENTL annotation. A GENTL annotations element contains at least one annotation. An annotation has a name and

**Listing 4.1** Formal Description of GENTL Annotations using EBNF

```
 annotations  =  annotation
  annotation  =  name, uuid, parent, {parent}, annAttr, annType;
     annType  =  extRef | serviceInv | simpleAnn;
      extRef  =  uri;
  serviceInv  =  reqUrl, staticParam, {dynParam};
   simpleAnn  =  {attribute};
     annAttr  =  {attribute};
 staticParam  =  {attribute};
   attribute  =  simpleAttr |  composAttr;
  composAttr  =  name, attribute, {attribute};
  simpleAttr  =  textAttr | intAttr | dateAttr | timeAttr | uriAttr | genAttr;
    textAttr  =  name, stringValue;
     intAttr  =  name, integerValue;
    dateAttr  =  name, dateValue;
    timeAttr  =  name, timeValue;
     uriAttr  =  name, uriValue;
     genAttr  =  name, genericValue;
    dynParam  =  name, datatype;
    datatype  =  'Date' | 'Integer' | 'String' | 'Time' | 'URI';
      parent  =  uuid;
```

a unique id and at least one parent element the annotation is attached to. A parent element can either be a topology, a component, a connector or a group.

Each annotation can either contain a simple annotation (containing a sequence of attributes), a reference to an external resource or a dynamic annotation which invokes a service call. Information about the annotation itself may be captured in optional annotation attributes.

The attributes in a simple annotation and the optional annotation attributes are of the same type as GENTL attributes, meaning they may be simple or composite attributes as discussed in Section 3.2.2.

An external reference annotation only captures the URI of a resource. In a service invocation annotation, the URL for the request is captured along with the necessary parameters for service invocation. The parameters are distinguished between static and dynamic parameters.

A static parameter captures a parameter which remains the same for any invocation of the service. Like the annotation attributes and the attributes in a simple annotation, they are captured as GENTL attributes.

Dynamic parameters represent information which has to be provided by the user at runtime. They consist of a name and a datatype. The name specifies the kind of parameter to be provided and the datatype can be used for type checking of the retrieved user input before service invocation.

### 4.3.2 Instantiation for Different Annotation Types

An instantiation of a GENTL annotation is a restriction on the available annotation types. An external reference could be instantiated by creating an annotation with annotationAttributes specifying the type of the referenced document or instructions on how to process the document.

To instantiate a documentation annotation, a simple annotation could be designed which contains a textAttribute named author, a dateAttribute and a timeAttribute named creationDate and creationTime and a textAttribute named documentationAnnotation. The annotationAttributes on the simple annotation could contain guidelines on how to structure the content of the documentationAnnotation field.

An instantiation of a service invocation annotation contains the request endpoint of the desired service. It incorporates all predefined request parameters as static parameters and the information which has to be retrieved from the user is represented by providing the name and datatype of the request parameter. Predefined request parameters could be for example the desired response format (e.g. JSON or XML) and dynamic parameters could be the configuration parameters for the service call.

# 5 Implementation

This section discusses the implementation of the GENTL environment. The requirements for the framework implementation are established in Section 5.1 and the possible implementation options are discussed in Section 5.2. A structural description of the implemented GENTL environment is given in Section 5.3 and the provided user interface is described in Section 5.4.

## 5.1 Requirements

This section lists the fundamental requirements for the implementation. The application topology annotation framework should be available to a wide range of users and it should provide the functionality to import application topologies from different existing topology languages and frameworks, display the imported topology graphs and add annotations of different types to nodes, edges or groups of nodes in the topology graph.

In consequence, the chosen form of implementation should provide ways to offer the following features:

- **Platform Independence:** Since the application topology framework should be available to a wide range of users, the implementation should be platform independent to enable easy access and usage.
- **Import and Model Transformation:** The import of application topologies and the transformation into GENTL topologies are considered core functionalities for the environment. Both importing and model transformation are widely used functionalities and different libraries/toolkits/plugins supporting these functionalities are available.
- **Proper Graph Layout:** To provide a graphical overview over the topology graph, the displayed nodes and edges should be positioned to avoid overlap as far as possible. This is not a trivial challenge, but many sophisticated algorithms for this matter have been implemented. Hence, the reuse of an existing implementation for graph visualization is highly desirable.

- **Interaction with the Topology Graph:** To annotate the topology graph, the user should be able to select nodes, edges as well as groups of nodes and edges and add different types of annotations. Navigation between topologies and sub-topologies, as well as editing of topologies and topology elements should be supported.

## 5.2 Implementation Options

Possible options for the implementation are: a stand alone application (in a platform independent programming language), a plugin for an existing development platform or a web application. Main criteria for the selection of a form for the implementation, is the availability of reusable features (libraries/plugins etc.) for the requirements listed in the previous section. The remainder of this section gives an outline of some reusable features for each implementation option.

### 5.2.1 Stand Alone Application

A stand alone application requires intense development effort for all parts of the implementation. The import and model transformation could either be implemented by importing XML-files, process them with a DOM-Parser and develop appropriate functionality for the transformations, or by the use of an XSL Transformation Engine.

A DOM-Parser processes a given XML document and transforms it into a Document Object Model (DOM), which is a platform- and language-neutral interface defined by the W3C in [DOM05]. It allows programs and scripts to dynamically access and update the content, structure and style of documents.

The transformation from a source model into a target model is not handled by the DOM-Parser but has to be implemented separately for the desired source models. For the transformation of a Blueprint into the developed application topology language, the Blueprint would be transformed into a Document Object Model, which then would have to be processed into a new model by a suitable matching function.

The other option, an XSL Transformation Engine, builds on the XSL Transformation defined by W3C in [XSL07]. An important role of XSLT is to add styling information to an XML source document by transforming it into a presentation-oriented format such as Extensible Stylesheet Language (XSL), HTML, XHTML or SVG, but it is used for a wide range of other transformation tasks.

A transformation in XSLT is done by a set of template rules which associate patterns that matches nodes in the source document, with a sequence constructor. The evaluation of the sequence constructor can result in the construction of new nodes, which can be used to produce part of a result tree. This way, different documents with similar source tree structures can be transformed into one or more result trees by the same stylesheet.

The graph visualization software Graphviz [gra] provides a way to visualize graphs using a simple text description language. Graphviz offers different style options for nodes and edges, as well as different layout options. Many different language bindings as well as generators and translators that transform other data sources and formats into Graphviz enable the reuse of Graphviz features in other applications.

### 5.2.2 Plugin for an existing Development Platform

Development of the GENTL environment as a plugin to an existing development platform provides the advantage that the framework for the environment is already given. Additional reusable plugins offering support for modeling and graphical representation are available for some platforms, since these functionalities are necessary for a wide range of tools and editors.

A possible Development Platform as basis for the plugin development is Eclipse. The Eclipse Platform offers various plugins and functionality for model driven development as well as for graphical editors.

In the Eclipse Modeling Project [ecl13a], the provision of a unified set of modeling frameworks, tooling, and standards implementations aims at the evolution and promotion of model-based development technologies within the Eclipse community.

The basis of the Eclipse Modeling Project is the Eclipse Modeling Framework (EMF). EMF offers a modeling framework and code generation functionality to support the development of applications which are based on structured data models.

The Graphical Editing Framework (GEF) bundles functionality for the creation of graphical editors and views for the Eclipse Workbench UI. It includes a layout and rendering toolkit, an interactive Model View Controler (MVC) framework and a visualization toolkit which supports the implementation of graphical views.

A set of generative components and runtime infrastructures for the development of graphical editors is provided by the Graphical Modeling Project (GMP) based on EMF and GEF. It includes a toolkit for the model-driven generation of graphical editors, an industry proven application framework for the development of graphical editors, a standard notational meta model and a graphics framework for the development of diagram editors for domain models.

Functionalities for XSLT and Graphviz as mentioned in Section refsec:standAlone are also available for the Eclipse framework. The XSLT Project [ecl13c] offers XSL Transformation and the Graphviz Eclipse plug-in [ecl13b] provides a Java API for Graphviz.

Another opportunity for plugin development is the NetBeans Platform. NetBeans doesn't have a comprehensive modeling project like Eclipse, but it offers functionality for graph visualization in the NetBeans Visual Library [net13a] and XSL Transformation in NetBeans XSL Support [net13b].

### 5.2.3 Web Application

An advantage of a web application over a stand alone application or a plugin for an existing development platform is that a web application can be accessed by a simple web browser, providing a higher degree of platform independence.

Since XSLT is often used for transformations into HTML and XHTML, the use of an XSLT Engine for the model transformation comes naturally for web application development. The use of Graphviz in web applications is possible either through the HTML binding webdot [web] or the HTML5/Javascript canvas viewer Canviz [can].

Another option for the implementation of a web application is the use of a web application framework like Django or Ruby on Rails. Web application frameworks reduce redundancy since similar pages can be generated after templates and don't have to be implemented individually.

Django is based on Python and Ruby on Rails is based on Ruby. There are numerous other frameworks based on different programming languages, but as all frameworks offer the same functionality in slightly different ways, the following discussion concentrates on Django as an example for a web framework.

The Django Framework [dja13] offers an object-relational mapper, which enables the definition of data models in python. The data models are stored in a database and can be accessed either through a dynamic database-access API or through SQL Statements.

For the required model transformation, the defined Application Topology Annotation Framework should be represented as a database schema. The mapping from the discussed topology languages and frameworks has to be implemented as a database import functionality with the appropriate data transformations.

The graph representation can be done by use of pydot [pyd], a python interface to Graphviz. Graphviz can generate SVG graphics with embedded URLs. This enables navigation in the web application through clickable nodes and edges in the topology graph.

| | **Stand Alone Application** | **Eclipse Plugin** | **Django Application** |
|---|---|---|---|
| Platform Independence | operating system independence | operating system independence | device independence |
| Import and Model Transformation | DOM-Parser XSLT Engine | EMF XSLT Project | |
| Proper Graph Layout | Graphviz APIs | GEF Graphviz Plugin | Pydot + Graphviz |
| Interactions with the Topology Graph | | GEF | embedded URLs in SVG files |

**Table 5.1:** Reusable Features for the Implementation Options

### 5.2.4 Options Evaluation

The reusable features described in Section 5.2.2 show that the Eclipse framework offers more features than the Netbeans Platform. As described in Section 5.2.3, web application frameworks like django can help reduce redundancy in web application development.

The decision for an implementation option narrows down to a decision between an implementation as a stand alone application, eclipse plugin or django application. Table 5.1 presents a summary of the reusable features for these options.

The Eclipse framework imposes the most extensive set of reusable features providing a clear advantage over an implementation as a stand alone application. While the django application offers a smaller set of reusable features than the eclipse framework, it also offers a higher degree of platform independence. A django application can be accessed by any device with a web browser.

Because of the huge set of plugins and features available for the eclipse framework, the eclipse plugin development provides a high threshold for beginners. Django offers a much smoother learning curve which puts the development efforts for the two options into another perspective. As a consequence, the GENTL Environment was decided to be implemented as a django application.

