Annex to D3.1

DOML Specification

Editor(s):	Adrian Noguero
Responsible Partner:	Go4It, Polimi
Status-Version:	V0.1
Date:	20.12.2021
Distribution level (CO, PU):	Public

Project Number:	101000162
Project Title:	PIACERE

Title of Deliverable:	Annex to D1.3
Due Date of Delivery to the EC	30.11.2021

Workpackage responsible for the Deliverable:	WP3 - Plan and create Infrastructure as Code
Editor(s):	Go4It
Contributor(s):	Go4It, Polimi
Reviewer(s):	Alfonso de la Fuente (Prodevelop)
Approved by:	All Partners
Recommended/mandatory readers:	WP4, WP5, WP6, WP7

Abstract:	This annex is accompanying document to Deliverable D3.1 - PIACERE Abstractions, DOML and DOML-E - v1. It includes the detailed specification of the DOML concepts. It will be updated periodically to account for the development of the language.
Keyword List:	DOML, Modelling abstractions
Licensing information:	This work is licensed under Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) http://creativecommons.org/licenses/by-sa/3.0/
Disclaimer	This document reflects only the author's views and neither Agency nor the Commission are responsible for any use that may be made of the information contained therein

Document Description

Version	Date	Modifications Introduced	
		Modification Reason	Modified by
v0.01	08.10.2021	First draft version	GO4IT
v0.02	10.12.2021	Completed the specification v1.0 of DOML. Added the concrete layer and all DOML-E mechanisms. Completed the properties and updated examples	GO4IT
V0.1	20.12.2021	Syntax definition added, revision of the whole content	Polimi

Table of contents

List of figures

Terms and abbreviations

CSP	Cloud Service Provider
DevOps	Development and Operation
DoA	Description of Action
EC	European Commission
GA	Grant Agreement to the project
IaC	Infrastructure as Code
IEP	IaC execution platform
IOP	IaC Optimization
KPI	Key Performance Indicator
SW	Software

Executive Summary

This document contains the main description of the DOML language specification, as well as its extension mechanisms (DOML-E). The goal of the document is to serve as cornerstone for the implementation of DOML based solutions in PIACERE, ranging from the IDE to the optimization algorithms.

1 Description of DOML

DOML specifies a common language for addressing the definition, deployment and operation of complex cloud-based applications inside the PIACERE framework. DOML is intended to be used by users with different degrees of expertise, therefore, it has been conceived to be easy to use by non-expert users, but also expressive enough to allow expert users to get the most out of it.

DOML is a declarative language, thus, each of the layers describe what the application and infrastructure should look like after all the deploying is done. However, DOML allows the user to integrate imperative scripts, to actually describe some specific configurations whenever needed. The main goal of DOML is to serve as a bridge to the many IaC languages that currently exist (e.g. Terraform, TOSCA, Ansible…), providing a degree of expressiveness that allows the PIACERE framework to generate IaC code in those mentioned formats easily.

In addition, DOML is intended to be used with the PIACERE optimization mechanisms. To achieve this DOML allows the user to define different application deployment configurations, as well as different infrastructure configurations, and it includes a specific layer to define optimization objevtices and constraints.

Finally, DOML in envisaged as an evolving entity capable of coping with the constant advancements in the cloud computing state-of-the-art. As such, DOML includes extension mechanisms built inside that allow the user and the tools using DOML to create new concepts for any of the layers in DOML, as well as extending existing ones with new properties and attributes. This extension mechanisms are collectively called DOML-E.

This specification is intended to be used as a reference for the implementation of all DOML related tools and guides.

1.1 DOML Layers

The DOML language specification is split into several packages, referred to as “layers”, which incrementally enrich the description of the cloud-based applications that will be managed inside PIACERE. Each layer provides a unique point of view of the applications; yet, all the layers build up for a comprehensive application description.

The **Commons Layer** contains the main abstract application agnostic concepts that are shared among different layers. The DOML extension mechanisms (DOML-E) are also addressed in this layer by setting up the basic elements that will allow creating new concepts and properties in the top layers.

The **Application Layer** contains the information to describe the components and building blocks that compose the applications, as well as the functional requirements of each of them in terms of software interfaces and APIs. Finally, this layer describes how the application is deployed into the different infrastructure components.

The **Infrastructure Layer** defines the abstract infrastructure elements that will be used to deploy the application components. Concepts in this layer will include information that is relevant to meet the requirements of the applications. However, most of the concepts in this layer will require a concretization, or in other words, a more concrete instance they will be mapped on. For example, a virtual machine in this layer must be mapped to a concrete virtual mechine instance, be it a VM from AWS or a specific VM deployed by the user.

The **Concrete Layer** provides the tools to concretize the infrastructure elements in the Infrastructure Layer and map them onto specific infrastructure instances either provided by cloud runtime providers, such as AWS or Google Cloud, or provided by the users.

The **Optimization Layer** defines all the information required for the optimizers to locate the best configurations for the cloud applications described in the DOML, as well as means to capture the optimization solutions.

2 Commons Layer

The following diagram shows the main elements of the Commons Layer in DOML:

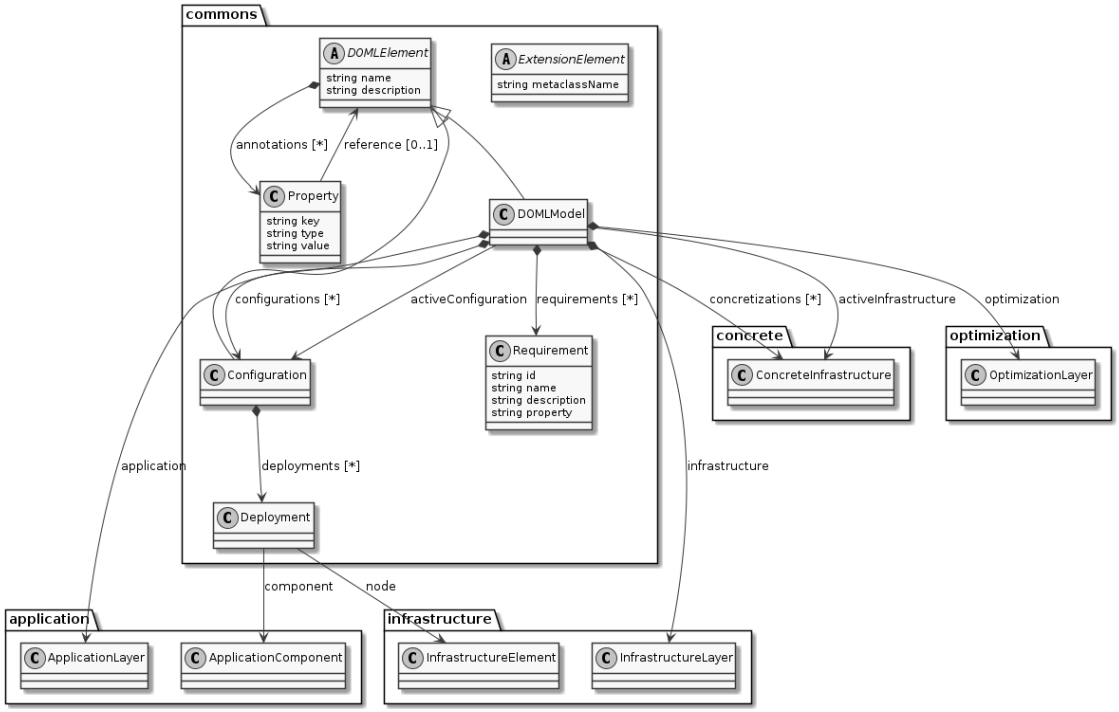

Figure 1. Commons Layer diagram (excluding Requirement subclasses)

2.1 DOMLElement Class (abstract)

A DOMLElement represents any element inside the DOML language and it is intended to be the top meta-element of the DOML.

Attributes

name: String [1]	An identifier for this DOML Element
description: String [0..1]	An optional textual description of the Element. Used for documenting the element or similar purposes.

Associations

annotations: Property [0..*]	A set of properties used to modify the semantics of this particular element. These properties will add to the final semantics of the element, refining the DOML element to which they are applied. These properties will serve as the main extension mechanisms for DOML.
contributesTo: Requirement [0..*]	The set of requirements this DOMLElement contributes to achieving. This association is derived from the predicatesOn association of the Requirement class.

Constraints

* All properties added to a DOML element must have different keys.

Usage

DOMLElement is the common parent of all elements in DOML but the Property class. It is also the enabler for DOML-E extensions through the use of Property elements.

2.2 Property Class

A Property represents an additional information added to any DOML Element to further refine their meaning or semantics.

Attributes

key: String [1]	An identifier for this Property
type: String [0..1]	An optional name defining the data type for this property
value: String [0..1]	An optional textual information associated with this Property instance.

Associations

reference: DOMLElement [0..1]	An optional link to another DOML Element relevant for this property.

Constraints

* All properties owned by a DOML element must have different keys.

Usage

Instances of the Property class are used to add information to DOML elements that cannot be described using that element's attributes and associations. Properties can be used as any or both attributes and associations to refine any DOML element.

2.3 DOMLModel Class

A DOMLModel represents the design and development space for a cloud application or set of applications. The DOML model provides access to the different points of view of this space through the use of the model layers.

Superclass

DOMLElement

Associations

application: ApplicationLayer [0..1]	A reference to the Application Layer instance associated with this model.
infrastructure: InfrastructureLayer [0..1]	A reference to the Concrete Infrastructure Layer instance associated with this model.

concretizations: ConcreteInfrastructure [0..*]	A list of concrete infrastructures that map on to abstract infrastructure layer elements.
activeInfrastructure: ConcreteInfrastructure [0..1]	The ConcreteInfrastructure considered active for the current DOML specification
optimization:OptimizationLayer [0..1]	A reference to the Optimization Layer instance associated with this model.
configurations: Configuration [0..*]	All possible configurations of the current DOML specification
activeConfiguration: Configuration [0..1]	The Configuration instance considered as active
requirements: Requirement [0..*]	The set of requirements that are applicable to the current DOML specification.

Usage

A DOML model is intended to be used as the container for all the DOML layers defined in a particular design space. Each of those layers will provide a different point of view of the design space of the application. This element should be used as the root element of any DOML model.

2.4 Configuration Class

A Configuration describes how an application is intended to be deployed on top of the cloud infrastructure, and what credentials, parameters, etc. will apply to each of the application and/or infrastructure elements.

Superclass

DOMLElement

Associations

deployments: [0..*]	Deployment	A set of Deployment instances describing each of the links between an application component and a node of the infrastructure.

Constraints

* There must not be two Deployment instances inside a Configuration element with the same source and target elements.

Usage

A configuration must fully describe how an application will operate on top of a particular infrastructure. Since the parameters associated to each DOML element, whether it is an application component or an infrastructure node, will differ, the configuration element will use the Property list to include them, using the reference Association of the Property to describe which model element that particular parameter affects to.

2.5 Deployment Class

A Deployment element describes an association between an application component (e.g. a web application) and the infrastructure element that will host it (e.g. a Virtual Machine).

Associations

source: ApplicationComponent [1]	The application component that will be deployed.
target: InfrastructureElement [1]	The infrastructure element that will host/support the application component.

Usage

The deployment is designed to establish 1 to 1 relationships between application and infrastructure elements.

2.6 ExtensionElement Class (abstract)

A ExtensionElement abstract metaclass is used as the common meta-type for all the classes that are part of DOML-E extension mechanisms.

Attributes

metaclassName: String [1]	The name of the metaclass that will be added to DOML by using the extension class instance.

Usage

The extension element class must never be instantiated nor subclassed. Instead all extension metaclasses in DOML (i.e. ExtApplicationComponent, ExtInfrastructureElement, ExtConcreteElement and ExtOptimizationObjective) extend this metaclass, in addition to other extensions.

2.7 Requirement Class

A Requirement represents an objective to be achieved by the current DOML specification. Requirements, whether they are functional, non-functional or optimization objectives, must be described in plain text and also annotations can be used to further qualify it, if needed.

Attributes

identifier: String [1]	A unique identifier for this requirement.
title: String [0..1]	An optional meaningful title for the requirement.
description: String [0..1]	A text further specifying the requirement.
property: String [0..1]	The property of the DOMLElement instances this requirement predicates on.

Associations

predicatesOn: [0..*]	DOMLElement	A reference to the set of DOMLElement instances this requirement predicates on.

Constraints

* All requirements in a DOML model must have different identifiers.

Usage

Requirements are used to model objectives and restrictions the current DOML design must meet. These objectives should be as formal as possible; however, they can also be used in a less formal way using the textual attributes. The way to define them in a formal way is by using the “property” and the “predicatesOn” members. The Requirement class is also the parent of all formal requirements defined in DOML. The following diagram shows the requirements section of the commons layer in DOML.

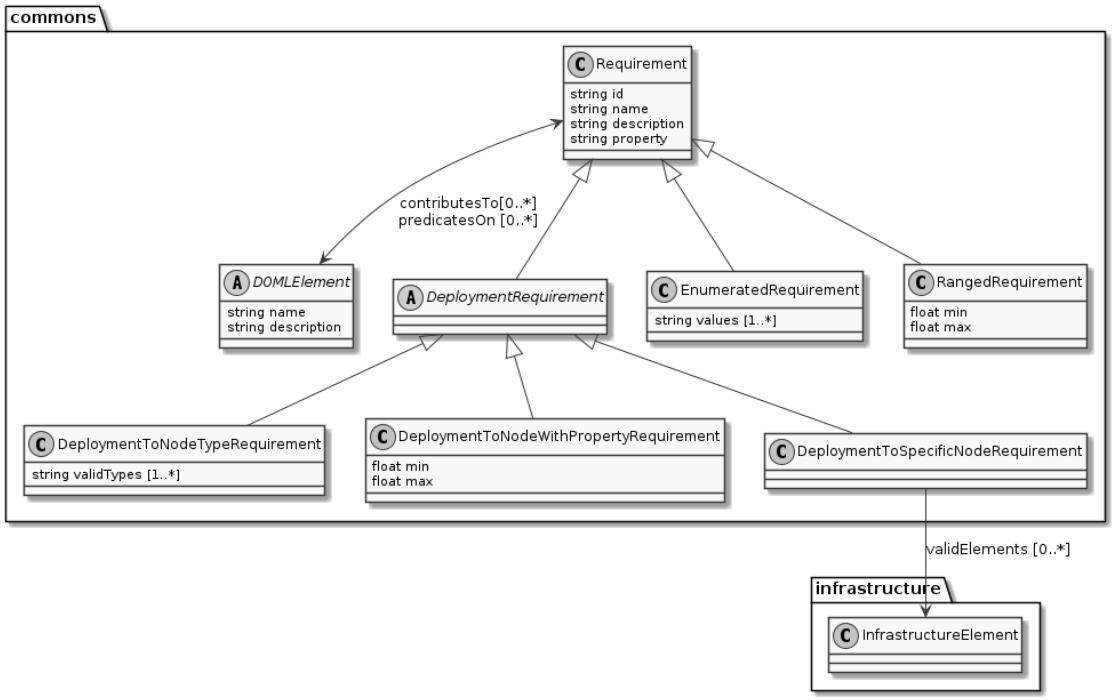

Figure 2. Commons Layer Requirements diagram

2.8 RangedRequirement Class

A RangedRequirement is a formal requirement instance which establishes a range of valid values to a property in a set of DOMLElements.

Superclass

Requirement

Attributes

min: Float [0..1]	The minimum value of the property.
max: Float [0..1]	The maximum value of the property.

Constraints

* The property attribute of a RangedRequirement must always be set.

* The predicatesOn association must always be linked to at least one DOMLElement for a RangedRequirement

* At least the max or the min attributes of a RangedRequirement mus be set.

* A ranged requirement can only be applied to numeric properties.

Usage

A ranged requirement should be used to establish limits to the numeric properties that need them.

2.9 EnumeratedRequirement Class

A EnumeratedRequirement describes a formal requirement that restricts the number of valid values a property of a certain DOML element may take.

Superclass

Requirement

Attributes

values: String [1..*]	The set of values that are valid for the property referred by this requirement.

Constraints

* The property attribute of a EnumeratedRequirement must always be set.

* The predicatesOn association must always be linked to at least one DOMLElement for a EnumeratedRequirement

* At least one value must be set in the values attribute.

Usage

A enumerated requirement is used to set a list of valid values for a particular property.

2.10 DeploymentRequirement Class (abstract)

A DeploymentRequirement class describes a restriction to be applied to the definition of configurations in the current DOML.

Superclass

Requirement

Constraints

* The predicatesOn association must always be linked to at least one DOMLElement for a DeploymentRequirement and they must all be ApplicationComponent instances.

Usage

A DeploymentRequirement is used as the common parent class to all deployment related formal requirements in DOML.

2.11 DeploymentToNodeTypeRequirement Class

A DeploymentToNodeTypeRequirement describes a formal requirement that restricts types of infrastructure elements an application component can be deployed to.

Superclass

DeploymentRequirement

Attributes

validTypes: String [1..*]	The set of valid meta-types the application components this requirement predicates on can be deployed to.

Constraints

* At least one value must be set in the validTypes attribute.

* Values in validTypes must all be valid names of meta-classes in DOML infrastructure layer that extend the InfrastructureElement class.

Usage

A requirement of this kind is used to make an application component or a set of components deployable only into certain types of infrastructure elements (for example, make a software package only deployable to physical nodes).

2.12 DeploymentToNodeWithPropertyRequirement Class

A DeploymentToNodeWithPropertyRequirement describes a formal requirement that restricts the infrastructure elements an application component can be deployed to according to the value of a property.

Superclass

DeploymentRequirement

Attributes

min: Float [0..1]	The minimum value of the property.
max: Float [0..1]	The maximum value of the property.
values: String [0..*]	The set of values that are valid for the property referred by this requirement.

Constraints

* The property attribute of a DeploymentToNodeWithPropertyRequirement must always be set.

* At least the max, the min or the values attributes of a requirement of this kind must be set.

* If values is not empty, then min and max cannot be set.

* If min and/or max are set, then values has to be empty.

Usage

A DeploymentToNodeWithPropertyRequirement is used to restrict the valid infrastructure nodes an application component can be deployed to according to the value of a property of the target infrastructure element (for example, a software interface can only be attached to a network interface with a minimum speed of 1Gbps, or a dbms component can only be deployed to a node with location equal to Europe).

2.13 DeploymentToSpecificNodeRequirement Class

A DeploymentToSpecificNodeRequirement describes a formal requirement that restricts the set of valid infrastructure element an application component can be deployed to a specific list.

Superclass

DeploymentRequirement

Associations

validElements: InfrastructureElement [1..*]	The set of elements the application component referred to by this requirement can be deployed to.

Usage

A DeploymentToSpecificNodeRequirement is used provide a valid set of infrastructure elements to be used to deploy an application component or a set of application components.