## 5.3 GENTL Environment Architecture

The GENTL Environment is a Django project consisting of a Topology App, a Transformation App and two Annotation Apps. The Topology App implements the topology data model and the graph visualization. Static annotation data is captured and displayed in the Static Annotation App and dynamic annotation data is handled by the Dynamic

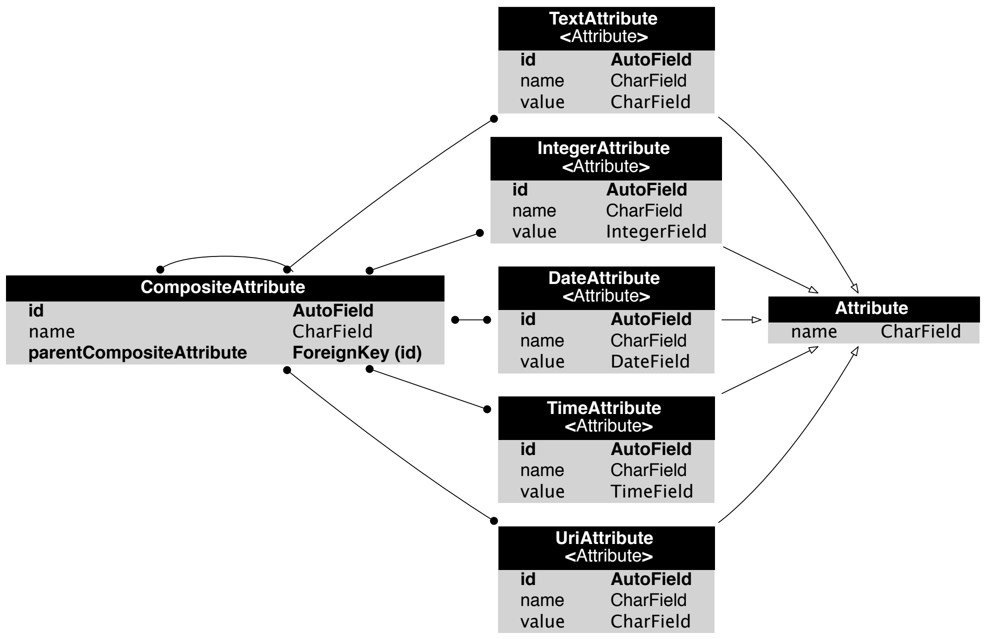


**Figure 5.1:** Attributes Datamodel

Annotation App. Import and export functionalities for topologies and annotations are bundled in the Transformation App.

As prerequisites, the GENTL Environment requires the Django Framework [dja13], the graph visualization software Graphviz [gra] and pydot [pyd] as python interface to Graphviz.

The remainder of this section is structured as follows: The structure of the Topology App is described in Section 5.3.1, Section 5.3.2 describes the contents of the Annotation Apps and the structure of the Transformation App is given in Section 5.3.3.

### 5.3.1 Topology App

The Topology App handles the topology data and the graph visualization. Topology elements and attributes are implemented as model classes. Model classes are python objects, which serve as basis for the generation of database tables containing the model instances. The model instances can be accessed and models can be queried through Djangos dynamic database-access API.

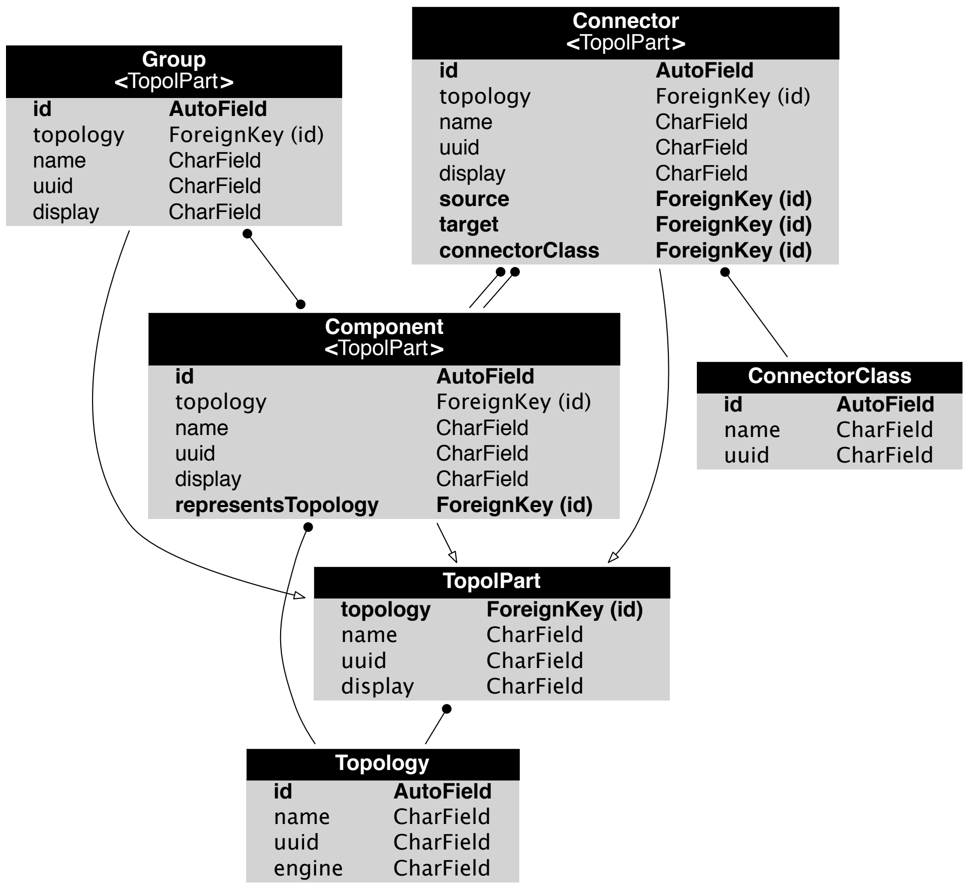


**Figure 5.2:** Datamodel of Topology Elements

Figure 5.1 displays the data model for simple and composite attributes. TextAttribute, IntegerAttribute, UriAttribute, TimeAttribute and DateAttribute classes are derived from an abstract Attribute class and differ only in the datatype of their respective value field. The CompositeAttribute class has many to many relationships to the simple attribute classes and a foreign key relationship to its own class, to enable hierarchical nesting of composite attributes.

Figure 5.2 displays the data model of the topology elements and their interconnections. The Component, Connector and Group classes are subclasses of the abstract class TopolPart. They have a foreign key relationship to the Topology class and many to many relationships to the different attribute classes (not included in the figure for reasons of clarity). Besides the properties defined in the GENTL language, a display field has been added to enable optional hiding of topology elements in the graph representation.

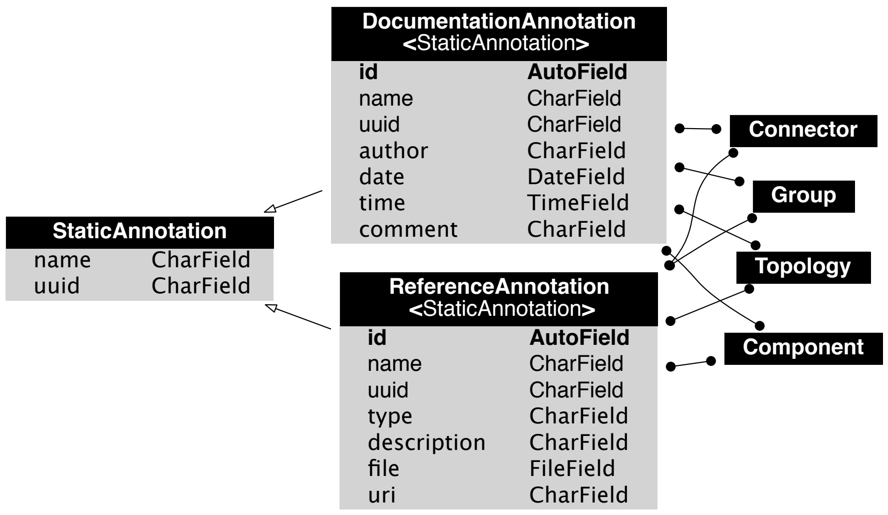


**Figure 5.3:** Datamodel for static Annotations

In the Connector class, the source and target properties are represented as foreign key relationships to the Component class and the connectorClass property is represented as foreign key relationship to the ConnectorClass module. The representsTopology property of a component is reflected in an additional foreign key relationship from the Component to the Topology class. A many to many relationship between Group and Component defines which components belong to which groups. Besides the GENTL topology properties, the Topology class contains an engine field which sets the layout orientation for the graph visualization engine.

The topology data is retrieved from the database and transformed into a graph representation in an svg file through pydot and Graphviz. After the graph generation, the created file is enhanced with javascript event listeners and functions to pass events to the parent document, which handles the display of additional information on topology elements.

### 5.3.2 Annotation Apps

#### Static Annotation App

The Static Annotation App realizes a documentation annotation as an example of a simple annotation and a reference annotation as an example for an external reference annotation. Figure 5.3 depicts the data model for both annotations. They are derived from an abstract Annotation class and have many to many relationships to the Component, Connector, Group and Topology classes from the Topology App.

The DocumentationAnnotation class contains a comment and an author field to capture simple comments for documentation purposes, and a date and a time field, which have the current date and time as default values.

The ReferenceAnnotation contains a type, a URI and an optional description field to capture references to external resources. It also contains an optional file field which can be used to upload an external resource into the GENTL framework.

#### Dynamic Annotation App

The Dynamic Annotation App contains the data model for a sequence of requests to the Nefolog System [Xiu13, AX13], which provides a set of decision support web services. The implemented annotation data model is designed for requests to the cost calculator service of Nefolog.

Figure 5.4 depicts the designed data model. The NefologOfferingAnnotation and the NefologServiceTypeAnnotation classes are derived from the abstract class NefologAnnotation, which contains a foreign key relationship to the Component class from the Topology App. The intended request sequence is to

1. create a NefologServiceTypeAnnotation by requesting the available service types from Nefolog and choosing the desired service type for a component,
2. use the chosen service type to request the available offerings and create a NefologOfferingAnnotation with the chosen offering,
3. retrieve the available configurations for the offering and create a NefologConfigurationAnnotation with the chosen configuration,
4. retrieve the available configuration variables, insert the desired values and create a NefologConfigurationVariablesAnnotation,
5. use the NefologConfigurationVariablesAnnotation to retrieve the cost calculation for the offering with the given configuration values

The NefologConfigurationAnnotation class has a foreign key relationship to the NefologConfigurationAnnotation class, and the NefologConfigurationVariablesAnnotation class has a foreign key relation to the NefologConfigurationAnnotation class. The configuration variables are represented through TextAttributes by a many to many relationship in the NefologConfigurationVariablesAnnotation.