3 Application Layer

The following diagram shows the main elements of the Application Layer in DOML:

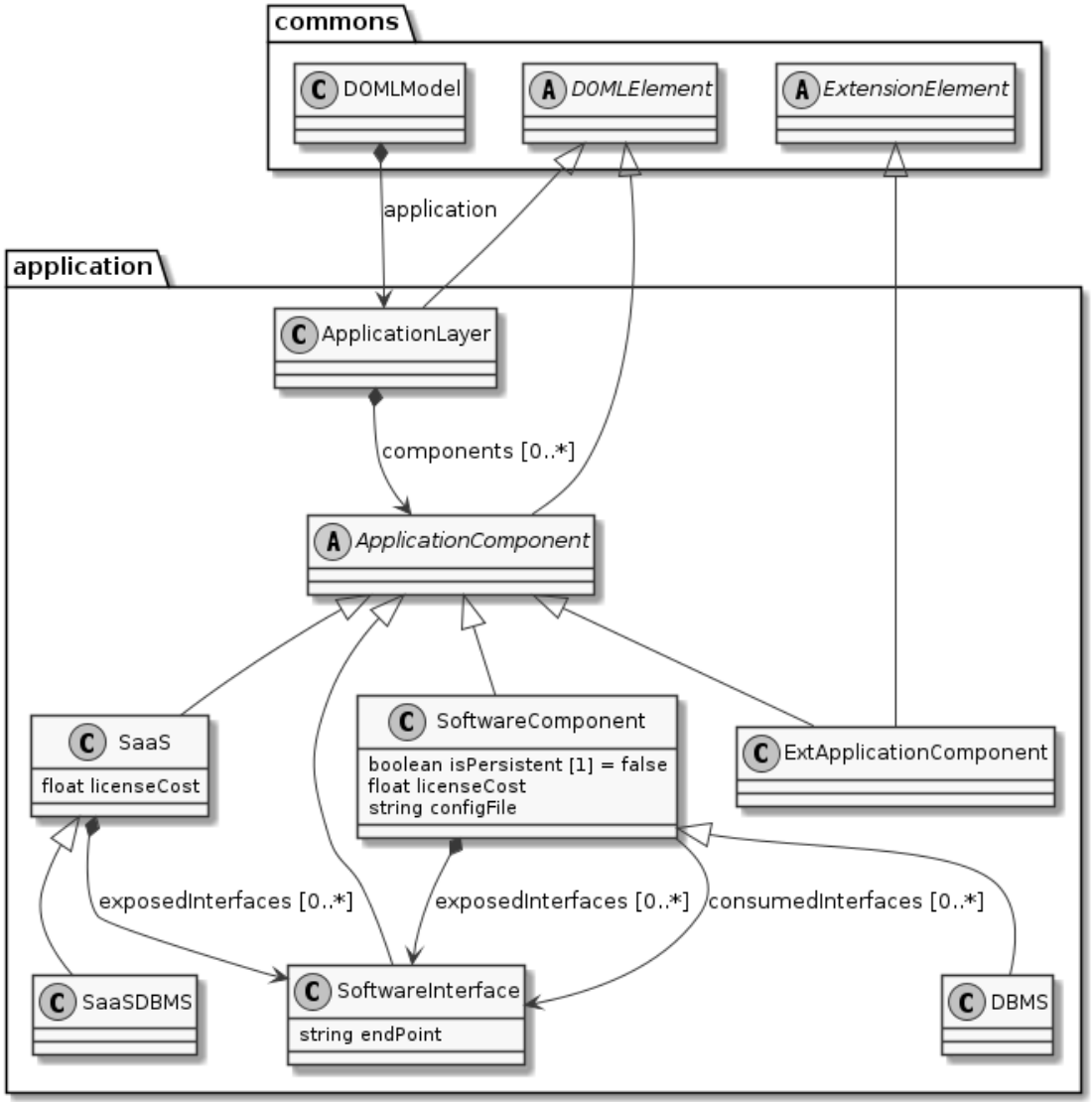

Figure 3 Application Layer diagram

3.1 ApplicationLayer Class

The Application class represent the container for all the components of the application in a DOML design. It is the representation of the Application Layer, and all the functional elements of the cloud application to be deployed must be defined as application components inside it.

Superclass

DOMLElement

Associations

components: ApplicationComponent [0..*]	A containment reference to all the application components that will be part of the current application layer.

Usage

The Application is designed to be a container for ApplicationComponent instances.

3.2 ApplicationComponent Class (abstract)

The ApplicationComponent describes anything meaningful to the application being deployed in DOML from the functional perspective (e.g. software components, services or APIs). Each application component is susceptible of being deployed to an infrastructure element in the infrastructure model.

Superclass

DOMLElement

Usage

The ApplicationComponent class is intended to be the common parent class for all elements in the application layer. Any common properties must always be specified on this class.

3.3 SoftwarePackage Class

The SoftwarePackage class describes any of the functional software components that conform an application in DOML. A software component may use or provide software interfaces, creating, this way, links among components, APIs and other functional elements in the application layer.

Superclass

ApplicationComponent

Attributes

isPersistent: Boolean [1]	A flag to indicate whether this component persists any information/state during operation. By default the value of this property is *false*.
licenseCost: Float [0..1]	An optional license cost (in Euro) associated to this software component.
configFile: String [0..1]	The path to the installation and configuration script (e.g. Ansible, Terraform, Shell...) for this software component. This information will be used by IaC generators.

Associations

exposedInterfaces: SoftwareInterface [0..*]	A set of software interfaces provided by this component for other software components to use.
consumedInterfaces: SoftwareInterface [0..*]	The set of software interfaces required by this component to fulfil its role.

Constraints

* Consumed interfaces must always refer to software interfaces exposed by other components or SaaS instances.

Usage

The SoftwarePackage class is intended to describe the main functional components or an application (e.g. web server, a REST API, etc.). It is important to note that software packages should be part of the components to be deployed in and are susceptible of having requirements attached to them.

3.4 SaaS Class

The SaaS class models an API that is external to our application, but relevant for functional purposes.

Superclass

ApplicationComponent

Attributes

licenseCost: Float [0..1]	An optional license cost (in Euro) associated to this SaaS.

Associations

exposedInterfaces: SoftwareInterface [0..*]	A set of software interfaces provided by this component for other software components to use.

Usage

The SaaS class is intended to describe APIs that are external to the current application, but are used by the software components inside it. SaaS components must not have requirements associated to them, the user has no control over them. SaaS instances may, however, define properties related to expected performance, response time, etc. if those are relevant for the current DOML model.

3.5 SoftwareInterface Class

The SoftwareInterface class models a software interface (e.g. a REST API, a TCP/IP connection, etc.) that connects two different application components in the application model.

Superclass

ApplicationComponent

Attributes

endPoint: String	The IP address / hostname / URL through which the service is accessed

Constraints

* A software interface must always be provided by one application component and used by at least one application component in the DOML model.

Usage

The SoftwareInterface class is intended to describe a connector between two different application components.

3.6 DBMS Class

The DBMS describes a software component that includes a Data Base Management System.

Superclass

SoftwarePackage

Constraints

* The isPersistent attribute of a DBMS component must always be set to *true*.

Usage

The DBMS is just a convenient subclass of the more generic SoftwarePackage class to model specifically DBMS.

3.7 SaaSDBMS Class

The SaaSDBMS describes an external API that will provide the DataBase Management System Functionality.

Superclass

SaaS

Usage

The SaaSDBMS class is just a convenient subclass of the more generic SaaS class to model specifically a DBMS providing SaaS.

3.8 ExtApplicationComponent Class

The ExtApplicationComponent describes an instance of a new application layer concept. This class is part of DOML-E extension mechanisms.

Superclasses

ApplicationComponent, ExtensionElement

Usage

The ExtApplicationComponent class is should be used to create instances of concepts and metaclasses not currently available in DOML.

4 Infrastructure Layer

The infrastructure layer describes the abstract infrastructure elements that will be supporting the execution of the application described in the ApplicationLayer. It is important to note that this abstract representation of the infrastructure is intended to be reused, mapping the elements on this layer to concrete instances in the infrastructure (e.g. an abstract virtual machine described in this layer will be mapped to a concrete VM instance, provided by a specific runtime provider, such as AWS or GoogleCloud).

The following diagram shows the main elements of the Infrastructure Layer in DOML related to infrastructure nodes:

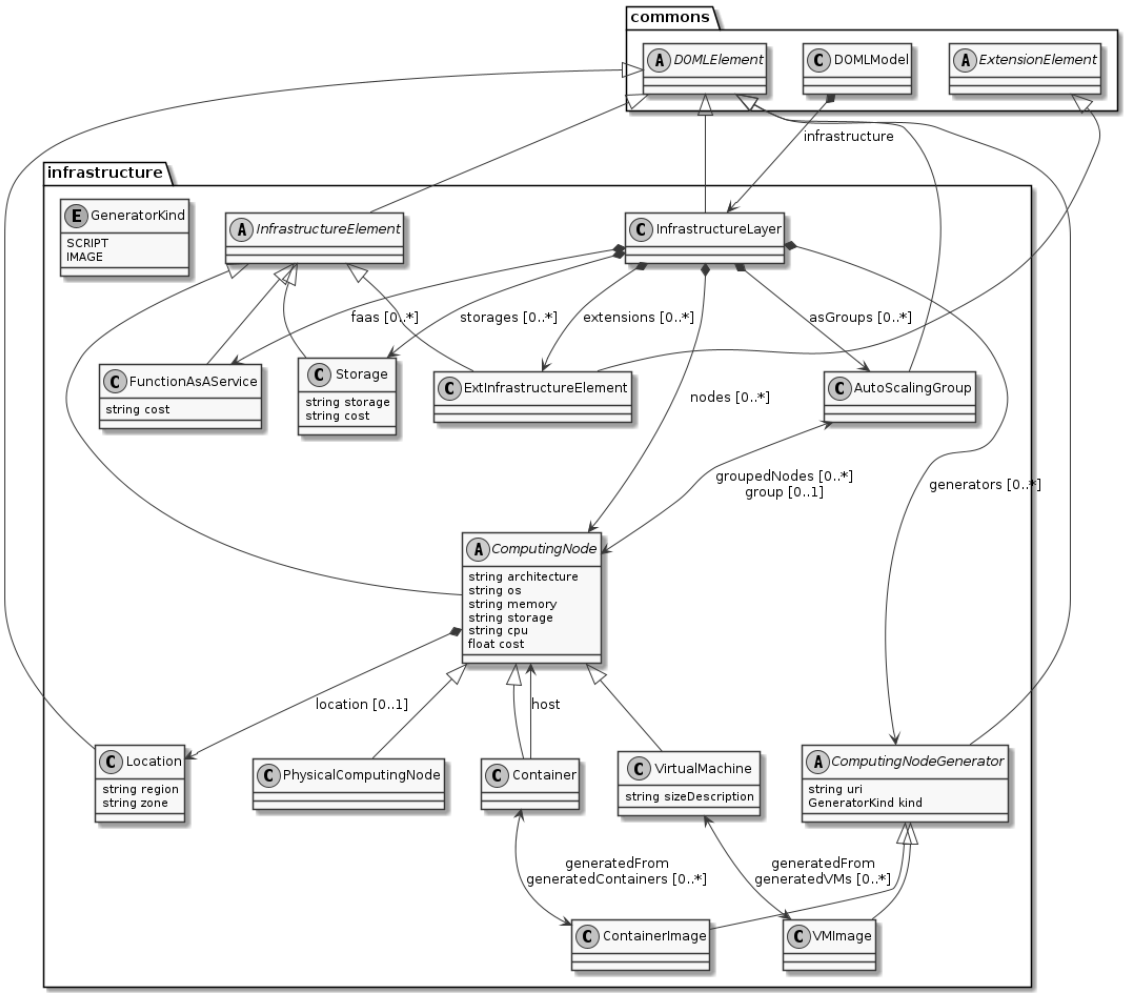

Figure 4. Infrastructure Layer diagram showing infrastructure nodes

4.1 InfrastructureLayer Class

The InfrastructureLayer class is the container for the catalog of infrastructure elements that will be available to the current DOML model.

Superclass

DOMLElement

Associations

providers: RuntimeProvider [0..*]	The list of runtime providers available in the catalogue
nodes: ComputingNode [0..*]	The list of independent computing nodes (not attached to a provider) available in the catalogue
generators: ComputingNodeGenerator [0..*]	The list of virtual machine and container images available in the catalogue
asGroups: AutoScalingGroup [0..*]	The list of independent auto scaling groups (not attached to a runtime provider) available in the catalogue
networks: Network [0..*]	The list of independent networks (not attached to a runtime provider) available in the catalogue
firewalls: Firewall [0..*]	The list of independent firewalls available in the catalogue
orchestrator: RuntimeOrchestrationEnvironment [0..1]	An optional orchestration environment available in the catalogue.
storages: Storage [0..*]	The list of storage resources that will be part of this abstract infrastructure model
faas: FunctionAsAService [0..*]	The list of faas services part of this DOML model infrastructure.

Usage

The InfrastructureLayer is a container element, used as the catalog of infrastructure elements available to deploy the final cloud application using DOML.

4.2 InfrastructureElement Class (abstract)

The InfrastructureElement class represents all infrastructure elements that can have an application component deployed to them.

Superclass

DOMLElement

Usage

The InfrastructureElement is intended to be used as the parent for more concrete elements of the infrastructure model.

4.3 ComputingNode Class (abstract)

The ComputingNode class represents any element that can be used for computing, from a dedicated host to an IoT node.

Superclass

InfrastructureElement

Attributes

architecture: String [0..1]	A string describing the internal architecture of the computing node (e.g. x86, x64, etc.).
os: String [0..1]	A string describing the operating system of this node (e.g. Windows 10, Ubuntu 20.04, etc.).
memory: String [0..1]	A string describing the total memory of this node (e.g. 4GB).
storage: String [0..1]	A string describing the total storage available in this node (e.g. 10TB).
cpu: String [0..1]	A string describing the CPU of the computing node.
cost: Float [0..1]	An optional cost value (in Euro).

Associations

group: AutoScalingGroup [0..1]	Derived property. A link to the group that owns this computing node.
ifaces: NetworkInterface [0..*]	The network interfaces owned by this computing node.
location: Location [0..1]	An optional location for this infrastructure element.

Usage

The CopmutingNode class is intended to be the common parent for all the infrastructure elements capable of executing code.

4.4 PhysicalComputingNode Class

The PhysicalComputingNode class represents a dedicated physical server.

Superclass

ComputingNode

4.5 ComputingNodeGenerator Class (abstract)

The ComputingNodeGenerator class represents all infrastructure elements that describe a virtual computing node.

Superclass

DOMLModel

Usage

The ComputingNodeGenerator is intended to be used as the commong parent for all elements defining the characteristics of virtual computing nodes. Often these generators rely on a file which defines them.

Usage

The PhysicalComputingNode is used to describe physical computing nodes available for the owner of a cloud application that are going to be used as part of the cloud deployment.

4.6 VirtualMachine Class

The VirtuualMachine class represents a virtual computing node running a on top of a supervisor software.

Superclass

ComputingNode

Attributes

sizeDescription: String [0..1]	An optional string describing the size of the VM.

Associations

generatedFrom: [0..1]	VMImage	The image used to generate this virtual machine.
location: Location [0..1]		An optional Location object to represent where the VM should be located

Usage

The VirtualMachine is used to describe virtual computing nodes running on a supervisor software. In order to be automatically configurable, the virtual machine must define the image that will generate it.

4.7 Location Class

The Location class represents the place where a computing node should be.

Superclass

DOMLModel

Attributes

region: String [1]	A string describing the region for this location.
zone: String [0..1]	An optional attribute to refine the location if the region is not precise enough.

Usage

The Location is intended to describe the location of infrastructure elements, more concretely virtual machines and physical machines.

4.8 Container Class

The Container class represents a virtual computing node running on top of another computing node.

Superclass

ComputingNode

Associations

generatedFrom: ContainerImage [0..1]	The image used to generate this container.
host: ComputingNode [1]	The computing node that will be the host of this container.

Usage

The Container is used to describe virtual computing nodes, such as Docker containers.

4.9 GeneratorKind Enum

The GeneratorKind enumeration describes the different computing node generation kinds.

Values

SCRIPT, IMAGE

4.10 ComputingNodeGenerator Class (abstract)

The ComputingNodeGenerator class represents all infrastructure elements that describe a virtual computing node.

Superclass

DOMLModel

Attributes

uri: String [0..1]	A URI to the file containing this computing node generation image or file.
kind: GeneratorKind [0..1]	An optional attribute to define whether this generator uses a node image (i.e. a VM image) or a file (i.e. docker file) to generate the computing node.

Usage

The ComputingNodeGenerator is intended to be used as the commong parent for all elements defining the characteristics of virtual computing nodes. Often these generators rely on a file which defines them.

4.11 VMImage Class

The VMImage class represents the image (i.e. the set of attributes and parameters) that can be used to generatae a virtual machine.

Superclass

ComputingNodeGenerator

Associations

generatedVMs: VirtualMachine [0..*]	The set of virtual machines that will be created using this image.

Usage

The VMImage is used for generation purposes, allowing the ICG to generate the scripts to generate VMs from a VM defining image.

4.12 ContainerImage Class

The ContainerImage class represents the image (i.e. the set of attributes and parameters) that can be used to generatae a container.

Superclass

ComputingNodeGenerator

Attributes

generatedContainers: Container [0..*]	The set of containers that have been generated using this conatiner image.

Usage

The ContainerImage is used for generation purposes, allowing the ICG to generate the scripts to generate containers from a the container defining image.

4.13 AutoScalingGroup Class

The AutoScalingGroup class represents an aggrupation of computing nodes with the auto scaling property.

Superclass

DOMLElement

Associations

supportedBy: RuntimeProvider [0..1]	The runtime provider that supports the group.
groupedNodes: ComputingNode [0..*]	The computing nodes

Usage

The AutoScalingGroup class allows to configure a set of nodes to act as a group.

4.14 Storage Class

The Storage class represents an infrastructure node that aims at incremening the overall storage available to the computing nodes in the infrastructure.

Superclass

InfrastructureElement

Attributes

storage: Float [0..1]	The size of the storage in GB
cost: Float [0..1]	The cost of this storage service in Euro

Associations

ifaces: NetworkInterface [0..1]	The network interfaces connected to this infrastructure node.

Usage

The Storage class allows to define a node that increments the storage of the application. The node cannot support any other functionality other than providing storage space.

4.15 FunctionAsAService Class

The FunctionAsAService class represents an pure software infrastructure component capable of executing functional algorithms through an API.

Superclass

InfrastructureElement

Attributes

cost: Float [0..1]	The cost of this service in Euro

Associations

ifaces: NetworkInterface [0..1]	The network interfaces connected to this infrastructure node.

Usage

The FunctionAsAService class allows to define a service used to execute pure business logic/algorithms on a set of input data.

4.16 ExtInfrastructureElement Class (abstract)

The ExtInfrastructureElement class is just used to represent an instance of a new infrastructure element concept that the user wants to add to DOML. This class is part of the DOML-E extension mechanisms.

Superclass

InfrastructureElement, ExtensionElement

Usage

The ExtInfrastructureElement class is should be used to creat instances of concepts and metaclasses not currently available in DOML.