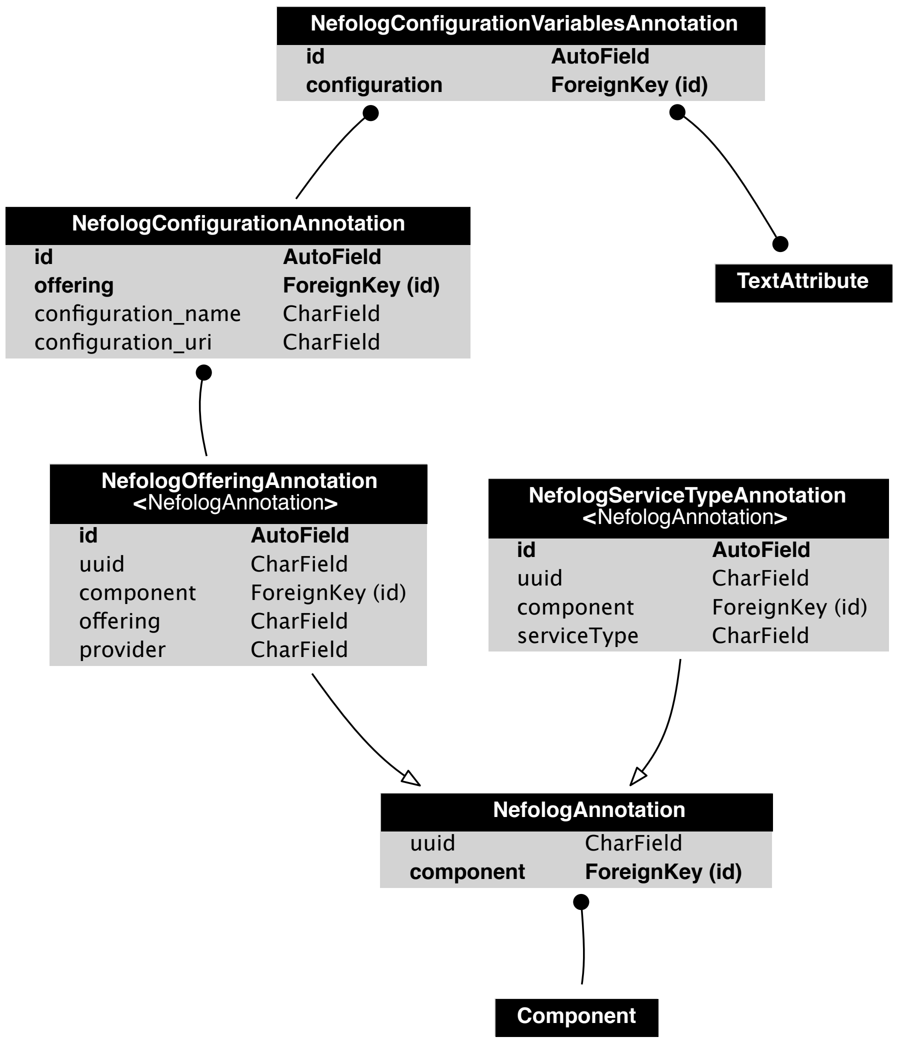

NefologConfigurationVariablesAnnotation
id AutoField
configuration ForeignKey (id)
TextAttribute
NefologConfigurationAnnotation
id AutoField
offering ForeignKey (id)
configuration_name CharField
configuration_uri CharField
NefologOfferingAnnotation
<NefologAnnotation>
id AutoField
uuid CharField
component ForeignKey (id)
offering CharField
provider CharField
NefologServiceTypeAnnotation
<NefologAnnotation>
id AutoField
uuid CharField
component ForeignKey (id)
serviceType CharField
NefologAnnotation
uuid CharField
component ForeignKey (id)
Component

**Figure 5.4:** Nefolog Annotation Datamodel

### 5.3.3 Transformation App

The Transformation App bundles the import functionalities for Blueprint and TOSCA topologies and the serialization and deserialization of GENTL topologies and annotations. The Blueprint import is based on the XML Schema for the Blueprint Language developed as part of the 4CaaSt project[1] and TOSCA import functionality relies on the schema of the TOSCA Version 1.0[2]. The XML Schema for the serialized representation of GENTL topologies and GENTL annotations is given in the appendix.

## 5.4 User Interface

The index page depicted in Figure 5.5 lists all topologies available in the environment. New topologies can be imported from TOSCA, Blueprint or GENTL files and existing topologies can be opened by a click on an item in the topology list.

Figure 5.6 depicts the topology page for the TaxiScenario BackEnd Blueprint. The topology page bundles all the functionalities offered for a specific topology. A graph representation of the topology is given on the left, and additional information on topology elements, annotations and view settings is given in tabs on the right. New components, connectors and groups can be added to the topology through the menu bar, which also provides the option to download the serialized XML representation of the GENTL topology or the annotations connected to the topology.

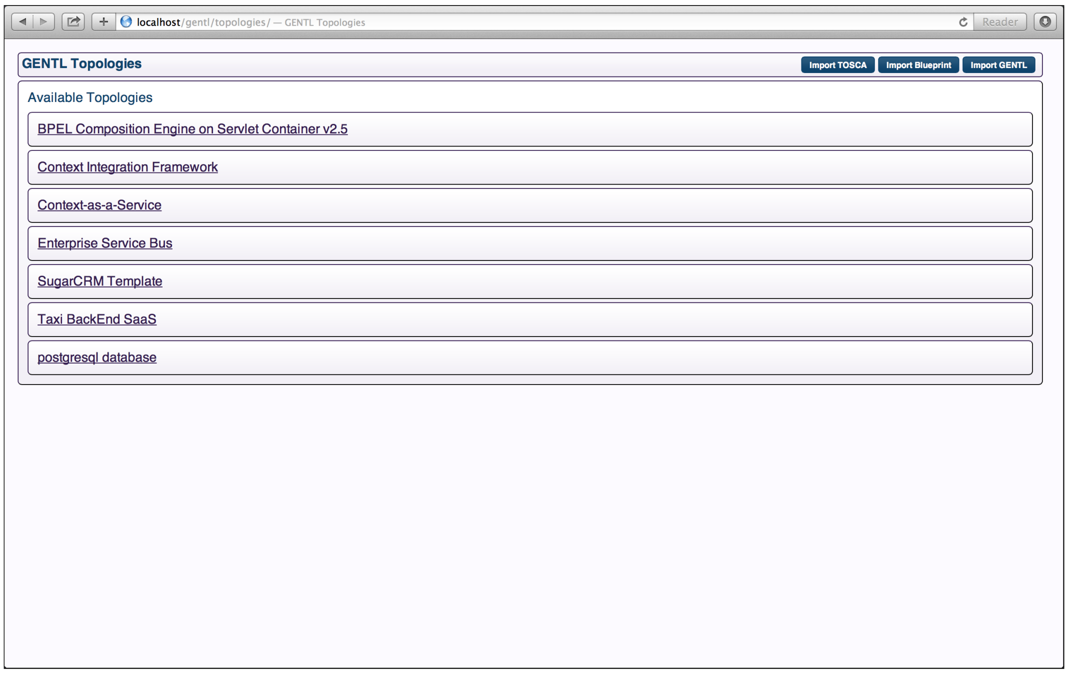
A click on the elements of the topology graph loads information on the clicked element to the tabs on the right. The Element Info tab displays the properties and attributes of the topology element (Figure 5.8(a)) and provides functionalities to edit or delete the selected element. Figure 5.8(b) displays the edit view for a component in the Element Info tab.

In the static annotation tab depicted in Figure 5.7(a), the documentation and reference annotations which are connected to the selected topology element are displayed. Single annotations can be disconnected from the element, existing annotations can be connected to the element and new annotations can be created.

The dynamic annotation tab (Figure 5.7(b)) displays the dynamic annotations connected to the selected topology element. New annotations can be added and existing annotations can be used to invoke service calls. The result from the service calls is also displayed in this tab.

[1]EU Project 4CaaSt: `http://www.4caast.eu`

[2]Topology and Orchestration Specification for Cloud Applications Version 1.0: `http://docs.oasis-open.org/tosca/TOSCA/v1.0/cos01/schemas/TOSCA-v1.0.xsd`

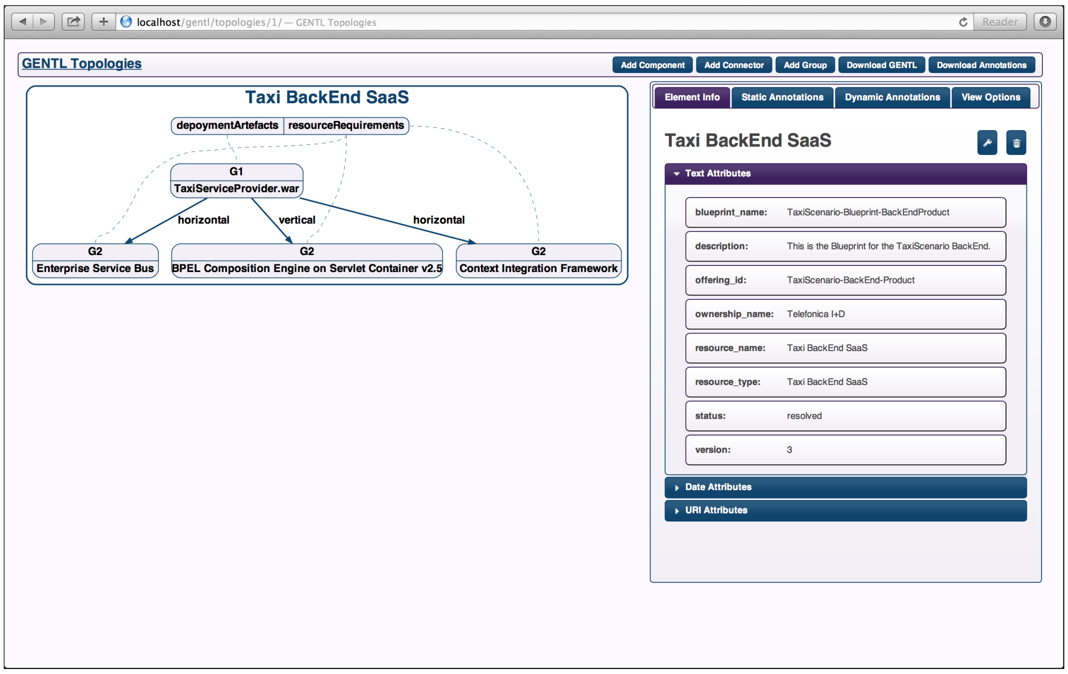


GENTL Topologies

Available Topologies

- BPEL Composition Engine on Servlet Container v2.5
- Context Integration Framework
- Context-as-a-Service
- Enterprise Service Bus
- SugarCRM Template
- Taxi BackEnd SaaS
- postgresql database

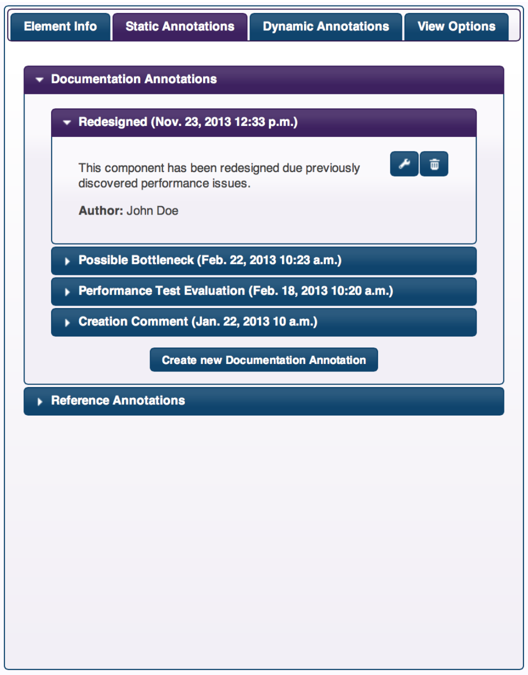
**Figure 5.5:** Index Page of the GENTL Environment