4.17 Network Class

The Network class represents the means to interconnect computing nodes. The concepts related to the network, as well as associations among them, is depicted in the following diagram:

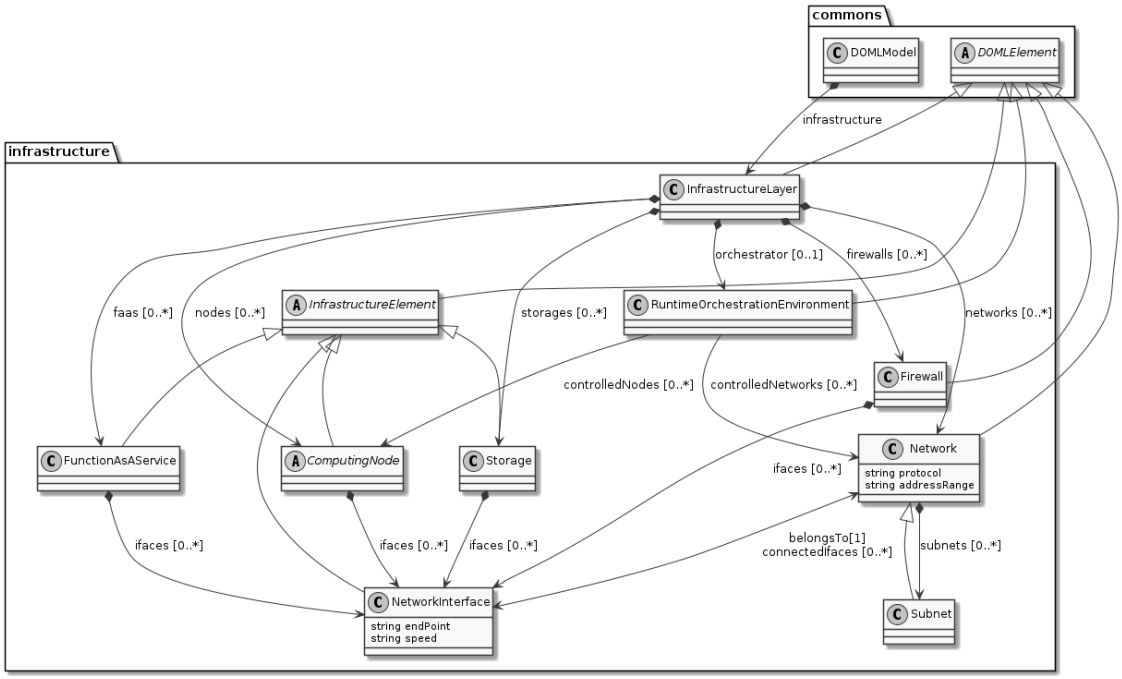

Figure 5. Infrastructure Layer diagram showing network related concepts

Superclass

DOMLElement

Attributes

protocol: String [0..1]	A string defining the protocol of the current network (e.g. TCP/IP).
addressRange: String [0..1]	A string describing the valid addresses in this particular network.

Associations

connectedIfaces: NetworkInterface [0..*]	The set of network interfaces connected to this network. This is a derived association.
subnets: Subnet [0..*]	The set of sub networks of the current one.

Usage

The Network describes a means to interconnect computing nodes as part of a cloud architecture.

4.18 Subnet Class

The Subnet class models a partition of a main network. A subnet is also a network.

Superclass

Network

Usage

The Subnet is used to describe partitions of main networks.

4.19 NetworkInterface Class

The NetworkInterface class represents the means to interconnect computing nodes.

Superclass

InfrastructureElement

Attributes

endPoint: String [0..1]	A string defining the endpoint (i.e. address) of this network interface inside the network.
speed: String [0..1]	A string defining the maximum speed of this network interface.

Associations

belongsTo: Network [1]	A reference to the network associated to this interface.

Usage

The Network describes a means to interconnect computing nodes as part of a cloud architecture.

4.20 Firewall Class

The Firewall class represents the means to interconnect computing nodes.

Superclass

DOMLElement

Associations

ifaces: NetworkInterface [0..*]	The interfaces of this Firewall element.

Constraints

* Iterfaces connected to a Firewall must be associated to different networks

Usage

The Firewall describes a device used to secure the access to a specific network.

4.21 RuntimeOrchestrationEnvironment Class

The RuntimeOrchestrationEnvironment class represents the environment that will be orchestrating the provisioning of computing nodes and the creation of the networks between these nodes.

Superclass

DOMLElement

Associations

controlledNetworks: [0..*]	Network	The networks controlled by the orchestrator.
controlledNodes: ComputingNode [0..*]		The computing nodes controlled by the orchestrator.

Usage

The RuntimeOrchestrationEnvironment class may be used in the case PIACERE will support multiple orchestration environments. In this case, in fact, it will be possible to identify each environment in a DOML model and to allow users to select a specific one.

5 Concrete Layer

The following diagram shows the main elements of the Concrete Layer in DOML:

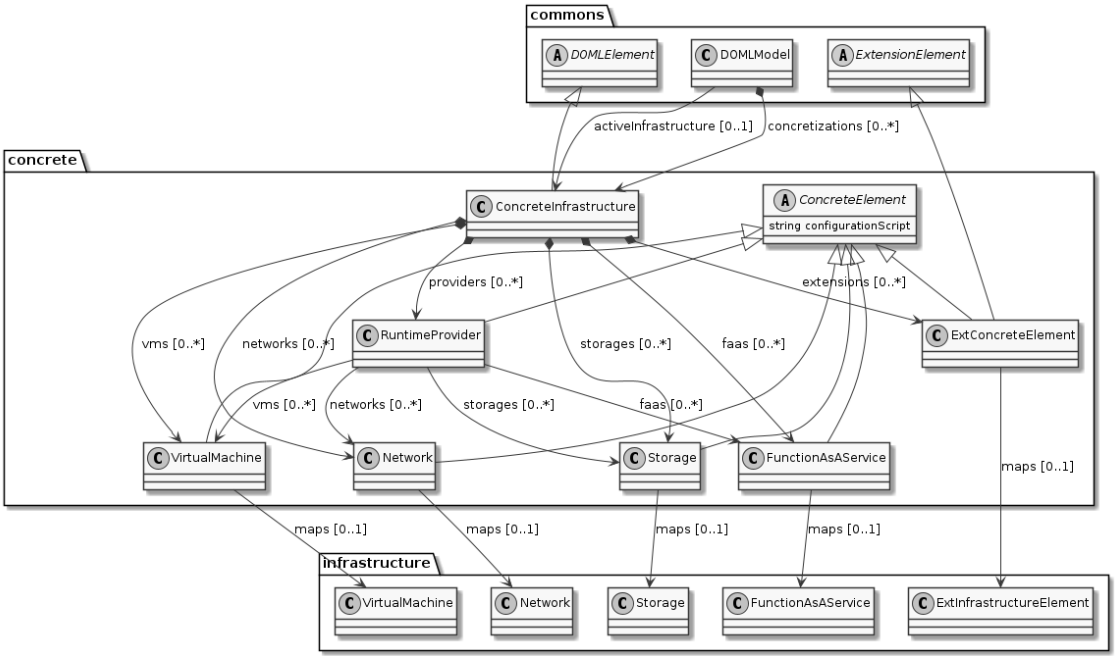

Figure 6. Infrastructure Layer diagram

5.1 ConcreteInfrastructure Class

The ConcreteInfrastructure class is the container for the catalog of concrete infrastructure elements that will be available to the current DOML configuration. Several concrete infrastructure instances may exist at the same time, each of them being part of a particular DOML solution.

Superclass

DOMLElement

Associations

providers: RuntimeProvider [0..*]	The list of runtime providers available in the catalogue
nodes: ComputingNode [0..*]	The list of independent computing nodes (not attached to a provider) available in the catalogue
generators: ComputingNodeGenerator [0..*]	The list of virtual machine and container images available in the catalogue
asGroups: AutoScalingGroup [0..*]	The list of independent auto scaling groups (not attached to a runtime provider) available in the catalogue
networks: Network [0..*]	The list of independent networks (not attached to a runtime provider) available in the catalogue

storages: Storage [0..*]	The list of storage resources that will be part of this abstract infrastructure model
faas: FunctionAsAService [0..*]	The list of concrete function as a service nodes provided by this concrete infrastructure

Usage

The ConcreteInfrastructure is a container element, used as the catalog of the concrete infrastructure elements used to deploy the final cloud application using DOML for a particular solution.

5.2 ConcreteElement Class (abstract)

The ConcreteElement class represents all concrete infrastructure elements that can have an abstract infrastructure element component mapped onto them.

Superclass

DOMLElement

Attributes

configurationScript: [0..1]	String	An optional URI to the script that has to be executed to correctly configure a node.

Usage

The ConcreteElement is intended to be used as the parent for more concrete elements of the concrete infrastructure model.

5.3 RuntimeProvider Class

The RuntimeProvider class describes a cloud resources provider (e.g. AWS).

Superclass

DOMLElement

Associations

supportedGroups: AutoScalingGroup [0..*]	The groups requested to the runtime provider.
vms: VirtualMachine [0..*]	The virtual machines that will be provided by the runtime provider.
networks: Network [0..*]	The networks requested to the runtime provider.
storages: Strorage [0..*]	The storages offered by this particular provided.
faas:FunctionAsAService [0..*]	The Faas services offered by this runtime provider

Usage

The RuntimeProvider is intended to model all the parameters related to a specific cloud IaaS provider.

5.4 VirtualMachine Class

The VirtualMachine class in the concrete layer represents a specific VM instance either provided by a runtime provider or configured by the user on their own infrastructure.

Superclass

ConcreteElement

Attributes

maps: infrastructure.VirtualMachine	The VM on the abstract infrastructure layer this concrete VM maps on.

Usage

The VirtualMachine is intended to be used as the concrete counterpart of the abstract VM defined in the infrastructure layer.

5.5 Network Class

The Network class in the concrete layer represents a specific network instance either provided by a runtime provider or configured by the user on their own infrastructure.

Superclass

ConcreteElement

Attributes

maps: infrastructure.Network	The network on the abstract infrastructure layer this concrete element maps on.

Usage

The Network is intended to be used as the concrete counterpart of the abstract Network defined in the infrastructure layer.

5.6 Storage Class

The Storage class in the concrete layer represents a specific storage service either provided by a runtime provider or configured by the user on their own infrastructure.

Superclass

ConcreteElement

Attributes

maps: infrastructure.Storage	The storage service on the abstract infrastructure layer this concrete storage maps on.

Usage

The Storage is intended to be used as the concrete counterpart of the abstract Storage defined in the infrastructure layer.

5.7 FunctionAsAService Class

The FunctionAsAService class in the concrete layer represents a specific functional logic service instance either provided by a runtime provider or configured by the user on their own infrastructure.

Superclass

ConcreteElement

Attributes

maps: infrastructure.FunctionAsAService	The faas instance on the abstract infrastructure layer this concrete element maps on.

Usage

The FunctionAsAService is intended to be used as the concrete counterpart of the abstract FunctionAsAService defined in the infrastructure layer.

5.8 AutoScalingGroup Class

The AutoScalingGroup class in the concrete layer represents a specific group instance either provided by a runtime provider or configured by the user on their own infrastructure.

Superclass

ConcreteElement

Attributes

maps: infrastructure.AutoScalingGroup	The group on the abstract infrastructure layer this concrete group maps on.

Usage

The AutoScalingGroup is intended to be used as the concrete counterpart of the abstract AutoScalingGroup defined in the infrastructure layer.

5.9 ExtConcreteElement Class

The ExtConcreteElement class is just used to represent an instance of a new infrastructure element concept that the user wants to add to DOML. This class is part of the DOML-E extension mechanisms.

Superclass

ConcreteElement, ExtensionElement

Usage

The ExtConcreteElement class is should be used to creat instances of concepts and metaclasses not currently available in DOML.

6 Optimization Layer

The following diagram shows the main elements of the Optimization Layer in DOML:

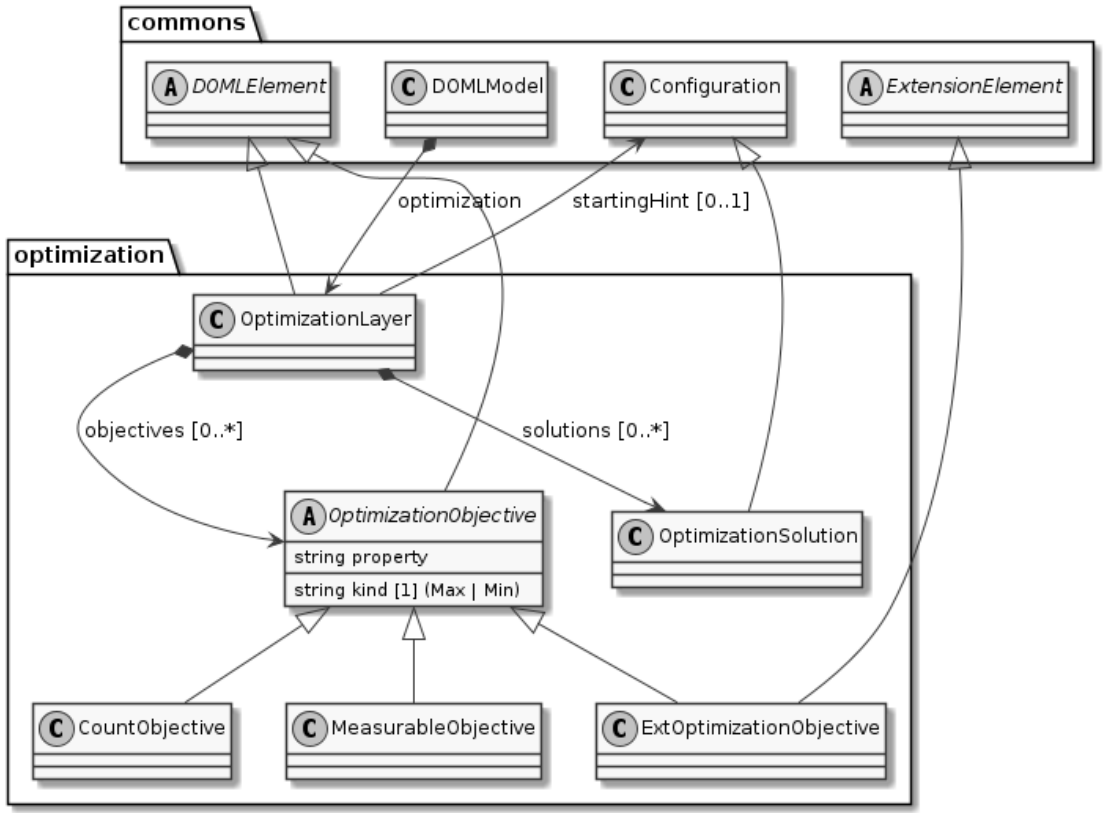

Figure 7. Optimization Layer diagram

6.1 OptimizationLayer Class

The OptimizationLayer class is the main cointainer for all the elements related to the definition and usage of the optimization algorithms in DOML.

Superclass

DOMLModel

Associations

objectives: [0..*]	OptimizationObjective	The set of objectives for the optimization algorithms.
solutions: OptimizationSolution [0..*]		All the solutions generated by the optimization algorithm.
startingHint: Configuration [0..1]		An optional configuration instance that will be used as a hint by the optimization algorithm.

Constraints

* At least one optimization objective should be provided to be able to use the model for optimization purposes.

Usage

The OptimizationLayer is intended to be used as a container for the objectives and solutions associated to the optimization algorithms for a DOML model.

6.2 OptimizationObjective Class (abstract)

The OptimizationObjective class represents a formal objective for an optimization algorithm. This objective will afterwards be used by the algorithms as an input to obtain a solution for the application deployment into the cloud infrastructure.

Superclass

DOMLElement

Attributes

property: String [1]	The property associated to this optimization objective.
kind: String [1]	The kind of objective, which can be either “max” or “min”.

Constraints

* The kind attribute may only have the “min” or “max” values.

Usage

The OptimizationObjetive is made abstract to serve as the basis for more concrete optimization objectives, such as objectives that measure a property, or objectives that are related to counting the number of different values of a property.

6.3 CountObjective Class

The CountObjective class represents an optimization objective that will count the different number of values associated to the property specified on them.

Superclass

OptimizationObjetive

Usage

The CountObjective is used to define optimization objectives which want to maximize or minimize the total number of values a property may take (e.g. minimize the number of locations for all the servers in a DOML solution).

6.4 MeasurableObjective Class

The MeasurableObjective class represents an optimization objective associated to the measurement of a particular property.

Superclass

OptimizationObjetive

Usage

The MeasurableObjective is used to define an optimization objetcive related directly to the value of a particular property (e.g. minimize the cost or maximize the throughput of a DOML solution).

6.5 OptimizationSolution Class

The OptimizationSolution class represents a Configuration of the current DOML model obtained through the usage of optimization algorithms.

Superclass

Configuration

Usage

The OptimizationSolution is a subclass of the main Configuration class in the commons package, as it is foreseen that any information related to the results obtained by the optimization algorithms (e.g. parameters, used requirements, etc.) could be added as additional information to this kind of Configuration instances.

6.6 ExtOptimizationObjective Class (abstract)

The ExtOptimizationObjective class is just used to represent an instance of a new optimization objective concept that the user wants to add to DOML. This class is part of the DOML-E extension mechanisms.

Superclass

OptimizationObjective, ExtensionElement

Usage

The ExtIOptimizationObjective class is should be used to creat instances of concepts and metaclasses not currently available in DOML.

7 DOML Text Syntax

The following figures define the current DOML syntax. This will evolve in the future releases based on the feedback by end users and the other PIACERE technical partners.

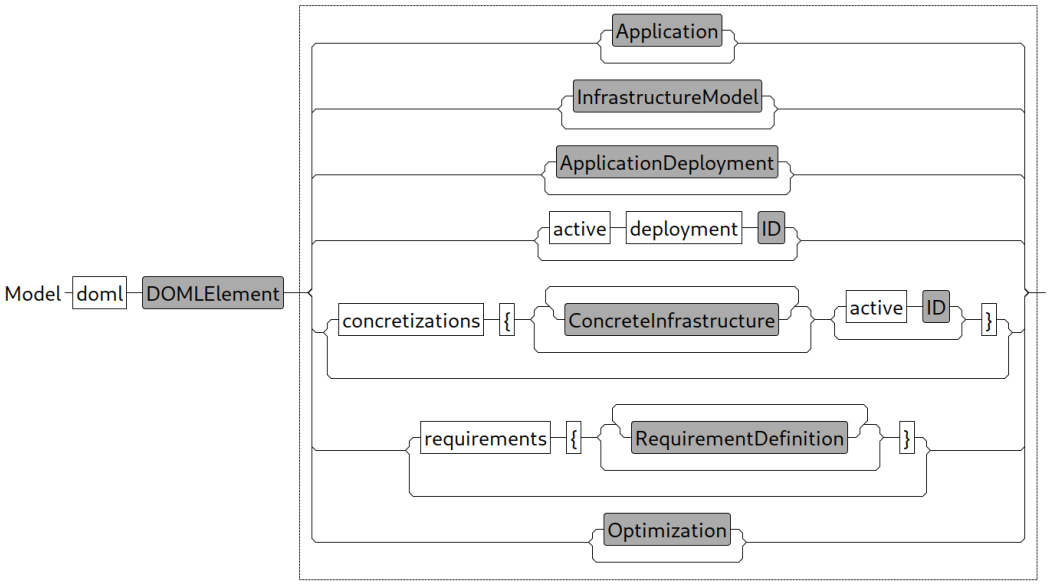

Figure 1. DOML top level model.

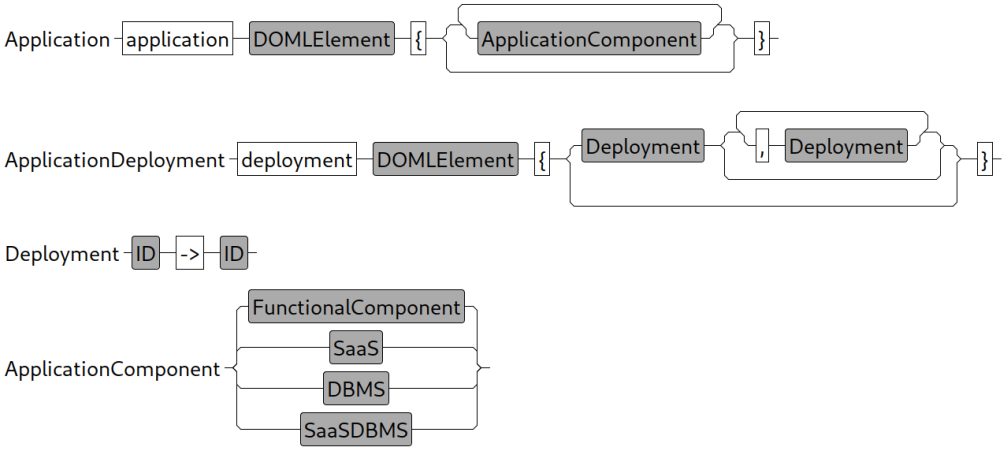

Figure 2. Application layer model.