Taxi BackEnd SaaS

| | |
|---|---|
| blueprint_name: | TaxiScenario-Blueprint-BackEndProduct |
| description: | This is the Blueprint for the TaxiScenario BackEnd. |
| offering_id: | TaxiScenario-BackEnd-Product |
| ownership_name: | Telefonica I+D |
| resource_name: | Taxi BackEnd SaaS |
| resource_type: | Taxi BackEnd SaaS |
| status: | resolved |
| version: | 3 |

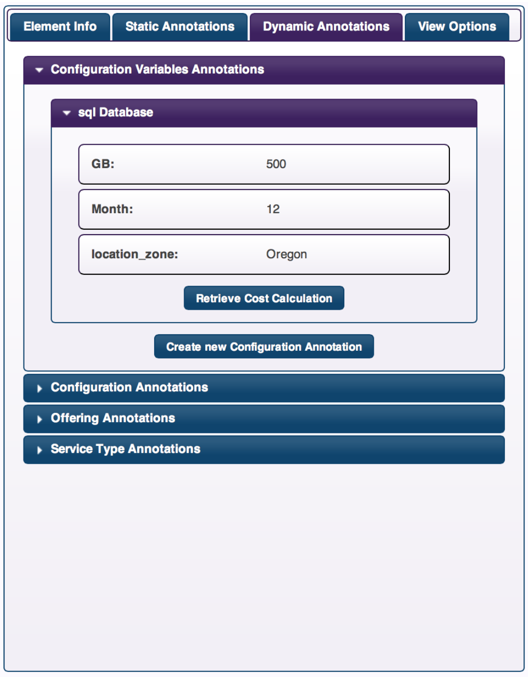
**Figure 5.6:** Topology Overview


Element Info
Static Annotations
Dynamic Annotations
View Options
Documentation Annotations
Redesigned (Nov. 23, 2013 12:33 p.m.)
Possible Bottleneck (Feb. 22, 2013 10:23 a.m.)
Performance Test Evaluation (Feb. 18, 2013 10:20 a.m.)
Creation Comment (Jan. 22, 2013 10 a.m.)
Create new Documentation Annotation
Reference Annotations

This component has been redesigned due previously discovered performance issues.

**Author:** John Doe

(a) Static Annotations

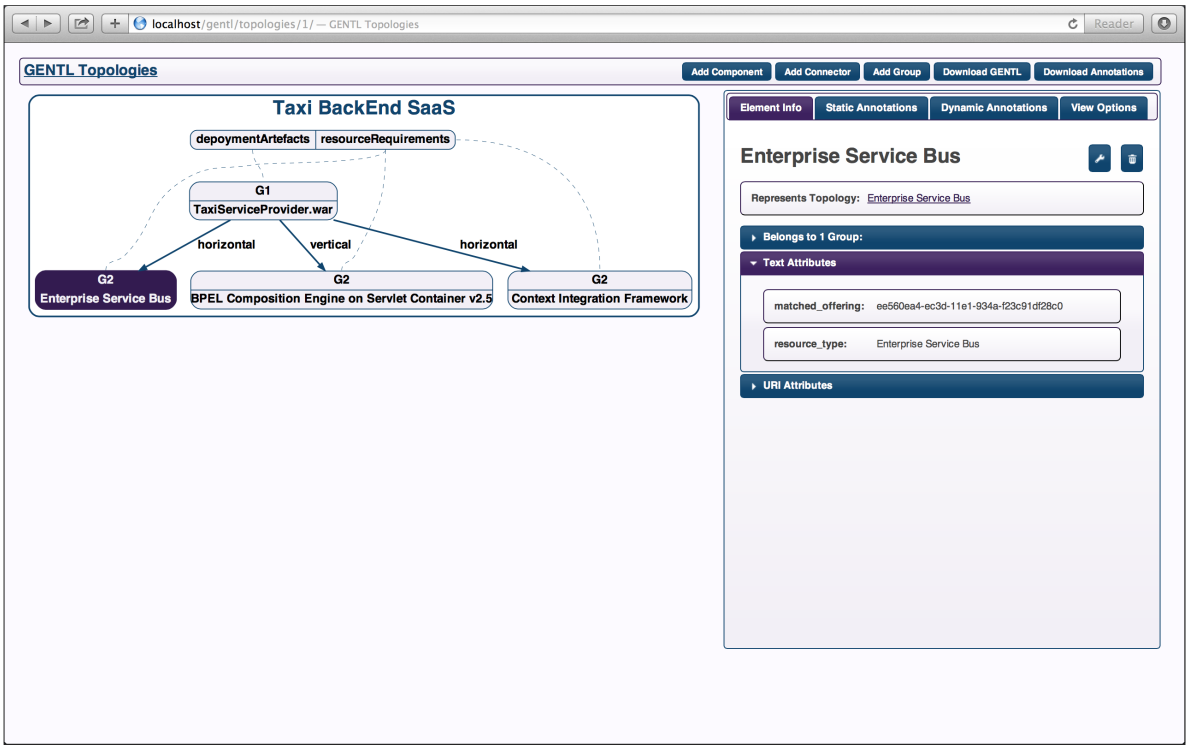

Element Info
Static Annotations
Dynamic Annotations
View Options
Configuration Variables Annotations
sql Database
Retrieve Cost Calculation
Create new Configuration Annotation
Configuration Annotations
Offering Annotations
Service Type Annotations

| | |
|---|---|
| **GB:** | 500 |
| **Month:** | 12 |
| **location_zone:** | Oregon |

(b) Nefolog Annotations

**Figure 5.7:** Contents of the Annotation Tabs

In the view options tab displayed in Figure 5.8(c), the settings for the topology graph display can be changed. This includes changing the layout direction for the topology graph between “top to bottom” and “left to right”. Each component, connector and group in the topology can be hidden individually, to limit the displayed elements and focus on specific parts of the topology.



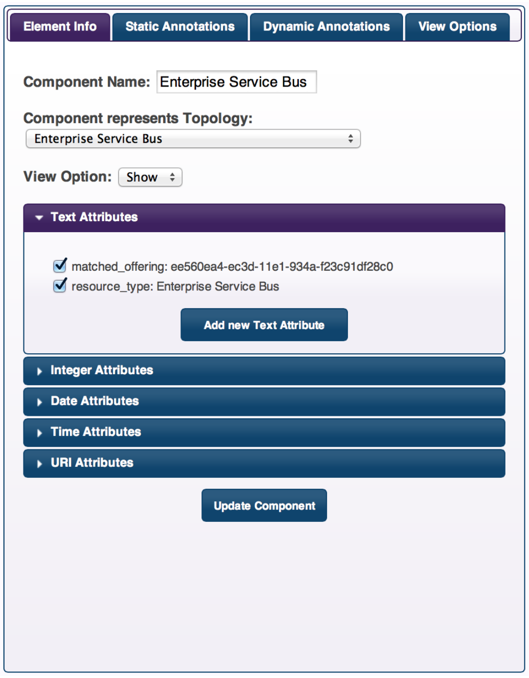
Enterprise Service Bus

Represents Topology: Enterprise Service Bus

matched_offering: ee560ea4-ec3d-11e1-934a-f23c91df28c0

resource_type: Enterprise Service Bus

(a) Component Detail Display



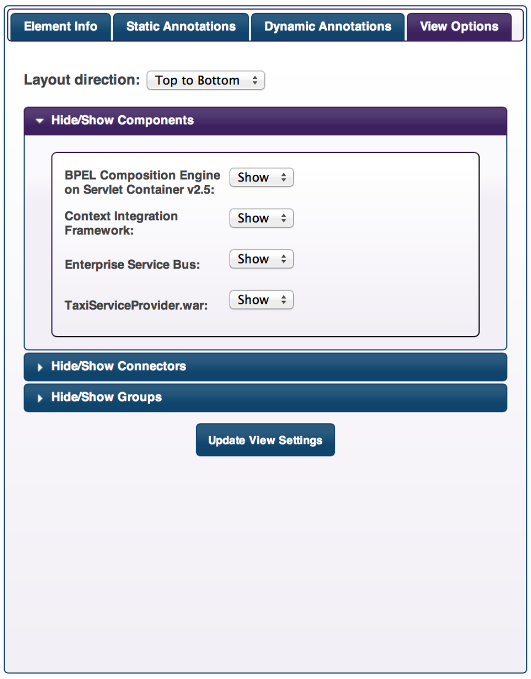
Component Name: Enterprise Service Bus

Component represents Topology: Enterprise Service Bus

View Option: Show

☑ matched_offering: ee560ea4-ec3d-11e1-934a-f23c91df28c0
☑ resource_type: Enterprise Service Bus

(b) Edit Component



Layout direction: Top to Bottom

BPEL Composition Engine on Servlet Container v2.5: Show

Context Integration Framework: Show

Enterprise Service Bus: Show

TaxiServiceProvider.war: Show

(c) View Settings

**Figure 5.8:** Contents of the Topology Information and View Tabs

# 6 Evaluation

The introduced framework offers functionality to import topologies from GENTL, TOSCA and Blueprint files and display the topologies as graphs. Annotations can be added to the topologies in form of documentation annotations, reference annotations and a set of service invocation annotations which retrieve information from the external Nefolog system.

The remainder of this section is structured as follows: In Section 6.1 the import and visualization of Blueprint and TOSCA topologies is evaluated. The implemented annotations are discussed in Section 6.2 and Section 6.3 specifies how the framework can be extended by adding additional annotations or import functionalities for other topology languages.

## 6.1 Topology Import and Visualization

A topology import from a GENTL file merely deserializes the contained data into the GENTL Environment. Imports from TOSCA and Blueprint topologies translate the topology data into the GENTL language and create the corresponding GENTL topologies. This section evaluates how TOSCA and Blueprint topologies are translated to GENTL and how the resulting topology graphs are visualized.

### 6.1.1 TOSCA Import

Figure 6.1 depicts the GENTL topology imported from the TOSCA ServiceTemplate of the SugarCRM application used for interoperability demonstration purposes[1]. The TOSCA node templates are represented as components, which belong to a group that corresponds to the type of the node template. Relationship types are denoted as connector classes and each relationship template is depicted as connector of the connector class representing its type.

[1]SugarCRM CSAR file: `https://www.oasis-open.org/committees/download.php/50158/SugarCRM-Interop-20130803.zip`

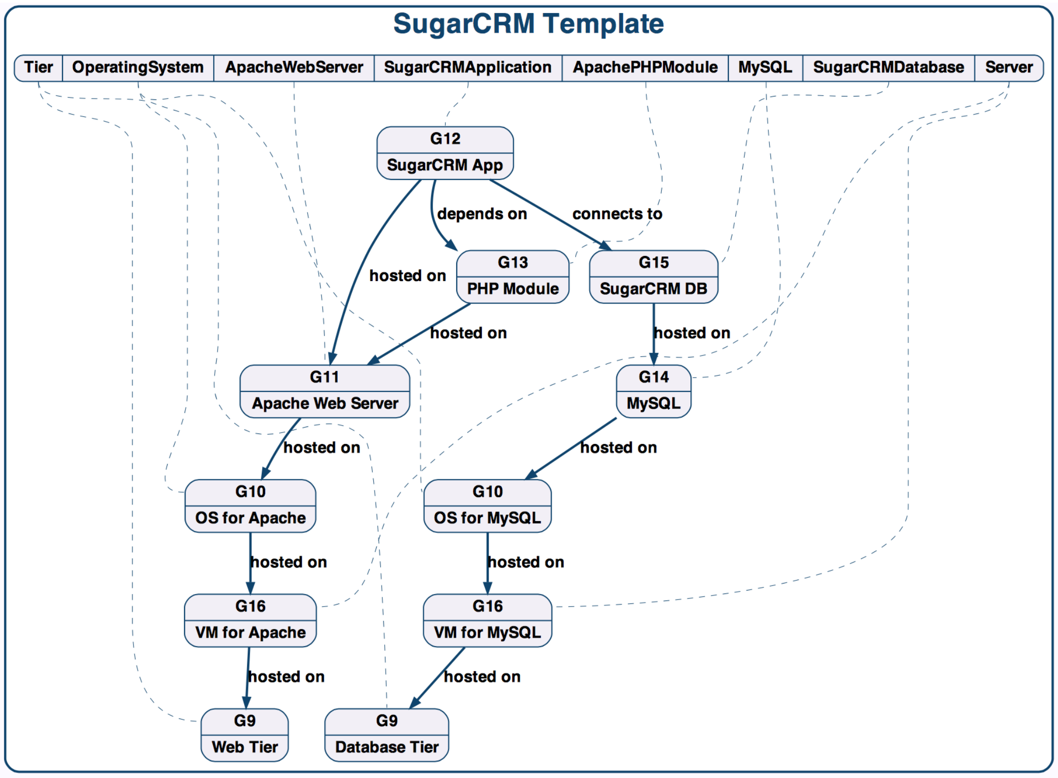
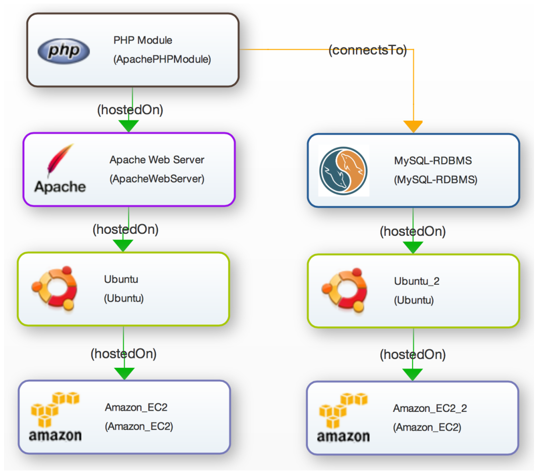
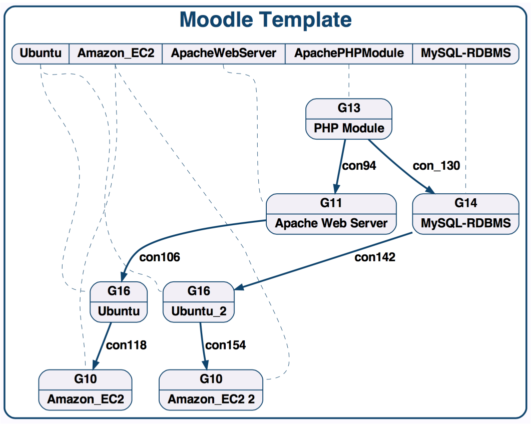


**Figure 6.1:** Sugar CRM Topology



(a) Moodle Topology in Winery [win13]



(b) Moodle Topology in GENTL

**Figure 6.2:** Comparison of Topology Displays

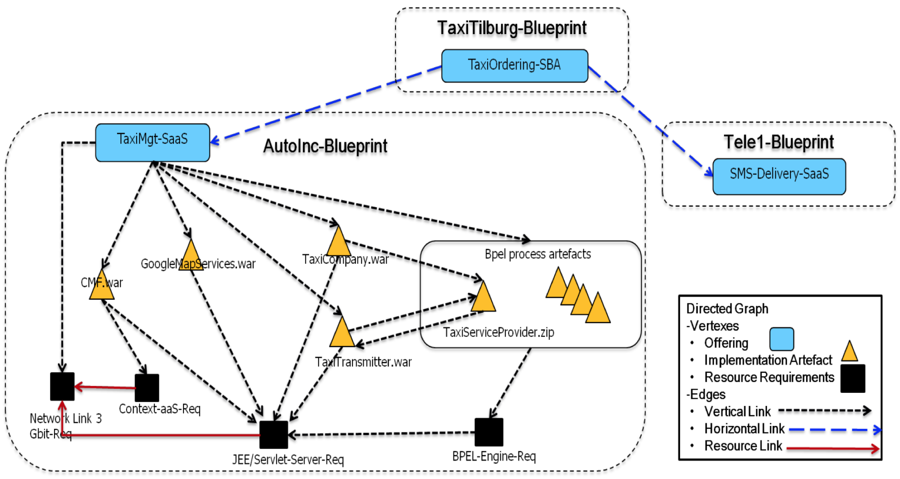


**Figure 6.3:** Sample Blueprint Virtual Architecture Graph [NLPH12].

In [BBK+12] a visual notation for TOSCA is proposed and a topology modeler for TOSCA is given in [win13]. Figure 6.2(a) shows a topology graph in the winery topology modeler and Figure 6.2(b) depicts the same topology as GENTL topology.

The winery topology modeler is designed for TOSCA topologies. Different node types can be distinguished by separate colors and specific icons in the node and each relationship type can be displayed in a different color. GENTL represents node types as groups of nodes and the language allows for a component to be in multiple groups. Hence the visualization of node types is decoupled from the topology component and displayed in a separate graph node with connections to the components of the certain node type.

## 6.1.2 Blueprint Import

The Blueprint Approach does not specify a visual notation for Blueprint topologies, but in [NLPH12], an example virtual architecture graph is depicted (see Figure 6.3). A Blueprint which is imported into the GENTL Environment results in a set of topologies. Each created topology matches an offering in the imported Blueprint.

The import of the TaxiScenario-Blueprint-BackEndProduct for the Taxi Application scenario in 4CaaSt results in six topologies. First the “Taxi BackEnd SaaS Topology” depicted in Figure 6.4 is created, which consists of one deployment artefact and three resource requirements. This topology represents the offering contained in the Blueprint.

The required blueprints section of the TaxiScenario-Blueprint-BackEndProduct contains three Blueprints, which specify the matched offerings for the resource requirements. Each of those Blueprints is imported which results in topologies for the included offerings

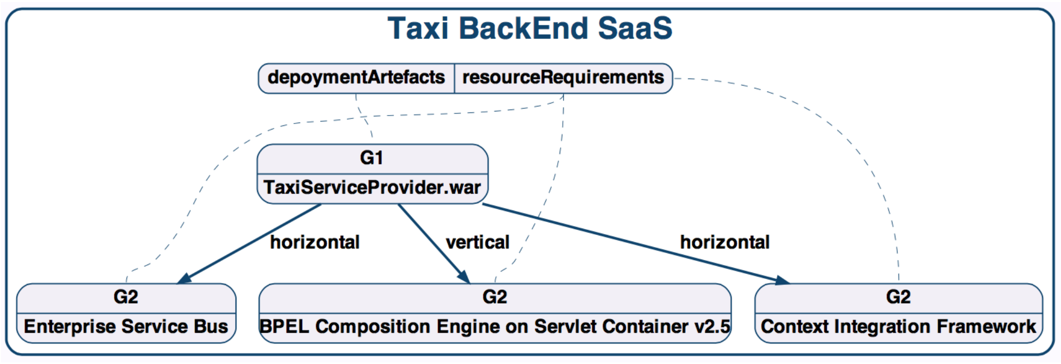


**Figure 6.4:** Taxi BackEnd SaaS Topology

BPEL Composition Engine on Servlet Container v2.5

**Figure 6.5:** BPEL Composition Engine Topology

and the required blueprints. Components reflecting resource requirements with a matched offering are connected to the topology of the matched offering through the representsTopology property.

Regarding the “Taxi BackEnd SaaS Topology”, this means that the “Enterprise Service Bus” component is connected to the topology depicted in Figure 6.7, the “BPEL Composition Engine on Servlet Container v2.5” component is connected to the topology depicted in Figure 6.5 and the “Context Integration Framework” component is connected to the topology in Figure 6.6.

The “BPEL Composition Engine on Servlet Container v2.5” topology graph (Figure 6.5) is empty, because the corresponding Blueprint contains neither deployment artefacts nor resource requirements. This results in an empty topology with Blueprint and offering properties as topology attributes.

The “Enterprise Service Bus” topology in Figure 6.7 and the “Context Integration Framework” topology in Figure 6.6 also contain resource requirements with matched offerings. The corresponding components are either connected to an existing topology in the framework (e.g. the “Enterprise Service Bus” component in the “Context Integration Framework is connected to the topology in Figure 6.7) or, if the matched offering has not been imported yet, a new topology is created.