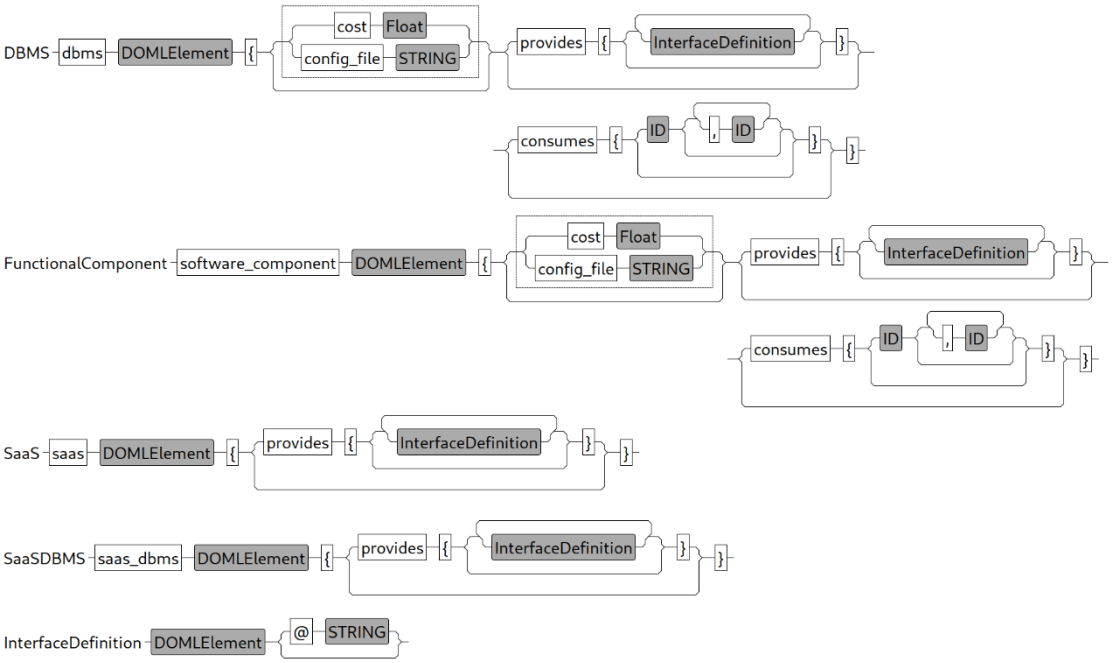

Figure 3. Application Components model.

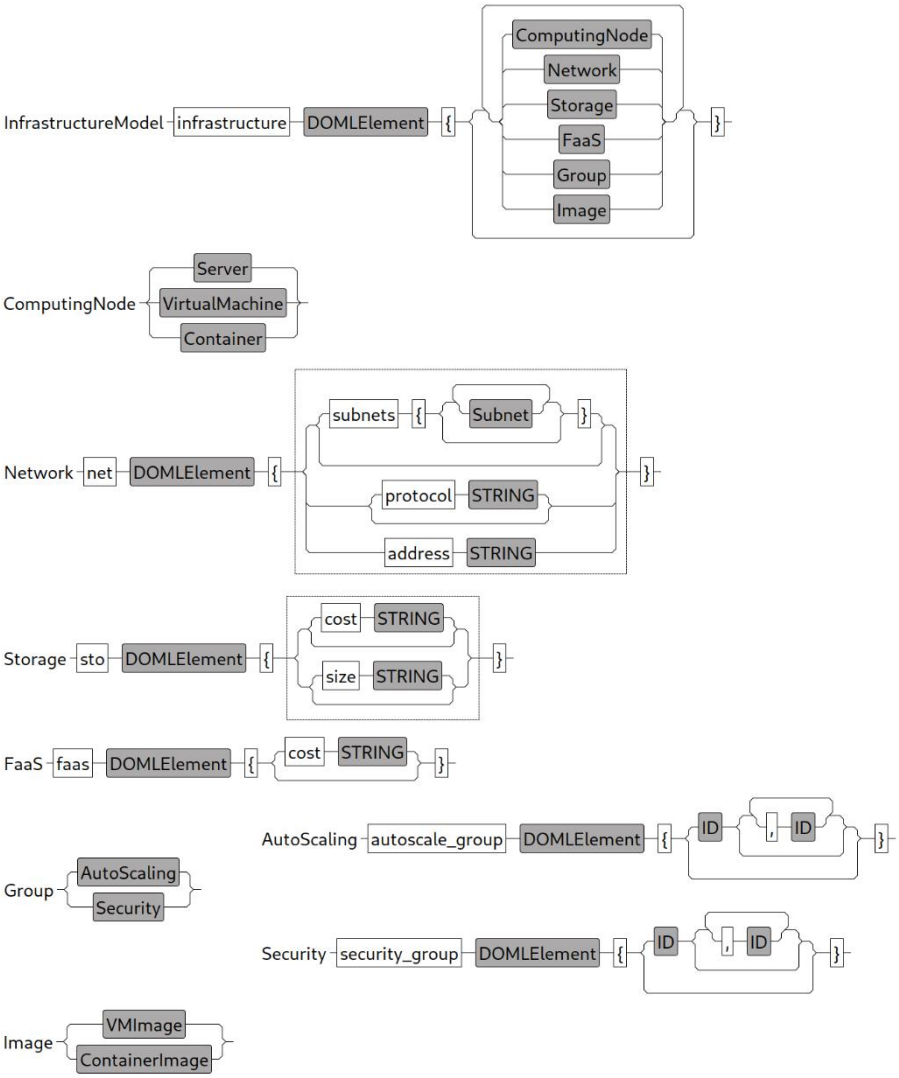

Figure 4. Abstract Infrastructure layer model.

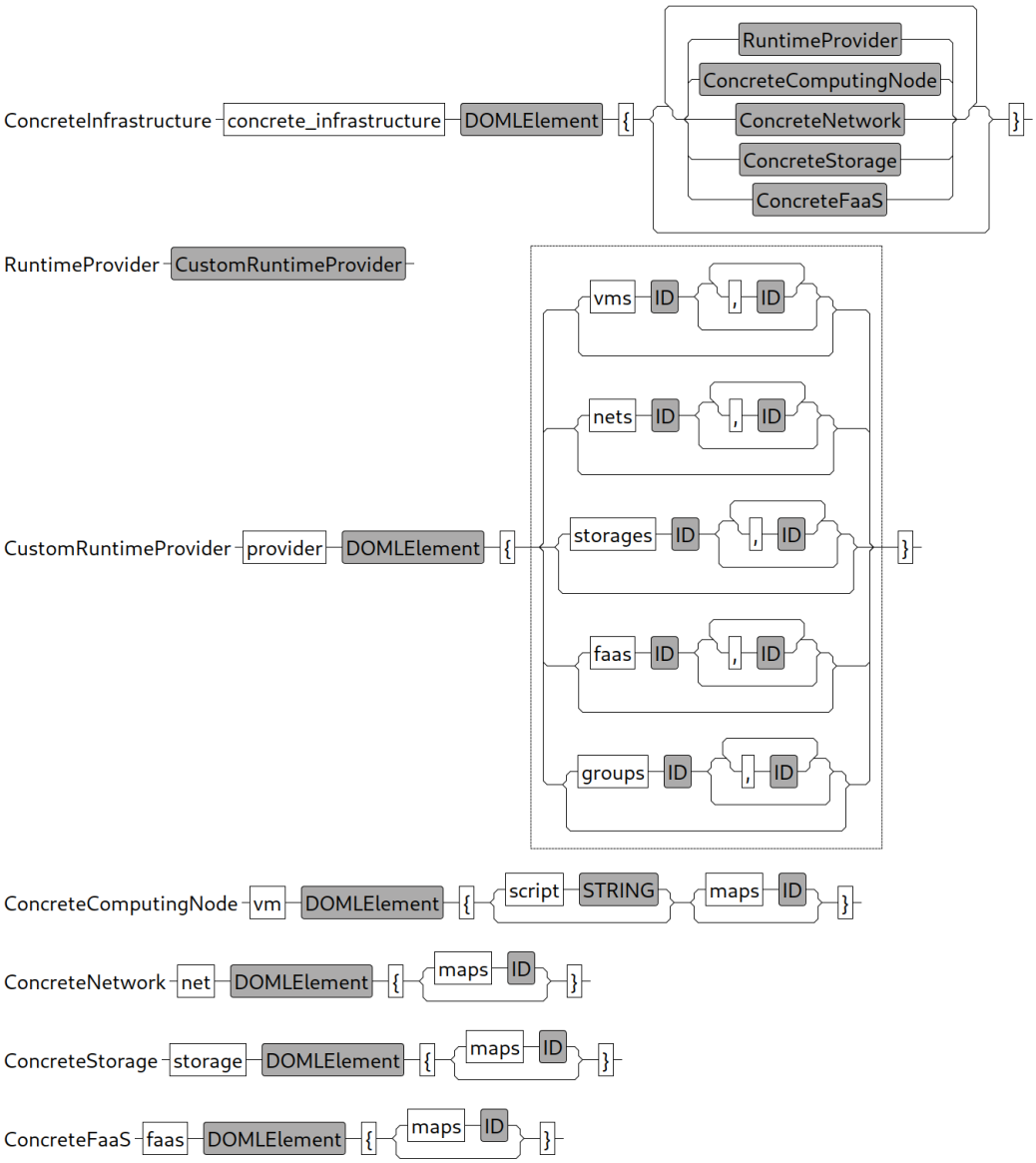

Figure 5. Concrete Infrastructure layer model.