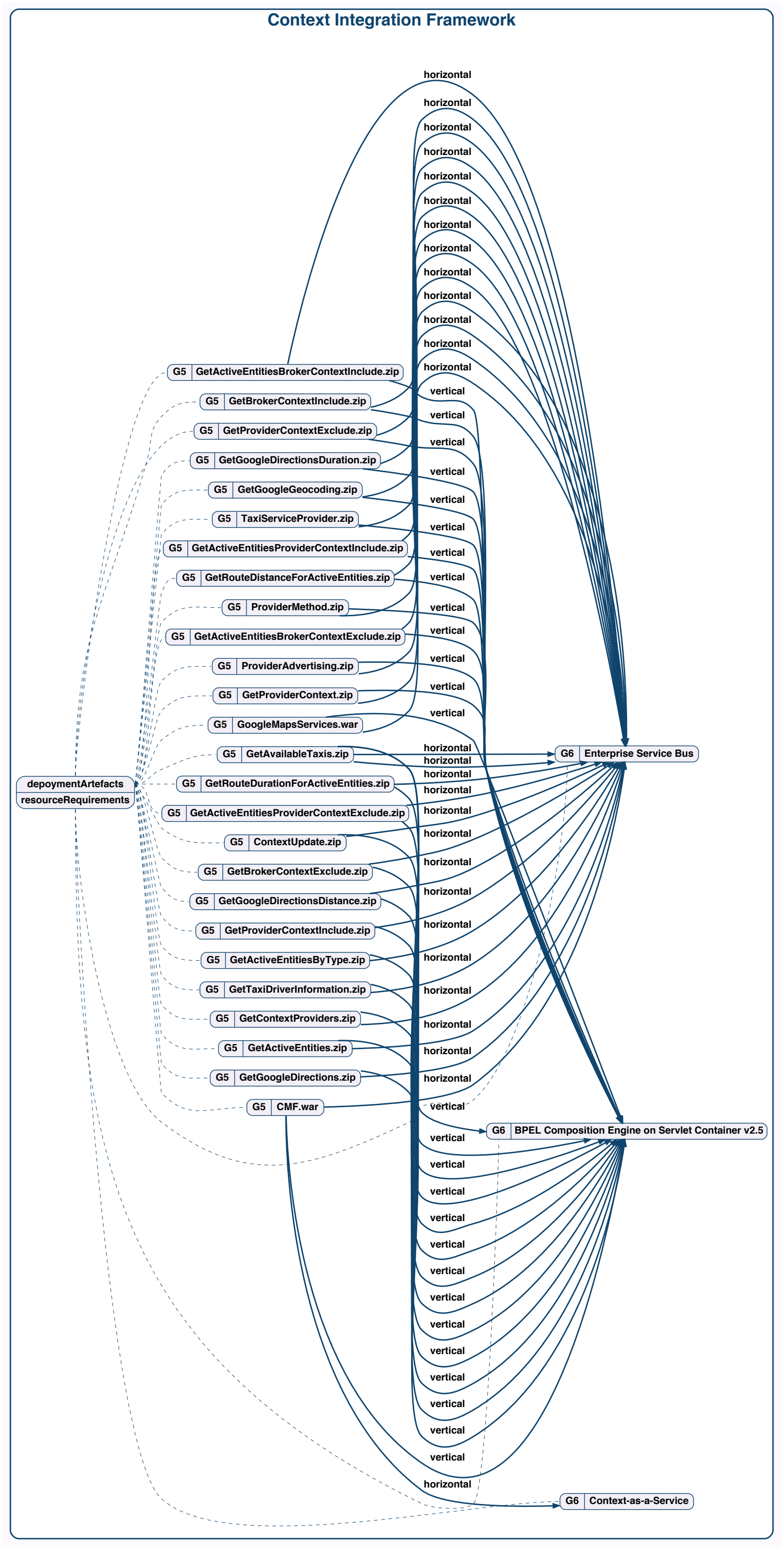

Context Integration Framework
horizontal
horizontal
horizontal
horizontal
horizontal
horizontal
horizontal
horizontal
horizontal
horizontal
horizontal
horizontal
horizontal
G5 GetActiveEntitiesBrokerContextInclude.zip
G5 GetBrokerContextInclude.zip
G5 GetProviderContextExclude.zip
G5 GetGoogleDirectionsDuration.zip
G5 GetGoogleGeocoding.zip
G5 TaxiServiceProvider.zip
G5 GetActiveEntitiesProviderContextInclude.zip
G5 GetRouteDistanceForActiveEntities.zip
G5 ProviderMethod.zip
G5 GetActiveEntitiesBrokerContextExclude.zip
G5 ProviderAdvertising.zip
G5 GetProviderContext.zip
G5 GoogleMapsServices.war
G5 GetAvailableTaxis.zip
G5 GetRouteDurationForActiveEntities.zip
G5 GetActiveEntitiesProviderContextExclude.zip
G5 ContextUpdate.zip
G5 GetBrokerContextExclude.zip
G5 GetGoogleDirectionsDistance.zip
G5 GetProviderContextInclude.zip
G5 GetActiveEntitiesByType.zip
G5 GetTaxiDriverInformation.zip
G5 GetContextProviders.zip
G5 GetActiveEntities.zip
G5 GetGoogleDirections.zip
G5 CMF.war
vertical
vertical
vertical
vertical
vertical
vertical
vertical
vertical
vertical
vertical
vertical
vertical
vertical
horizontal
horizontal
horizontal
horizontal
horizontal
horizontal
horizontal
horizontal
horizontal
horizontal
horizontal
horizontal
horizontal
horizontal
depoymentArtefacts
resourceRequirements
G6 Enterprise Service Bus
vertical
G6 BPEL Composition Engine on Servlet Container v2.5
vertical
vertical
vertical
vertical
vertical
vertical
vertical
vertical
vertical
vertical
vertical
vertical
vertical
horizontal
G6 Context-as-a-Service

**Figure 6.6:** Context Integration Framework Topology

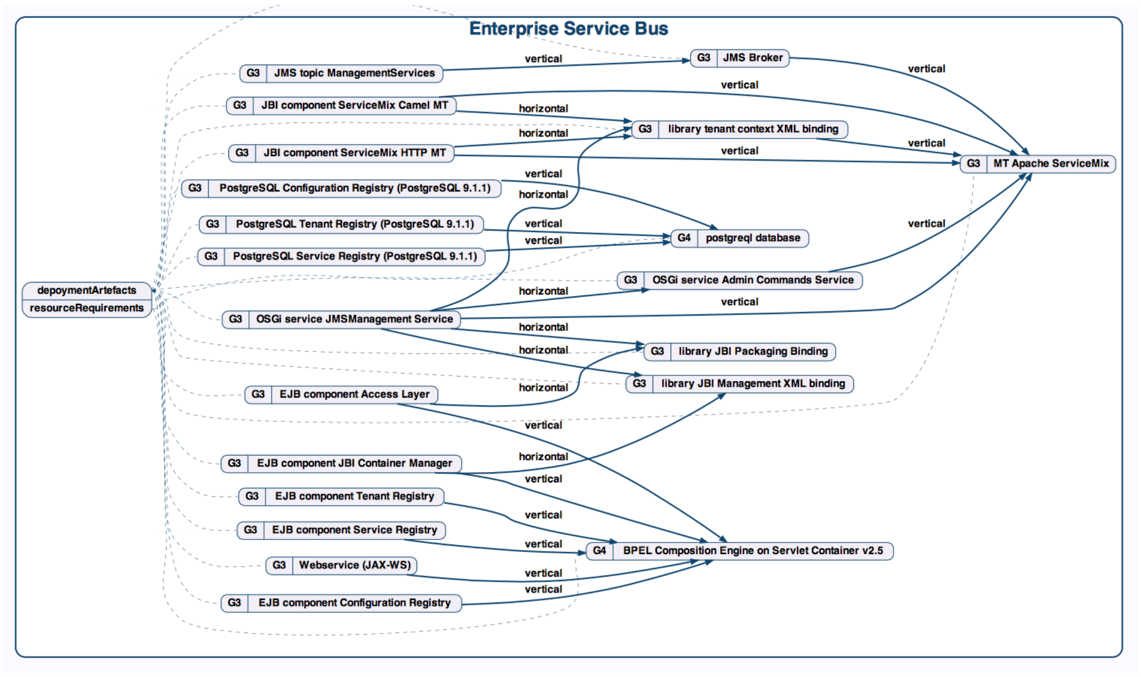

Enterprise Service Bus
G3 JMS Broker
G3 JMS topic ManagementServices
G3 JBI component ServiceMix Camel MT
G3 library tenant context XML binding
G3 JBI component ServiceMix HTTP MT
G3 MT Apache ServiceMix
G3 PostgreSQL Configuration Registry (PostgreSQL 9.1.1)
G3 PostgreSQL Tenant Registry (PostgreSQL 9.1.1)
G4 postgreql database
G3 PostgreSQL Service Registry (PostgreSQL 9.1.1)
depoymentArtefacts
resourceRequirements
G3 OSGi service Admin Commands Service
G3 OSGi service JMSManagement Service
G3 library JBI Packaging Binding
G3 library JBI Management XML binding
G3 EJB component Access Layer
G3 EJB component JBI Container Manager
G3 EJB component Tenant Registry
G3 EJB component Service Registry
G4 BPEL Composition Engine on Servlet Container v2.5
G3 Webservice (JAX-WS)
G3 EJB component Configuration Registry
vertical
horizontal

**Figure 6.7:** Enterprise Service Bus Topology

## 6.2 Topology Annotations

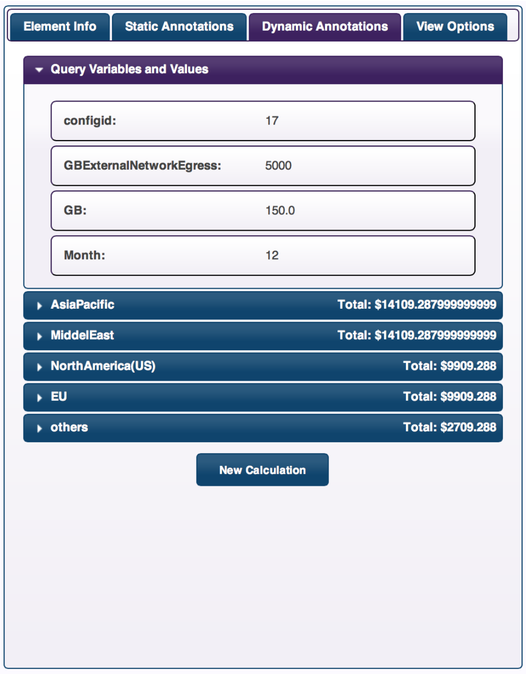
### 6.2.1 Static Annotations

The GENTL Environment offers two kinds of static annotations. The documentation annotation can be used to capture comments on topologies and topology elements and consist of an author textfield, a comment textfield and a date and time field. The reference annotation is designed to attach arbitrary external resources to topologies and topology elements. Information on the contents of the external resource can be captured in the annotation through a type and a description field.

### 6.2.2 Dynamic Annotations

The dynamic annotations introduced in the framework form a sequence of annotation creation steps and service calls, which can be used to retrieve cost calculations for a specific component in a topology.

In a first step, the service type of a component is defined by retrieving the available service types and choosing one of them for an annotation to the component. The created

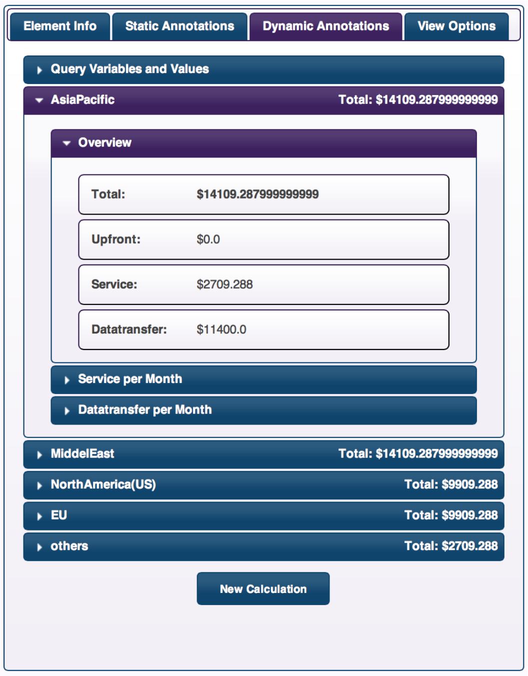


Query Variables and Values

| | |
|---|---|
| configid: | 17 |
| GBExternalNetworkEgress: | 5000 |
| GB: | 150.0 |
| Month: | 12 |

| | |
|---|---|
| AsiaPacific | Total: $14109.287999999999 |
| MiddelEast | Total: $14109.287999999999 |
| NorthAmerica(US) | Total: $9909.288 |
| EU | Total: $9909.288 |
| others | Total: $2709.288 |

(a) Query Variables and Values



AsiaPacific Total: $14109.287999999999

Overview

| | |
|---|---|
| Total: | $14109.287999999999 |
| Upfront: | $0.0 |
| Service: | $2709.288 |
| Datatransfer: | $11400.0 |

| | |
|---|---|
| MiddelEast | Total: $14109.287999999999 |
| NorthAmerica(US) | Total: $9909.288 |
| EU | Total: $9909.288 |
| others | Total: $2709.288 |

(b) Cost Calculation

**Figure 6.8:** Cost Calculation Response

annotation may be used to retrieve the available offerings for the service type and annotate the component with one of the offerings. Based on the offering annotation, available configurations can be retrieved and a suitable configuration may be saved as annotation to the component. Available configuration variables can be retrieved for a configuration annotation and with the values for those variables, the costs for the specified service can be retrieved.

The retrieved cost calculation contains the Query Variables and Values used for the calculation and cost information for different location zones as displayed in Figure 6.8(a) and 6.8(b).

The sequence of service calls provides the opportunity to retrieve cost calculations based on different settings. Each step in the cost calculation process may be repeated with a different selection, leading to a new annotation based on different settings.

## 6.3 Extending the Framework

The framework can be extended by import functions for additional languages, or by new kinds of static or dynamic annotations. To introduce a new language import, the

Transformation App has to be extended. The app contains a package with a set of basic functions for the import of topologies and contained elements, which can be reused for additional languages.

This reduces the import of new languages to the design of a sufficient mapping to the elements of the GENTL language and the implementation of the functionality to retrieve the source data. The implemented language imports are both XML based, but additional imports for different language representations could extend the framework. Available APIs of cloud frameworks for example could be used to design import functions for the managed topologies.

Additional annotations can be added either as extension of one of the existing annotation apps, or included in the framework as a separate app. In both cases, the serialization and deserialization of the annotation has to be added to the Transformation App.

The GENTL annotation model provides a generic structure for annotations which has to be refined to provide a detailed model for a specific annotation. This refinement process includes the design of an XML Schema definition for the serialized representation, the design of a corresponding data model used in the environment and the specification of the meaning and intended usage for the annotation.

To include a new annotation into the framework, the data model and the import and export functionality has to be implemented. The views to display, create, edit and delete annotations can either be added to one of the existing annotation tabs or provided in a separate tab.

For a new simple annotation or a new kind of reference annotation the extension of the existing Static Annotation App and the corresponding view template is advisable. If the new annotation consists of a series of connected annotations like the introduced nefolog annotations, the design of a new separate app and the display in an extra tab should be the preferred option.

# 7 Conclusion

In this thesis, a set of common concepts in application topology languages have been identified on the basis of the description of existing application topologies and frameworks. The new application topology language GENTL has been introduced as an abstraction from language specific details to provide a generalized topology description which can capture topologies described in different languages. Mappings from existing topology languages and frameworks to the designed language have been given.

The designed language describes a topology and the contained elements in a very generic way, which provides the opportunity to capture different topology concepts in the same model. A GENTL topology contains components, connectors between components, groups of components and connector classes.

Components have a representsTopology property which enables the nesting of an existing topology as a component into a new topology. Connectors represent a directed link between a source and a target component and are grouped into distinct connector classes specifying the expressed type of the connection. Groups of components can be used to specify certain types of components as well as to express common properties of different kinds of components. Components can be part of multiple groups.

Additional information on topologies, components, connectors and groups are captured as generic but typed attributes on the respective elements. Attributes can be simple attributes consisting of a name and value pair, as well as composite attributes which may contain a sequence of simple and composite attributes.

Different existing topology annotations have been described and categorized into Discovery, Provision and Management and Design Support Annotations, based on their intended application area, as well as into static and dynamic annotations. The processing modes automatic processing, human processing and hybrid processing have been distinguished.

An annotation scheme for GENTL has been designed to capture static and dynamic annotations with a focus on annotations supporting design decisions. The abstract annotation scheme has been instantiated for exemplary static and dynamic annotations.

An extensible application topology framework with a corresponding topology annotation scheme has been designed and implemented as a Django web application. The framework offers import functions for GENTL, Blueprint and TOSCA topologies.

Topologies can be annotated with documentation annotations, reference annotations providing connections to external resources and dynamic annotations which can be used to retrieve cost calculations from the external decision support system Nefolog. Topologies and annotations can be exported to the XML representation of GENTL and GENTL annotations.

The framework is designed to be extensible. Extensions can be included as new import functionalities for different application topology languages, and as new static and dynamic annotations, providing additional options for decision support.

The presented framework provides the basis for decision support during application architecture design for the cloud. It provides the means to extend the framework with further decision support functionalities building towards a comprehensive support system for cloud application design.

New import functionalities for additional languages, like Cafe or CloudML as well as import options for information captured in cloud management tools may be introduced in the future to complement the existing import functionality for Blueprint, TOSCA and GENTL topologies.

The environment may also be extended to benefit from additional annotations e.g. to support the attachment of design patterns and CAP properties (as introduced in Section 2.3.5) or annotations for performance calculations like the performance characteristics included in the Descartes Meta-Model.

Export functionalities from GENTL to other languages could be introduced with accompanying language-specific annotations to facilitate the transformation. Those transformations could be implemented as hybrid-processing annotations, which perform the translation to the target language based on the GENTL topology and some user provided matching information. For an export to TOSCA the matching information would include e.g. the node types of the components in the topology, and the relationship types of the connectors.

# 8 Appendix

## XML Schema for GENTL and GENTL Annotations

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:simpleType name="UUID">
    <xs:restriction base="xs:string"/>
  </xs:simpleType>
  <xs:complexType name="innerReference">
    <xs:attribute name="uuid" type="UUID" use="required"/>
  </xs:complexType>
  <xs:element name="topology">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="topologyAttributes" type="attributeType"/>
        <xs:element ref="component" maxOccurs="unbounded"/>
        <xs:element ref="connectorClass" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="connector" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="group" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="uuid" type="UUID" use="required"/>
      <xs:attribute name="name" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="group">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="groupAttributes" type="attributeType"/>
        <xs:element name="component" type="innerReference" minOccurs="0"
            maxOccurs="unbounded"/>
        <xs:element name="connector" type="innerReference" minOccurs="0"
            maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="uuid" type="UUID" use="required"/>
      <xs:attribute name="name" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="component" type="componentType"/>
  <xs:element name="connector" type="connectorType"/>
  <xs:element name="connectorClass" type="connectorClassType"/>
  <xs:complexType name="componentType">
```

```
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:element name="componentAttributes" type="attributeType"/>
    </xs:sequence>
    <xs:attribute name="uuid" type="UUID" use="required"/>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="representsTopology" type="UUID" use="optional"/>
  </xs:complexType>
  <xs:complexType name="connectorClassType">
    <xs:attribute name="uuid" use="required"/>
    <xs:attribute name="name" use="required"/>
  </xs:complexType>
  <xs:complexType name="connectorType">
    <xs:sequence>
      <xs:element name="source" type="innerReference"/>
      <xs:element name="target" type="innerReference"/>
      <xs:element name="class" type="innerReference"/>
      <xs:element name="connectorAttributes" type="attributeType"/>
    </xs:sequence>
    <xs:attribute name="uuid" type="UUID" use="required"/>
    <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:complexType name="attributeType">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="simpleAttribute" type="simpleAttributeType"/>
      <xs:element name="compositeAttribute" type="compsiteAttributeType"/>
    </xs:choice>
  </xs:complexType>
  <xs:complexType name="simpleAttributeType">
    <xs:choice>
      <xs:element name="textAttribute" type="textAttributeType"/>
      <xs:element name="intAttribute" type="intAttributeType"/>
      <xs:element name="referenceAttribute" type="referenceAttributeType"/>
      <xs:element name="dateAttribute" type="dateAttributeType"/>
      <xs:element name="timeAttribute" type="timeAttributeType"/>
      <xs:element name="genericAttribute" type="genericAttributeType"/>
    </xs:choice>
  </xs:complexType>
  <xs:complexType name="compsiteAttributeType">
    <xs:sequence maxOccurs="unbounded">
      <xs:choice>
        <xs:element name="simpleAttribute" type="simpleAttributeType"/>
        <xs:element name="compositeAttribute" type="compsiteAttributeType"/>
      </xs:choice>
    </xs:sequence>
    <xs:attribute name="name" use="required"/>
  </xs:complexType>
  <xs:complexType name="genericAttributeType">
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="value" type="xs:anySimpleType" use="required"/>
  </xs:complexType>
```

```
<xs:complexType name="textAttributeType" final="restriction">
  <xs:complexContent>
    <xs:restriction base="genericAttributeType">
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="value" use="required">
        <xs:simpleType>
          <xs:restriction base="xs:string"/>
        </xs:simpleType>
      </xs:attribute>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="intAttributeType">
  <xs:complexContent>
    <xs:restriction base="genericAttributeType">
      <xs:attribute name="value" type="xs:int" use="required"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="referenceAttributeType">
  <xs:complexContent>
    <xs:restriction base="genericAttributeType">
      <xs:attribute name="value" type="xs:anyURI" use="required"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="timeAttributeType">
  <xs:complexContent>
    <xs:restriction base="genericAttributeType">
      <xs:attribute name="value" type="xs:time" use="required"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="dateAttributeType">
  <xs:complexContent>
    <xs:restriction base="genericAttributeType">
      <xs:attribute name="value" type="xs:date" use="required"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:element name="annotations">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="annotation" type="annotationType"
          maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:complexType name="annotationType">
  <xs:sequence>
```

```
      <xs:element name="parent" type="UUID" maxOccurs="unbounded"/>
      <xs:element name="annoationAttributes" type="attributeType"/>
      <xs:choice>
        <xs:element name="externalReference" type="xs:anyURI"/>
        <xs:element name="serviceInvocation">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="requestURL" type="xs:anyURI"/>
              <xs:element name="staticParameters" type="attributeType"/>
              <xs:element name="dynamicParameters">
                <xs:complexType>
                  <xs:sequence minOccurs="0" maxOccurs="unbounded">
                    <xs:element name="requestParameter"
                        type="requestParameterType"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="staticAnnotation" type="attributeType"/>
      </xs:choice>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="uuid" type="UUID" use="required"/>
  </xs:complexType>
  <xs:complexType name="serviceInvocationType">
    <xs:sequence>
      <xs:element name="requestUrl" type="xs:anyURI"/>
      <xs:element name="parameters"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="requestParameterType">
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="datatype" type="dataType" use="required"/>
  </xs:complexType>
  <xs:simpleType name="dataType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="String"/>
      <xs:enumeration value="Integer"/>
      <xs:enumeration value="URI"/>
      <xs:enumeration value="Date"/>
      <xs:enumeration value="Time"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

# Bibliography

[AFG$^+$09] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, M. Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, 2009. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html`. (Cited on page 13)

[ASFL13] V. Andrikopoulos, S. Strauch, C. Fehling, F. Leymann. CAP-Oriented Design for Cloud-native Applications, 2013. To be published. (Cited on pages 7, 45, 46 and 47)