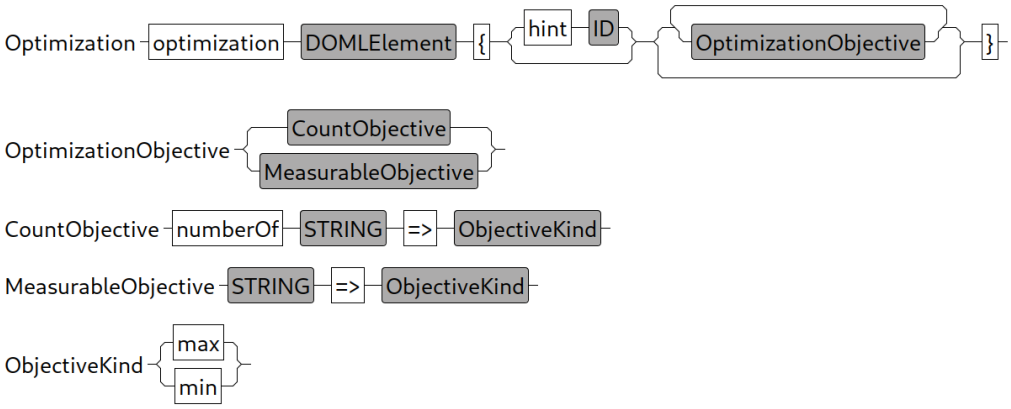

Figure 6. Optimization layer model.

8 DOML Examples

In this section we provide two simple DOML examples, showing their definition in the textual syntax and the corresponding translation in an XMI notation. Other examples are available in Deliverable D3.1.

8.1 Simple Web Application

This example describes a very simple DOML model with a web application that accesses an external API and a database and 2 IoT nodes that provide information to the application. The main software components will be running on virtual machines provided by a runtime provider, while the iotUnits will be running on physical nodes. The configuration has been made manually by the user.

A possible textual representation of this DOML model would be as follows:

```
doml simple
application simpleApp {
	dbms oracle {
		provides {
			db
		}
	}
	functional webapp {
		provides {
			logMessage
		}
		consumes { db, getWeather }
	}
	functional iotProvider {
		consumes { logMessage }
	}
	saas meteoAPI {
		provides {
			getWeather @ "https://api.mymeteo.com/get"
		}
	}
}
infrastructure infra {
	provider AWS {
		vms vm1, vm2
	}
	vm vm1;
	vm vm2;
	node iotNode1;
	node iotNode2;
}
deployment config1 {
	oracle -> vm1,
	webapp -> vm2,
	iotProvider -> iotNode1,
	iotProvider -> iotNode2
}
active config1
```

The example above shows the 2 main layers of DOML: application and infrastructure, as well as some elements in the commons layer (i.e. the configuration). As described in the example, the application layer contains 4 application components:

- The Oracle database software, defined by the DBMS metaclass. This component provides a software interface to access the database.
- The main web application software component, defined by the SoftwarePackage metaclass. As described in the example definition, the component consumes the

database interface and the external meteoAPI service. It also provides an interface for IoT components to send messages.

- The IoT software component, which will be deployed to all IoT nodes and that uses the logMessage interface of the web application to upload their data.
- The external API required by the web application, described by the SaaS metaclass. In the case of the external API, the service that it provides has been given an end point using a URL.

The infrastructure layer, which defines all the infrastructure elements catalogue available for the application layer, is also modelled as describen in the example:

- Two virtual machines are defined.
- The AWS runtime provider is also described, as the provider for the virtual machines.
- Two physical nodes are modelled as the IoT nodes deployed with the application.

Finally, a configuration is done by simply linking the application component to the corresponding infrastructure elements, and that configuration is configured as the active configuration.

The XML representation of the above model is shown below:

```
<?xml version="1.0" encoding="ASCII"?>
<commons:DOMLModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:app="http://www.piacere-
project.eu/doml/application" xmlns:commons="http://www.piacere-project.eu/doml/commons"
xmlns:infra="http://www.piacere-project.eu/doml/infrastructure" name="simple"
activeConfiguration="//@configurations.0">
  <application name="simpleApp">
    <components xsi:type="app:DBMS" name="oracle">
      <exposedInterfaces name="db"/>
    </components>
    <components xsi:type="app:SoftwarePackage" name="webapp"
consumedInterfaces="//@application/@components.0/@exposedInterfaces.0
//@application/@components.3/@exposedInterfaces.0">
      <exposedInterfaces name="logMessage"/>
    </components>
    <components xsi:type="app:SoftwarePackage" name="iotProvider"
consumedInterfaces="//@application/@components.1/@exposedInterfaces.0"/>
    <components xsi:type="app:SaaS" name="meteoAPI">
      <exposedInterfaces name="getWeather" endPoint="https://api.mymeteo.com/get"/>
    </components>
  </application>
  <infrastructure name="infra">
    <providers name="AWS" providedVMs="//@infrastructure/@nodes.0
//@infrastructure/@nodes.1"/>
    <nodes xsi:type="infra:VirtualMachine" name="vm1"/>
    <nodes xsi:type="infra:VirtualMachine" name="vm2"/>
    <nodes xsi:type="infra:PhysicalComputingNode" name="iotNode1"/>
    <nodes xsi:type="infra:PhysicalComputingNode" name="iotNode2"/>
  </infrastructure>
  <configurations name="config1">
    <deployments component="//@application/@components.0"
node="//@infrastructure/@nodes.0"/>
    <deployments component="//@application/@components.1"
node="//@infrastructure/@nodes.1"/>
    <deployments component="//@application/@components.2"
node="//@infrastructure/@nodes.2"/>
    <deployments component="//@application/@components.2"
node="//@infrastructure/@nodes.3"/>
  </configurations>
</commons:DOMLModel>
```

8.2 Optimization Problem Example

This example describes a DOML model that will be fed to an optimization service to obtain an optimal configuration for the cloud application according to a set of formal requirements. In this case the software components need to be deployed into some/all of the infrastructure elements in the catalogue.

The textual representation of the DOML model is shown below:

```
doml ^optimization
application optimizationApp {
      functional comp1 {

      }
      functional comp2 {

      }
      functional comp3 {

      }
}
infrastructure catalogue {
      provider AWS {
            vms size1, size2
      }
      vm size1 $"cost"="50",$"performance"="100",$"location"="Europe";
      vm size2 $"cost"="100",$"performance"="200",$"location"="America";
      provider Google {
            vms g1, g2
      }
      vm g1 $"cost"="15",$"performance"="50",$"location"="Europe";
      vm g2 $"cost"="75",$"performance"="120",$"location"="Asia";
}
optimization opt {
      numberOf "location" => min
      "performance" => max
      "cost" => min
}
```

As described before, the DOML captures all 3 software components in the application, and proposes 4 different virtual machines, from 2 different providers, that will be used to deploy the application. The latter model should be then sent as an input to the optimization algorithms to obtain a configuration that matches the objectives described in the optimiization layer.

The equivalent XML definition of the DOML model is shown below:

```
<?xml version="1.0" encoding="ASCII"?>
<commons:DOMLModel          xmi:version="2.0"          xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"      xmlns:app="http://www.piacere-
project.eu/doml/application"   xmlns:commons="http://www.piacere-project.eu/doml/commons"
xmlns:infra="http://www.piacere-project.eu/doml/infrastructure"
xmlns:optimization="http://www.piacere-project.eu/doml/optimization"
name="optimization">
  <application name="optimizationApp">
    <components xsi:type="app:SoftwarePackage" name="comp1"/>
    <components xsi:type="app:SoftwarePackage" name="comp2"/>
    <components xsi:type="app:SoftwarePackage" name="comp3"/>
  </application>
  <infrastructure name="catalogue">
    <providers            name="AWS"            providedVMs="//@infrastructure/@nodes.0
//@infrastructure/@nodes.1"/>
    <providers           name="Google"           providedVMs="//@infrastructure/@nodes.2
//@infrastructure/@nodes.3"/>
    <nodes xsi:type="infra:VirtualMachine" name="size1">
      <annotations key="cost" value="50"/>
      <annotations key="performance" value="100"/>
      <annotations key="location" value="Europe"/>
    </nodes>
    <nodes xsi:type="infra:VirtualMachine" name="size2">
```

```
      <annotations key="cost" value="100"/>
      <annotations key="performance" value="200"/>
      <annotations key="location" value="America"/>
    </nodes>
    <nodes xsi:type="infra:VirtualMachine" name="g1">
      <annotations key="cost" value="15"/>
      <annotations key="performance" value="50"/>
      <annotations key="location" value="Europe"/>
    </nodes>
    <nodes xsi:type="infra:VirtualMachine" name="g2">
      <annotations key="cost" value="75"/>
      <annotations key="performance" value="120"/>
      <annotations key="location" value="Asia"/>
    </nodes>
  </infrastructure>
  <optimization name="opt">
    <objectives xsi:type="optimization:CountObjective" kind="min" property="location"/>
    <objectives          xsi:type="optimization:MeasurableObjective"          kind="max"
property="performance"/>
    <objectives xsi:type="optimization:MeasurableObjective" kind="min" property="cost"/>
  </optimization>
</commons:DOMLModel>
```

9 Conclusions

This document has described the specification of the DOML language. The DOML has been conceived as a declarative language to make it easier for non-expert users, but it includes mechanisms to include imperative scripts and advanced features for expert user profiles.

DOML has also been designed taking into account the fast evolution of the cloud computing state-of-the-art, including mechanisms to extend itself easily, adding more concepts and properties to existing ones.