[ASL13] V. Andrikopoulos, S. Strauch, F. Leymann. Decision Support for Application Migration to the Cloud: Challenges and Vision. In *Proceedings of the 3rd International Conference on Cloud Computing and Service Science (CLOSER'13)*, pp. 1–7. SciTePress, 2013. URL `http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2013-18&engl=`. (Cited on page 11)

[AX13] V. Andrikopoulos, M. Xiu. The Nefolog MiDSuS Systems. Technical report, Universität Stuttgart, Institut für die Architektur von Anwendungssystemen, 2013. (Cited on page 73)

[BBK$^+$12] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, D. Schumm. Vino4TOSCA: A Visual Notation for Application Topologies Based on TOSCA. In *OTM 2012, Part I*, volume 7565 of *Lecture Notes in Computer Science (LNCS)*, pp. 416–424. Springer-Verlag, 2012. doi: 10.1007/978-3-642-33606-5_25. URL `http://dx.doi.org/10.1007/978-3-642-33606-5_25`. (Cited on page 81)

[BBK$^+$13] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, M. Wieland. Policy-Aware Provisioning of Cloud Applications. In *SECURWARE 2013, The Seventh International Conference on Emerging Security Information, Systems and Technologies*, pp. 86–95. IARIA, Stuttgart, 2013. URL `http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2013-48&engl=0`. (Cited on pages 7, 42 and 43)

[BBKL13] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann. Pattern-based Runtime Management of Composite Cloud Applications. In F. Desprez, D. Ferguson, E. Hadar, F. Leymann, M. Jarke, M. Helfert, editors, *CLOSER*, pp. 475–482. SciTePress, 2013. (Cited on page 42)

[ben13] Bento Boxes | Cloud Management Software and Cloud Orchestration Software | Flexiant, 2013. URL `http://www.flexiant.com/tag/bento-boxes/`. (Cited on page 26)

[BGPCV12] L. Badger, T. Grance, R. Patt-Corner, J. Voas. Cloud Computing Synopsis and Recommendations. Technical report, U.S. Department of Commerce Gary Locke, Secretary National Institute of Standards and Technology Patrick D. Gallagher, Director, 2012. URL `http://csrc.nist.gov/publications/nistpubs/800-146/sp800-146.pdf`. (Cited on pages 13 and 14)

[BMM12a] E. Brandtzæg, P. Mohagheghi, S. Mosser. Towards a Domain-Specific Language to Deploy Applications in the Clouds. In *CLOUD COMPUTING 2012: The Third International Conference on Cloud Computing, GRIDs and Virtualization*. 2012. (Cited on pages 7, 28 and 29)

[BMM12b] E. Brandtzæg, S. Mosser, P. Mohagheghi. Towards CloudML, a Model-based Approach to Provision Resources in the Clouds. In *Model-Driven Engineering for and on the Cloud workshop (co-located with ECMFA'12)(CloudMDE'12), workshop*, , pp. 18–27. DTU, Copenhaghen, Danemark, 2012. (Cited on pages 7, 28 and 29)

[Bre00] E. A. Brewer. Towards Robust Distributed Systems, 2000. URL `http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf`. (Cited on page 45)

[can] canviz - JavaScript library for drawing Graphviz graphs to a web browser canvas - Google Project Hosting. URL `http://code.google.com/p/canviz/`. (Cited on page 68)

[clo13] AWS CloudFormation, 2013. URL `http://aws.amazon.com/cloudformation/`. (Cited on pages 7, 25 and 26)

[dja13] The Web framework for perfectionists with deadlines | Django, 2005-2013. URL `https://www.djangoproject.com`. (Cited on pages 68 and 70)

[DOM05] W3C Document Object Model, 1997-2005. URL `http://www.w3.org/DOM/`. (Cited on page 66)

[ecl13a] Eclipse Modeling Project, 2013. URL `http://www.eclipse.org/modeling/`. (Cited on page 67)

[ecl13b] Graphviz Eclipse plug-in | Free Development software downloads at SourceForge.net, 2013. URL `http://sourceforge.net/projects/eclipsegraphviz/`. (Cited on page 68)

[ecl13c] XSLT Project - Eclipsepedia, 2013. URL `http://wiki.eclipse.org/XSLT_Project`. (Cited on page 68)

[FL07] J. Farrell, H. Lausen. Semantic Annotations for WSDL and XML Schema. W3crecommendation, World Wide Web Consortium, 2007. URL `http://www.w3.org/TR/sawsdl/`. (Cited on page 39)

[FLR$^+$11] C. Fehling, F. Leymann, R. Retter, D. Schumm, W. Schupeck. An Architectural Pattern Language of Cloud-based Applications. In *Proceedings of the 18th Conference on Pattern Languages of Programs, PLoP 2011*, pp. 1–11. ACM, 2011. URL `http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2011-66&engl=1`. (Cited on pages 7, 43, 44 and 45)

[GL02] S. Gilbert, N. Lynch. Brewer’s Conjecture and the Feasibility of Consistent Available Partition-Tolerant Web Services. In *In ACM SIGACT News*, p. 2002. 2002. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.20.1495&rep=rep1&type=pdf`. (Cited on page 45)

[gra] Graphviz | Graphviz - Graph Visualization Software. URL `http://www.graphviz.org`. (Cited on pages 67 and 70)

[Gro11] O. M. G. Group. UML Specification, Version 2.4.1, 2011. (Cited on page 31)

[HKP$^+$09] P. Hitzler, M. Krötzsch, B. Parsia, P. F. Patel-Schneider, S. Rudolph. OWL 2 Web Ontology Language Primer. W3C Recommendation, World Wide Web Consortium, 2009. URL `http://www.w3.org/TR/owl2-primer/`. (Cited on page 40)

[KBH12] S. Kounev, F. Brosig, N. Huber. Descartes Meta-Model (DMM). Technical report, Karlsruhe Institute of Technology (KIT), 2012. URL `http://www.descartes-research.net/metamodel/`. To be published. (Cited on pages 7 and 30)

[Kec05] C. Kecher. *UML 2.0: Das umfassende Handbuch*. Galileo Computing, 1 edition, 2005. (Cited on pages 31 and 32)

[Ley09] F. Leymann. Cloud Computing: The Next Revolution in IT. In *Photogrammetric Week ‘09*, pp. 3–12. Wichmann Verlag, 2009. (Cited on pages 13 and 14)

[LFM⁺11] F. Leymann, C. Fehling, R. Mietzner, A. Nowak, S. Dustdar. Moving Applications to the Cloud: An Approach based on Application Model Enrichment. *International Journal of Cooperative Information Systems*, 20(3):307–356, 2011. doi:10.1142/S0218843011002250. (Cited on pages 7, 23 and 24)

[LKD⁺02] H. Ludwig, A. Keller, A. Dan, R. P. King, R. Franck. Web service level agreement (WSLA) language specification. Technical report, IBM Corporation, 2002. (Cited on pages 7, 40, 41 and 42)

[Mie10] R. Mietzner. *A method and implementation to define and provision variable composite applications, and its usage in cloud computing.* Ph.D. thesis, University of Stuttgart, 2010. (Cited on pages 7, 22, 23 and 37)

[MUL09] R. Mietzner, T. Unger, F. Leymann. Cafe: A Generic Configurable Customizable Composite Cloud Application Framework. In R. Meersman, T. Dillon, P. Herrero, editors, *CoopIS 2009 (OTM 2009)*, volume 5870 of *Lecture Notes in Computer Science*, pp. 357–364. Springer-Verlag, Berlin, Heidelberg, 2009. (Cited on pages 22 and 44)

[net13a] NetBeans Visual Library, 2013. URL `http://platform.netbeans.org/graph/`. (Cited on page 68)

[net13b] NetBeans xml: XSL Support, 2013. URL `https://xml.netbeans.org/xsl/`. (Cited on page 68)

[NLPH12] D. K. Nguyen, F. Lelli, M. P. Papazoglou, W.-J. van den Heuvel. Blueprinting Approach in Support of Cloud Computing. *Future Internet*, 4(1):322–346, 2012. (Cited on pages 7, 8, 15, 16, 17, 56 and 81)

[ope13a] OpenNebula - Flexible Enterprise Cloud Made Simple, 2002-2013. URL `http://opennebula.org`. (Cited on page 27)

[ope13b] Software » OpenStack Open Source Cloud Computing Software, 2013. URL `http://www.openstack.org/software/`. (Cited on pages 7, 26 and 27)

[PH11] M. Papazoglou, W. van den Heuvel. Blueprinting the Cloud. *Internet Computing, IEEE*, 15(6):74–79, 2011. doi:10.1109/MIC.2011.147. (Cited on page 15)

[pyd] pydot - Python interface to Graphviz's Dot language. - Google Project Hosting. URL `http://code.google.com/p/pydot/`. (Cited on pages 68 and 70)

[SABL13] S. Strauch, V. Andrikopoulos, T. Bachmann, F. Leymann. Migrating Application Data to the Cloud Using Cloud Data Patterns. In *Proceedings of the 3rd International Conference on Cloud Computing and Service*

*Science, CLOSER 2013, 8-10 May 2013, Aachen, Germany*, pp. 0–11. SciTePress, 2013. (Cited on page 45)

[Spe10] S. Speiser. Semantic Annotations for WS-Policy. In *Web Services (ICWS), 2010 IEEE International Conference on*, pp. 449–456. 2010. doi:10.1109/ICWS.2010.15. (Cited on page 40)

[tos13] Topology and Orchestration Specification for Cloud Applications Version 1.0. OASIS Committee Specification 01, 2013. URL `http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html`. (Cited on pages 7, 18, 19, 20 and 21)

[USD11] Unified Service Description Language 3.0 (USDL): Overview. Technical report, SAP Research, 2011. URL `http://www.internet-of-services.com/fileadmin/IOS/user_upload/pdf/USDL-3.0-M5-overview.pdf`. (Cited on pages 7, 32 and 33)

[WCL05] S. Weerawarana, F. Curbera, F. Leymann. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Prentice Hall PTR, 2005. (Cited on pages 13, 14, 15, 38, 39 and 61)

[web] WebDot Home Page. URL `http://www.graphviz.org/webdot/index.html`. (Cited on page 68)

[win13] Winery, 2013. URL `http://winery.opentosca.org`. (Cited on pages 80 and 81)

[wsa04] Web Services Architecture. W3C Working Group Note, 2004. URL `http://www.w3.org/TR/ws-arch/`. (Cited on pages 13 and 14)

[WSD07] Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. W3C Recommendation, 2007. URL `http://www.w3.org/TR/wsdl20/`. (Cited on page 14)

[WSP07] Web Services Policy Framework (WSPolicy). W3C Recommendation, 2007. URL `http://www.w3.org/TR/ws-policy`. (Cited on pages 38 and 39)

[Xiu13] M. Xiu. *Decision support for different migration types of applications to the Cloud*. Diplomarbeit, University of Stuttgart, 2013. (Cited on page 73)

[XSL07] XSL Transformations (XSLT) Version 2.0. W3C Recommendation, 2007. URL `http://www.w3.org/TR/xslt20/`. (Cited on page 66)

All links were last followed on November 26th, 2013.